# Meteor/Sopremo: An Extensible Query Language and Operator Model

Arvid Heise*[1] Astrid Rheinländer†[2] Marcus Leich‡[3] Ulf Leser†[4] Felix Naumann*[5]
*Hasso Plattner Institute Potsdam, Germany †Humboldt-Universität zu Berlin, Germany
‡ Technische Universität Berlin, Germany
[1,5]{arvid.heise, felix.naumann}@hpi.uni-potsdam.de [2,4]{rheinlae,leser}@informatik.hu-berlin.de
[3]marcus.leich@tu-berlin.de

## ABSTRACT

Recently, quite a few query and scripting languages for MapReduce-based systems have been developed to ease formulating complex data analysis tasks. However, existing tools mainly provide basic operators for rather simple analyses, such as aggregating or filtering. Analytic functionality for advanced applications, such as data cleansing or information extraction can only be embedded in user-defined functions where the semantics is hidden from the query compiler and optimizer. In this paper, we present a language that treats application-specific functions as first-class operators, so that operator semantics can be evaluated and exploited for optimization at compile time.

We present Sopremo, a semantically rich operator model, and Meteor, an extensible query language that is grounded in Sopremo. Sopremo also provides a programming framework that allows users to easily develop and integrate extensions with their respective operators and instantiations. Meteor's syntax is operator-oriented and uses a Json-like data model to support applications that analyze semi- and unstructured data. Meteor queries are translated into data flow programs of operator instantiations, i.e., concrete implementations of the involved Sopremo operators. Using a real-world example, we show how operators from different applications can be combined for writing complex analytical queries.

## 1. INTRODUCTION

Today, we are flooded with data that is generated in various scientific areas, on the web, or in business applications. For example, Twitter produces 340M messages per day as of March 2012[1], which can be analyzed in order to gain insights on emerging trends [21]. Often, the available data sets contain (near-exact) duplicates, unstructured text, or both in combination. Therefore, extracting relevant information and removing duplicates are fundamental steps in various data analysis pipelines [1].

Data flow systems [3, 4, 19] as a generalization of the MapReduce programming model [11] have gained much attention in this context, as they promise to ease writing scalable code for analytical tasks on huge amounts of data. However, developing data flow programs for non-relational workloads like information extraction (IE), data cleansing (DC), or data mining (DM) can become quite complex. For example, IE often involves complex preprocessing steps such as text segmentation, linguistic annotation, or stop word removal before the actual entities or relationships can be determined. IE algorithms themselves are often complex, and a re-use of existing algorithms and tools is often necessary for ad-hoc data and text analysis.

Quite a few languages for expressing complex analysis data flows in form of queries or scripts [6, 9, 14, 23, 27] have been developed recently. These tools often provide only basic operators for simple, SQL-like analyses, such as aggregating, joining, or filtering data. Analytic functionality, which is required for more complex tasks, must be embedded in user-defined functions (UDF) where the UDF's semantics is hidden from the query compiler and optimizer. Moreover, a combination of operators that were initially developed for different use cases and application domains is often difficult, since operators may differ in terms of expected input, produced output, or background information that is needed by an operator to work properly.

In this paper, we propose to treat use case specific functions as first-class operators of a data flow scripting language instead of writing UDFs. A main advantage of this approach is that the operator's semantics can be accessed at compile time and potentially be used for data flow optimization, or for detecting syntactically correct, but semantically erroneous queries. We present Meteor and Sopremo, an extensible query language and a semantically rich operator model for the Stratosphere data flow system. Both tools are integrated in the system's software stack and can be tested by downloading Stratosphere[2].

Sopremo provides a programming framework that allows users to define custom packages, the respective operators and their instantiations. Sopremo already contains a set of base operators from the relational algebra, plus two sets of additional operators for IE and DC. Further operators, e.g., for performing statistical tests and for boilerplate detection on web data, are in development.

[1]http://blog.twitter.com/2012/03/twitter-turns-six.html

[2]http://www.stratosphere.eu


*Int. Workshop on End-to-end Management of Big Data* 2012 Istanbul, Turkey

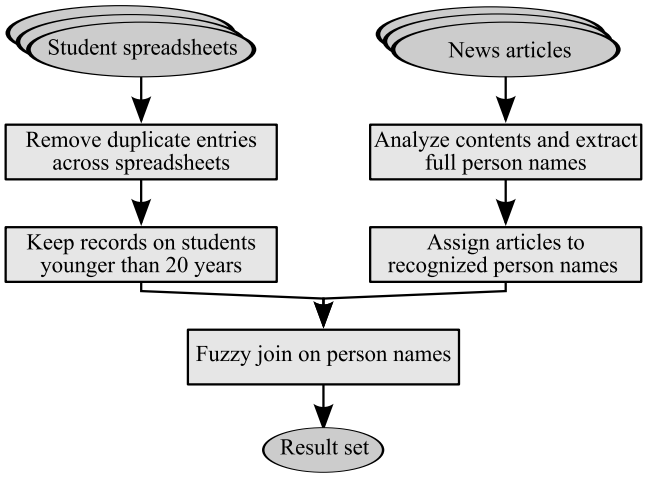


**Figure 1: Abstract workflow for retrieving mentions of teen-aged students in news articles.**

Data analysis programs for Sopremo are specified in Meteor, a declarative language inspired by Jaql [5]. Meteor builds upon the Json data model, which allows the system to process a variety of different types of data. Meteor contains several simplifications of Jaql that are necessary to seamlessly add new operators to the language and to support n-ary input and output operators. Meteor queries are parsed and translated into data flow programs of Sopremo operator instantiations, i.e., concrete implementations of the involved operators.

We use the following scenario as our running example in this paper. We give only an abstract description of the task here, details on how to express this use-case with Meteor and Sopremo are explained in Sections 4 and 5.

