

SOFA: An Extensible Logical Optimizer for UDF-heavy Dataflows

Astrid Rheinländer
Humboldt-Universität zu Berlin
Berlin, Germany
rheinlae@informatik.hu-berlin.de

Arvid Heise
Hasso Plattner Institut
Potsdam, Germany
arvid.heise@hpi.uni-potsdam.de

Fabian Hueske
Technische Universität Berlin
Berlin, Germany
fabian.hueske@tu-berlin.de

Ulf Leser
Humboldt-Universität zu Berlin
Berlin, Germany
leser@informatik.hu-berlin.de

Felix Naumann
Hasso Plattner Institut
Potsdam, Germany
felix.naumann@hpi.uni-potsdam.de

Abstract

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

SOFA is a novel and extensible optimizer for UDF-heavy dataflows. 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 dataflows from different domains and compare its performance to three other algorithms for dataflow optimization. Our experiments reveal that SOFA finds efficient plans, outperforming the best plans found by its competitors by a factor of up to 6.

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, predictive modeling, or graph traversal on large texts, graphs, semi-structured datasets, 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 [7].

A variety of dataflow languages has been proposed that aim at (a) making the definition of complex analytics tasks easier and at (b) allowing flexible deployment of such dataflows on diverse hardware infrastructures, especially compute clusters or compute clouds [24]. Many of these languages support UDFs [2, 14, 20]. Research has shown that proper optimization of such dataflows can improve the execution times by orders of magnitude [3, 16, 27]. However, most of these systems focus on relational operators and treat UDFs essentially as black boxes, considerably hampering optimization. At the same time, non-standard applications in areas such as information extraction, graph analysis, or predictive modeling often utilize UDF-rich dataflows. 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 behavior, which are difficult to describe in an abstract, optimizer-enabling manner.

Different approaches have been proposed to overcome this problem. Two broad classes can be discerned: Approaches that require manual annotations of UDFs [1, 17],

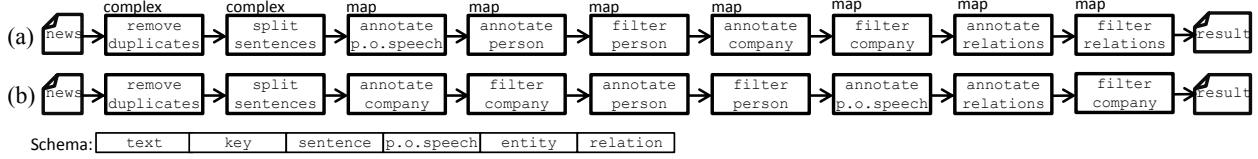


Figure 1: High-level dataflow for employee relationship analysis. (a) Initial dataflow, (b) reordered dataflow based on operator semantics.

and approaches performing code analysis to automatically infer optimization options with UDFs [3, 11, 16]. We present SOFA, a semantics-aware optimizer for UDF-heavy dataflows¹. Compared to previous work, SOFA features a richer, yet concise set of properties for automatic and manual UDF annotation. It is capable of finding a much larger and more efficient set of semantically equivalent execution plans for a given dataflow than other systems. Given a concrete dataflow, 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 dataflow for this task is shown in Figure 1(a). The first operator performs duplicate removal 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 ↔ 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 dataflow is composed of nine steps: seven maps, and two complex operators (first and second from left).

If UDFs are treated as black boxes, this dataflow cannot be reordered in any way. 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 dataflow, as annotations produced by this operator are necessary for relationship annotation only. Moreover, the company and person annotation operators are commutative, as they independently add annotations to the text, but never delete existing annotations. Thus, both annotation operators can be reordered for early filtering. Figure 1(b) displays an equivalent dataflow 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 dataflow as possible. As we see in Sections 3 and 7, existing dataflow optimization techniques, such as [16, 20], 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 datasets with custom, domain-specific operator packages [1]. Available packages for information extraction (IE) and data cleansing (DC) already contain 38 and 9 operators, respectively. Further packages, e.g., for machine-learning and web analytics, are in development. Defining semantic properties for each of these operators individually would result in an unacceptable burden to the designer and would considerably limit extensibility and maintainability. Furthermore, the optimizer needs rewrite rules for operator pairs that take operator semantics into account. 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 modeled. Presto considerably eases extensions, as novel operators can be hooked to the system by specifying

¹SOFA is only a vague acronym but more of a metaphor.

a single subsumption relationship to an existing operator exhibiting the same behavior with respect to optimization; these new operators are immediately optimized in the same manner as their parent. If desired and appropriate, more rewrite rules describing specific properties of the new operator may be introduced later in a “pay-as-you-go” manner [22].

