Elsevier Editorial System(tm) for Information Systems
Manuscript Draft

Manuscript Number: IS-D-15-41

Title: SOFA: An Extensible Logical Optimizer for UDF-heavy Data Flows

Article Type: Regular Article

Keywords: data flow optimization; user-defined operators; map/reduce

Corresponding Author: Ms. Astrid Rheinländer,

Corresponding Author's Institution: Humboldt-Universität zu Berlin

First Author: Astrid Rheinländer

Order of Authors: Astrid Rheinländer; Arvid Heise; Fabian Hueske; Ulf Leser, Prof. Dr.; Felix Naumann, Prof. Dr.

Abstract: Recent years have seen an increased interest in large-scale analytical data flows on non-relational data. These data flows are compiled into execution graphs scheduled on large compute clusters. In many novel application areas the predominant building blocks of such data flows are user-defined predicates or functions (UDFs). However, the heavy use of UDFs is not well taken into account for data flow optimization in current systems.

SOFA is a novel and extensible optimizer for UDF-heavy data flows. It builds on a concise set of properties for describing the semantics of Map/Reduce-style UDFs and a small set of rewrite rules, which use these properties to find a much larger number of semantically equivalent plan rewrites than possible with traditional techniques.
A salient feature of our approach is extensibility: We arrange user-defined operators and their properties into a subsumption hierarchy, which considerably eases integration and optimization of new operators. We evaluate SOFA on a selection of UDF-heavy data flows from different domains and compare its performance to three other algorithms for data flow optimization. Our experiments reveal that SOFA finds efficient plans, outperforming the best plans found by its competitors by a factor of up to six.

SOFA: An Extensible Logical Optimizer for UDF-heavy Data Flows

Astrid Rheinländer[a,*], Arvid Heise[b], Fabian Hueske[c], Ulf Leser[a], Felix Naumann[b]

[a] *Humboldt-Universität zu Berlin*
[b] *Hasso-Plattner-Institut for Software Systems Engineering Potsdam*
[c] *Technische Universität Berlin*

Abstract

Recent years have seen an increased interest in large-scale analytical data flows on non-relational data. These data flows are compiled into execution graphs scheduled on large compute clusters. In many novel application areas the predominant building blocks of such data flows are user-defined predicates or functions (UDFs). However, the heavy use of UDFs is not well taken into account for data flow optimization in current systems.

SOFA is a novel and extensible optimizer for UDF-heavy data flows. It builds on a concise set of properties for describing the semantics of Map/Reduce-style UDFs and a small set of rewrite rules, which use these properties to find a much larger number of semantically equivalent plan rewrites than possible with traditional techniques. A salient feature of our approach is extensibility: We arrange user-defined operators and their properties into a subsumption hierarchy, which considerably eases integration and optimization of new operators. We evaluate SOFA on a selection of UDF-heavy data flows from different domains and compare its performance to three other algorithms for data flow optimization. Our experiments reveal that SOFA finds efficient plans, outperforming the best plans found by its competitors by a factor of up to six.

Keywords: data flow optimization, user-defined operators, map/reduce

*Corresponding Author: Astrid Rheinländer, Humboldt-Universität zu Berlin, Department of Computer Science, Unter den Linden 6, 10099 Berlin, Germany; Phone: +49 30 2093 3024

Email addresses: rheinlae@informatik.hu-berlin.de (A. Rheinländer), arvid.heise@hpi.uni-potsdam.de (A. Heise), fabian.hueske@tu-berlin.de (F. Hueske), leser@informatik.hu-berlin.de (U. Leser), felix.naumann@hpi.uni-potsdam.de (F. Naumann)

Information Systems

The Editor

February 3rd, 2015

Dear Editor,

we hereby submit our manuscript entitled "SOFA: An Extensible Logical Optimizer for UDF-heavy Data Flows" for consideration for publication in Information Systems.

In many Big Data scenarios, user-defined functions (UDFs) are the predominant building blocks of analytical data flows. Such UDF-heavy data flows require specific optimization techniques as they usually carry a semantic that is very different from traditional relational operators. Our manuscript addresses exactly this problem, i.e., logically optimizing UDF-heavy data flows. We introduce SOFA, a novel and extensible optimizer for UDF-heavy data flows that builds upon a concise set of properties for describing the semantics of UDFs. SOFA finds a much larger number of semantically equivalent data flows than possible with traditional techniques. A salient feature of our approach is extensibility: We arrange user-defined operators and their properties into a subsumption hierarchy, which considerably eases extension of the system with new operators.

We note that SOFA was described in two other papers. A fully functional demonstration of SOFA was shown at SIGMOD 2014; the corresponding paper is four pages long, has many screenshots, and omits any technical details. Thus, the overlap with our submission here is very small. A previous version of our article appeared as technical report in the CoRR archive (CoRR/abs:1311.6335). As you certainly know, CoRR reports are not undergoing peer-review nor do they create any copyright issues. Compared to this report, the submitted manuscript is significantly extended and advanced: We added novel scalability experiments and a formal problem statement, extended the discussion on the operator annotations, re-draw almost all figures, and now provide an extensive appendix.

Thank you for considering our manuscript for review. We appreciate your time and look forward to hearing from you.

Kind regards,

Astrid Rheinländer
Arvid Heise
Fabian Hueske
Ulf Leser
Felix Naumann

Potential Peer Reviewers

We suggest the following persons for peer reviewing our submission due to their expert knowledge in data flow optimization, handling and optimizing UDF-heavy data flows, and large-scale data management in general.

Prof. Anastasios Gounaris
Aristotle University of Thessaloniki
Dept. of Informatics
541 24 Thessaloniki
GREECE
gounaria@csd.auth.gr

Dr. Paolo Missier
Newcastle University
School of Computing Science
Claremont Tower 4.24
Newcastle upon Tyne, NE1 7RU
United Kingdom
paolo.missier@ncl.ac.uk

Prof. Marta Mattoso
Federal University of Rio de Janeiro
Programa de Sistemas, COPPE/UFRJ
P.O. Box 68511
21941-972 Rio de Janeiro - RJ
Brazil
marta@cos.ufrj.br

Prof. Dr. Bernhard Mittschang
University of Stuttgart
Institute for Parallel and Distributed Systems
Universitätsstraße 38
D-70569 Stuttgart
Germany
Bernhard.Mitschang@ipvs.uni-stuttgart.de

SOFA: An Extensible Logical Optimizer for UDF-heavy Data Flows

Astrid Rheinländer[a,*], Arvid Heise[b], Fabian Hueske[c], Ulf Leser[a], Felix Naumann[b]

[a] *Humboldt-Universität zu Berlin*
[b] *Hasso-Plattner-Institut for Software Systems Engineering Potsdam*
[c] *Technische Universität Berlin*

Keywords: data flow optimization, user-defined operators, map/reduce

Highlights

- We identify a small yet powerful set of common Map/Reduce-style UDF properties influencing important aspects of data flow optimization.
- We show how these properties can be arranged in a concise taxonomy to ease UDF annotation, to enable automatic property inference, and to enhance extensibility of data flow languages.
- We present a novel optimization algorithm that is capable of rewriting DAG-shaped data flows given proper operator annotations.
- We evaluate our approach using a diverse set of data flows across different domains. We show that SOFA subsumes three existing data flow optimizers by enumerating a larger plan space and by finding more efficient plans with factors of up to six.
- Our experiments show that optimization as carried out with SOFA is even more beneficial when working on very large input data.

*Corresponding Author: Astrid Rheinländer, Humboldt-Universität zu Berlin, Department of Computer Science, Unter den Linden 6, 10099 Berlin, Germany; Phone: +49 30 2093 3024

Email addresses: rheinlae@informatik.hu-berlin.de (A. Rheinländer), arvid.heise@hpi.uni-potsdam.de (A. Heise), fabian.hueske@tu-berlin.de (F. Hueske), leser@informatik.hu-berlin.de (U. Leser), felix.naumann@hpi.uni-potsdam.de (F. Naumann)

Previous publication (Demo paper)

Click here to download Supplemental: p685-rheinlander.pdf

Previous publication (CoRR technical report)

Click here to download Supplemental: 1311.6335.pdf

SOFA: An Extensible Logical Optimizer for UDF-heavy Data Flows

Astrid Rheinländer[a,*], Arvid Heise[b], Fabian Hueske[c], Ulf Leser[a], Felix Naumann[b]

[a] *Humboldt-Universität zu Berlin*
[b] *Hasso-Plattner-Institut for Software Systems Engineering Potsdam*
[c] *Technische Universität Berlin*

Abstract

Recent years have seen an increased interest in large-scale analytical data flows on non-relational data. These data flows are compiled into execution graphs scheduled on large compute clusters. In many novel application areas the predominant building blocks of such data flows are user-defined predicates or functions (UDFs). However, the heavy use of UDFs is not well taken into account for data flow optimization in current systems.

SOFA is a novel and extensible optimizer for UDF-heavy data flows. It builds on a concise set of properties for describing the semantics of Map/Reduce-style UDFs and a small set of rewrite rules, which use these properties to find a much larger number of semantically equivalent plan rewrites than possible with traditional techniques. A salient feature of our approach is extensibility: We arrange user-defined operators and their properties into a subsumption hierarchy, which considerably eases integration and optimization of new operators. We evaluate SOFA on a selection of UDF-heavy data flows from different domains and compare its performance to three other algorithms for data flow optimization. Our experiments reveal that SOFA finds efficient plans, outperforming the best plans found by its competitors by a factor of up to six.

Keywords: data flow optimization, user-defined operators, map/reduce

1. Introduction

In recent years, the characteristics of data analysis tasks have changed significantly. One change is the increase in the typical amounts of data to be processed; in addition, the diversity of the data to be analyzed and the heterogeneity of analysis tasks has grown considerably. While in the past most analytics were performed on structured data using relational data processors, such as SQL OLAP, many applications today require complex analyses, such as heavy-weight machine learning, or graph traversal on large texts, graphs, semi-structured data sets, etc. Data processing systems support such analyses by providing system APIs for user-defined functions (UDF). Commonly, these UDFs are specified with imperative code, registered with the system, and called during execution. In fact, a large portion of Map/Reduce's popularity can be accounted to its widespread support for custom data processing [1].

A variety of data flow languages has been proposed that aim at (a) making the definition of complex analytics tasks easier and at (b) allowing flexible deployment of such data flows on diverse hardware infrastructures, especially compute clusters or compute clouds [2]. Many of these languages support UDFs [3, 4, 5]. Research has shown that proper optimization of such data flows can improve the execution times by orders of magnitude [6, 7, 8]. However, most of these systems focus on relational operators and treat UDFs essentially as black boxes, which considerably hampers optimization. At the same time, non-standard applications in areas such as information extraction, graph analysis, or predictive modelling often utilize UDF-heavy data flows. Traditional optimizers focus on relational operations, because their semantics in terms of optimization are well understood. In contrast, Map/Reduce-style UDFs can exhibit all sorts of behaviour, which are difficult to describe in an abstract, optimizer-enabling manner.

We address the three most prevailing challenges in optimizing UDFs on parallel execution engines, namely (1) defining the right UDF properties, since properly optimizing UDFs requires properties beyond the classical relational ones, (2) defining appropriate rewrite rules, since novel properties require novel ways of transforming queries, and (3) finding the right set of properties from describing a given UDF. This paper contributes to all three challenges.We present SOFA, a semantics-aware optimizer for UDF-heavy data flows. Compared to previous work, SOFA features a richer, yet concise set of general operator properties for automatic and manual UDF annotation. It is

*Corresponding Author: Astrid Rheinländer, Humboldt-Universität zu Berlin, Department of Computer Science, Unter den Linden 6, 10099 Berlin, Germany; Phone: +49 30 2093 3024

Email addresses: rheinlae@informatik.hu-berlin.de (A. Rheinländer), arvid.heise@hpi.uni-potsdam.de (A. Heise), fabian.hueske@tu-berlin.de (F. Hueske), leser@informatik.hu-berlin.de (U. Leser), felix.naumann@hpi.uni-potsdam.de (F. Naumann)

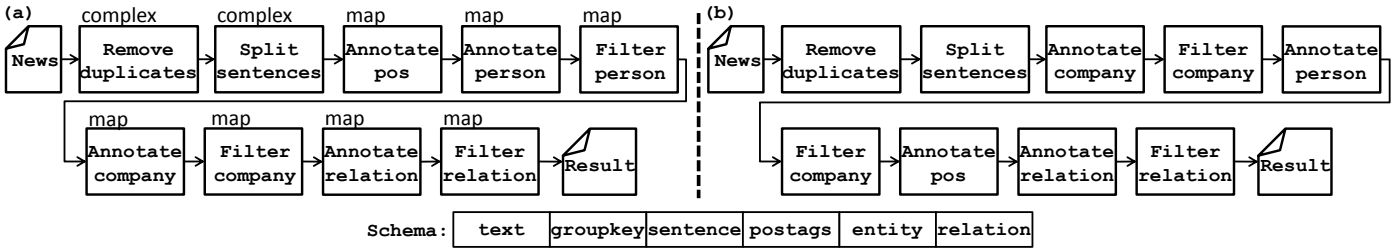


Figure 1: High-level data flow for employee relationship analysis. (a) Initial data flow, (b) reordered data flow based on operator semantics.

capable of finding a much larger and more efficient set of semantically equivalent execution plans for a given data flow than other systems. Given a concrete data flow, both automatically detected and manually created annotations are evaluated by a cost-based optimizer, which uses a concise set of rewrite templates to infer semantically equivalent execution plans.

Example. We use the following *running example* to explain the principles of Sofa throughout this paper: A large set of news articles shall be analyzed to identify persons, companies, and associations of persons to companies. We assume the articles stem from a web crawl and have already been stripped of HTML tags, advertisements, etc.; still the set contains many duplicate articles, as different news articles are often copied from reports prepared by a news agency.

An exemplary data flow for this task is shown in Figure 1(a). The first operator removes duplicates by first computing a grouping key followed by an analysis of each group for similar documents, such that detected duplicates are filtered out per group. Next, a series of operators performs linguistic analysis (sentence splitting and part-of-speech tagging), entity recognition (persons and companies), and relation identification (persons $\leftrightarrow$ companies). After each annotation operator, filter operators remove texts with no person, no company, or no person-company relation, respectively. As displayed in Figure 1(a), the data flow is composed of nine steps: seven maps, and two complex operators (first and second from left).