**Example:** Suppose a county school-board wants to find out which of its teen-aged students carry out voluntary work and thus have appeared in recent news articles. Our input is a set of spreadsheets containing information on all students from all schools in the county and a corpus of news articles. The task is to return all news articles that mention at least one student being under 20. As displayed in Fig. 1, we have to perform multiple operations to accomplish this task. First, we have to eliminate duplicates across the spreadsheets, since students might have changed school. Afterwards, we need to filter the spreadsheets and keep only records of teenagers. Concerning the news data, we need to analyze the textual contents, extract the names of people mentioned in the articles, and group the news articles by person names. Finally, we need to join both data sets on the persons' full names and return the result.

The remainder of this paper is structured as follows. In Sect. 2, we present background information on the Stratosphere system and illustrate how Sopremo and Meteor fit into the system's architecture. We give insights into technical details and base operators of Sopremo as well as language features of Meteor in Sect. 3. In Sect. 4, we show how both Sopremo and Meteor can be extended. Section 5 discusses related work. Finally, we conclude and present ideas for future work in Sect. 6.

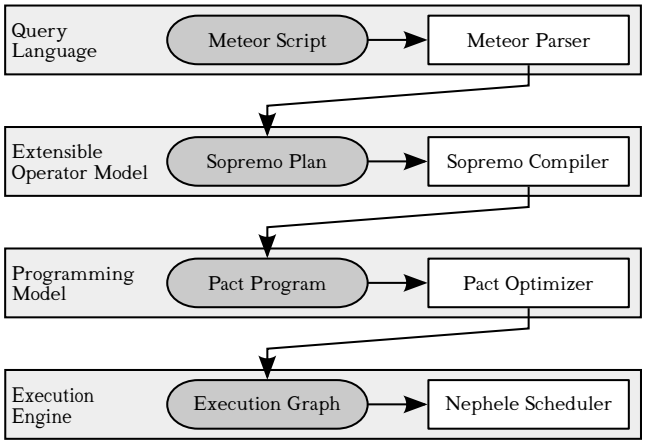


**Figure 2: Architecture of the Stratosphere system.**

## 2. THE STRATOSPHERE SYSTEM

We implemented Meteor and Sopremo within Stratosphere, a system for parallel data analysis. Stratosphere consists of four major components; next to Meteor and Sopremo, it comprises the *Pact* programming model [2, 3], and the *Nephele* execution engine [25].

Figure 2 displays the complete stack of the Stratosphere system. End-users specify data analysis tasks by writing Meteor queries. Such a query is parsed and translated into a Sopremo plan, a directed acyclic graph (DAG) of interconnected high level data processing operators. Each operator consumes and processes one or multiple input data sets and produces one or multiple output data sets. Operators are semantically rich, i.e., the concrete instantiation (data processing algorithm) of an operator defines in which way data sets are partitioned and processed in parallel. Often, there are different instantiations available for each operator, which are expressed with Pacts. These instantiations have different properties with respect to runtime, memory consumption, quality, etc. Choosing the 'right' algorithm will be part of a cost-based optimizer (see Sect. 6).

Pact is a generalization of the MapReduce programming paradigm [11]. A Pact operator consists of a second-order function (*InputContract*), a first-order function (the UDF), and a so-called *OutputContract*. Next to Map and Reduce, Pact defines three additional second-order functions that each process two input streams, namely *Match*, *CoGroup*, and *Cross*. Each Pact is responsible for partitioning the input data and calling a user-defined first-order function. Pact uses schema-free tuples as data model, i.e., the schema of any tuple is up to the interpretation of the user code. Pact programs are compiled into data flow graphs, which are then deployed and executed by Nephele. During translation, the Pact compiler performs physical optimization and reorderings [18] to improve the parallel execution.

Finally, Nephele [25] interprets data flow graphs and distributes tasks to the computation nodes. Nephele is designed to run in heterogeneous environments, and is also capable of exploiting the elasticity of clouds by booking and releasing machine instances at runtime.

Both Pact and Nephele may also act as a starting point for experienced users to perform data analyses, i.e., by writing Java programs against the Pact API or by providing

data flow graphs as a configuration for Nephele. However, this is cumbersome and requires intimate knowledge on the system's execution layer. With Meteor, we focus on the end-user's perspective when working with Stratosphere. Specifically, we present the language layer Meteor and the algebraic layer Sopremo, which together enable end-users to formulate and execute queries in Stratosphere. Plus, the modular design of Sopremo and Meteor also allows programmers to develop their own operator algebra and language extensions, to plug-in different scripting languages such as Pig or Hive, or even to create graphical user interfaces where analysis workflows can be specified.

# 3. OPERATOR AND LANGUAGE LAYER

In this section, we show how users write queries in Meteor. We also explain how such a query is first translated into a Sopremo plan and finally into a Pact program. First, we informally define the basic concepts used in Meteor and Sopremo.

The data model builds upon Json to support unstructured and semi-structured applications. A Sopremo or Meteor *value* thus represents a tree structure consisting of objects, arrays, and atomic values. A *data set* is an unordered collection of such trees under bag semantics. The base model does not support any constraints such as schema definition in the first place, but those may be enforced implicitly through specific operators.

An *operator* acts as a building block for a query. It consumes one or more input data sets, processes the data, and produces one or more output data sets. *Operator sub-types* share common characteristics, but are specialized versions of the generic operator, e.g., IE operators for entityand relationship annotation require different algorithms. Operators are *instantiated* to reflect specific configuration and adjustments, e.g., a join is an operator but a join over the attribute *id* is an *operator instantiation*. A *Sopremo plan* is a directed acyclic graph of interconnected operator instantiations.

## 3.1 The Meteor Query Language

Meteor is an operator-oriented query language that focuses on analysis of large-scale, semi- and unstructured data. Users compose queries as a sequence of operators that reflect the desired data flow. By importing application-specific operators, users can use Meteor to process data for a wide range of applications. The internal data format build upon Json, but Meteor supports additional input and output formats.

We start by showing and explaining the Meteor query for our running example (see Fig. 1), but we defer the discussion of operators that are specific to IE and DC applications to Sect. 4.

Our running example involves operators from the domains of data cleansing and information extraction. Thus, Line 1f first imports the packages *cleansing* and *ie*. Consequently, all operators and functions become accessible that are defined in these packages.