In summary, our work includes the following contributions:

1. We identify a small yet powerful set of common UDF properties influencing important aspects in terms of dataflow optimization.
2. We show how properties of UDFs in Map/Reduce-style systems can be arranged in a concise taxonomy to simplify UDF annotation, and to enhance extensibility of dataflow languages.
3. We present a novel optimization algorithm that is capable of rewriting DAG-shaped dataflows given proper operator annotations.
4. We evaluate our approach using a diverse set of dataflows cutting through different domains. We show that SOFA subsumes three existing dataflow optimizers by enumerating a larger plan space and by finding more efficient plans.
5. SOFA outperforms the best plans found with other approaches in many situations with factors of up to 6.

This paper is structured as follows: Section 2 describes preliminaries including a brief overview of Stratosphere with a focus on its rich set of domain-specific UDFs. Section 3 gives an overview of our approach for dataflow 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 dataflows on a semi-structured data model, such as JSON or XML. A *record* is a potentially nested value tree consisting of objects, arrays, and atomic values. An unordered bag of records is called *dataset*. An *operator* o transforms a list of input datasets $I = (I_1, \dots, I_n)$ into a list of output datasets $O = (O_1, \dots, O_m)$ by applying a UDF f to I . We denote with $o_{in,i}$ and $o_{out,i}$ the i -th input and output of operator o . We call $S(o_{in,i})$ *input schema* and $S(o_{out,i})$ *output schema* of the i -th input and output of o , respectively.

A *dataflow* 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 concrete operators. For any directed edge (u, v) connecting two operators we demand $S(u_{out,j}) \supseteq S(v_{in,i})$, and $S(v_{in,i})$ must meet the schema requirements of the i -th input of v ; according conditions hold for edges from an input dataset or to a data sink. 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 call two deterministic dataflows D, D' *semantically equivalent* (denoted $D \equiv D'$), if D and D' produce the same output sets T , given the same input datasets I .

2.1 Stratosphere

The SOFA optimizer is integrated into Stratosphere, a full-fledged system for massively parallel data analytics. Meteor [14], a dataflow-oriented declarative scripting language resides at the top of the stack (see Figure 2). 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 system’s algebra called Sopro. 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 dataflow optimization (see Section 5).

Meteor and Sopro are designed for extensibility. Operators are defined in domain-specific packages, which are self-contained libraries of the operator implementations, their syntax, and semantic annotations. Algebraic Sopro plans are compiled into Pact programs, which may consist of different parallelization primitives, such as map or reduce, and the associated user-defined function [1]. Subsequently, Pact programs are physically optimized and translated into an execution graph, 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. A detailed description of Stratosphere can be found in [1]. Here, we describe only the algebraic optimizer.

2.2 User-Defined Operators

Before we introduce our UDF annotation schema and dataflow optimizer, we first introduce a subset of the op-

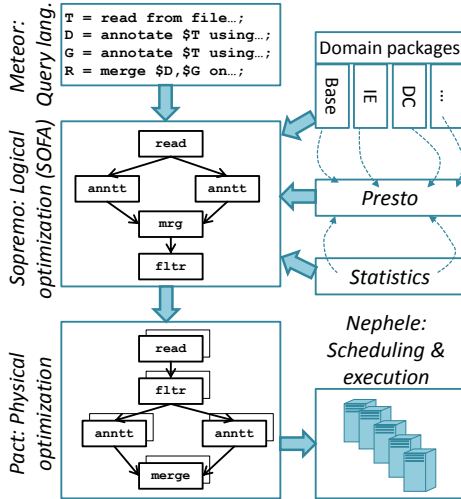


Figure 2: Architecture of Stratosphere.

erators currently used in Stratosphere to highlight their diversity and resulting optimization challenges. User-defined operators are organized into *packages* specific to a certain application domain. Operators can be either abstract or concrete; for instance, *anntt-ent-pers* (see Table 1) is an abstract operator for annotating person names in texts, and its instantiations are different concrete algorithms and tools for performing this task. All instantiations of a given abstract operator share the same language syntax. Note that concrete operators may use very different implementations; for instance, the recognition of person names may be performed by using dictionary-, pattern-, or machine-learning-based methods.

Concrete operators can either be elementary or complex. Elementary operators are implemented using a single stub function, complex operators are composed of several 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 dataflow optimization, since a complex operator may exhibit other semantics than those of its elements (see below).

Stratosphere currently contains three packages, namely a base package, a package for IE, and a package for DC. Packages for machine-learning and web data analytics are under development. Table 1 shows example operators from all three packages together with information describing their requirements and behavior. The base package

Operator	Abbreviation	#inputs	Processing type	Preserves schema?
filter	<i>fltr</i>	1	Record	Yes
project	<i>prjt</i>	1	Record	No
(un)nest	<i>nst</i>	1	Record	No
join	<i>join</i>	n	Record	No
(co)group	<i>grp</i>	1/2	Bag	No
...				
annotate entities	<i>anntt-ent</i>	1	Record	Yes
annotate tokens	<i>anntt-tok</i>	1	Record	Yes
merge	<i>mrg</i>	2	Bag	Yes
split sentences	<i>splt-sent</i>	1	Record	Yes
extract relations	<i>extr-rel</i>	1	Record	No
remove stopwords	<i>rm-stop</i>	1	Record	Yes
...				
scrub	<i>scrub</i>	1	Bag	Yes
split records	<i>sptrc</i>	1	Record	No
transform records	<i>trfrc</i>	1	Record	No
detect duplicates	<i>ddup</i>	1	Bag	No
remove duplicates	<i>rdup</i>	1	Bag	No
fuse	<i>fuse</i>	1	Record	No
...				