If Udfs are treated as black boxes, this data flow cannot be reordered. But when provisioned with proper information, such as data dependencies or operator commutativities, an optimizer has multiple options for reordering. For example, the part-of-speech tagger can be pushed multiple steps toward the end of the data flow, as annotations produced by this operator are necessary for relationship annotation only. Moreover, the entity annotation operators are commutative, as they independently add annotations to the text, but never delete existing ones. Thus, both annotation operators can be reordered for early filtering. Figure 1(b) displays an equivalent data flow with prospectively smaller costs as the most selective filters are executed as early as possible and expensive predicates are moved as much to the end of the data flow as possible. As we show in Sections 3 and 7, existing data flow optimizers cannot infer this plan.

A major obstacle to the optimization of Map/Reduce-style Udfs is their diversity. Our algorithm is developed in Stratosphere, a platform for data analytics on massive data sets [9]. In addition to the basic relational algebra operators, Stratosphere contains a diverse set of custom, domain-specific operators, each implementing one or more of basal stub types (e.g., map, reduce, join) available in many state-of-the-art large-scale data analytics systems. Available packages for general purpose (Base), information extraction (IE), and data cleansing (DC) already contain 14, 38, and 9 operators, respectively. Further packages, for example for machine-learning and web analytics, are in development. Defining semantic properties for each of these operators and rewrite rules that take operator semantics into account for each possible pair of operators would result in an unacceptable burden to the designer and would considerably limit extensibility and maintainability. Thus, every new operator in principle has to be analyzed with respect to every existing operator to specify possible rewritings. Sofa solves this problem by means of an extensible *taxonomy* of operators, operator properties, and rewrite templates called *Presto*. Sofa uses the information encoded in Presto to reason about relationships between operators during plan optimization; specifically, it leverages subsumption relationships between operators to derive reorderings not explicitly modelled. Presto considerably eases extensions, as novel operators can be hooked into the system by specifying a single subsumption relationship to an existing operator exhibiting the same behaviour with respect to optimization; these new operators are immediately optimized in the same manner as their parent. If desired and appropriate, more rewrite rules and operator properties describing the new operator may be introduced later in a "pay-as-you-go" manner [10].

In summary, our work includes the following contributions:

1. We identify a small yet powerful set of common Map/Reduce-style Udf properties influencing important aspects of data flow optimization.
2. We show how these properties can be arranged in a concise taxonomy to ease Udf annotation, to enable automatic property inference, and to enhance extensibility of data flow languages.

3. We present a novel optimization algorithm that is capable of rewriting DAG-shaped data flows given proper operator annotations.
4. We evaluate our approach using a diverse set of data flows across different domains. We show that SOFA subsumes three existing data flow optimizers by enumerating a larger plan space and by finding more efficient plans with factors of up to six.
5. Our experiments show that optimization as carried out with SOFA is even more beneficial when working on very large input data.

This paper is structured as follows: Section 2 describes preliminaries. Section 3 gives an overview of our approach for data flow optimization. Details on Presto and SOFA are explained in Sections 4 and 5. Section 6 discusses related work. We evaluate our approach in Section 7 and conclude in Section 8.

2. Preliminaries

We study the optimization of deterministic data flows. A *record* is a potentially nested value tree consisting of objects, arrays, and atomic values. An unordered bag of records is called *data set*. An *operator* o transforms a list of input data sets $I = (I_1, \ldots, I_n)$ into a list of output data sets $O = (O_1, \ldots, O_m)$ by applying a UDF f to I. A *data flow* is a connected directed acyclic graph $D = (V, E)$ with the following properties: Vertices in D are either operators, data sources, or data sinks. Sources have no incoming and sinks no outgoing edges, respectively. Inner nodes with both incoming and outgoing edges are operators. Our data model does not require a schema definition in the first place, but operators might require that the processed records have a certain schema.

We say two operators $o_i, o_j \in V(D)$ are in a *precedence relation* if a path from o_i to o_j in D exists and o_j accesses information contained in attributes $a_n, \ldots, a_m$ that were modified or created by o_i. The set of precedence constraints, i.e., the set of all precedence relations between pairs of operators in D is denoted with $P(D)$. We call two deterministic data flows D, D' *semantically equivalent* (denoted with $D \equiv D'$), if D and D' produce the same output sets O given the same input datasets I. Note that intermediate results may differ. Now, we can formulate the problem we address in this paper as follows:

Problem Statement: Given a data flow D and a cost function *costs*, find the set of precedence constraints $P(D)$ for operator execution orders in D to infer a set of semantically equivalent data flows $\mathcal{S}$, such that for each data flow $D' \in S$ $P(D) = P(D')$ holds. We call a set of data flows $S' \subseteq S \cup \{D\}$ optimal with respect to *costs* if argmin $costs(D')$ holds for each $D' \in S'$. Finally, one data flow $D' \in S'$ is selected for parallel execution.

In contrast to well-known relational operators, user-defined operators[1] can exhibit all sorts of behaviours. Stratosphere [9] currently contains more than 60 user-defined operators from three different domains. These operators will be used to explain our approach for data flow optimization throughout this paper. For a complete overview of the operators and a description of the Stratosphere system stack, see Appendix A.

User-defined operators can be either *abstract* or *concrete*; for instance, an operator for annotating person names in texts is abstract, and its instantiations are different concrete algorithms and tools for performing this task. Concrete operators may use very different implementations; e.g., the recognition of person names may be performed using dictionaries, patterns, or machine-learning-based methods.

Concrete operators can either be *elementary* or *complex*. Elementary operators are implemented using a single second-order function with a concrete execution semantics, complex operators are composed of multiple elementary operators similar to macros in programming languages. Complex operators are of high practical relevance, as they provide a shortcut for adding multiple elementary operators to a query. They are also important for data flow optimization, since a complex operator may exhibit different semantics than its elements.

The algebraic plan of the data flow for our running example is shown in Figure 2(a) together with properties and inferred schema information (cf. Figure 2(b)), which are used for optimization. Figure 2(c) displays that a complex operator may exhibit different properties than its elementary components: the complex operator *splt-sent* has different read/write set annotations and different I/O ratios than its elementary components. In the following section, we demonstrate how this plan can be reordered substantially by exploiting information on operator semantics.

3. Data flow Optimization

SOFA is an optimizer for rewriting UDF-heavy data flows into semantically equivalent data flows whose expected efficiency is higher, according to a cost model. Rewriting depends on a set of rewrite rules, each defining valid manipulations of data flow sub-plans, such as a reordering of two filter operations [11]. The novelty of SOFA lies in its flexible and extensible treatment of Map/Reduce-style UDFs going far beyond the capabilities of existing approaches. In this section, we highlight the advantages of SOFA by means of our running example.

Starting from a data flow D, data flows semantically equivalent to D may be produced using different transformation techniques. SOFA is capable of introducing, removing, and reordering operators. Complex operators are

[1] We use the terms *user-defined operators* and *user-defined functions* synonymously in this paper to refer to non-relational operators integrated into parallel data analytics systems by developers.

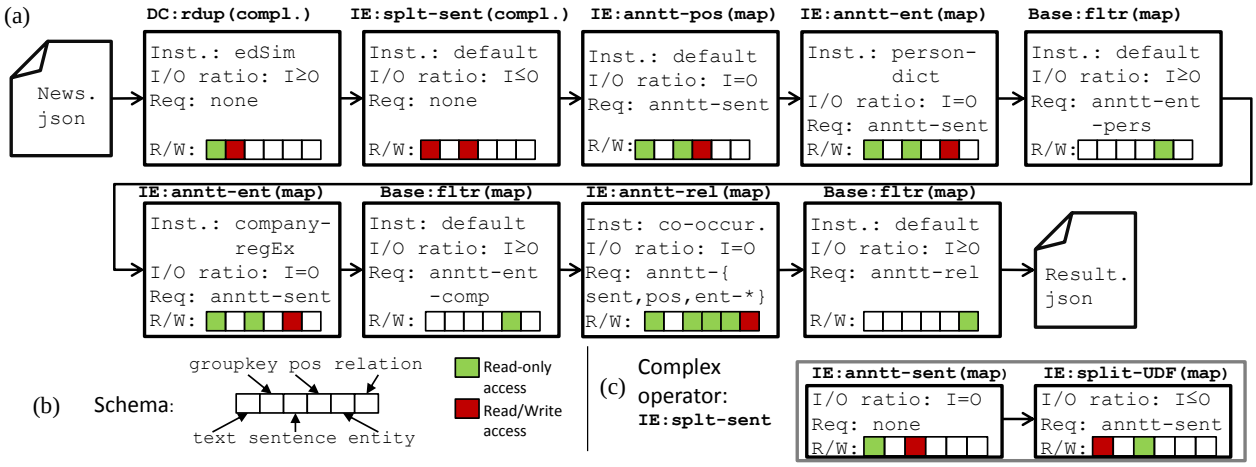


Figure 2: Algebraic data flow for the running example. (a) displays concrete operator instantiations together with properties relevant for optimization. Coloured boxes indicate read/write access on record attributes, which are part of the global schema shown in (b). (c) shows the resolution of the complex *splt-sent* operator (second operator in (a)) into its components *anntt-sent* and *split-UDF*.

optimized both as a whole and after expansion. In the following, we focus on reordering and treatment of complex operators, which are the most intricate and most effective optimization techniques.

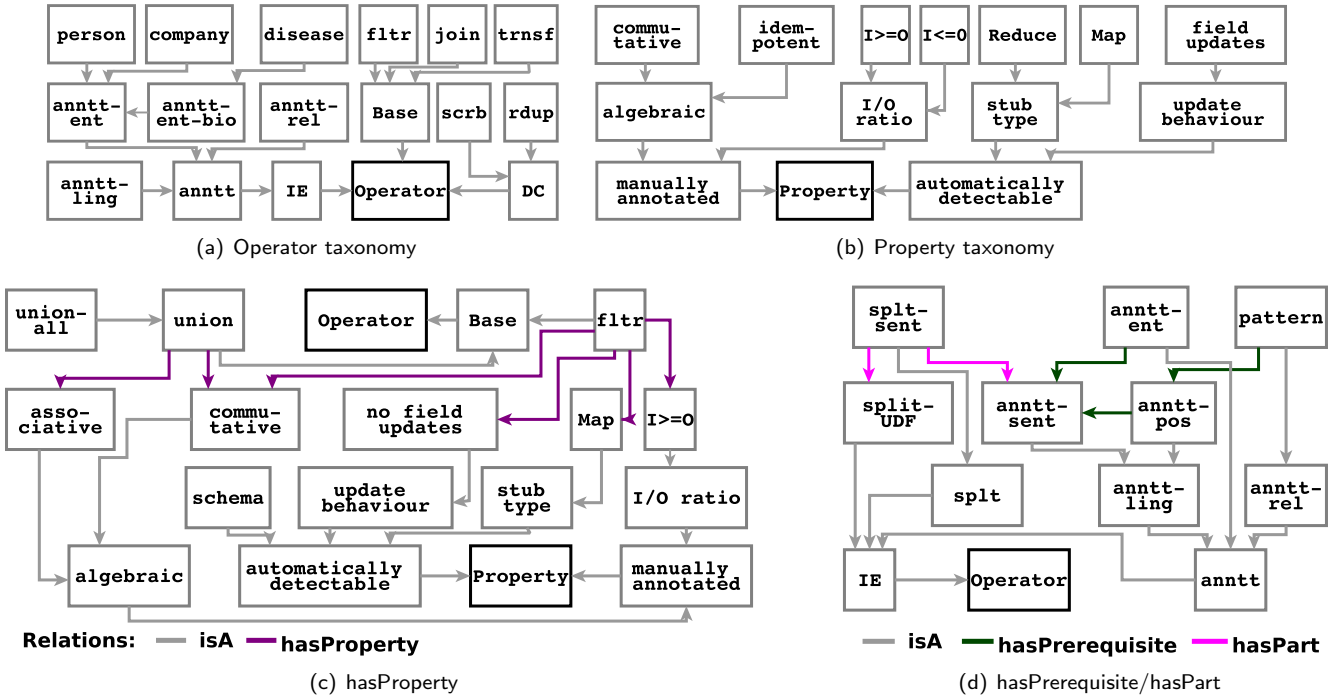
Existing approaches for data flow optimization enable reorderings by using either manually defined rewrite rules for relational operators [5] or by performing some kind of code analysis [6, 7]. The approach of Hueske et al. [7] probably is the most general, as it automatically derives data flow reorderings based on read/write set analysis of individual UDFs. In particular, the order of two subsequent tuple-at-a-time operators may safely be switched if they have no read/write or write/write conflicts on any attribute. The data flow shown in Figure 2 allows only one such beneficial reordering: The *anntt-pos* operator that annotates part-of-speech tags and stores them in the fourth attribute (first row, third from right) can be pushed before the *anntt-rel* operator (second row, second from right), because part-of-speech annotations are accessed only during relation annotation. This reordering most likely saves time, because the different *fltr* operators are now executed before *anntt-pos* and thus fewer sentences have to be annotated. Moving *anntt-pos* towards the start of the data flow is not possible, because it reads annotations produced by the complex *splt-sent* operator.

Semantics-aware rewrite rules allow to reorder the data flow in Figure 2 more extensively. Consider the two *anntt-ent* operators. Both write into the same attribute (the fifth), which collects all entity information. If the optimizer knows that annotation operators only *add* values and never delete or update existing values, these operators together with their subsequent filter operators may be reordered. The best order very likely is the one that filters the most sentences first; this decision can make use of selectivity and execution time estimates at the operator level (see Section 5.3). Furthermore, the optimizer can decompose complex operators and reorder the components individually. For instance, *splt-sent* consists of an *anntt-sent* operator and a UDF splitting the input text into separate sentences based on the annotations produced by *anntt-sent*. As shown in Figure 2(c), the two components of *splt-sent* differ in terms of read and write access on attributes and I/O ratio. Pushing split-UDF some steps towards the end of the plan is valid, because all succeeding *anntt* operators perform their analyses sentence-based, since all *anntt* operators for entity, relation, and part-of-speech annotation read sentence annotations. This reordering is likely beneficial, as the split-UDF generates multiple output records for each incoming record, depending on the number of annotated sentence boundaries.

In summary, semantics-aware plan rewriting allows us to pick a plan (with respect to cost estimates) from a larger set of equivalent data flows compared to other existing approaches. For instance, SOFA finds 4,545 distinct plans for the running example, compared to only 512 plans found with the read/write-set analysis of [7] (see Section 7 for a detailed comparison).