Line 4ff reads the spreadsheet of students and associates the variable `$students` with that data set. Then all students are removed that are listed in multiple schools (see Sect. 4.2). Line 9 filters all students that have an age less than 20.

Further, Line 12ff reads the articles, annotates the persons in the articles (see Sect. 4.1), and restructures the data set such that each entry of the data set `$peopleInNews` consists

```
1  using cleansing;
2  using ie;
3
4  $students = read from 'students.csv';
5  $students = remove duplicates $students
6    where average(levenshtein(name),
7      dateSim(birthDay)) > 0.95
8      retain maxDate(enrollmentDate);
9  $teens = filter $stud in $students
10   where (now()-$stud.birthDay).year < 20;
11
12 $articles = read from 'news.json';
13 $articles = annotate sentences $articles
14   using morphAdorner;
15 $articles = annotate entities in $articles
16   using type.person and regex 'names.txt';
17 $peopleInNews = pivot $articles around
18     $person=$article.annotations[*].entity
19   into {
20     name: $person,
21     articles: $articles
22   };
23
24 $teensInNews = join $teen in $teens,
25     $person in $peopleInNews
26   where $teen.name == $person.name
27   into {
28     student: $teen,
29     articles: $person.articles[*].url
30   };
31
32 write $teensInNews to 'result.json';
```

**Listing 1: Meteor query for running example.**

of the person's name and a list of articles mentioning him or her. Finally, both data sets are joined on person name in Line 24 and written to a file.

The goal of Meteor is to support a large variety of applications, each with its own specialized operators that are dynamically imported. To facilitate the correct specification of a query with arbitrary, dynamically imported operators, all operators in Meteor have a uniform syntax. The general form of operators is given by the excerpt of the EBNF in Listing 2. Basically, the specification of an operator starts with the multi-word operator name, followed by the list of its inputs. It concludes with a list of property specifications that each consist of the property name and the corresponding value. Additionally, inputs may be assigned aliases that act as iteration variables and are especially useful to identify the left and right input in self-joins.

```
1 operator ::= name+ inputs? properties? ';'
2 inputs ::= (alias 'in')? variable (',' inputs)?
3 properties ::= property properties?
4 property ::= property_name expression
5 variable ::= '$' name
```

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

The script in Listing 1 consists of eight operators. In the following, we highlight some operators to explain the grammar rules. The first operator **read** has no input and one property *from* with a constant string expression. The third operator **filter** has one input `$students` with alias `$stud` and the property *where* with a boolean expression. Finally, **join** has two inputs with respective aliases and a

| Operator | Inputs | Properties | Comment |
|---|---|---|---|
| filter | 1 | condition | |
| transform | 1 | projection | |
| join | n | condition, projection | |
| group | n | keys, aggregation | co-group with n > 1 |
| intersect | n | - | set intersection |
| union | n | - | set union |
| subtract | n | - | set difference |
| union all | n | - | bag union |
| replace | 1 | path, dictionary | dictionary lookup |
| pivot | 1 | path | (un)nesting of tree |
| split | 1 | path | denormalizes arrays |

**Table 1: Core operators.**

*where* and an *into* property.

A complete list of the mostly relational, standard operators is given in Table 1. The third columns shows the configurable properties of the operators, e.g., filter has a property *condition*, which is specified in Meteor after the *where* keyword. All operators except *intersect*, *union*, and *subtract* use bag semantics similar to relational DBMS. The three exceptions implicitly convert incoming bags into sets and always return sets. In contrast, *union all* performs only a concatenation of the inputs. The three operators *replace*, *pivot*, and *split*, are especially designed to work with semi-structured data.

To import application-specific operators, Meteor users import packages with **using** <package>. Users can also access operators with a package prefix without prior import, e.g., ie:**annotate**. Thus, users may specify which operator to use in cases where two packages use the same name for different operators. We explain the underlying mechanism for supporting name spaces in Sect. 3.2.

Additionally to operators, Meteor allows users to define and import functions. Function definitions inside a Meteor script have the same expressive power as a new Meteor script. These functions serve to shorten a script that repeatedly uses a given sequence of operators or expressions. Additionally, users can register Java functions with the Meteor function *javaudf* if they need functionality that cannot be expressed in Meteor or is more efficient in Java. Java functions are also automatically registered during package import if the package contains built-in, application-specific functions, such as a function *levenshtein* in our running example that computes the Levenshtein distance.

The syntax of Meteor is inspired by Jaql [6], a scripting language for Hadoop developed by IBM. Jaql follows a functional approach to let users specify the operators in a query (except for six base operators, such as join or filter). In contrast, Meteor users configure operators in a more object-oriented way. Also, the syntax of Meteor is tailored towards the package concept and thus simplifier than Jaql in two ways. First, Meteor enforces the convention that all variables must start with $ to help humans and machines to better distinguish between operators and variables. For example, the assignment *filter = group filter by filter* is valid