Table 1: Selected general purpose (top), IE (middle), and DC (bottom) operators available for Stratosphere. See [14] for details.

contains 16 operators, the IE package 38 operators, and the DC package 9 operators. 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 *splt-sent* 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 *mrq* 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 [13]. 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 *scrub* operator that either repairs these values or filters invalid records. Fuzzy duplicates are found with *ddup* and *lnkrc* within one relation or across several relations, respectively. 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 one data source.

The algebraic plan for the dataflow for our running example is shown in Figure 3(a) together with properties and schema information (cf. Figure 3(b)), which are used for optimization. Figure 3(c) displays that complex operators 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 Dataflow Optimization

SOFA is an optimizer for rewriting UDF-heavy dataflows into semantically equivalent dataflows whose expected efficiency is higher, according to a cost model. Rewriting depends on a set of rewrite rules, each defining valid manipulations of dataflow sub-plans, such as a reordering of two filter operations [9]. The novelty of SOFA is 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 dataflow *D*, dataflows semantically equivalent to *D* may be produced using different transformation techniques. SOFA is capable of introducing, removing, and reordering operators. Complex operators

are 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 dataflow optimization enable reorderings by using either manually defined rewrite rules for relational operators [20] or by performing some kind of code analysis [3, 16]. The approach of Hueske et al. [16] probably is the most general, as it automatically derives dataflow 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 dataflow shown in Figure 3 allows only one such 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 dataflow is not possible, because it reads annotations produced by the complex *splt-sent* operator.

Semantics-aware rewrite rules allow to reorder the dataflow in Figure 3 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 3(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, because the split-UDF generates multiple output records for each

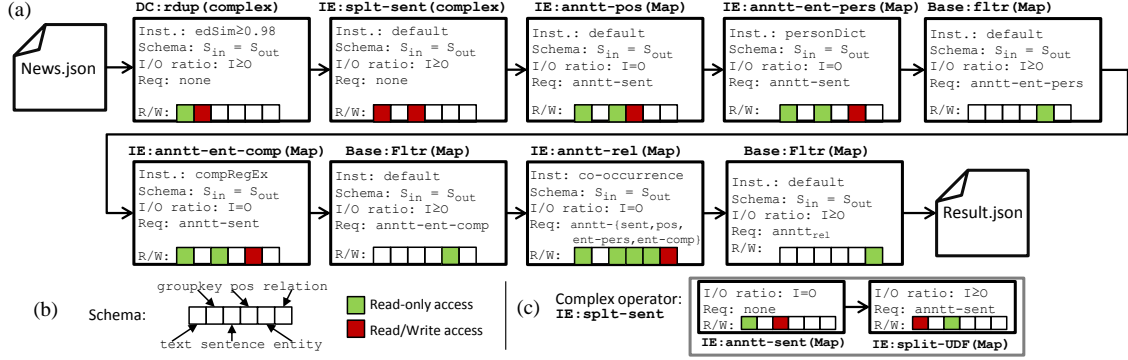


Figure 3: Algebraic dataflow for the running example. (a) displays concrete operator instantiations together with properties relevant for optimization. Colored boxes indicate read/write access on record attributes, which are part of the global schema shown in (b). (c) shows the resolution of the complex $split_{sent}$ operator (second operator in (a)) into its components $anntt_{sent}$ and $split-UDF$.

incoming record, depending on the number of annotated sentence boundaries.

Note that sometimes the expansion of complex operators may make possible reorderings impossible. For instance, think of an IE operator $anntt-syns$ for expanding existing entity annotations with all synonyms from a dictionary. Subsequently, another operator $repl-repr$ selects one of these synonyms as representative and deletes all others. Both operators change the existing list of entity annotations, so reordering based on read/write-set analysis is limited. However, a complex operator $norm-ent$ for normalizing entity annotations composed of both $anntt-syns$ and $repl-repr$ is optimizable, since we know that the number of entities does not change. Therefore, SOFA always tries to reorder complex operators both as a whole and after their expansion.

In summary, semantics-aware plan rewriting allows us to pick the best plan (with respect to cost estimates) from a larger set of equivalent dataflows 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 [16] (see Section 7 for a detailed comparison).