4. Annotating and Rewriting Operators with Presto

To enable optimizations such as those shown in the previous section, operators need to be annotated with meta data, for instance to describe selectivity estimates or semantic properties, such as associativities or commutativities. In this section, we introduce Presto, an extensible taxonomy for annotating and rewriting operators. Presto consists of two major components, the operator-property graph for modelling relationships between operators and properties (see Section 4.1), and a set of rewrite templates for data flow rewriting (see Section 4.2). When designing Presto, we paid special attention to extensibility by allowing enhancements to the semantic operator descriptions



(a) Operator taxonomy (b) Property taxonomy

(c) hasProperty (d) hasPrerequisite/hasPart

Figure 3: Exemplary subgraphs of Presto; root nodes are displayed in bold.

over time, to more and more unleash their optimization potential (see Section 4.3).

4.1. Operator-Property Graph

The operator-property graph in Presto contains two taxonomies for classifying *operator* and *properties*. Both taxonomies are self-contained and model generalization–specialization relationships (*isA*) between operators and properties, respectively. Figures 3(a) and 3 (b) display subgraphs of Presto. For example, the *anntt* operator has two specializations: *anntt-ent* and *anntt-rel* shown in Figure 3(a). Each leaf in the operator taxonomy describes a concrete implementation of the parent operator. Concrete implementations of operators can be very diverse: Similar to the relational world where multiple algorithms for the join operator exist, a non-relational operator for removing duplicates can be implemented naïvely in a theta join, by using a multi-pass sorted neighborhood approach, or through other advanced algorithms [12]. These implementations have different properties, e.g., a theta join is an elementary operator which is implemented in a "cross" second order function, whereas the sorted neighborhood implementation is a complex operator consisting of multiple elementary operators. Depending on which implementation of the duplicate removal operator is chosen (this can either be defined by the user or the system selects a default implementation), the potential for data flow reordering and the concrete data flow transformations may differ. The design of the operator taxonomy allows us to uniquely identify available operator instantiations, to use subsumption to effectively assign properties and relationships to operators, and to deduce rewrite options. Note that such abstraction-implementation relationships are an established concept in relational optimizers. However, in the relational world the hierarchies are very flat; they become much deeper when dealing with domain-specific UDFs.

As shown in Figure 3(b), we distinguish between automatically detectable properties and properties that are annotated by the package developer. The latter comprises algebraic properties (e.g., commutativity, associativity), processing type (record-at-a-time, bag-at-a-time), and the ratio between the number of input and output records. Automatically detectable properties comprise the parallelization function of the operator implementation (e.g., map, reduce), schema information available at query compile time, the number of inputs, and the read/write behaviour. Note that Presto defines only three manual and four automatically detectable property types; assigning these in an effective and intuitive manner to large sets of UDFs is the core rationale behind Presto.

Relationships connect operators and properties. Each specialization inherits all properties and relationships that are defined for the corresponding generalizations. For instance, the *union-all* shown in Figure 3(c) is a specialization of the *union* operator and thus inherits the algebraic properties defined for *union*. Complex operators can be characterized with respect to their components using the *hasPart* relation (Figure 3(d)). For example, the complex operator *splt-sent* consists of the two components *anntt-sent* and *split-UDF*.

Next to *isA* and *hasPart*, we define a *hasProperty* and a *hasPrerequisite* relation. *HasProperty* is a binary relation

between an operator and a property and is used to characterize operator semantics. For instance, the following properties are attached to *fltr* (Figure 3(c)):

- is implemented with a Map function,
- does not modify inside fields,
- input $\geq$ output,
- is commutative to other *fltr* instantiations.

Precedence constraints between operators are captured with *hasPrerequisite(X, Y)*, which states that operator X must be executed before operator Y. In Figure 3(d) it is shown that *anntt-rel* based on linguistic patterns requires part-of-speech and entity annotations to be performed in advance. Since *anntt-ent* itself requires sentence annotation and *hasPrerequisite* is a transitive relation, it is necessary to apply *anntt-sent* before *anntt-rel*.

The *isA* relationship very much simplifies deriving novel rewrite options for operators that are initially not well annotated. Suppose, the data scrubbing operator *scrb* from the DC package is initially not equipped with any *hasProperty* relationships. Later, the developer may see that *scrb* is a specialization of the well-annotated Base operator *trnsf*, i.e., both operators perform write operations in attributes of the incoming records. By formally specifying this through an *isA* relationship, *scrb* inherits all properties defined for *trnsf* (not shown in Figure 3(a)).

Though the complete Presto graph is too large to show here, it is still rather small and easy to understand: The property taxonomy contains 32 nodes and the operator taxonomy 78 nodes. Note that new packages mostly extend the operator taxonomy, while the property taxonomy is a fairly stable structure in our experience.

4.2. Rewrite Templates

We perform data flow rewriting using a set of rules specifying semantically valid reorderings, insertions, or deletions of operators. Because rewrite rules apply to combinations of operators, and because the different independently developed and maintained packages available for Stratosphere already contain more than 60 individual operators, it is practically impossible to define all rewrite rules across the different packages one-by-one. Instead, we define a concise set of rewrite templates using Presto relationships and abstract operators as building blocks. Reasoning along relationships allows Sofa to automatically instantiate the templates with concrete operators and thus enables us to derive individual rewrite options for concrete operator combinations on-the-fly. Currently, Sofa requires only 11 rewrite templates, which are expanded to over 150 individual rewrite rules.

Listing 1 displays a subset of the available templates in Datalog notation; further rules cover different reorderings based on algebraic properties as well as insertion and removal of operators . The complete set of rewrite templates with examples can be found in Appendix B. The first three templates are static and can be evaluated at package loading time, whereas the last two templates are dynamic and are evaluable at query compile time only. The first template covers commutative operators and expresses that two consecutive appearances of operators X annotated as associative in Presto can be safely reordered. Specifically, the goal `reorder(X,X)` evaluates to true if Presto contains a *hasProperty*-relationship of X with the property *associative*. Note that associativity does not necessarily need to be defined directly on X; the rule also applies if some ancestor of X in Presto is marked as associative. This fact is inferred using inheritance rules for reasoning over the Presto graph. The second template (Line 4–5) enables reordering of operators based on the *isA* relation and states that for any three operator instantiations X, Y, Z, the operators X, Y are reorderable given that X is not a prerequisite of Y, X is a specialization of Z, and Y, Z are reorderable. We include the goal `not hasPrerequisite(Y,X)` in the templates to ensure that operator precedences are respected. The third template (Lines 7–8) enables reorderings of consecutive *anntt* operators X, Y, when X is not a prerequisite of Y.

```
1  reorder(X,X) :- hasProperty(X,'associative'),
2    isA(X,'operator').
3
4  reorder(X,Y) :- not hasPrerequisite(Y,X),
5    isA(X,Z), reorder(Z,Y).
6
7  reorder(X,Y) :- not hasPrerequisite(Y,X),
8    isA(X,'annotate'),isA(Y,'annotate').
9
10 reorder(X,Y) :- hasProperty(X,'single-in'),
11   hasProperty(X,'RAAT'), hasProperty(Y,'RAAT'),
12   hasProperty(Y,'single-in'),
13   noReadWriteConflicts(X,Y).
14
15 reorderWithLeft(X,Y) :-
16   hasProperty(X,'dual-input'),
17   hasProperty(Y,'single-input'),
18   hasProperty(Y,'RAAT'),
19   not contains(readSet(Y),readSet(X)),
20   not contains(rightInputSchema(X),readSet(X)).
```

Listing 1: Rewrite template examples.

Dynamic rewrite templates are partly based on information not available before the query is compiled, for example, information on concrete attribute access by operators is available only after posing a query. Template 4 (Lines 10–13) enables reordering of two single-input record-at-a-time operators if these operators have no read/write conflicts. This single rule essentially covers most optimization options achieved by [7], which shows the power of our approach.

While most rules in Presto are generic and apply to many operator combinations, other rules are more specific. Suppose, we are given a data flow that consists of an equi-join of two data sources I_1, I_2 followed by *trnsf* that transforms only attributes of I_1, which are not part of the join condition. This data flow can be rewritten into an equivalent data flow, which first applies *trnsf* to I_1 and afterwards joins I_1 and I_2:

Similar to extending Presto with new operators and properties, package developers can also extend the set of rewrite templates to enable data flow optimization for their concrete application domain. For example, the third template was added by the IE package developer, since it enables reordering *anntt* instantiations, which is not supported by any other Presto template.

4.3. Pay-as-you-go annotation of operators

A key feature of Sofa is its extensible design, which significantly reduces the effort for annotating properties of new operators based on the subsumption hierarchy and inheritance mechanisms contained in Presto. By adding an *isA*-relationship for some operators X and Y, X inherits all properties from Y, and thus, X is optimized in the same manner as Y. Similarly, the property taxonomy can be extended with new properties (e.g., requires sorted input, computes aggregate function) and novel rewrite templates using these properties can be added if required for optimizing novel operators. Consider a new package for web analytics to be integrated into Stratosphere, containing an operator *rmark* for detecting and removing HTML markup in web pages. Initially, this operator would probably not be equipped with any Presto annotations. In this case, the Sofa optimizer can infer only automatically detectable properties, i.e., reordering can be performed only on the basis of read/write-set analysis. Later, the package developer invests some thought and annotates that *rmark* outputs as many records as incoming ($|I| = |O|$). Sofa infers from the set of automatically detectable properties, that *rmark* is a single-input operator implemented with a map function. Taken these properties together, the last template of Listing 1 becomes applicable to *rmark*. A full specification of *rmark* would include the definition of *isA* relationships to other operators. Actually, *rmark* has the same semantics as the *trnsf* operator from the Base package, as it essentially performs a transformation of the input texts. Now all templates valid for *trnsf* become applicable, such as the rule for reordering a *join* and a *trnsf* operator introduced in Section 4.2. Given that *rmark* accesses only attributes present in input I_1 that are not part of the join condition, Sofa can then reorder a data flow containing *rmark* and *join* as follows:

5. Optimization Algorithms

Sofa enumerates plan alternatives for a given data flow using algebraic transformations based on the available rewrite templates. Instead of explicitly formulating each possible order of executing any two operators as done for the concise set of relational operators, our rewrite rules consist of rather general operator properties modelled in the Presto taxonomy. Given a data flow D, Sofa performs two passes of the following three steps. First, D is analyzed for precedence relationships between operators based on rewrite templates and operator properties contained in Presto. This analysis yields a precedence graph, which is used in the plan enumeration phase, to secondly enumerate and thirdly rank valid plan alternatives based on a cost model. Afterwards, the complex operators contained in D are resolved into their elementary components and the three steps are repeated. Finally, the best plan is selected, translated, and physically optimized for parallel execution by the underlying execution engine (see [9] for details on this step). Note that the enumeration algorithm of Sofa is not complete as it does not explore any possible combination of complex and elementary operators together in a plan, so we cannot guarantee that Sofa finds the best possible plan. Yet, we will show in the following that Sofa always picks the plan with smallest estimated costs from the search space.

5.1. Precedence analysis

Presto models dependencies between operators either explicitly on the basis of the *hasPrecedence* relation or inferred, if the goal `reorder(X,Y)` fails for two operator instantiations X, Y.

The precedence graph construction starts by creating the directed transitive closure D^+ of the given data flow, which explicitly models all pairwise operator execution orders in D. The algorithm then inspects D^+ to detect and remove edges that are not logically required. It retains all edges incident to a data source or a data sink to prevent reordering of sources and sinks. The goal `reorder(X,Y)` is instantiated with start and end node of each edge (u, v) and the inference mechanism tries to resolve the goal based on the operator properties and rewrite templates stored in Presto. If successful, both nodes are reorderable and the edge (u, v) is removed from the precedence graph. Precedence analysis is a polynomial time algorithm; its complexity is determined by computing the transitive closure in $O(|V|^3)$ using the Floyd-Warshall algorithm and the data complexity of stratified non-recursive Datalog, which we use for reasoning in Presto [13].

Figure 4 shows the final precedence graph for our running example (omitting data sources and sinks for readability). The displayed graph reflects precedences between DC, IE, and Base operators, e.g., *rdup* and *anntt-entperson* are a prerequisite for the $\mathit{fltr}_{person>0}$ operator, and *anntt-rel* is in a *hasPrerequisite* relation with *anntt-pos* (cf. Figure 3(d)). The graph contains edges between *anntt* and *fltr* reflecting that the concrete instantiations of *fltr* have read/write conflicts with their preceding *anntt* operators.

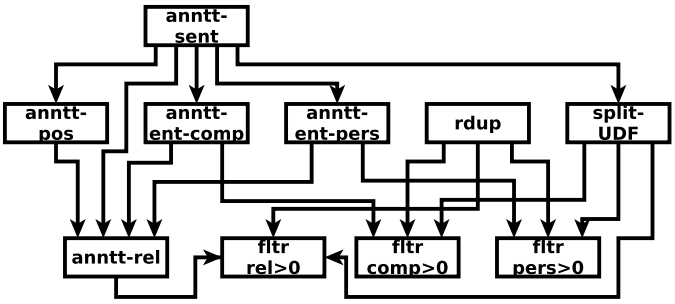


Figure 4: Precedence graph for running example with complex operator resolution.

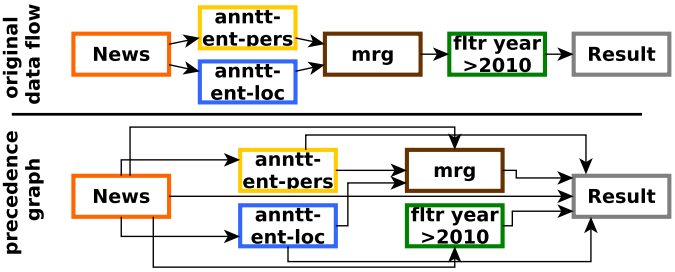


Figure 5: DAG-shaped data flow (top) and corresponding precedence graph (bottom) inspired by the running example.

5.2. Plan enumeration

Plan enumeration essentially generates different topological orders constrained by the precedence graph, while performing cost-based pruning. In contrast to topological sorting, the outcome are not full orders but DAG-shaped plans. The main idea is to iteratively construct alternative plans from data sinks to sources for a given data flow D by analyzing the corresponding precedence graph for operators that have no outgoing edges. Such operators are not required by any other operator and can therefore be added to the emerging partial plans. If multiple operators have no outgoing edges, the algorithm creates a set of alternative partial plans. The algorithm continues to pursue each alternative, removing the newly added operator from the precedence graph, estimating the costs of the partial plan (see Section 5.3), and pruning costly partial plan alternatives where possible.

We explain its principles using the simplified data flow shown in Figure 5 (top). Note that this data flow is DAG-shaped, which poses no problem to Sofa. The data flow performs task-parallel annotation of persons and companies. Annotations are subsequently merged, and the result set is filtered for articles published after 2010. The resulting precedence graph is displayed in Figure 5 (bottom).