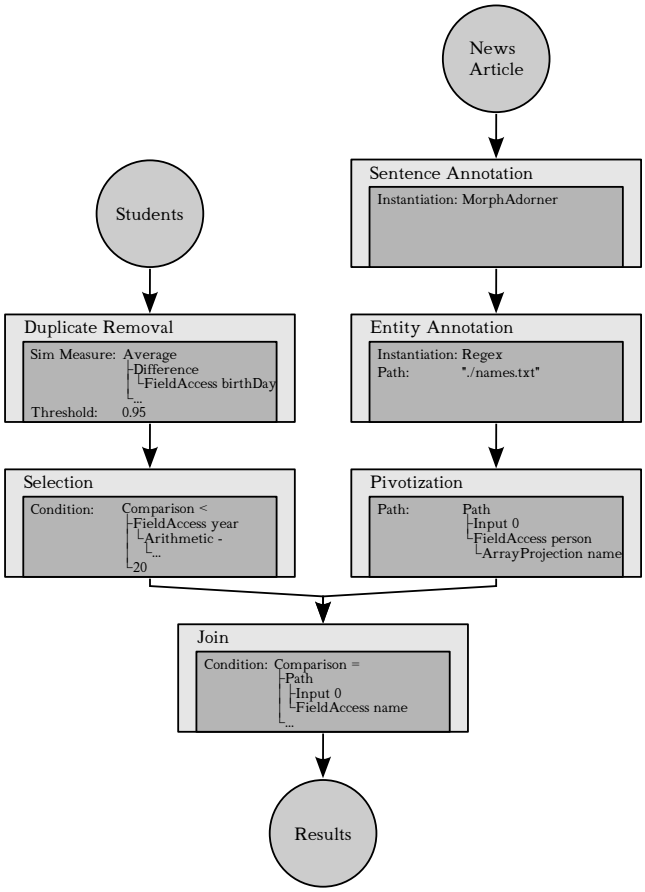


**Figure 3: Sopremo plan for query in Listing 1.**

Jaql but would not compile in Meteor. Second, we dropped the pipe notation of Jaql, e.g., `$teens = $students ->` **filter**. Pipe notation makes sense only for operators that have one input and one output but Sopremo also supports operators that have multiple inputs and multiple outputs.

The meteor parser parses any given Meteor query into an abstract syntax tree and then translated it into a logical execution plan of Sopremo operators. The plan is then handled by the Sopremo layer.

## 3.2 The Sopremo Operator Model

Sopremo is a framework to manage an extensible collection of semantically rich operators organized into packages. It acts as a target for the Meteor parser and ultimately produces an executable Pact program. Nevertheless, we designed Sopremo to be a common base for additional query languages such as XQuery or even GUI builders.

Figure 3 depicts the Sopremo plan that Meteor returns for our running example. Relational operators co-exist with application-specific operators, e.g., data cleansing and information extraction operations such as **remove duplicates** and **annotate** persons. All variables in a Meteor script are replaced by edges, which represent the flow of data.

Operators may have several properties, e.g., the **remove duplicates** operator has a similarity measure and a threshold as properties. The values of properties belong to a set of expressions that process individual Sopremo values or groups thereof. These expressions can be nested to form

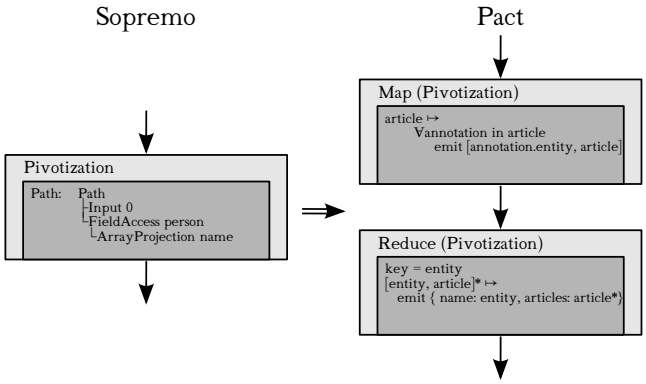


**Figure 4: Transformation of pivotization. Left: Sopremo operator, right: partial Pact plan.**

trees to perform more complex calculations and transformations. For example, the selection condition of the example plan (see Fig. 3) compares the year of the calculated age with the constant 20 for each value in the data set `$students`.

Sopremo operators and packages can be developed independently. To be able to use and combine operators from different packages, operators must be self-descriptive in two ways. First, each operator provides meta information about itself including their own configurable properties and certain characteristics that can be used during optimization. Second, all operator instantiations must define in which way they are executed and parallelized. In particular, each operator must provide at least one Pact workflow that executes the desired operation.

Figure 4 shows how the instantiated operator **pivot** side is implemented as a *partial* Pact plan that consists of a Map and a Reduce. The Pact plan for operators are only partial because they cannot be executed due to the missing sources and sinks. Further, these partial Pact plans have the same amounts of incoming and outgoing data flows as the corresponding Sopremo operator. For complex Sopremo operators such as **remove duplicates**, partial plans may easily consist of 20 Pacts.

Operator instantiations may also reuse other operators in their implementation. These *composite* operators recursively translate the reused operators into partial Pact plans and rewire the input and outputs to form even larger partial Pact plans. Composition of operators reduces implementation complexity. Future improvements of a reused operator also improve the composite operator.

Additionally, operators may also have different implementation strategies. The strategy can either be selected by properties or by an optimizer. For example, a Sopremo join with an arbitrary join condition may require a theta join with a cross Pact, while a join with a equality condition can be efficiently executed with a match Pact.

## 3.3 Query Compilation

In Sopremo, all operator instantiations have a direct Pact implementation. Thus, the compilation of a complete Sopremo plan consists of two steps. First, all operator instantiations in the Sopremo plan are translated into partial Pact plans. Second, the inputs and outputs of the partial Pact plans are rewired to form a single, consistent Pact plan. The result of the translation process for our running example can