4 Annotating and Rewriting Operators with Presto

To enable optimizations like 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 modeling relationships between operators and properties (see Section 4.1), and a set of rewrite templates for dataflow-rewriting (see Section 4.2). When designing Presto, we paid special attention to extensibility by allowing enhancements to the semantic operator descriptions 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 properties*. Both taxonomies are self-contained and model generalization-specialization relationships (*isA*) between operators and properties, respectively. Figures 4(a) and (b) display subgraphs of Presto. For example, the $anntt$ operator has two specializations: $anntt-ent$ and $anntt-rel$ shown in Figure 4(a). Each leaf in the operator taxonomy describes a concrete implementation of the parent operator, like different implementations for a join. This design 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

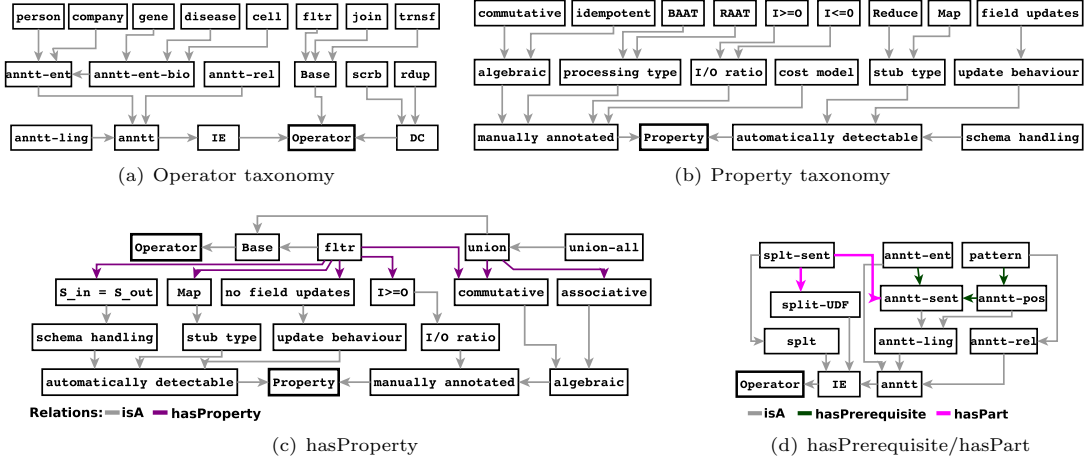


Figure 4: Exemplary subgraphs of Presto; root nodes are displayed in bold.

become much deeper when dealing with domain-specific UDFs.

As shown in Figure 4(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, or idempotence), cost model (e.g., cost functions, resource consumption), 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, and the read/write behavior at attribute level.

Relationships connect operators and properties. As usual, each specialization inherits all properties and relationships that are defined for the corresponding generalizations. For instance, the *union-all* shown in Figure 4(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 4(d)). For example, the complex operator *spltt-sent* consists of the two components *anantt-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 4(c)):

- does not modify the schema of incoming records,
- is implemented with a Map function,
- does not modify inside fields,
- input \geq output,
- processes one record at a time, and
- 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 4(d) it is shown that *anantt-rel* based on linguistic patterns requires part-of-speech and entity annotations to be performed in advance. Since *anantt-ent* itself requires sentence annotation and *hasPrerequisite* is a transitive relation, it is necessary to apply *anantt-sent* before *anantt-rel*.

The *isA* relationship very much simplifies deriving novel rewrite options for operators that are initially not well annotated. Suppose, the data scrubbing operator *scrub* from the DC package is initially not equipped with any *hasProperty* relationships. Later, the developer may see that *scrub* 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, *scrub* inherits all properties defined for *trnsf* (not shown in Figure 4(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 ex-

tend the operator taxonomy, while the property taxonomy is a fairly stable structure in our experience.

4.2 Rewrite Templates

We perform dataflow-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 10 rewrite templates, which are expanded to over 150 individual rewrite rules.

Figure 5 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 (not shown here for brevity). 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 commutative 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 `commutative`. Note that the commutativity does not necessarily need to be defined directly on X ; the rule also applies if some ancestor of X in Presto is marked as commutative. The second template (Line 4) 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 6–7) enables reorderings of consecutive `anntt` operators X, Y , when X is not a prerequisite of Y .

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 only available after posing a query. Tem-

```

1 %% static rules (package loading time) %%
2 reorder(X,X):-hasProperty(X,'commutative').% 1
3
4 reorder(X,Y):-not hasPrerequisite(Y,X),isA(X,Z),reorder(Z,Y).%2
5
6 reorder(X,Y):-isA(X,'anntt'),isA(Y,'anntt'),
7             not hasPrerequisite(Y,X). % 3
8
9 %% dynamic rules (query compile time) %%
10 reorder(X,Y):-hasProperty(X,'single-in'),hasProperty(X,'RAAT'),
11             hasProperty(Y,'single-in'),hasProperty(Y,'RAAT'),
12             not readWriteConflicts(X,Y). % 4
13
14 reorder(X,Y):-hasProperty(X,'single-in'),
15             hasProperty(X,'|I|=|O|'),
16             hasProperty(X,'S_in contains S_out'),
17             hasProperty(X,'no field updates'),
18             hasProperty(Y,'single-in'),
19             hasProperty(Y,'|I|>=|O|'),
20             hasProperty(Y,'S_in = S_out'),
21             not hasPrerequisite(Y,X),accessedFields(Y, ACCY),
22             S_out(X,OUTX), contains (OUTX, ACCY). % 5