The recursive plan enumeration algorithm is displayed in Figure 6. It takes as input the original data flow, the corresponding precedence graph, and a partial plan, which initially is empty (Lines 1–2). First, the algorithm selects the set of nodes from the precedence graph that have out-degree 0 (Line 8). These operators are not a prerequisite of any remaining operator and can thus be added to the partial plan (Lines 11–13) without violating precedence constraints. For each of these operators, alternative partial plans are constructed in the following loop (Lines 10–34). In our example, only the data sink can be selected. Once added to the partial plan, the selected node is removed from the precedence graph (Line 11-12). We determine the set *inputNodes* of operators contained in the partial plan having open inputs, i.e., at least one of the input channels of such an operator i is not connected to the output channel of some other operators or a data source preventing a proper functioning of i (Line 14). Since the partial plan was empty before adding the data sink, we cannot insert any edges in the partial plan and therefore, plan enumeration is recursively invoked again (Lines 16–17). Now, *mrg* and *fltr* both have no outgoing edges any more and are therefore added to the set of candidate nodes. Each candidate node is processed individually, added to the partial plan and removed from the precedence graph. This yields in two alternative partial plans, which are both inspected further.

The complete enumeration of all plan alternatives is shown in Appendix D. We exemplarily follow the plan with the *mrg* operator due to space constraints. The operator is added to the plan and subsequently, the set of *inputNodes* is divided into required and optional nodes (Lines 19–24). Required nodes are those nodes that have the currently added node as its direct predecessor in the original data flow, optional successors are all other operators contained in *inputNodes*. In our example, the set of required nodes is empty, and the set of optional nodes contains *fltr*. For each required node m, we create an edge (n, m) for the newly added node n, add it to the edge set of our partial plan, estimate the costs of the partial plan, and recursively call the plan enumeration algorithm (Lines 25–29). Each optional node l is processed individually. We iteratively create edges (n, l), estimate the costs of the new partial plan, and again recursively call the plan enumeration algorithm if necessary (Lines 30–34). A recursive invocation of the plan enumeration algorithm terminates either if the precedence graph is empty and an alternative plan has been found (Lines 4–7), or if no alternative plans with smaller costs compared to the initial plan were found (Line 33).

Pruning. The plan enumeration algorithm has exponential worst-case complexity (consider for instance a precedence graph without any edges). We included a simple technique for search space pruning in our algorithm preventing completion of partial plans whose estimated costs are higher than the estimated costs for the current best data flow. Once a cheaper plan was found, we update the costs of the best plan, in a manner similar to accumulated cost pruning in top-down query optimization [11, 14]. If no alternative plan with lower estimated costs compared to the best plan could be constructed, we terminate (cf. Figure 6, Line 33).

```
enumAlternatives(Graph precedGraph, Graph plan,
                     Graph partialPlan) {

  if (isEmpty(precedGraph)) {
    addPlanToResultSet(partialPlan);
    return;
  }
  candNodes = getNodesWithOutDegreeZero(precedGraph);

  foreach(Node n in candNodes) {
    addNodeToPartialPlan(n);
    removeNodeAndIncidentEdgesFromPrecedenceGraph(n);

    inputNodes = getNodesWithOpenInputs(partialPlan);

    if (isEmpty(inputNodes))
      enumAlternatives(precedGraph, plan, partialPlan);

    foreach(Node m in inputNodes){
      if (inputGraphcontainsEdge(n,m))
        addNodeToRequiredNodes(m);
      else addNodeToOptionalNodes(m);
    }
    if(not isEmpty(requiredNodes)) {
      addEdgesToAllRequiredNodesInPartialPlan(m);
      if (costs(partialPlan) < costs(originalPlan))
        enumAlternatives(precedGraph,plan,partialPlan);
    }
    foreach(Node l in optionalNodes) {
      addEdgeToPartialPlan(n,l);
      if (costs(partialPlan) < costs(originalPlan))
        enumAlternatives(precedGraph,plan,partialPlan);
    }
  }
  addPlanToResultSet(plan);
  return;
}
```

Figure 6: Plan enumeration with SOFA.

5.3. Cost estimation

To estimate costs and result sizes of a data flow, SOFA depends on estimates for key figures of operators, which can either be provided by the developer by adding appropriate annotations to Presto, by sampling from the input data, or by runtime monitoring of previously executed data flows. We estimate the costs of a plan by computing the weighted sum of estimated ship data volume, I/O volume, and CPU usage of the UDFs per stub call.

Specifically, the costs of an operator o_i are estimated as follows: let c_i be the average CPU usage of o_i per invocation, s_i the estimated startup costs of o_i, and r_i the estimated number of processed input items of o_i. Including startup times of operators is particularly important for complex non-relational UDFs, as many IE and DC operators need a long startup time for instance to load large dictionaries, or to assemble trained models. Furthermore, let d_i denote the estimated I/O costs of an input item processed by o_i, n_i the estimated shipping costs of an output item produced by o_i, and sel_i the selectivity of o_i. The estimated number of items r_i processed by an operator o_i is calculated as $r_i = \sum_{(h,i)\in E(D)} r_h * sel_h$. The costs of an operator o_i are estimated as the weighted sum of the estimated ship data volume, I/O volume, and CPU usage of o_i using the formula $costs(o_i) = w*(c_i*r_i+s_i)+u*(d_i*r_i)+v*(n_i*r_i*sel_i)$, where u, v, w denote weight constants for each cost component. Note that the formula for operator costs can be replaced with custom cost functions, which are added to Presto by the package developers. For example, to accurately estimate the costs for data flows containing IE operators, we also capture the *projectivity* of *anntt* operators, i.e., the average number of annotations produced by an *anntt* instantiation. Consequently, the selectivity of $fltr_{anntt}$ is denoted as $sel(fltr_{anntt}) = r_{i-1} * proj(anntt)$.

Finally, the total costs of a data flow D are estimated as $costs(D) = \sum_{i=1}^{n} costs(o_i)$. Note that our cost model optimizes for total computation time, disregarding parallelization in the underlying execution engine. Physical optimization of data placement and shipment between nodes handled downstream by the underlying parallel execution engine. During logical optimization with SOFA, we have no access to the information which concrete shipping strategy will be chosen. Therefore, we assume in our cost model that if two operators o_1, o_2 are implemented in a map stub and there is a data flow from o_1 to o_2, the data is not transferred over the network. In all other cases, we assume that the data is shipped over the network. If some dual-input operator o_3 receives inputs from operators o_4 and o_5, we compare the estimated size of the outputs of o_4 and o_5 and assume that the smaller output is transferred over the network. However, we see in Section 7 that this approach already allows us to correctly rank enumerated plan alternatives in many cases.

6. Related Work

Query optimization is a prominent topic in database research and many facets of our problem setting have been addressed before. The optimizer of the Starburst project leverages an extensible set of rules to rewrite query expressions [15]. Rules are specified as pairs of condition and action functions implemented in a procedural language [16]. Volcano [11] and Cascades [14] have extensible sets of transformation and implementation rules, which rewrite query expressions and compile them to physical execution plans. Chaudhuri et al. proposed declarative rewrite rules to improve the optimization of queries with external functions [17]. All aforementioned approaches deal with the optimization of relational queries and assume that rewrite rules are manually added to the optimizer framework. The optimization of relational queries with user-defined predicates has been another focus of research [18, 19, 20]. In these works, the semantics of the UDFs to be reordered is assumed to be uniform, i. e., filtering tuples. Consequently, these approaches focus on the problem of identifying the optimal order of filters and do not address the question whether general UDFs can be reordered or not; besides, they disregard the effects of second-order functions on first-order UDFs. Our work addresses the optimization of UDF-heavy Map/Reduce-style data flows for which additional information is required to answer this question. In our approach, this information is provided by the developer or inferred from Presto. Plus, our work does not require the explicit definition of

condition–action rules for new operators, which considerably improves extensibility.

Several methods to optimize more general data flows have been proposed. Ogasawara et al. propose an algebraic approach to define scientific workflows [21] and optimize their execution. Operators are classified as UDFs with strict templates or as relational expressions. While their classification of UDFs is somewhat similar to particular combinations of our operator properties, we regard our approach as more general as it features a richer set of properties that can be freely composed to precisely reflect the characteristics of an operator. Furthermore, their work does not contain a transformation-based optimizer. Simitsis et al. present an approach for optimizing ETL processes [22]. They introduce three different optimization techniques, namely reorderings of adjacent single-input/single-output operators that have no read/write-conflicts, merging and splitting of operators, and factorization of operators. Our approach is more general as SOFA is able to optimize arbitrary data flows and it is able to reorder any operator combinations given that precedence constraints are respected. Stubby is an optimizer for workflows constructed of multiple Map/Reduce jobs [23]. This approach is orthogonal to SOFA, as it leverages manual annotations of Map and Reduce functions to merge Map/Reduce jobs while preserving the logical order of operations. The MRQL framework performs physical algebraic optimizations of Map/Reduce data flows consisting only of relational operators [24], but disregard general UDFs. Both approaches identify valid transformations from static rules and operator properties, such as in- and output schema. Our approach differs in using an extensible taxonomy of a richer set of operator properties and rewrite rules to deduce precedence constraints. Work presented by Hueske et al. [7] on reordering operators in Map/Reduce-style data flows is based on specific information about the operator's behaviour in terms of accessed data attributes obtained by static code analysis. We included this idea in SOFA, but also show how semantic annotations that describe the behaviour of UDFs more precisely can help to further increase the space of possible rewritings; besides, our approach allows to rewrite DAG-shaped data flows, which is not possible with the algorithm presented in [7]. The SUDO optimizer [25] combines manual annotation and code analysis to analyze UDF properties with respect to data partitioning to avoid unnecessary data shufflings. This problem is orthogonal to SOFA, where we analyze semantic operator properties to reduce execution times by reordering operators. Pig Latin [5], JAQL [3], and Hive [26] are higher-level languages for Hadoop/Tez featuring limited optimization techniques. Both Pig and JAQL do not perform cost-based optimization, but apply a limited set of heuristic transformation rules (e.g., filter push downs, function and variable in-lining, or rules for optimizing field access) that most likely is beneficial and relies otherwise on the decisions of the programmer. Hive contains three mechanisms for data flow optimization: partition pruning, filter push-downs, and cost-based join reordering. Hortonworks is working on integrating the Apache Calcite query planning framework into Hive, which provides cost-based optimization for relational operators, but disregards the optimization of UDFs [2].

In summary, we believe that SOFA is the first extensible, fully functional optimizer for arbitrary DAG-shaped data flows for Map/Reduce-style systems.

7. Evaluation

We evaluated SOFA on a 28-node cluster, each equipped with a 6-core Intel Xeon E5 processor, 24 GB RAM, and 1TB HDD using Stratosphere 0.2.1.

Queries. We implemented, optimized, and executed seven UDF-heavy Meteor queries originating from different application domains. These queries are translated into algebraic data flows and handed to SOFA for logical optimization. The Meteor scripts we used for the evaluation are listed in Appendix D. **Q1** adopts the data flow described in our running example for relationship extraction from biomedical literature using IE and DC UDFs. **Q2** performs topic detection by computing term frequencies in a corpus grouped by year. The query first splits the input data into sentences, reduces terms to their stem, removes stop words, splits the text into tokens, and aggregates the token counts by year. **Q3** extracts NASDAQ-listed companies that went bankrupt between 2010 and 2012 from a subset of Wikipedia. This query takes article versions from two different points in time, annotates company names in both sets and applies different *fltr* operators and a *join* to accomplish the task. **Q4** corresponds to the data flow shown in Figure 5 and performs task-parallel annotation of person and location names. **Q5** analyzes DBpedia to retrieve politicians named 'Bush' and their corresponding parties using a mixture of DC and base operators. **Q6** is a relational query inspired by the TPC-H query 15. It filters the lineitem table for a time range, joins it with the supplier table, groups the result by join key, and aggregates the total revenue to compute the final result. **Q7** uses two complex IE operators to split incoming texts into sentences and to extract person names.

Data sets. We evaluated Q1 on a set of 10 million randomly selected citations from Medline, Q2 was evaluated on a set of 100,000 full-text articles from the English Wikipedia initially published between 2008 and 2012, Q3 was evaluated on two sets of English Wikipedia articles of 50,000 articles each, one set from 2010 and one set from 2012, Q4 and Q7 on a set of 100,000 full-text articles from the English Wikipedia downloaded in 2012, Q5 on the full DBpedia data set v. 3.8, and Q6 was evaluated on a 100GB

[2]Information is taken from the Hive online documentation (http://hive.apache.org) and slides provided by Julian Hyde (http://slideshare.net/julianhyde/w-435phyde-3). Last access of both sources: 2015-10-15.

relational data set generated using the TPC-H data generator. For each experiment, we report the average of three runs. Estimates on operator selectivities, projectivities, startup costs, and average execution times per input item were derived from 5% random samples of each data set.

Competitors. Although data flow languages for Big Data are a hot topic in current research, surprisingly few systems actually optimize the data flow at the logical level as we do. Thus, detecting appropriate competitors is difficult, because optimizers are commonly deeply coupled to a particular system. We reimplemented the ideas of three current data flow optimizers, namely techniques presented by Hueske et al. [7], Olston et al. [5], and Simitsis et al. [22]. We compare the number of plan alternatives found and the achieved runtime improvements. For each method, we disabled rules and information on operator properties stored in Presto and replaced them with the appropriate rewrite rules described in [7, 5, 22]. For the method of Olston et al., we referred to the online documentation of rewrite rules for Apache Pig, version 0.11.1. For Hueske et al., we enabled annotation of read- and write-sets, but disabled reordering of DAG-shaped plans.

Optimization time. The time needed to optimize a given data flow with Sofa depends heavily on the number of contained operators. For our evaluation queries, Sofa needed between 0.5 (Q6) and 14 seconds (Q3) to analyze and optimize the query. During optimization, most time is spent on the Datalog-based reasoning along relationships in Presto. However, time needed for optimization pays off quickly for data analytics tasks at large scale. In all our queries, the time spent on optimization amounts to a very small fraction of the time needed for executing the actual queries. For example, the non-optimized version of Q1 needs more than 18 hours to analyze 100k full-text documents, and the optimized version of Q1 analyzes this set of documents in less than 7 hours, whereas Sofa needed roughly 12 seconds to retrieve the best plan.

7.1. Finding optimal plans

A large number of semantically equivalent plans for a concrete data flow has the potential to contain the most effective variant. Therefore, we first evaluate Sofa to all three competitors with respect to the number of alternative plans found with each method. We turned search space pruning off and enumerated the complete space of alternative data flows for all queries. In Section 4, we explained how complex operators can be resolved into a series of interconnected elementary operators. Q1, Q2, and Q7 contain complex operators, thus, we enumerated the plan space for these queries both using only elementary operators and using combinations of elementary and complex operators. For the methods presented in [7, 22, 5], we used complex operators only, as these methods do not provide mechanisms for operator expansion.