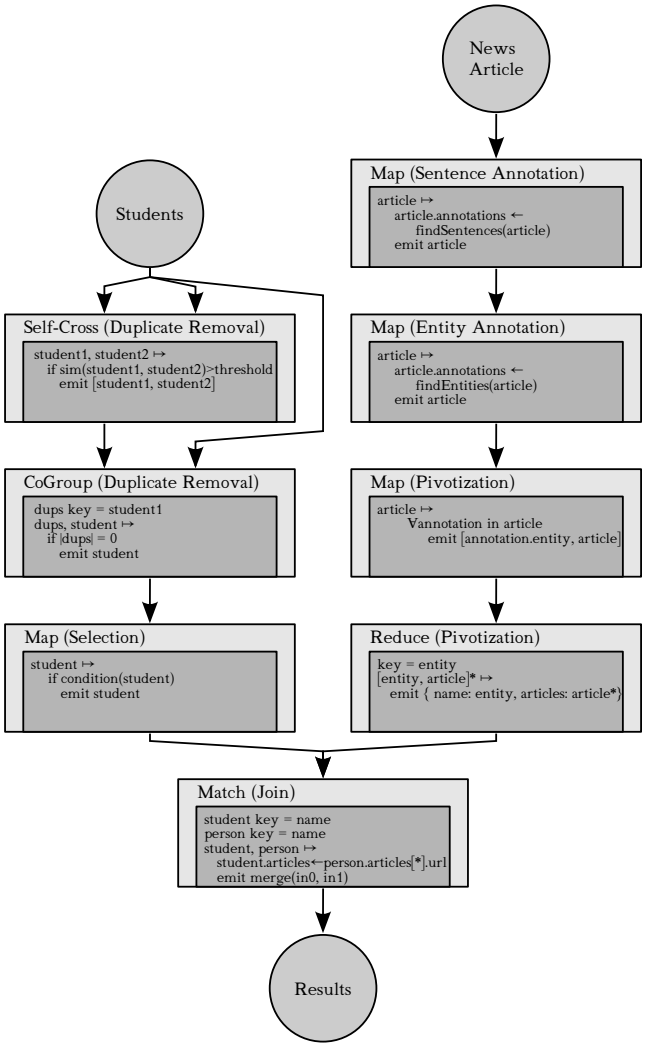


**Figure 5: Pact plan.**

be seen in Fig. 5. Please note that we here used naive duplicate detection for visualization purposes only. Section 4.2 discusses further alternatives.

To improve the runtime efficiency of a compiled plan, the translation process is augmented with two more steps: First, a Sopremo plan is logically optimized (work in progress, see Sect. 6). A separate, physical optimization is performed later in the Pact layer [18]. Second, Pact uses a flat, schemaless data model that is necessary to reorder Pacts. For pure Pact programs, the schema interpretation is performed by Pact users in their UDFs. However, the additional semantics of Sopremo operator allows Sopremo to infer an efficient data layout and bridge the gap between the flat data model of the Pact model and the nested data model of Sopremo. Meteor or Sopremo users thus do not have to specify the data layout explicitly.

Figure 6 summarizes the complete process of translating a Meteor query into a Pact plan. Meteor users formulate a query that is parsed into a Sopremo plan. To import packages, Meteor requests the package loader of Sopremo to inspect the packages and register the discovered operators and predefined functions. Meteor uses this information to

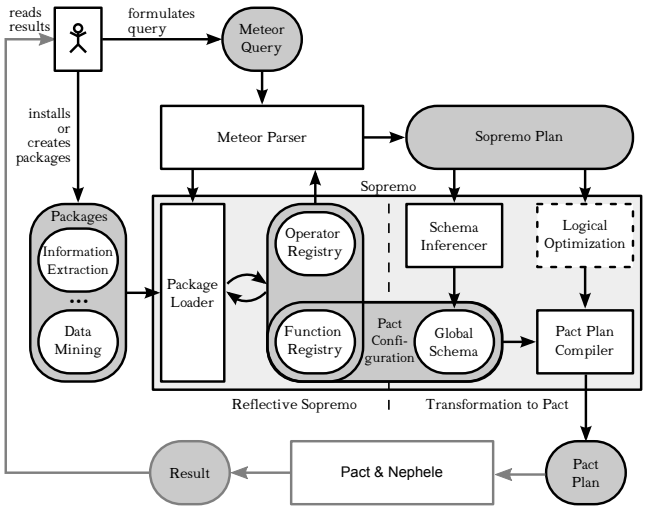


**Figure 6: Architecture of Sopremo.**

validate the script and translate it into a Sopremo plan. The plan is analyzed by the schema inferencer to obtain a global schema that is used in conjunction with the Sopremo plan to create a consistent Pact plan.

## 3.4 Query execution

The previous sections describe how a Meteor query is successively translated into a Pact program. This happens once at query time on the master node of the cluster. During this process, Sopremo injects code into all Pacts to bridge the conceptional gap between Pacts and Sopremo operators during execution time. In this section, we briefly outline how Sopremo influences the actual execution of a translated query on a cluster.

The standard execution of a Pact plan involves four steps: First, compute nodes are allocated by Nephele and the UDFs are distributed to each allocated node. Second, the UDF is instantiated on each worker thread and a callback is called to allow the UDF to configure itself with information specified at query time. Third, the UDF is called for each input tuple or partition and finally, the UDF is disposed. Sopremo adds glue code to the second and third step of that process.

Most importantly, Sopremo uses the configuration callback to deserialize the context of the current Pact. The context includes the globally inferred schema, the function registry with all user-defined and imported functions, and also the values of the properties. The context additionally contains debug information that allow users to exactly identify which (composite) operator and which step a Pact belongs to in case of errors.

Afterwards, the context is used to properly deserialize incoming Pact values into Sopremo values for each call of the UDFs of the Pacts. The values are then used by the implementation of the corresponding operator to calculate outgoing Sopremo values. Finally, these values are serialized again into schema free tuples.

When the query successfully finishes, it produces one or more output files that are encoded in Json unless otherwise specified. In case of an error, Meteor users receive two kinds of feedback depending on the type of error. Firstly, a Meteor script may be invalid. Operators check their configuration

| Operator | # Inputs | Subtypes | Properties |
|---|---|---|---|
| **annotate** | 1 | entities, relations, sentences, tokens, pos, stems, lemmas | instantiation, path |
| **replace** | 1 | token, entity, fields, | instantiation, projection condition path |
| **unnest** | 1 | | fields |
| **join** | n | | condition |
| **aggregate** | n | | key |
| **split into** | 1 | sentences, tokens, ngrams | condition, instantiation |
| **extract** | 1 | entity, relation | instantiation |

**Table 2: Information extraction operators. Top: basic operators, bottom: complex operators.**

before their translation and may issue warnings or errors if properties are conflicting or if prerequisites are missing. Secondly, Sopremo operators may contain errors. In this case, Meteor shows a detailed stack trace of the erroneous operator on the master node. This essentially means that the Sopremo plan is reconstructed at execution time and stack traces from the cluster nodes are transferred to the master node. Furthermore, Sopremo optionally adds debug information to allow values to be traced along a script execution to ease debugging of Sopremo operators as well as Meteor scripts.

# 4. CONCRETE PACKAGES

Meteor and Sopremo are both extensible with application-specific packages and operators. In order to avoid conflicts when combining operators from different packages, Meteor supports namespaces. In this section, we introduce two packages for IE and DC and explain available operators in each package.