```

Figure 5: Rewrite template examples.

plate 4 (Lines 10–12) 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 [16], 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 dataflow 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 dataflow can be rewritten into an equivalent dataflow, which first applies *trnsf* to I_1 and afterwards joins I_1 and I_2 :

$$\begin{array}{ccc}
 I_1 \rightarrow \text{join} \rightarrow \text{trnsf} \rightarrow O & & I_1 \rightarrow \text{trnsf} \rightarrow \text{join} \rightarrow O \\
 I_2 \nearrow & \Leftrightarrow & I_2 \nearrow
 \end{array}$$

Similar to extending Presto with new operators and properties, package developers can also extend the set of rewrite templates to enable dataflow 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. 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 only be performed 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 and does not change the schema of the incoming records. Taken these properties together, the last template of Figure 5 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 dataflow containing *rmark* and *join* as follows:

$$\begin{array}{c} I_1 \rightarrow \text{join} \rightarrow \text{rmark} \rightarrow O \\ I_2 \nearrow \end{array} \quad \Leftrightarrow \quad \begin{array}{c} I_1 \rightarrow \text{rmark} \rightarrow \text{join} \rightarrow O \\ I_2 \nearrow \end{array}$$

5 Algorithms

Given a dataflow D , the SOFA optimizer 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 [1] for details on this step). In the following, we discuss Phases 2–4 in more detail.

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 of the given dataflow, which explicitly models all pairwise operator execution orders in D . The algorithm then in-

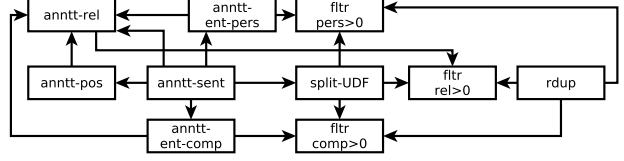


Figure 6: Precedence graph for running example with complex operator resolution.

spects this graph to detect and remove edges that are not logically required. It retains all edges incident to a data source or a 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 [6].

Figure 6 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-ent-person* are a prerequisite for the *filtr_{pers>0}* operator, and *anntt-rel* is in a *hasPrerequisite* relation with *anntt-pos* (cf. Figure 4(d)). The graph contains edges between *anntt* and *filtr* reflecting that the concrete instantiations of *filtr* have read/write conflicts with their preceding *anntt* operators.

5.2 Plan enumeration

Plan enumeration essentially generates different topological orders given by the precedence graph, while performing cost-based pruning. On contrast to topological sorting, the outcome are not full orders but DAG-shaped plans. The main idea is to construct alternative plans for a given dataflow D iteratively 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 dataflows. 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

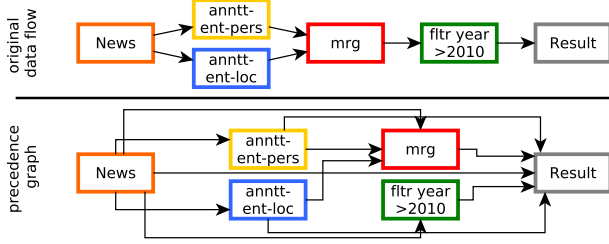


Figure 7: DAG-shaped dataflow (top) and corresponding precedence graph (bottom) inspired by the running example.

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 dataflow shown in Figure 7 (top). Note that this dataflow is DAG-shaped, which poses no problem to SOFA. The dataflow 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 7 (bottom).

The recursive plan enumeration algorithm is displayed in Figure 8. It takes as input the original dataflow, 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. In our example, only the data sink can be selected. Once added, the selected node is removed from the precedence graph (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 anymore 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.

We exemplarily follow the plan with the *mrg* operator. 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

```

1 enumAlternatives(Graph precdGraph, Graph plan,
2   Graph partialPlan) {
3
4   if (isEmpty(precdGraph)) {
5     addPlanToResultSet(partialPlan);
6     return;
7   }
8   candidateNodes = getNodesWithOutDegreeZero(precdGraph);
9
10  foreach(Node n in candidateNodes) {
11    inputNodes = getNodesWithOpenInputs(partialPlan);
12
13    addNodeToPartialPlan(n);
14    removeNodeAndIncidentEdgesFromPrecedenceGraph(n);
15
16    if (isEmpty(inputNodes))
17      enumAlternatives(precdGraph, plan, partialPlan);
18
19    foreach(Node m in inputNodes){
20      /*split inputNodes into required and optional successors*/
21      if (inputGraphcontainsEdge(n,m))
22        addNodeToRequiredNodes(m);
23      else addNodeToOptionalNodes(m);
24    }
25    if(not isEmpty(requiredNodes)) {
26      addEdgesToAllRequiredNodesInPartialPlan(m);
27      if (costs(partialPlan) < costs(originalPlan))
28        enumAlternatives(precdGraph, plan, partialPlan);
29    }
30    foreach(Node l in optionalNodes) {
31      addEdgeToPartialPlan(n,l);
32      if (costs(partialPlan) < costs(originalPlan))
33        enumAlternatives(precdGraph, plan, partialPlan);
34    }
35  }
36  addPlanToResultSet(plan);
37  return;
38 }

```

Figure 8: Plan enumeration with SOFA.

in the original dataflow, 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).