As displayed in Table 1, Sofa enumerates the largest plan space in all cases. Note that Q1 and Q3 translate to data flows with 10 and 12 operators, respectively, which both contain many degrees of freedom. For example, Q1 and Q3 contain 3 and 5 filter operators (see Appendix D for the concrete scripts). Each filter can be positioned differently in the data flows yielding a high number of alternative plans. The method presented by Hueske et al. is unable to rewrite Q2, Q4, Q5, and Q7, because it is neither capable of rewriting DAG-shaped data flows (Q4, Q5) nor of expanding complex operators (Q2, Q7). The approach of Olston et al. can rewrite only Q3, Q4, and Q6, because these are the only methods that involve filter push-ups. Simitsis et al. find no alternative plans for Q2 and Q7, as in these cases, no adjacent single-input/single-output operators were reorderable. For Q3 and Q6, Sofa and [7] both enumerate the largest plan space, as for both queries the predominant rewrite options concerned *fltr* operators.

To evaluate the correctness of plan ranking performed by Sofa, we enumerated the complete plan space and ranked the resulting plans ascending by estimated costs for each query. For each query, we selected and executed differently ranked plans and report estimated costs and observed runtimes for these plans.

As shown in Figure 7, Sofa ranks the different algebraic execution plans correctly, and for Q1, Q2, Q5, and Q7, the best ranked plans were retrieved only with Sofa. We also observed a large optimization potential for most tasks. For example, the best ranked plans for Q1–Q4 outperform the worst ranked plans with factors in the range of 4.2 (Q2) to 9.1 (Q1). For the remaining queries Q5–Q7 we observed differences in execution times of 23 to 28 % between the best and worst plan. Note that these three queries were the shortest running in our experiments with total runtimes between 10 to 30 minutes, and a significant portion of these runtimes can be attributed to system initialization and communication. Thus, we expect that these queries benefit much more from optimization on larger data sets. Although we used rather small data sets for evaluating the correctness of the ranking, we see a large impact of choosing a good plan on the overall performance of a query. For example, consider Q1 and Q3, where the worst ranked plans were very expensive even for rather small data set of 100,000 full-text articles due to bad placement of expensive operators in the data flows. Specifically, the worst ranked plan for Q1 took more than 2 days to finish and the worst ranked plan for Q3 took more than 4 days to finish, whereas the best ranked plans for these queries were executed in about 6 and 13 hours, respectively.

7.2. Pruning

Table 1 displays the plan space with search space pruning enabled in brackets. For queries spanning the largest plan space (Q1 and Q3), pruning helps to significantly reduce the enumerated plan space. For the methods presented in [5, 22], which both enumerate significantly smaller plan spaces than Sofa, pruning as performed by our enumeration algorithm does not reduce the plan space in most cases. For each tested query, the optimization time with

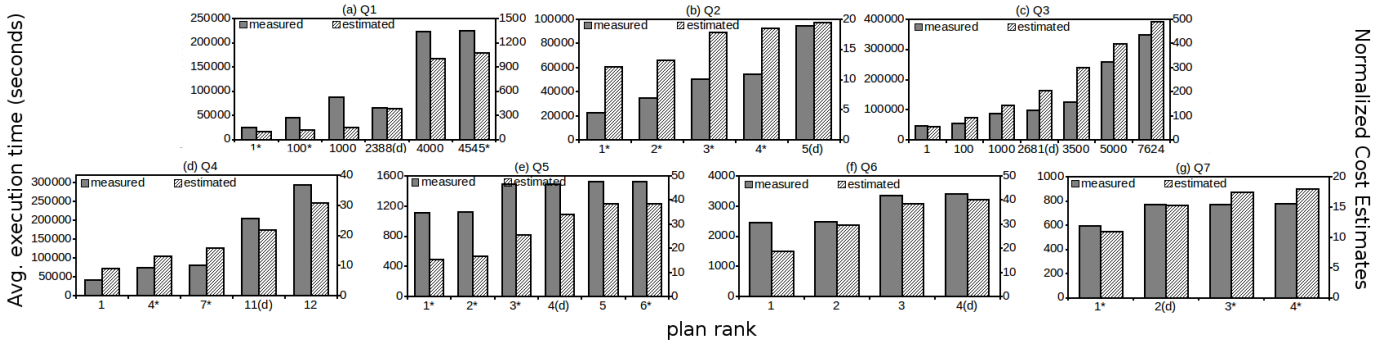


Figure 7: Estimated costs (right y axis) and observed execution times (left y axis) of selected plans ranked by cost estimates. Ranks marked with a '*' denote plans found only with Sofa, ranks marked with '(d)' point to the time required by executing the query without any optimization.

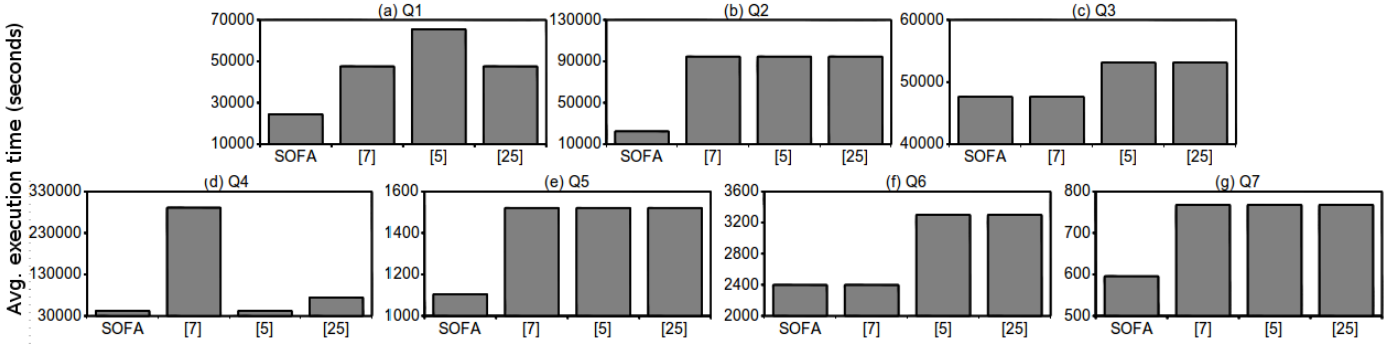


Figure 8: Execution times of best plans found with SOFA and best plans found by three competitors.

	Sofa	[7]	[5]	[22]
Q1	**4545 (783)**	512 (214)	1 (1)	38 (38)
Q2	**5 (5)**	1 (1)	1 (1)	1 (1)
Q3	**7624 (844)**	**7624 (844)**	240 (192)	240 (192)
Q4	**12 (10)**	1 (1)	**6 (6)**	4 (4)
Q5	**6 (4)**	1 (1)	1 (1)	2 (2)
Q6	**4 (4)**	**4 (4)**	2 (2)	2 (2)
Q7	**4 (2)**	1 (1)	1 (1)	1 (1)

Table 1: Number of plan alternatives per query. Counts in braces denote the number of plans considered with pruning enabled. Bold numbers indicate the plan space containing the fastest plan.

pruning enabled takes not longer than 2.5 seconds with Sofa. Enumerating the complete plan space for each query takes at most 10 seconds, which is negligible compared to the execution times of our long-running evaluation queries. Note that the largest part of these optimization times can be attributed to reasoning along Presto relationships, which could be improved using known Datalog optimization techniques [27].

7.3. Optimization benefits

In our third experiment, we evaluated to which extent data flow optimization benefits from information on operator semantics. Figure 8 displays the execution times of the best ranked plan found with Sofa as well as the methods described in [7, 5, 22]. For each tested query, Sofa finds the fastest plan, and for Q1, Q2, Q5, and Q7, Sofa finds significantly faster plans than competitors: the best plan found with Sofa outperforms the best plans found by [7] with factors of up to 6.8 (Q4), by [5] and [22] with factors up to 4.2 (Q2). The method of Hueske et al. performs as well as Sofa for Q3 and Q6, because both methods enumerate the same plan space for these two queries. The rewrite rules of Olston et al. and Simitsis et al. find the same best plan as Sofa for Q4. In these cases, plan optimization involves only reordering filter operators, which is addressed equally well in these methods as in Sofa. Note that the method of Hueske et al. cannot rewrite Q4, as this query is DAG-shaped. All other queries involve rewriting general Udfs and expansion of complex operators, and thus, optimization benefits notably from semantic information that is available in Sofa.

7.4. Scalability

To evaluate scalability, we executed the unoptimized and optimized data flows for Q2 (Topic detection), Q6 (TPC-H), and Q7 (Entity extraction) on data sets of increasing sizes. Particularly, we manifold the Medline data set we used to evaluate Q2 several times from scale factor 1 (20 GB) to scale factor 100 (2 TB), the TPC-H data set for Q6 from scale factor 2 (2 GB) to scale factor 2000 (2 TB), and the Wikipedia data set from scale factor 1 (12 GB) to

	Scale factor	Input size	Optimized (avg. runtime in seconds)	Unoptimized (avg. runtime in seconds)	Gain in %
Q2	1	20 GB	734.44	1,018.26	39
	10	200 GB	5,221.14	6,674.69	28
	50	1 TB	25,057.47	53,934.33	115
	100	2 TB	49,456.58	124,322.50	151
Q6	2	2 GB	175.48	218.60	25
	20	20 GB	225.30	268.16	19
	200	200 GB	674.18	781.97	16
	2,000	2 TB	7,497.36	19,466.59	160
Q7	1	12 GB	237.21	1,113.75	369
	5	60 GB	658.53	4,410.41	570
	10	120 GB	1,190.27	8,679.26	629

Table 2: Scalability measurements of optimized and unoptimized plans for selected queries.

scale factor 10 (120 GB). Each query was tested on a 12-node cluster with 144 threads and 20GB RAM available on each node. As shown in Table 2, data flow optimization as carried out with Sofa is more beneficial the larger data sets grow. Particularly, the optimized plan for Q2 is executed more than twice as fast as the implemented query on 1 TB of input data, whereas on the original data set (20 GB of text data), the optimized plan is 39% faster compared to the unoptimized plan. Similarly, optimizing Q6 achieves a decrease of runtime of 160 percent on 2 TB of input data, compared to 25% of improvement at 2 GB of input data. On Q7, we observe the highest acceleration with factors of between 4.5 on 12 GB and 7.29 on 120 GB of input data, which is due to a possible operator deletion detected by Sofa. The increase of performance gain with larger data sets is due to the vanishing effect of the start-up costs of Stratosphere. These constant costs are responsible for a large fraction of runtimes on smaller data sets, but count less and less the larger the overall runtime of a query.

7.5. Extensibility

Finally, we concretize the example from Section 4.3 to quantify the effect of pay-as-you-go annotation of operators in Sofa. Recall the novel *rmark* operator, which replaces HTML tags in web pages by a series of '%' of the same length as the removed tags to retain text length and markup position. Imagine a query Q8 that first replaces HTML markup in websites, computes term frequencies from the websites content, and finally filters terms starting with a series of '%'. The high-level data flow looks as follows:

$I \rightarrow rmark \rightarrow splt\text{-}sent \rightarrow stem \rightarrow rm\text{-}stop \rightarrow splt\text{-}tok \rightarrow grp \rightarrow fltr \rightarrow O$

Initially, *rmark* is annotated only with an *isA*-relationship to the abstract Presto concept *operator*. In this case, Sofa can analyze only read and write access on attributes similar to the method presented in [7], which yields in 10 semantically equivalent plans for Q8. After adding the information that *rmark* is a record-at-a-time operator implemented with a Map function, Presto already finds 18 equivalent algebraic plans. Finally, when *rmark* is fully specified, including an *isA* relationship to the Base operator *trnsf*, Sofa would find 75 alternative plans.

8. Conclusions

We address the problem of logical optimization for Udf-heavy data flows and present Sofa, a novel, extensible, and comprehensive optimizer. Sofa builds on a concise set of properties describing the semantics of Map/Reduce-style Udfs and a small set of rewrite templates to derive equivalent plans.

A unique characteristic of our approach is extensibility: we arrange operators and their properties into taxonomies, which considerably eases integration and optimization of new operators. We implemented our solution in Stratosphere, a fully-functional system for large-scale data analytics. Our experiments reveal that Sofa is able to reorder acyclic data flows of arbitrary shape (pipeline, tree, DAG) from different application domains, leading to considerable runtime improvements. We also show that Sofa finds plans that clearly outperform found by other techniques.

Sofa was implemented on top of the Stratosphere system, which executes data flow programs made of second-order functions. Sofa optimizes logical data flows consisting of such algebraic operators, which can be compiled into physical data flows consisting of second-order functions. Our approach is equally applicable to other parallel data analytics platforms that build upon such data flows, in particular those using the Map/Reduce paradigm as implemented in Hadoop. For example, Pig [5] compiles a query into a logical operator plan, which is translated into a physical data flow consisting of map and reduce second-order functions. Similar to Stratosphere, SOFA could optimize such an operator plan based on inferred or annotated properties. The parallelization primitives used in PIG/Hadoop are a subset of the primitives supported by Stratosphere and therefore already accounted for during optimization. Thus we believe that our contributions here can also be fruitful for many other large-scale data analytics systems.

Acknowledgments

This research was funded by the German Research Foundation under grant "FOR 1036: Stratosphere-Information Management on the Cloud". We thank Martin Beckmann and Anja Kunkel for help with implementing the Meteor queries and we thank Volker Markl and Stephan Ewen for valuable discussions.

Bibliography

[1] J. Dean, S. Ghemawat, MapReduce: Simplified data processing on large clusters, in: Proc. of the Symp. on Operating Systems Design and Implementation, 2004, pp. 137–150.

[2] S. Sakr, A. Liu, D. Batista, M. Alomari, A survey of large scale data management approaches in cloud environments, IEEE Comm. Surveys Tutorials 13 (3) (2011) 311–336.

[3] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Y. Eltabakh, C.-C. Kanne, F. Özcan, E. J. Shekita, JAQL: A scripting language for large scale semistructured data analysis, Proc. of the VLDB Endowment 4 (12) (2011) 1272–1283.

[4] A. Heise, A. Rheinländer, M. Leich, U. Leser, F. Naumann, Meteor/Sopremo: An Extensible Query Language and Operator Model, in: Proc. of the Int. Workshop on End-to-End Management of Big Data (BigData) in conjunction with VLDB, 2012.