## 4.1 Information Extraction

Generally, an IE operator transforms some input text data into some output by applying a function to the input. Different operators are dedicated to different purposes. As displayed in Table 2, we distinguish between basic and complex operators. The set of basic operators comprises *annotate*, *replace*, *unnest*, *join*, and *aggregate*. The set of complex operators contains by now a *split into* and an *extract* operator. Operators can be specified further by sub-types. This also allows for an easy extension of operators by creating new sub-types if desired. In the following, we briefly describe the available operators and sub-types.

The *annotate* operator is used to add information to a given input text. By now, we support annotating *entities*, binary *relations* between entities, and the annotation of structural information, such as *sentences* or *tokens*, and linguistic information on *pos* tags, *stems*, or *lemmas*. For each sub-

class there are different IE algorithms available that differ heavily in terms of runtime, startup costs, memory consumption, and quality. Both baseline variants and specialized third-party libraries are available as operator instantiations. A user can specify a variant in his/her query by adding the keyword "using" and the name of the algorithm. For example, *extract relations using svm* performs relation extraction using a support vector machine based approach, whereas *extract relations using co-occ* performs relation extraction based on co-occurrences of entities in the same scope.

*Replace* is responsible for either replacing existing annotations (*tokens* or *entities*) or entire *fields* in the Json record *with* some user-defined value. Example applications of *replace* are stop word removal (e.g., by specifying a list of stop words that is searched for and an empty string as replacement for each matched stop word), entity blinding as a preprocessing step for relation extraction, or entity normalization. Compared to the replace operator in the core package, this replace operator is more extensive, since it not only performs string replacements, but is also capable of replacing entire fields or complex annotations.

We use *unnest* to flatten nested annotations, e.g., to split up a bag of entity annotations of the same type. *Join* is used to merge records or annotations. The semantics of the join operator is that if record a and record b shall be joined into record $c$, and $a$ and $b$ contain annotations of the same type, these annotations are unioned in the output record $c$. The decision, whether $a$ and $b$ shall be joined is taken by evaluating a join condition, which might be fuzzy. Finally, the *aggregate* operator merges records and existing annotations based on a user-defined key that is specified by the keyword *by*. The merge semantics of aggregate is similar to join, i.e., if records $a, ..., k$ share the same key, all records are merged into a single record $l$, such that all annotation objects contained in $a, ..., l$ are unioned by field type.

The complex operators *split into* and *extract* can be composed from basic operators. For example, *split into sentences* can be performed by first annotating sentence boundaries in a given input text (i.e., a whole document). Internally, sentence annotations are stored in a list that contains start and end position of the individual sentences. This list needs to be unnested, such that a single Json record is produced for each sentence annotation. Finally, the original input text needs to be replaced by a substring of itself, which is defined by start and end position of a sentence. Similarly, the *extract* operator for entities or relations can be expressed using a series of *annotate*, *filter*, *replace* and *unnest* operations. However, for both complex operators and their subtypes, there are also more efficient instantiations available that perform unnesting, filtering, and replacements of fields or annotations in a single step. In particular, operators and their sub-types can be improved by means of quality and runtime incrementally, i.e., simply by adding a new operator instantiation.

Lines 11 to 15 in Listing 1 display a fully specified Meteor query for annotating person names in news articles as part of our running example. In Line 12f., we annotate sentence boundaries in the input text by applying the MorphAdorner[3] sentence splitting algorithm to the input (specified by *using*). Sentence splitting is in our example a preprocessing step for person name annotation, because we require that a person name does not go beyond the scope of a sentence. In Line 14f., we annotate person names using an algorithm based on matching regular expressions. The expression "type.person" specifies the concrete entity type that is annotated and it is part of the Json object that will be created and added to the input by the annotation operator. Information on concrete entity types are useful for subsequent IE operators, such as an *annotate relation* operator that annotates relations between persons and companies. We also define the set of relevant expressions for person names by specifying the filename "names.txt". Internally, the regular expression based entity annotation algorithm loads this file first and then matches the input text.

All underlying operator implementations are built using the Sopremo programming framework. The implementation of the basic operators *annotate*, *replace*, and *unnest* and their sub-types build upon Map contracts, which process each input item independently. For *merge*, we use a Match contract, and *aggregate* is built using CoGroup and Reduce.

The different sub-type instantiations might have dependencies on other annotation operator variants that need to be executed in advance. For example, a text tokenization algorithm might need information on sentence boundaries and thus, *annotate sentences* needs to be performed first. By now, users need to resolve these dependencies by hand, but we plan to integrate a dependency resolving algorithm in the Sopremo compiler (see Section 6).

[3]http://morphadorner.northwestern.edu

| Operator | Inputs | Properties |
|---|---|---|
| scrub | 1 | rules |
| split records | 1 | projections |
| detect duplicates | 1 | sim, threshold, sorting or partition key |
| fuse | 1 | strategies, weights |
| remove duplicates | 1 | sim, threshold, sorting or partition key, strategy |
| link/cluster records | n | sim, threshold, sorting or partition key, projection |

**Table 3: Data cleansing operators. Top: basic operators, bottom: complex operators.**

## 4.2 Data Cleansing

The DC package comprises operators that are commonly used to improve the data quality. Important tasks are duplicate detection, integration of several independent data sources, data fusion, and schema alignment. Similar to IE operators, the package consists of basic operators and complex operators as listed in Table 3. The table also shows all properties that can be configured by users in the last column.

The basic operators improve the quality of one data source. The *scrub* operator applies a set of rules to a data source to correct data errors and normalize values for later steps. Further, *split records* help to align schemata by projection the data set into smaller data sets. In contrast to a simple projection, this operator maintains relationships between the fragments of one records. Next, *detect duplicates* aims to identify all pairs of records that represent to the same real-world entity. Finally, *fuse* resolves data conflicts between duplicates using data fusion strategies.

The complex operators are composed of the previously mentioned operators. We already briefly introduced the *remove duplicates* operator in Section 3 in our running example. The operator combines the basic operators *detect duplicates* and *fuse*. Further, the two operators *link records* and *cluster records* find duplicates across multiple data sources and are used to integrate data sources or to generate connections between data sources, e.g., *owl:sameAs* links in the LOD cloud. The latter operator additionally calculates the transitive closure of the found duplicates, for instance by using an adaption of the three phase algorithm of Katz and Kider [20].