Figure 9 shows all stages of enumerating the plan space for the dataflow from Figure 7. In Stage 1, only the data sink can be selected (grey box) and is thus removed in Stage 2. The algorithm can now either add *mrg* or *fltr* (red and green boxes) as both have no outgoing edges in

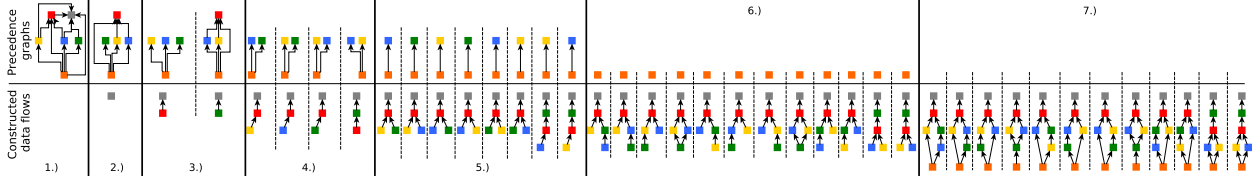


Figure 9: Plan enumeration for the DAG-shaped dataflow from Figure 7. Columns are alternative partial plans grouped into stages of the algorithm. Boxes correspond to operators with isochromatic frames as defined in Figure 7. Stage 7 contains all valid plans for this dataflow.

Stage 3. In Stage 4, the *mrg* plan results in three further alternatives, 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.

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 dataflow. 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 [9,10]. If no alternative plan with lower estimated costs compared to the best plan could be constructed, we terminate (cf. Figure 8, Line 33).

5.3 Cost estimation

To estimate costs and result sizes of a dataflow, 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 dataflows. 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 dataflows 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 dataflow 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. 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 [12]. Rules are specified as pairs of condition and action functions implemented in a procedural language [21]. Volcano [9] and Cascades [10] 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 [4]. All aforementioned approaches deal with the optimization of relational queries and assume that rewrite rules are manually added to the optimizer framework. Our work differs as it focuses on UDF-heavy query, also deals with DAG-shaped plans, and does not require the explicit definition of condition-action rules for new operators but automatically infers them from a concise set of operator properties. This considerably increases extensibility.

The optimization of relational queries with user-defined predicates has been another focus of research [5, 15, 26]. 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 parallelization functions on UDFs. In contrast, our work addresses the optimization of Map/Reduce-style dataflows with user-defined operators for which additional information is required to answer this question. In our approach, this information is provided by the developer or inferred from Presto.

While common query optimizers use tree-shaped query execution plans, our work focuses on DAG-shaped dataflows in the spirit of [18]. There, Neumann investigated the problem of generating and executing DAG-shaped query evaluation plans for relational queries in order to enable sharing of intermediate results. This is different from our setting as the input of our optimizer are already DAG-shaped dataflows in contrast to declarative relational queries.

Several methods to optimize more general dataflows have been proposed. Ogasawara et al. propose an algebraic approach to define scientific workflows [19] 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 [25]. 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 dataflows 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 [17]. 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 dataflows consisting only of relational operators [8], 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. [16] on reordering operators in Map/Reduce-style dataflows is based on specific information about the operator’s behavior 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 behavior of UDFs more precisely can help to further increase the space of possible rewritings; besides, our approach allows to rewrite DAG-shaped dataflows, which is not possible with the algorithm presented in [16]. The SUDO optimizer [28] 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 [20] is a procedural higher-level language for Hadoop and features a “safe” optimizer. It applies a limited set of heuristic transformation rules, such as filter push down, that most likely is beneficial and relies otherwise on the decisions of the programmer.

In summary, we believe that SOFA is the first extensible, fully functional optimizer for arbitrary DAG-shaped dataflows 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 queries originating from different ap-

	Q1	Q2	Q3	Q4	Q5	Q6	Q7
SOFA	4545 (783)	5 (5)	7624 (844)	12 (10)	6 (4)	4 (4)	4 (2)
Hueske et al. [16]	512 (214)	1 (1)	7624 (844)	1 (1)	1 (1)	4 (4)	1 (1)
Olston et al. [20]	1 (1)	1 (1)	240 (192)	6 (6)	1 (1)	2 (2)	1 (1)
Simitis et al. [25]	38 (38)	1 (1)	240 (192)	4 (4)	2 (2)	2 (2)	1 (1)

Table 2: 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 measured plan (see Figure 11).

plication domains². Q1, Q2, and Q7 are pipeline-shaped, Q3 and Q6 are tree-shaped, and Q4 and Q5 are DAG-shaped.