[5] C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins, Pig Latin: a not-so-foreign language for data processing, in: Proc. of the Int. Conf. on Management of Data, 2008, pp. 1099–1110.

[6] M. J. Cafarella, C. Ré, Manimal: relational optimization for data-intensive programs, in: Proc. of the ACM SIGMOD Workshop on the Web and Databases, 2010, pp. 10:1–10:6.

[7] F. Hueske, M. Peters, M. J. Sax, A. Rheinländer, R. Bergmann, A. Krettek, K. Tzoumas, Opening the black boxes in data flow optimization, Proc. of the VLDB Endowment 5 (11) (2012) 1256–1267.

[8] S. Wu, F. Li, S. Mehrotra, B. C. Ooi, Query optimization for massively parallel data processing, in: Proc. of of Int. Symp. on Cloud Computing, 2011, pp. 12:1–12:13.

[9] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, D. Warneke, The Stratosphere Platform for Big Data Analytics, VLDB Journal (2014) 1–26.

[10] M. T. Roth, P. M. Schwarz, Don't scrap it, wrap it! A wrapper architecture for legacy data sources, in: Proc. of the Int. Conf. on Very Large Databases, 1997, pp. 266–275.

[11] G. Graefe, Volcano – an extensible and parallel query evaluation system, IEEE Transactions on Knowledge and Data Engineering 6 (1) (1994) 120–135.

[12] M. A. Hernández, S. J. Stolfo, The Merge/Purge Problem for Large Databases, in: SIGMOD Conference, 1995, pp. 127–138.

[13] E. Dantsin, T. Eiter, G. Gottlob, A. Voronkov, Complexity and expressive power of logic programming, ACM Computing Surveys 33 (3) (2001) 374–425.

[14] G. Graefe, The Cascades framework for query optimization, IEEE Data Engineering Bulletin 18 (3) (1995) 19–29.

[15] L. M. Haas, J. C. Freytag, G. M. Lohman, H. Pirahesh, Extensible query processing in Starburst, in: Proc. of the Int. Conf. on Management of Data, 1989, pp. 377–388.

[16] H. Pirahesh, J. M. Hellerstein, W. Hasan, Extensible/rule based query rewrite optimization in Starburst, in: Proc. of the Int. Conf. on Management of Data, 1992, pp. 39–48.

[17] S. Chaudhuri, K. Shim, Query optimization in the presence of foreign functions, in: Proc. of the Int. Conf. on Very Large Databases, 1993, pp. 529–542.

[18] S. Chaudhuri, K. Shim, Optimization of queries with user-defined predicates, ACM Transactions on Database Systems 24 (2) (1999) 177–228.

[19] J. M. Hellerstein, M. Stonebraker, Predicate migration: Optimizing queries with expensive predicates, in: Proc. of the Int. Conf. on Management of Data, 1993, pp. 267–276.

[20] U. Srivastava, K. Munagala, J. Widom, R. Motwani, Query optimization over web services, in: Proc. of the Int. Conf. on Very Large Databases, 2006, pp. 355–366.

[21] E. S. Ogasawara, D. de Oliveira, P. Valduriez, J. Dias, F. Porto, M. Mattoso, An algebraic approach for data-centric scientific workflows, Proc. of the VLDB Endowment 4 (12) (2011) 1328–1339.

[22] A. Simitsis, P. Vassiliadis, T. K. Sellis, Optimizing ETL processes in data warehouses, in: Proc. of the Int. Conf. on Data Engineering, 2005, pp. 564–575.

[23] H. Lim, H. Herodotou, S. Babu, Stubby: a transformation-based optimizer for MapReduce workflows, Proc. of the VLDB Endowment 5 (11) (2012) 1196–1207.

[24] L. Fegaras, C. Li, U. Gupta, An optimization framework for map-reduce queries., in: Proc. of the Int. Conf. on Extending Database Technology, 2012, pp. 26–37.

[25] J. Zhang, H. Zhou, R. Chen, X. Fan, Z. Guo, H. Lin, J. Y. Li, W. Lin, J. Zhou, L. Zhou, Optimizing data shuffling in data-parallel computation by understanding user-defined functions, in: Proc. of the USENIX Conf. on Networked Systems Design and Implementation, 2012, pp. 22–22.

[26] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, R. Murthy, Hive: A warehousing solution over a map-reduce framework, Proc. of the VLDB Endowment 2 (2) (2009) 1626–1629.

[27] Y. Sagiv, Optimizing datalog programs, in: Proc. of the Symp. on Principles of Database Systems, 1987, pp. 349–362.

[28] A. Heise, F. Naumann, Integrating open government data with Stratosphere for more transparency, Web Semantics 14 (0) (2012) 45–56.

A. The Stratosphere System

SOFA is part of Stratosphere, a full-fledged system for massively parallel data analytics. As displayed in Figure A.9, the Stratosphere system comprises three layers, namely *Meteor/Sopremo*, a declarative scripting language and algebraic operator model, the *Pact* programming model, and the parallel execution engine *Nephele*. Each layer is equipped with its own programming model and and specific components responsible for different tasks during data flow processing. In this section, we give a brief overview of the whole system focussing on Meteor and Sopremo. Details on the Stratosphere system can be found in [9].

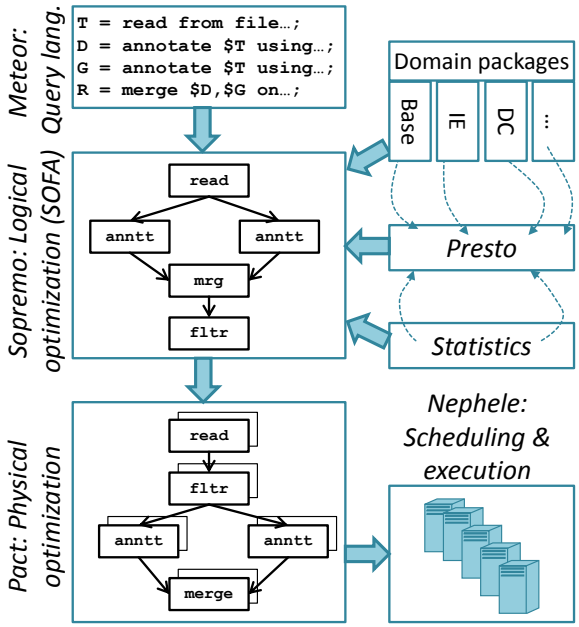


Figure A.9: Architecture of Stratosphere.

A.1. Meteor/Sopremo

Meteor [4] is a data flow oriented declarative scripting language that resides at the top of the stack. Meteor builds upon a semi-structured data model that extends JSON and has similar objectives as other query languages, (e.g., Pig [5] or Jaql [3]), but is distinguished by the semantically rich operator model Sopremo. A main advantage of this approach is that the operator's semantics can be accessed at compile time and thus is available during data flow optimization carried out with SOFA.

Listing 2 summarizes the general Meteor syntax in EBNF form. The result of an operator invocation can be assigned to a variable, which either refers to a materialized data set or to a logical intermediate data set. Variables start with a dollar sign ($) to ease distinguishing between data sets and operator definitions. Each operator invocation starts with the unique name of the operator (bold in all listings contained in Appendix D) and is typically followed by a list of inputs and a set of operator properties, which are configured with a list of name/expression pairs.

```
inputs::= (alias 'in')? variable (',' inputs)?
properties::= property properties?
property::= property_name expression
variable::= '$'name
```

Listing 2: Excerpt of Meteor's grammar in EBNF.

A Meteor query is parsed into an abstract syntax tree composed of basic or complex operators (see Section 3) and then compiled into a logical execution plan in the Sopremo algebra. SOFA resides on the algebraic layer of Stratosphere and employs information on properties and semantics of operators stored in the Presto taxonomy (see Section 4) as well as statistics on the operators to perform data flow optimization (see Section 5).

Meteor and Sopremo are designed for extensibility. Through Sopremo, package developers and domain specialists can integrate application-specific functionality by extending Sopremo's set of operators. When adding their new operators to Preso, they can enable cross-domain data flow optimization and extend the optimization potential of their operators in a pay-as-you-go manner (see Section 4.3). Operators are defined in domain-specific packages, which are self-contained libraries of the operator implementations, their syntax, and semantic annotations.

A.2. Sopremo operator packages

User-defined operators are organized into *packages* specific to a certain application domain. Stratosphere currently contains three packages, namely a base package containing 16 operators, a package for information extraction with 38 operators, and a package for data cleansing with nine operators. A description of all operators contained in each package, together with the corresponding Meteor keywords is contained in Table A.3.

The base package mostly comprises typical relational operators, such as filter, projection, transformation, join, and group. These operators are complemented by operators for semi-structured data, such as nest or unnest.

The IE package comprises three classes of operators: One for producing text annotations, one to merge annotations, and one for complex operators. Operators analyze the text and add, remove, or update annotations to the record. They may also transform records, e.g., the *spltsent* operator takes as input single records formed of documents and outputs a set of records formed of sentences. The most abstract operator in the annotation class is *anntt*. Specializations can be distinguished between those performing linguistic annotations, semantic annotation of entities, or semantic annotations of relationships between entities. Each of these classes consists of multiple concrete operators; e.g., operators for tokenization, or part-of-speech tagging (first group), for recognizing persons, companies, or biomedical entities (second group), and for detecting binary or n-ary relationships between entities (third group). Specializations of *anntt* write to designated

attributes in the output record; for instance, all entity operators write to a list-valued field "entities". Some *anntt* specializations are in precedence relations with other *anntt* variants, for example, annotating relations between entities requires that entity annotations are already present in the input records. The merge operator *mrg* merges records a, b from two input sets A, B based on a user-defined merge condition. The set of complex operators comprises six operators, such as operators for splitting text into sentences, stemming, entity extraction, and stopword removal. Each of these internally consists of an *anntt* operator, a *trnsf* operator, and occasionally a *fltr* operator.

The DC package comprises six different classes of operators for data cleansing and data integration [28]. They address common challenges of dirty or heterogeneous data sources, such as inconsistent representation of equivalent values, fuzzy duplicates, typographic errors, or missing values. Inconsistencies and missing values can be fixed with the *scrb* operator that either repairs these values or filters invalid records. Fuzzy duplicates are found with *ddup* within one data set. These duplicates can subsequently be coalesced into a single record with *fuse*. The complex duplicate removal operator (*rdup*) combines duplicate detection and fusion of duplicates to solve the common task of removing all duplicate records within a data source.

A.3. Pact

Algebraic Sopremo plans are compiled into Pact programs , where each Sopremo operator is translated into one or more Pact operators. Before translation, each complex Sopremo operator is recursively decomposed into its components until only elementary operators remain. Elementary operators can be translated directly into second-order functions such as Map and Reduce, which are provided in the Pact programming model. Alike Map/Reduce, the Pact programming model grounds on second-order functions, each providing certain guarantees on what subsets of the input data will be processed together by the user-defined function. Pact programs may consist of different parallelization primitives, such as map or reduce, and the associated user-defined function. Next to map and reduce, Pact contains three additional second order functions, i.e., match, cross, and cogroup, which support an efficient implementation of dual-input operators.

A.4. Nephele

A Pact program is physically optimized and translated into a parallel data flow program, which is deployed on the given hardware by means of Nephele, a system for scheduling, executing, and monitoring DAG structured execution graphs on distributed systems. Similar to Sopremo and Pact, data flow programs for Nephele are DAGs where nodes represent individual tasks and edges modelling the data flow between tasks. However, Nephele data flow programs are equipped with a customized execution strategy, i.e., a suggested degree of parallelism for each task and data partitioning instructions, which adapts to the given compute environment and the data to be analyzed. Nephele is also responsible for executing of data flow programs, in particular, it allocates necessary hardware resources for executing the data flow program, schedules individual tasks among the resources, monitors execution, and enables task recovery in the event of failure.

Operator	Meteor keyword	Implementation	Processing type	Description
Selection	`filter`	map	record-at-a-time	Filters the input by retaining only those elements where the given predicate evaluates to true.
Projection	`transform`	map	record-at-a-time	Transforms each element of the input according to a given expression.
(Un)nesting	`nest` `unnest`	map	bag-at-a-time	Flattens or nests incoming records according to a given output schema.
Join	`join`	complex	record or bag at a time (depending on join type)	Joins two or more input sets into one result-set according to a join condition. Provides algorithms for anti-, equi-, natural-, left-outer-, right-outer-, full-outer-, semi-, and theta-joins.
Grouping	`group`	reduce	bag-at-a-time	Groups the elements of one or more inputs on a grouping key into one output, such that the result contains one item per group. Aggregate functions, such as count() or sum(), can be applied.
Set/Bag Union	`union` `union all`	co-group	bag-at-a-time	Calculates the union of two or more input streams under set or bag semantics.
Set Difference	`subtract`	co-group	bag-at-a-time	Calculates the set-based difference of two or more input streams.
Set Intersection	`intersect`	co-group	bag-at-a-time	Calculates the set-based intersection of two or more input streams.
Replace (All)	`replace` `replace all`	map	record-at-a-time	Replaces atomic values with with a defined replacement expression.
Sorting	`sort`	reduce	bag-at-a-time	Sorts the input stream globally.
Splitting	`split`	map	record-at-a-time	Splits an array, an object, or a value into multiple tuples and provides a means to emit more than one output record.
Unique	`unique`	reduce	bag-at-a-time	Turns a bag of values into a set of values.
Sentence Annotation / Splitting	`annotate/split sentences`	map	record-at-a-time	Annotates sentence boundaries in the given input text and optionally splits the text into separate records holding one sentence each.
Token Annotation / Splitting	`annotate/split tokens`	map	record-at-a-time	Annotates token boundaries in the given input sentencewise and optionally splits the input into separate tokens. Requires sentence boundary annotation.
Part-of-speech Annotation	`annotate pos`	map	record-at-a-time	Annotates part-of-speech tags in the given input sentencewise. Requires sentence and token boundary annotations.
Parse tree Annotation	`annotate structure`	map	record-at-a-time	Annotates the syntactic structure of the input sentencewise. Requires sentence boundary annotations.
Stopword Annotation / Removal	`annotate/remove stopwords`	map	record-at-a-time	Annotates stopwords occurring in the given input and optionally replaces stopword occurrences with a user-defined string.
Ngram Annotation / Splitting	`annotate/split ngrams`	map	record-at-a-time	Annotates token or character ngrams with user-defined length n in the given input. Optionally, the input can be split into ngrams.
Entity Annotation / Extraction	`annotate/extract entities`	map	record-at-a-time	Annotates entities in the given input and optionally extracts recognized entity occurrences. Supports general-purpose entites (e.g., persons, organizations, places, dates), biomedical entities (e.g., genes, drugs, species, diseases), and user-defined regular expressions and dictionaries. Requires sentence and token boundary annotations.