We already described the data integration operators in Reference [15], namely *scrub*, *split records*, *link/cluster records*, and *fuse*. In the following, we will explain the *detect duplicates* and the *remove duplicates* operators and their properties in more detail.

In the running example (see Listing 1), we receive a list of enrolled students of all schools in the county. Naturally the database is dirty: students change schools for various reasons even within a school year and may be listed in two schools. Hence, we use the *remove duplicates* operator to detect duplicate entries and choose the best representation of the student.

Duplicate detection would naively require comparing all entries with each other resulting in huge computational costs. In practice, various candidate selection techniques have been proposed that specifically select potential duplicates to avoid the Cartesian product and speed up duplicate detection at the cost of recall. However, they introduce new properties that the user must set or that need to be inferred automatically. For example, the popular sorted neighborhood method [16] needs sorting keys and a window size as additional parameters, while standard blocking [12] requires a blocking key.

To decide whether a candidate pair is a duplicate, the similarity of the entries is calculated and compared against a threshold. The properties similarity measure and threshold are specified in one comparison expression in Meteor but can be individually used for optimization. For instance, a similarity measure involving the Jaccard distance and a high threshold can be efficiently executed with a set similarity join [24].

Finally, in our running example, we use data fusion to choose the record with the most current enrollment date and drop all other records in the cluster. However, in general, we can specify in detail which datum we prefer for which attribute. We may choose to complete missing data, to use the minimum or maximum value, to aggregate all values, or we can even use other attributes or meta-information to decide which value to use. We implemented most of the strategies that were presented by Bleiholder and Naumann [7].

The presented data cleansing package covers a wide range of data cleansing applications but will be continuously improved to incorporate new operators and new implementations.

# 5. RELATED WORK

Building a high level language layer that abstracts from underlying massively parallel data processing systems is a quite common concept. For *Hadoop* [26] there exist several such layers like *Apache Pig* [14, 22], *Jaql* [5, 6], and *Hive* [23].

*Pig* is a parallel processing environment for data flow programs [14, 22]. Programs are formulated in *Pig Latin*, a declarative language that can describe directed acyclic graphs (DAG) of operators. In these DAGs data originates from sources, is passed on to operators that may filter, join, group, or project data until the data reaches a sink, which is used to store the results. Data passed between operators has a flexible structure that allows atomic values, tuples, bags, and maps. Each field of complex types may be of any data type with the only exception being map, which requires atomic keys. Operators can be parameterized with expressions that allow for UDFs and thus enable the extension of the language with user code. During compilation *Pig Latin* programs are first transformed into a DAG of operators, which resembles logical query plans of relational database systems and facilitates similar optimization strategies. Optimized logical plans are compiled into a sequence of MapReduce jobs that can be submitted to the Hadoop job manager for execution. In addition to functions, Meteor and Sopremo allow for the definition of completely new operators. As a result, Meteor programs can be highly expressive since the desired functionality does not need to be expressed with a fixed set of operators.

*Jaql* [5, 6] is a software stack similarly structured as Pig. It consists of a functional scripting language, a MapReduce compiler and the Hadoop execution engine. Similar to Pig and Meteor, a Jaql program also forms a DAG of operators. However, in Jaql, operators are expressed as functions. Programs are therefore expressed as function compositions. Just like in other functional languages, functions are first class citizens in Jaql and can be assigned to variables. The data model of Jaql is Json, which supports the use of atomic values, arrays, and records. Extensibility of the Jaql language is provided directly at the language level. Physical Jaql execution plans are expressed with plain Jaql syntax. In contrast to Meteor and Sopremo, Jaql needs no language extensions to express new functionality. The source to source translator, that converts a Jaql program into an execution plan, applies greedy rewriting rules logical optimization, e.g. filter push-downs. Currently, Jaql does not have a physical optimizer. Since Sopremo plans are compiled to Pact programs, Sopremo benefits both from logical and physical optimization components in Stratosphere.

*Hive* provides data warehouse functionality and is modeled like a traditional relational database system [23]. Hive relies on Hadoop for query processing and uses HDFS as storage layer. Queries are formulated in *HiveQL*, a SQL dialect that includes a subset of standard SQL and adds Hive-specific extensions. Queries are compiled and rewritten by a rule-based optimizer and executed as MapReduce jobs. Hive's data model is not strictly relational since it supports structured attributes that include nested lists and maps. In terms of extensibility, Hive supports UDFs similar to SQL. Again, in contrast to Meteor and Sopremo no additional operators can be defined and physical optimization is currently not supported.

The *Asterix Query Language* (AQL) is the high level query layer of the Asterix system, which executes on the Hyracks data processing engine [4, 8]. AQL queries are centered around nested FLWOR expressions, which originate from XQuery. The language offers a rich data model that supports nested records, lists, and enumerations. Queries are compiled to an intermediate representation, Algebricks, which is in turn is compiled to a DAG of operators, which are

executed parallel in on Hyracks. In terms of the features provided, AQL and Meteor/Sopremo/Stratosphere are very similar, both support UDFs. In contrast to Meteor customized operators are currently not supported. Some high level languages for expressing IE and DC tasks declaratively have also been developed. *SystemT*'s *AQL* [10] and *AJAX* [13] are such languages and are both heavily influenced by SQL. Both languages provide relational operators in addition to operators tailored for the extraction and aggregation of relevant document spans or data cleansing, respectively. *XClean* is tailored to perform DC in XML files [17]. Here, DC programs are written in *XClean/PL* using a library of pre-defined operators. XClean programs are then compiled to XQuery and executed using a standard XQuery processor.

# 6. CONCLUSION

In this paper, we presented a high level language layer for parallel data processing engines. Meteor, a declarative scripting language influenced by Jaql, enables end users to express sophisticated data flows. Sopremo, the underlying operator layer provides a modular and highly extensible set of application-specific operators. Currently, pre-defined relational operators as well as packages for IE and DC are available. We have shown that operators from these packages can be easily used together in a single Meteor program to implement advanced use cases that require functionality from both packages.