Q1 adopts the dataflow described in our running example for relationship extraction from biomedical literature using IE and DC UDFs. **Q2** performs an advanced word count 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. **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 *filtr* operators and a *join* to accomplish the task. **Q4** corresponds to the dataflow shown in Figure 7 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.

Datasets. 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 dataset v. 3.8, and Q6 was evaluated on a 100GB relational dataset 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 dataset.

Competitors. Although dataflow languages for Big Data are a hot topic in current research, surprisingly few systems actually optimize the dataflow 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 dataflow optimizers, namely techniques presented by Hueske et al. [16], Olston et al. [20], and Simitis et al. [25]. 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 [16, 20, 25]. 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.

7.1 Finding optimal plans

A large number of semantically equivalent plans for a concrete dataflow 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 dataflows 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 [16, 20, 25], we used complex operators only, as this methods do not provide mechanisms for operator expansion.

As displayed in Table 2, SOFA enumerates the largest plan space in all cases. The method presented by Hueske

²Queries may be found at <http://bit.ly/1dP9Nm2>, datasets are available on request.

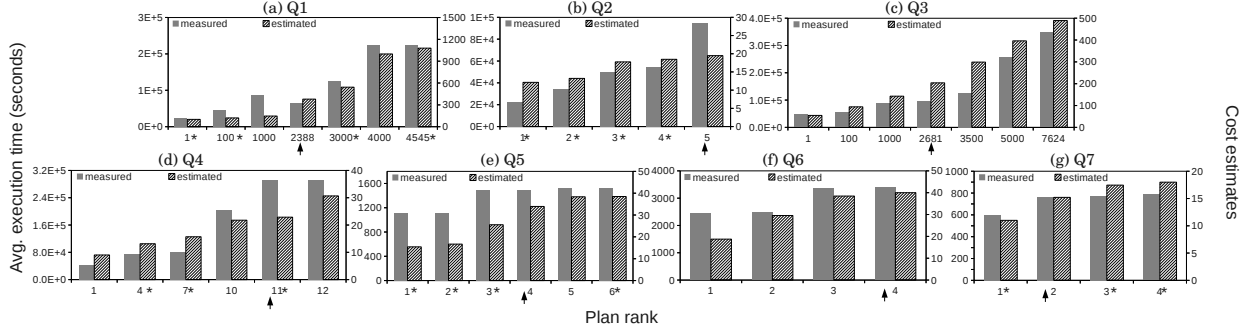


Figure 10: Cost estimates and execution time for queries. Ranks marked with a '*' denote plans found only with SOFA. Arrows below each figure point to the time required by executing the query without any optimization.

et al. is unable to rewrite Q2, Q4, Q5, and Q7, because it is neither capable of rewriting DAG-shaped dataflows (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. Simitis 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 [16] 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 sorted the complete plan space for each query plan ascending by estimated costs, selected different plans from a rank interval, and report estimated costs and observed runtimes for these plans. As shown in Figure 10, 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 datasets.

7.2 Pruning

Table 2 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 [20, 25], 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 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 in many known ways [23].

7.3 Optimization benefits

In our third experiment, we evaluated to which extent dataflow optimization benefits from information on operator semantics. Figure 11 displays the execution times of the best ranked plan found with SOFA as well as the methods described in [16, 20, 25]. 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 [16] with factors of up to 6.8 (Q4), by [20] and [25] 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 Sim-

itsis 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 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 dataflow looks as follows:

I ▶ *rmark* ▶ *splt-sent* ▶ *stem* ▶ *rm-stop* ▶ *splt-tok* ▶ *grp* ▶ *fltr* ▶ *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 [16], which yields in 10 semantically equivalent plans for Q8. After adding the information that *rmark* is a schema preserving 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 addressed the problem of logical optimization for UDF-heavy dataflows 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 a taxonomy, 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 those from other techniques.

9 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 we used for evaluation, and we thank Volker Markl and Stephan Ewen for valuable discussion and feedback.

References

- [1] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/PACTs: a programming model and execution framework for web-scale analytical processing. In *SOCC 2010*, pages 119–130.
- [2] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Y. Eltabakh, C.-C. Kanne, F. Özcan, and E. J. Shekita. JAQL: A scripting language for large scale semistructured data analysis. *Proc. of the VLDB Endowment*, 4(12):1272–1283, 2011.
- [3] M. J. Cafarella and C. Ré. Manimal: relational optimization for data-intensive programs. In *Proc. of the ACM SIGMOD Workshop on the Web and Databases (WebDB)*, pages 10:1–10:6, 2010.
- [4] S. Chaudhuri and K. Shim. Query optimization in the presence of foreign functions. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 529–542, 1993.
- [5] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. *ACM Transactions on Database Systems (TODS)*, 24(2):177–228, 1999.
- [6] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425, 2001.
- [7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. of the Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–150, 2004.
- [8] L. Fegaras, C. Li, and U. Gupta. An optimization framework for map-reduce queries. In *Proc. of the Int. Conf. on Extending Database Technology (EDBT)*, pages 26–37, 2012.
- [9] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 6(1):120–135, 1994.
- [10] G. Graefe. The Cascades framework for query optimization. *IEEE Data Engineering Bulletin*, 18(3):19–29, 1995.