Relation Annotation / Extraction	`annotate/ extract relations`	map	record-at-a-time	Annotates relations sentencewise in the given input and optionally extracts recognized relationships using co-occurrence- or pattern-based algorithms. Requires sentence, part-of-speech, and entity annotations.
Merge Records	`merge`	match	record-at-a-time	Merges existing annotations of records which share the same ID.
Data Scrubbing	`scrub`	complex	record-at-a-time	Enforces declaratively specified rules for (nested) attributes and filters invalid records.
Entity Mapping	`map entities`	complex	bag-at-a-time	Uses a set of schema mappings to restructure multiple data sources into multiple sinks. Usually used to adjust the data model of a new data source to a global data schema.
Duplicate Detection	`detect duplicates`	complex	bag-at-a-time	Finds fuzzy duplicates within a data set.
Record Linkage	`link records`	complex	bag-at-a-time	Finds fuzzy duplicates across multiple (clean) data sources.
Data Fusion	`fuse`	complex	bag-at-a-time	Fuses duplicate representations to one consistent entry with declaratively specified rules.
Duplicate Removal	`remove duplicates`	complex	record-at-a-time	Performs duplicate detection, subsequent fusion, and retains non-duplicates.

Table A.3: Overview of available Sopremo operators. Top: Base, Middle: Information Extraction, Bottom: Data Cleansing.

B. Rewrite Rules

This section lists and explains the usefulness of all rewrite rules available in SOFA and based on the following scenario. Assume we have three datasets R, S, and T, where R exhibits the JSON data formats shown in Listing 3, and S and T exhibit the format shown in Listing 4. We exemplify the usage of our rewrite rules by applying different small data flows to R, S, and T.

The first two rewrite rules (cf. Listing 5 and 6) cover operator reorderings based on read/write set analysis similar to [7]. In contrast to [7], we define the rewrite rules not based on the type of the second order function used for operator implementation (e.g., map, reduce). Instead, we use the more general properties *processing type* and *number of inputs*, which now allows us to reorder complex operators exhibiting these properties.

```
{
  'id' : integer,
  'text' : string,
  'author' : string,
  'annotation' : {
    'linguistic' : {
      'sentence' : array,
      'token' : array,
      'pos' : array
    },
    'entity' : [{
      'id' : string,
      'text' : string,
      'type' : string
    }]
  }
}
```

Listing 3: Json format of dataset R

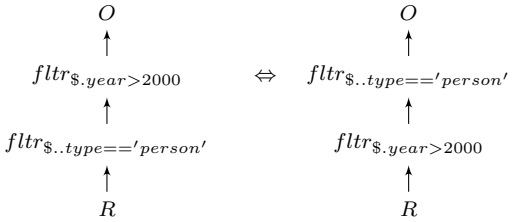
```
{
  'id' : integer,
  'income' : double,
  'department' : string
}
```

Listing 4: Json format of datasets S and T

```
1 reorder(X,Y) :-
2   hasProperty(X,'single-in'),
3   hasProperty(X,'RAAT'),
4   hasProperty(Y,'single-in'),
5   hasProperty(Y,'RAAT'),
6   noReadWriteConflicts(X,Y).
7
8 noReadWriteConflicts(X,Y) :-
9   intersection(writeSet(X),readSet(Y),N),
10  intersection(readSet(X),writeSet(Y),M),
11  length(N,0), length(M,0).
```

Listing 5: Rewrite rule 1.

Specificially, the rewrite rule shown in Listing 5 states that two record-at-a-time, single-input operators can be reordered if they have no read/write conflicts. This allows us for example to reorder the two filter operators contained in the following data flow:



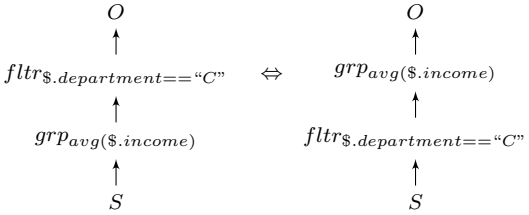
It also allows us to rewrite a data flow that uses two complex operators to split text contained in the attribute $\$.text$ of items in dataset R into tokens and to remove stopwords:

The second rewrite rule (see Listing 6) allows us to reorder bag-at-a-time with record-at-a-time operators similar to [7] given that both operators have no read/write conflicts and the record-at-a-time operator preserves key groups, i.e., retains or removes all items belonging to the same key group processed with the bag-at-a-time operator. As reordering based on read/write set analysis is symmetrical, the second rewrite rule consists of two parts to enable partial data flows $X \rightarrow Y$ where either X is the bag-at-a-time and Y the record-at-a-time operator or vice versa.

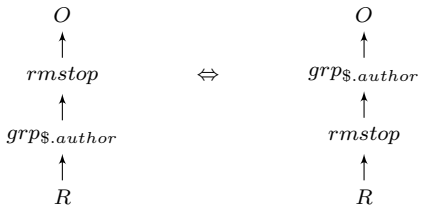
```
1  reorder(X,Y) :-
2    hasProperty(X,'single-in'),
3    hasProperty(X,'BAAT'),
4    hasProperty(Y,'single-in'),
5    hasProperty(Y,'RAAT'),
6    noReadWriteConflicts(X,Y),
7    preservesKeyGroup(Y).
8
9  reorder(X,Y) :-
10   hasProperty(X,'single-in'),
11   hasProperty(X,'RAAT'),
12   hasProperty(Y,'single-in'),
13   hasProperty(Y,'BAAT'),
14   noReadWriteConflicts(X,Y),
15   preservesKeyGroup(X).
```

Listing 6: Rewrite rule 2.

For example, a data flow that analyzes dataset S to determine the average income of people in department "C" can be reordered as follows:



Analog to rule 1 and opposed to read/write-set analysis as proposed in [7], we are can rewrite single-input bag-at-a-time operators with complex single-input record-at-a-time operators using the second rewrite rule, if no read/write conflicts exist and the complex record-at-a-time operator either removes or keeps all items belonging to the same key group. Consider a data flow that groups articles from data set R by author and removes stop words from the texts using the complex, single-input, record-at-a-time operator *rmstop*. This data flow can be rewritten as follows:

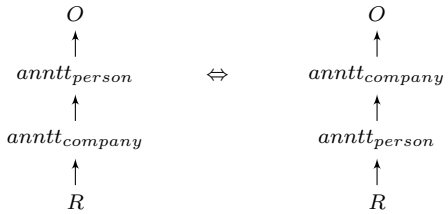


The third rewrite rule (see Listing 7) is specific to annotation operators as contained in the IE package. It enables reordering of any annotation operators X, Y given that X is not a prerequisite of Y. This rewrite rule is most useful for IE and NLP data flows containing many annotation operators.

```
1 reorder(X,Y) :-
2   not hasPrerequisite(Y,X),
3   isA(X,'annotate'),
4   isA(Y,'annotate').
```

Listing 7: Rewrite rule 3.

For example, a data flow that annotates person and company names in items from dataset R can be reordered despite read/write conflicts (i.e., both *anntt* operators write to $\$annotation.entity$) in the following way:

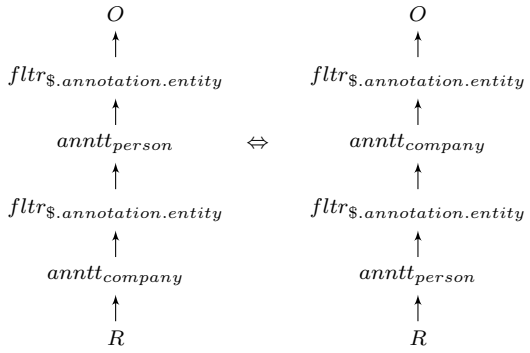


In IE data flows, we often observe that operators for entity or relation annotation are followed by $fltr$ operators that keep only those data items containing entities or relations, respectively. To enable reordering of filter operators togegher with their corresponding annotation operators, we added a quite specific rule (see Listing 8) to our rule set.

```
1 reorderAnnotateAndFilter(W,X,Y,Z) :-
2   isA(W,'annotate'),
3   isA(X,'filter'),
4   isA(Y,'annotate'),
5   isA(Z,'filter'),
6   rwConflictsAnnotateFilter(W,X),
7   rwConflictsAnnotateFilter(Y,Z).
8
9 rwConflictsAnnotateFilter(X,Y) :-
10  not isEmpty(
11    intersection(writeSet(X),readSet(Y))).
```

Listing 8: Rewrite rule 4.

Consider the following data flow, that annotates person and company names in the $\$.text$ attribute of dataset R. Each annotation operator is followed by a filter operator, which keeps only those items actually containing a person or company, respectively. Here, the goal `reorderAnnotateAndFilter` evaluates to true and thus, we allow to reorder the data flow as follows by removing the edge $fltr_{company} \rightarrow anntt_{person}$ from the corresponding precedence graph (see Section 5 for details):



The rewrite rule shown in Listing 9 enables reorderings of operators X based on inherited properties from an ancestor Z of X. Specificially, the rule states that if operator X is not a prerequisite of operator Y, X is a descendant of operator Z, and Z and Y are reorderable, then X and Y are also reorderable. This rule enables reordering of operators, which are not well annotated with property information, but which are declared as a specialization of some other operator.

```
1 reorder(X,Y) :-
2   not hasPrerequisite(Y,X), isA(X,Z),
3   reorder(Z,Y).
```

Listing 9: Rewrite rule 5.

Consider two operators $detect_{link}$, and $detect_{structure}$ from a package for web analytics that detect outgoing links and structured information (e.g., tables, lists) contained in websites. Both operators save the link and structure information in certain attributes $\$.link$ and $\$.structure$, respectively. For our example, we assume that both $detect_{link}$ and $detect_{structure}$ are not well annotated with properties, but are declared as specializations of the abstract IE annotation operator *anntt*. Now, all rules that are applicable to

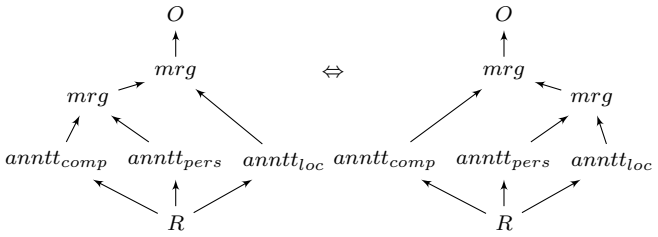
anntt (e.g., rule 4 and 5) become applicable to $detect_{link}$ and $detect_{structure}$.

SOFA also employs rewrite rules based on algebraic operator properties for data flow rewriting, see Listings 10–13. Reordering of associative operators is covered in the sixth rule.

```
1 reorder(X,X) :- hasProperty(X,'associative'),
2   isA('operator').
```

Listing 10: Rewrite rule 6.

For example, a data flow containing two inner joins can be reordered in the same manner as in relational databases. Similarly, a data flow that performs taskparallel annotation of the three different entity types person, company, and location in dataset R and that subsequently merges these annotations can be rewritten, based on the associativity of the *mrg* operator:

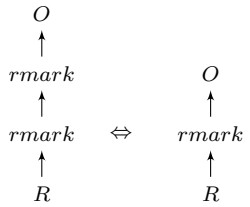


Operator removals are possible if two adjacent operators of the same type and same instantiation are configured identically and furthermore annotated as idempotent (see Listing 11).

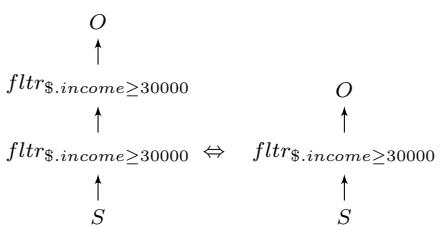
```
1 removeX(X,Y):-X=Y, hasProperty(X,'idempotent').
```

Listing 11: Rewrite rule 7.

Assume that dataset R is a collection of HTML websites where all markup shall be removed from the attribute $\$.text$ of each input item. A data flow containing an operator *rmark* for markup removal twice can be rewritten:



Similarly, two adjacent filter operators performing the same filter operation can be rewritten as follows:

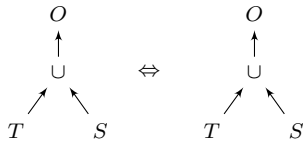


The rewrite rule shown in Listing 12 enables to interchange the left and right input of commutative dual input operators. In Stratosphere, the operators for unioning, intersecting, joining, and merging datasets are commutative.

```
1 rewireInputs(X) :-
2   hasProperty(X, 'commutative'),
3   hasProperty(X,'dual-input').
```

Listing 12: Rewrite rule 8.

Suppose that the datasets S and T shall be unioned and therefore, the following two data flows are equivalent:

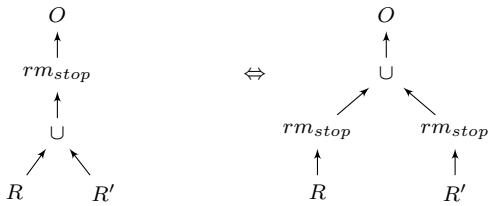


Distributive dual-input bag-at-a-time operators can be rewritten with single-input record-at-a-time operators as indicated by the nineth rewrite rule of SOFA shown in Listing 13.

```
1 rewriteDistributive(X,Y) :-
2   hasProperty(X, 'distributive'),
3   hasProperty(X,'BAAT'),
4   hasProperty(Y,'RAAT'),
5   hasProperty(X,'dual-input'),
6   hasProperty(Y,'single-input').
```

Listing 13: Rewrite rule 9.

This rule applies to quite a few operator combinations, for example consider two datasets R and R' exhibiting the same schema that shall be unioned and afterwards, stopwords shall be removed from the attribute $\$.text$ by applying a regular-expression based stopword removal operator rm_{stop}. Such a data flow can be rewritten as follows:

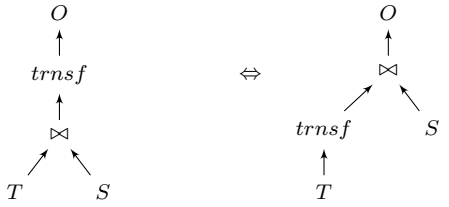


Rewrite rule 10 is again a very specific rule that allows to prepone a single-input record-at-a-time operator Y before a dual-input operator X given that Y only accesses attributes of the left input of Y. In this cases, if `reorderWithLeft` evaluates to true, Y can be placed directly before the left input of X (i.e., by removing the edge from the right input of X to Y and the edge $X \rightarrow Y$ from the corresponding precedence graph).

```
1 reorderWithLeft(X,Y) :-
2   hasProperty(X,'dual-input'),
3   hasProperty(Y,'single-input'),
4   hasProperty(Y,'RAAT'),
5   not contains(readSet(Y),readSet(X)),
6   not contains(rightInputSchema(X),readSet(X)).
```

Listing 14: Rewrite rule 10.