Future work on Meteor and Sopremo will focus on the extensions of its current capabilities as well as on optimization techniques and improved compatibility.

One of our objectives is to provide packages that facilitate a variety of use cases like scientific data processing or business intelligence. Instead of relying on different domain specific languages (DSL) for each use case, Meteor and Sopremo are meant to reach the efficiency of DSLs without sacrificing the flexibility of general purpose languages by providing tailor-made, yet interoperable domain specific operator packages.

Our current focus is the development of sophisticated natural language processing solutions. Beside the raw functionality, we also plan to create of a metadata system that models precisely both the required input and the generated output properties of the data that is passed to and from each operator. The resulting ontology will allow for type checking to recognize the use of operators on incompatible data before runtime, e.g., the application of a syntactic parser that requires POS tags to a collection of documents without such tags.

Building on top of the metadata system we would also like to optimize the execution of meteor scripts to runtime parameters of its execution environment. Specifically, we intend to account for concurrent and repeated execution of Meteor programs. If one considers several Meteor programs that perform analysis tasks on periodically updated documents, it is very likely that these tasks share certain processing steps such as word tokenization, sentence splitting or language detection. In this case, it could be beneficial to merge the corresponding portions of the programs into a single Meteor job that passes its result to the remaining portions of the original jobs.

In a similar fashion, the execution engine could employ the metadata system to monitor how often certain computations are performed on certain data sets. If frequent re-computations are observed, the system could decide to materialize the corresponding data set and replace the computations in the corresponding Meteor programs with data access routines.

Finally, we would like to enable compatibility with different execution environments. While currently each non-composite Sopremo operator can return its Pact implementation, we would like to extend this functionality to cover other execution engines as well. By adding implementations for different execution engines, such as Hyracks or Hadoop, for all non-composite operators, entire Sopremo programs could be executed on these engines.

# 7. ACKNOWLEDGMENTS

Arvid Heise and Astrid Rheinländer are funded by the German Research Foundation under grant "FOR 1036: Stratosphere – Information Management on the Cloud." Marcus Leich is funded by the European Institute of Technology (EIT). We would like to thank all members of the Stratosphere team, especially Volker Markl and Fabian Hueske, for valuable discussions and support.

# 8. REFERENCES

[1] D. Agrawal, P. Bernstein, E. Bertino, S. Davidson, U. Dayal, M. Franklin, J. Gehrke, L. Haas, A. Halevy, J. Han, H. Jagadish, A. Labrinidis, S. Madden, Y. Papakonstantinou, J. M. Patel, R. Ramakrishnan, K. Ross, C. Shahabi, D. Suciu, S. Vaithyanathan, and J. Widom. Challenges and Opportunities with Big Data. A community white paper developed by leading researchers across the United States. http://imsc.usc.edu/research/bigdatawhitepaper.pdf, 2012.

[2] A. Alexandrov, D. Battré, S. Ewen, M. Heimel, F. Hueske, O. Kao, V. Markl, E. Nijkamp, and D. Warneke. Massively parallel data analysis with PACTs on nephele. *Proceedings of the VLDB Endowment (PVLDB)*, 3(2), 2010.

[3] 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 *Symposium on Cloud Computing (SoCC)*, 2010.

[4] A. Behm, V. R. Borkar, M. J. Carey, R. Grover, C. Li, N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou, and V. J. Tsotras. Asterix: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, 29(3), 2011.

[5] 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. *Proceedings of the VLDB Endowment (PVLDB)*, 4(12), 2011.

[6] K. S. Beyer, V. Ercegovac, R. Krishnamurthy, S. Raghavan, J. Rao, F. Reiss, E. J. Shekita, D. E. Simmen, S. Tata, S. Vaithyanathan, and H. Zhu. Towards a scalable enterprise content analytics platform. *IEEE Data Engineering Bulletin*, 32(1), 2009.

[7] J. Bleiholder and F. Naumann. Declarative data fusion – syntax, semantics, and implementation. In *Advances in Databases and Information Systems (ADBIS)*, 2005.

[8] V. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2011.

[9] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment (PVLDB)*, 1(2), 2008.

[10] L. Chiticariu, R. Krishnamurthy, Y. Li, S. Raghavan, F. R. Reiss, and S. Vaithyanathan. SystemT: an algebraic approach to declarative information extraction. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, 2010.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[12] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2007.

[13] H. Galhardas, D. Florescu, D. Shasha, and E. Simon. AJAX: An extensible data cleaning tool. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2000.

[14] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of Map-Reduce: the Pig experience. *Proceedings of the VLDB Endowment (PVLDB)*, 2(2), 2009.

[15] A. Heise and F. Naumann. Integrating open government data with Stratosphere for more transparency. *Web Semantics: Science, Services and Agents on the World Wide Web*, 2012.

[16] M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 1995.

[17] M. Herschel and I. Manolescu. Declarative XML data cleaning with XClean. In *Proceedings of the Conference on Advanced Information Systems Engineering (CAiSE)*, 2007.

[18] F. Hueske, M. Peters, M. J. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas. Opening the black boxes in dataflow optimization. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2012.

[19] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *Proceedings of the ACM International Conference on Special Interest Group on Operating Systems (SIGOPS)*, 41(3), 2007.

[20] G. J. Katz and J. T. Kider, Jr. All-pairs shortest-paths for large graphs on the GPU. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, 2008.

[21] B. O'Connor, R. Balasubramanyan, B. R. Routledge, and N. A. Smith. From tweets to polls: Linking text sentiment to public opinion time series. In *Proceedings of the Fourth International Conference on Weblogs and Social Media (ICWSM)*, 2010.

[22] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2008.

[23] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment (PVLDB)*, 2(2), 2009.

[24] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using MapReduce. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2010.

[25] D. Warneke and O. Kao. Nephele: Efficient parallel data processing in the cloud. In *Workshop on Many-Task Computing on Grids and Supercomputers (SC-MTAGS)*, 2009.

[26] T. White. *Hadoop: The Definitive Guide*. first edition edition, 2009.

[27] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.