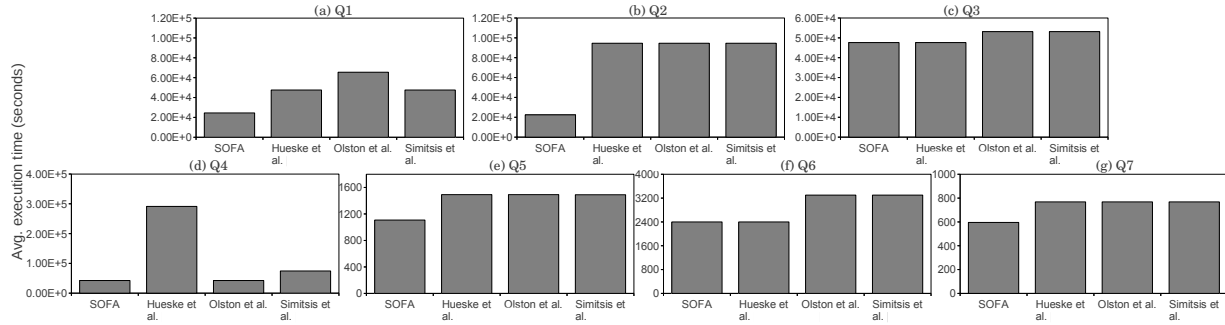


Figure 11: Execution times of best plans found with SOFA and best plans found by three competitors.

- [11] Z. Guo, X. Fan, R. Chen, J. Zhang, H. Zhou, S. McDirmid, C. Liu, W. Lin, J. Zhou, and L. Zhou. Spotting code optimizations in data-parallel pipelines through PeriSCOPE. In *Proc. of the Symposium on Operating Systems Design and Implementation (OSDI)*, pages 121–133, 2012.
- [12] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible query processing in Starburst. In *Proc. of the Int. Conf. on Management of Data (SIGMOD)*, pages 377–388, 1989.
- [13] A. Heise and F. Naumann. Integrating open government data with Stratosphere for more transparency. *Web Semantics*, 14(0):45–56, 2012.
- [14] A. Heise, A. Rheinländer, M. Leich, U. Leser, and F. Naumann. Meteor/Sopremo: an extensible query language and operator model. In *BigData 2012*, Istanbul, Turkey.
- [15] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proc. of the Int. Conf. on Management of Data (SIGMOD)*, pages 267–276, 1993.
- [16] F. Hueske, M. Peters, M. J. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas. Opening the black boxes in data flow optimization. *Proc. of the VLDB Endowment*, 5(11):1256–1267, 2012.
- [17] H. Lim, H. Herodotou, and S. Babu. Stubby: a transformation-based optimizer for MapReduce workflows. *Proc. of the VLDB Endowment*, 5(11):1196–1207, 2012.
- [18] T. Neumann. *Efficient generation and execution of DAG-structured query graphs*. PhD thesis, 2005.
- [19] E. S. Ogasawara, D. de Oliveira, P. Valduriez, J. Dias, F. Porto, and M. Mattoso. An algebraic approach for data-centric scientific workflows. *Proc. of the VLDB Endowment*, 4(12):1328–1339, 2011.
- [20] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *Proc. of the Int. Conf. on Management of Data (SIGMOD)*, pages 1099–1110, 2008.
- [21] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in Starburst. In *Proc. of the Int. Conf. on Management of Data (SIGMOD)*, pages 39–48, 1992.
- [22] M. T. Roth and 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 (VLDB)*, pages 266–275, 1997.
- [23] Y. Sagiv. Optimizing datalog programs. In *Proc. of the Symposium on Principles of Database Systems (PODS)*, pages 349–362, 1987.
- [24] S. Sakr, A. Liu, D. Batista, and M. Alomari. A survey of large scale data management approaches in cloud environments. *IEEE Comm. Surveys Tutorials*, 13(3):311–336, 2011.
- [25] A. Simitsis, P. Vassiliadis, and T. K. Sellis. Optimizing ETL processes in data warehouses. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*, pages 564–575, 2005.
- [26] U. Srivastava, K. Munagala, J. Widom, and R. Motwani. Query optimization over web services. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 355–366, 2006.
- [27] S. Wu, F. Li, S. Mehrotra, and B. C. Ooi. Query optimization for massively parallel data processing. In *Proceedings of the Int. Symposium on Cloud Computing (SOCC)*, pages 12:1–12:13, 2011.
- [28] J. Zhang, H. Zhou, R. Chen, X. Fan, Z. Guo, H. Lin, J. Y. Li, W. Lin, J. Zhou, and 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 (NSDI)*, pages 22–22, 2012.