Suppose, we are given a data flow that consists of an equi-join of two datasets S , T followed by $trnsf$ that transforms only attributes of S, which are not part of the join condition. This data flow can be rewritten into an equivalent data flow, which first applies $trnsf$ to S and afterwards joins S and T:

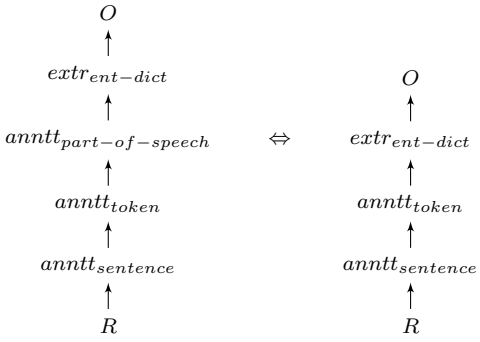


Finally, the rewrite rule shown in Listing 15 enables to remove an operator X in case X has only one outgoing edge to some other operator Y and X modifies only attributes that are neither accessed by Y and not contained in the output schema of Y. In Sofa, plan optimizations involving operator removals is handled as follows: During precedence analysis, the rule for operator removal is evaluated before all other rules for plan transformation. If `removeX(X,Y)` evaluates to true for some operators X and Y, X and all incident edges of X are removed from the directed transitive closure D^+ of the given plan D. Sofa continues with analyzing precedence constraints in the newly formed graph and eventually enumerates plan alternatives as described in Section 5.2.

```
1 removeX(X,Y) :-
2   hasProperty(X,'RAAT'),
3   hasProperty(Y,'RAAT'),
4   hasProperty(X,'I=O'),
5   hasProperty(X,'input-schema=output-schema'),
6   not contains(readSet(Y),writeSet(X)),
7   not contains(outSchema(Y),writeSet(X)).
```

Listing 15: Rewrite rule 11.

Consider an IE data flow that first annotates linguistic structure in the text (i.e., sentences, tokens, and part-of-speech tags) and afterwards extracts entities using a dictionary-based entity extractor. In this scenario, the $annott_{part-of-speech}$ operator can be removed since the annotations produced by $annott_{part-of-speech}$ are not read by the complex $extr_{ent-dict}$ operator and are not contained in the output schema produced by $extr_{ent-dict}$:



C. Plan enumeration by example

Plan enumeration with Sofa generates different topological orders given by the precedence graph, while performing cost-based pruning. In contrast to topological sorting, the outcome are not full orders but DAG-shaped plans. The main idea is to iteratively construct alternative plans for a given data flow D by analyzing the corresponding precedence graph for operators that have no outgoing edges. Such operators are not required by any other operator and can therefore be added to the emerging physical data flows. If multiple operators have no outgoing edges, the algorithm creates a set of alternative partial plans. The algorithm continues to pursue each alternative, removing the newly added operator from the precedence graph, estimating the costs of the partial plan (see Section 5.3), and pruning costly partial plan alternatives where possible.

The recursive plan enumeration algorithm as displayed in Figure 6 is fully explained in Section 5. Here, we exemplarily show its principles by enumerating all alternatives for the data flow shown in Figure 5 (top) and the corresponding precedence graph as shown in Figure 5 (bottom). Figure C.10 shows all stages of enumerating the plan space for our data flow. Columns are alternative partial plans grouped into stages of the algorithm. Boxes correspond to operators with isochromatic frames as defined in Figure 5.

The recursive plan enumeration algorithm takes as input the original data flow, the corresponding precedence graph, and a partial plan, which initially is empty (Stage 1). First, the algorithm selects the set of nodes from the precedence graph that have out-degree 0. These operators are not a prerequisite of any remaining operator and can thus be added to the partial plan without violating precedence constraints. Once added, the selected node and all its incoming edges are removed from the precedence graph. In Stage 1, only the data sink can be selected (grey box), which is subsequently removed from the precedence graph in Stage 2. Since the partial plan was empty before adding the data sink, we cannot insert any edges in the partial plan and therefore, plan enumeration is recursively invoked again. The algorithm can now either add *mrg* or *fltr* (brown and green boxes) as both have no outgoing edges in Stage 3. Each candidate node is processed individually, added to the partial plan and removed from the

precedence graph. This yields in two alternative partial plans, which are both inspected further.

We exemplarily follow the plan with the *mrg* operator. The *mrg* operator is added to the plan and the set of $inputNodes$ is divided into required and optional nodes (Lines 19–24). Required nodes are those nodes that have the currently added node as its direct predecessor in the original data flow, optional successors are all other operators contained in $inputNodes$. In our example, the set of optional nodes is empty, and the set of required nodes contains *sink*. For each required node m, we create an edge (n, m) for the newly added node n, add it to the edge set of our partial plan, estimate the costs of the partial plan, and recursively call the plan enumeration algorithm. In Stage 4, the *mrg* plan has three further alternatives for adding a node, namely *fltr*, *anntt-ent-comp*, and *anntt-ent-pers*. At that point, only the source cannot be added to the plan yet and consequently the remaining two operators can be added in arbitrary order. Thus, the four alternatives are expanded to eight plans in Stage 5. Finally, the last operator and source is added in Stage 6 and 7 resulting in 12 different alternatives.

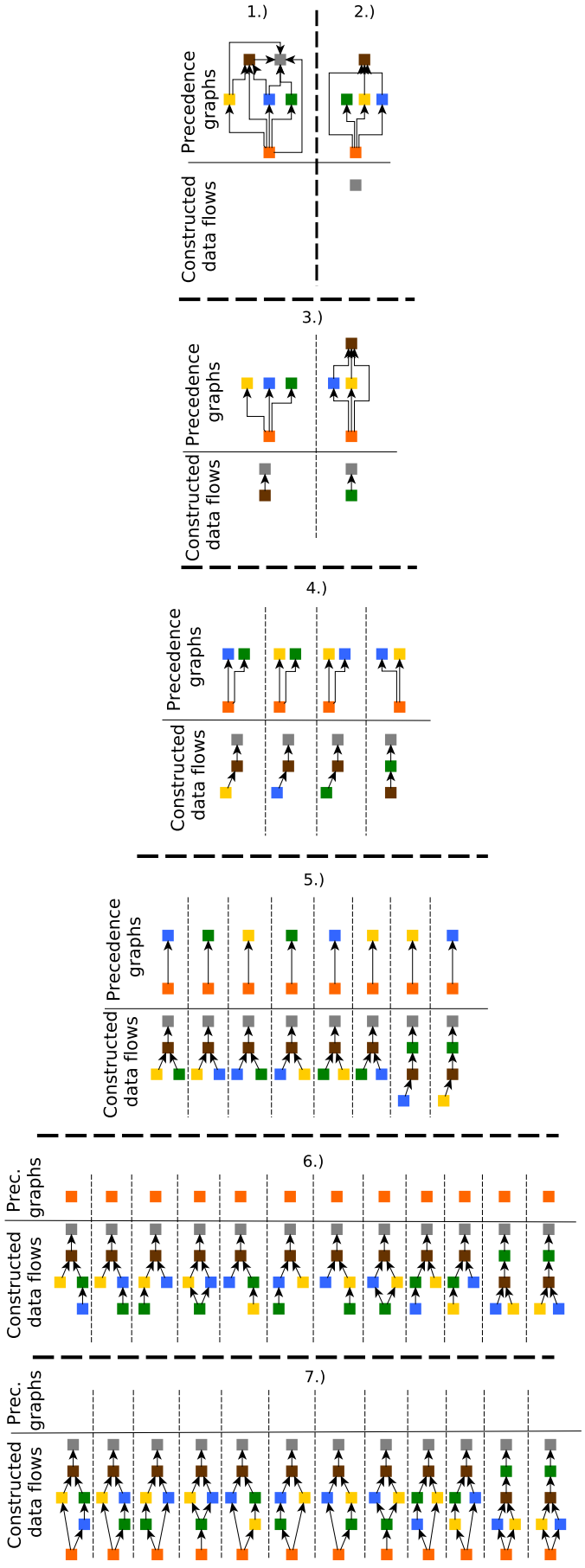


Figure C.10: Plan enumeration for the DAG-shaped data flow from Figure 5. Columns are alternative partial plans grouped into stages of the algorithm. Boxes correspond to operators with isochromatic frames as defined in Figure 5.

D. Evaluation benchmark

Query 1 annotates relationships between drugs and genes on a collection of biomedical texts, which might contain duplicate texts. Therefore, duplicate removal is performed in advance as a preprocessing step.

```
using base,ie,cleansing;

$article = read from
    "hdfs://192.168.127.43:50040/medline/";
$article = rdup $article;

$article = annotate sentences $article;
$article = annotate tokens $article;
$article = annotate postags $article;

$article = annotate biomedical entities $article use algorithm 'regex' type 'drug';
$article = filter $article where
    $article.annotation_entity.drugs;

$article = annotate biomedical entities $article use algorithm 'regex' type 'gene';
$article = filter $article where
    $article.annotation_entity.genes;

$article = annotate biomedical relations
    $article;
$article = filter $article where
    $article.annotation_relation;

write $article to
    "hdfs://192.168.127.43:50040/results/q1/";
```

Listing 16: Query 1.

Query 2 performs topic detection by computing term frequencies in a corpus grouped by year. The query first splits the input data into sentences, reduces terms to their stem, removes stopwords, splits the text into tokens, and aggregates the token counts by year.

```
using base,ie;

$input = read from
    'hdfs://192.168.127.43:50040/medline/';
$input = split sentences $input;
$input = annotate tokens $input;

$input = annotate stems $input;
$input = replace with stems $input;

$input = remove stopwords $input;
$input = split text $input on "TOKENS";

$input = group $input by
    { $input.text, $input.year }
    into {
        text:max($input.text),
        year:max($input.year),
        wordcount: count($input.id)
    };

$input = filter $i in $input where
    length($i.text)>3;

write $input to
    'hdfs://192.168.127.43:50040/results/q2/';
```

Listing 17: Query 2.

Query 3 extracts NASDAQ-listed companies that went bankrupt between 2010 and 2012 from a subset of Wikipedia. This query takes article versions from two different points in time, annotates company names in both sets and applies different *fltr* operators and a *join* to accomplish the task.

```
using base,ie;

$articles = read from
    'hdfs://192.168.127.43:50040/wiki2012/';

$articles = annotate sentences $articles;
$articles = annotate tokens $articles;
$articles = annotate entities $articles
    type "organization";

$articles = filter $article in $articles where
    ($article.annotation_entity.organization);

$articles = filter $article in $articles where
    (strpos($article.text,"bankrupt")>=0);

$old_articles = read from
    'hdfs://192.168.127.43:50040/wiki2010/';

$old_articles = annotate sentences $old_articles;
$old_articles = annotate token $old_articles;
$old_articles = annotate entities $old_articles
    type "organization";

$old_articles = filter $article in $old_articles where ($article.annotation_entity.organization);

$old_articles = filter $article in $old_articles where (strpos($article.text,"NASDAQ")>=0);

$old_articles = filter $article in $old_articles where (strpos($article.text,"bankrupt")<0);

$results = join $articles, $old_articles where
    ($articles.id==$old_articles.id);

write $results to
    'hdfs://192.168.127.43:50040/results/q3/';
```

Listing 18: Query 3.

Query 4 corresponds to the data flow shown in Figure 5 and performs task-parallel annotation of person and location names in Wikipedia articles created in 21013 or later.

```
using base,ie;

$articles = read from
    'hdfs://192.168.127.43:50040/wikipedia/';
$articles = annotate sentences $articles;
$articles_loc = annotate entities $articles
    type "location";
$articles_pers = annotate entities $articles
    type "person";
$articles_mrg = merge $articles_comp, $articles_pers
  where ($articles_comp.id == $articles_pers.id);

$result = filter $article in $article_mrg where
    ($article.year >= "2013");
write $result to
    'hdfs://192.168.127.43:50040/results/q4/';
```

Listing 19: Query 4.

Query 5 analyzes DBpedia to retrieve politicians named 'Bush' and their corresponding parties using a mixture of DC and base operators.

```
using base,cleansing;

$input = read csv from
    'hdfs://192.168.127.43:50040/dbpedia/'
    columns ['subject', 'predicate', 'object']
    delimiter ' '
    encoding 'iso-8859-1'
    quote true;

$parties = filter $input where
    $input.predicate == '<http://dbpedia.org/ontology/party>';

$names = filter $input where
    $input.predicate == '<http://xmlns.com/foaf/0.1/name>';

$politicianWithName = join $p in $parties,
    $n in $names
    where $p.subject == $n.subject
    into {
        url: $p.subject,
        name: substring($n.object, 0, -3),
        party: $p.object
    };

$scrubbed = scrub $politicianWithName
    with rules {name: &normalizeUnicode};

$filtered = filter $p in $scrubbed where
    like($p.name, "%%Bush");

write $filtered to
    'hdfs://192.168.127.43:50040/results/q5/';
```

Listing 20: Query 5.

Query 6 is a relational query inspired by the TPC-H query 15. It filters the lineitem table for a time range, joins it with the supplier table, groups the result by join key, and aggregates the total revenue to compute the final result.

```
$li = read from
    'hdfs://192.168.127.43:50040/tpc-h/lineitem/';
$s = read from
    'hdfs://192.168.127.43:50040/tpc-h/supplier/';

$join = join $li, $s where
    $s.s_suppkey == $li.l_suppkey into {
        $s.s_suppkey,
        $s.s_name,
        $s.s_address,
        $s.s_phone,
        $li.l_shipdate,
        $li.l_extendedprice,
        $li.l_discount};

$fli = filter $join where
    ($join.l_shipdate >= '1996-01-01' and
        $join.l_shipdate < '1996-04-01');

$revenue = group $fli by $fli.s_suppkey into {
    supplier_no: $fli[0].s_suppkey,
    total_revenue: sum($fli[*].l_extendedprice
                    * (1-$fli[*].l_discount))
};

write $revenue to
'hdfs://192.168.127.43:50040/results/q6/';
```

Listing 21: Query 6.

Query 7 uses two complex IE operators to split incoming texts into sentences and to extract person names.

```
using ie;

$input = read from
    'hdfs://192.168.127.43:50040/wikipedia/';

$input = split sentences $input;

$results = extract entities $input type "person";

write $results to
    'hdfs://192.168.127.43:50040/results/q7/';
```

Listing 22: Query 7.