The VLDB Journal
DOI 10.1007/s00778-014-0357-y

REGULAR PAPER

The Stratosphere platform for big data analytics

Alexander Alexandrov · Rico Bergmann · Stephan Ewen · Johann-Christoph Freytag · Fabian Hueske · Arvid Heise · Odej Kao · Marcus Leich · Ulf Leser · Volker Markl · Felix Naumann · Mathias Peters · Astrid Rheinländer · Matthias J. Sax · Sebastian Schelter · Mareike Höger · Kostas Tzoumas · Daniel Warneke

Received: 10 July 2013 / Revised: 18 March 2014 / Accepted: 1 April 2014

Abstract We present Stratosphere, an open-source software stack for parallel data analysis. Stratosphere brings together a unique set of features that allow the expressive, easy, and efficient programming of analytical applications at very large scale. Stratosphere's features include "in situ" data processing, a declarative query language, treatment of user-defined functions as first-class citizens, automatic program parallelization and optimization, support for iterative programs, and a scalable and efficient execution engine. Stratosphere covers a variety of "Big Data" use cases, such as data warehousing, information extraction and integration, data cleansing, graph analysis, and statistical analysis applications. In this paper, we present the overall system architecture design decisions, introduce Stratosphere through example queries, and then dive into the internal workings of the system's components that relate to extensibility, programming model, optimization, and query execution. We experimentally compare Stratosphere against popular open-source alternatives, and we conclude with a research outlook for the next years.

Stratosphere is funded by the German Research Foundation (DFG) under grant FOR 1306.

A. Alexandrov · S. Ewen · F. Hueske · M. Höger · O. Kao · M. Leich · V. Markl · S. Schelter · K. Tzoumas (✉)
Technische Universität Berlin, Berlin, Germany
e-mail: kostas.tzoumas@tu-berlin.de

A. Alexandrov
e-mail: alexander.alexandrov@tu-berlin.de

S. Ewen
e-mail: stephan.ewen@tu-berlin.de

F. Hueske
e-mail: fabian.hueske@tu-berlin.de

M. Höger
e-mail: mareike.hoger@tu-berlin.de

O. Kao
e-mail: odej.kao@tu-berlin.de

M. Leich
e-mail: marcus.leich@tu-berlin.de

V. Markl
e-mail: volker.markl@tu-berlin.de

S. Schelter
e-mail: sebastian.schelter@tu-berlin.de

R. Bergmann · J.-C. Freytag · U. Leser · M. Peters · A. Rheinländer · M. J. Sax
Humboldt-Universität zu Berlin, Berlin, Germany
e-mail: bergmann@informatik.hu-berlin.de

J.-C. Freytag
e-mail: freytag@informatik.hu-berlin.de

U. Leser
e-mail: leser@informatik.hu-berlin.de

M. Peters
e-mail: mathias.peters@informatik.hu-berlin.de

A. Rheinländer
e-mail: rheinlae@informatik.hu-berlin.de

M. J. Sax
e-mail: mjsax@informatik.hu-berlin.de

A. Heise · F. Naumann
Hasso Plattner Institute, Potsdam, Germany
e-mail: arvid.heise@hpi.uni-potsdam.de

F. Naumann
e-mail: felix.naumann@hpi.uni-potsdam.de

D. Warneke
International Computer Science Institute, Berkeley, CA, USA
e-mail: warneke@icsi.berkeley.edu

Published online: 06 May 2014

Keywords Big data · Parallel databases · Query processing · Query Optimization · Data cleansing · Text mining · Graph processing · Distributed systems

1 Introduction

We are in the midst of a "Big Data" revolution. The plunging cost of hardware and software for storing data, accelerated by cloud computing, has enabled the collection and storage of huge amounts of data. The analysis and exploration of these data sets enable data-driven analysis that has the potential to augment or even replace ad-hoc business decisions. For example, web companies track user behavior to optimize their business. Large scientific experiments and simulations collect huge amounts of data, and scientists analyze these to form or validate hypotheses.

Commercial RDBMS products cannot cope with the scale and heterogeneity of the collected data sets, and their programming and data model is not a good fit to the new analysis workflows. These reasons have led to a reconsideration of methods for managing data at scale, leading to new software artifacts developed by academia and industry. The "Big Data" software ecosystem includes distributed file systems [29], parallel data analysis platforms [8,12,15,22,44,70,74], data programming languages [13,19,37,57,69], and more specialized tools for specific data domains [53,54].

We present Stratosphere, a data analytics stack that enables the extraction, analysis, and integration of heterogeneous data sets, ranging from strictly structured relational data to unstructured text data and semi-structured data. The Stratosphere system can perform information extraction and integration, traditional data warehousing analysis, model training, and graph analysis using a single query processor, compiler, and optimizer.

Stratosphere brings together a unique set of features that we believe are an essential mix for supporting diverse analytical applications on "Big Data."

First, we believe data analysts are more productive when using declarative, high-level languages rather than low-level languages. Stratosphere includes such a query language. In addition, the system can serve as a suitable compilation platform for several other languages for different domains. By offering an extensible intermediate layer, and by exposing several layers of the system stack as programming models with an underlying optimizer, query languages can be compiled to Stratosphere with less effort (e. g., often without implementing an own optimizer for the specific language), and this compilation can lead to better performance.

Second, Stratosphere enables "in situ" data analysis by connecting to external data sources, such as distributed file systems that often act as the "landing points" of heterogeneous data sources from various organizations. That way, an expensive data loading process is not needed; Stratosphere does not store data, but only converts it to optimized binary formats after the initial scans.

Third, Stratosphere uses a richer set of primitives than MapReduce, including primitives that allow the easy specification, automatic optimization, and efficient execution of joins. This makes the system a more attractive compilation platform for data warehousing languages and applications.

Fourth, Stratosphere treats user-defined functions (UDFs) as first-class citizens throughout the system's stack, including the system's optimizer. This widens the applicability and extensibility of the system to problem domains beyond traditional data warehousing queries, such as information extraction from textual data and information integration.

Fifth, Stratosphere includes a query optimizer that automatically parallelizes and optimizes data analysis programs. The programmer does not need to worry about writing parallel code or hand-picking a join order.

Sixth, Stratosphere includes support for *iterative programs*, programs that make repeated passes over a data set updating a model until they converge to a solution. This enables the specification, optimization, and execution of graph analytics and statistical applications *inside* the data processing engine. Such applications are integrated with data pre- and postprocessing within a single analytical pipeline, cross-optimized, and executed by the same system.

Finally, Stratosphere uses an execution engine that includes external memory query processing algorithms and natively supports arbitrarily long programs shaped as directed acyclic graphs. Stratosphere offers both pipeline (inter-operator) and data (intra-operator) parallelism.

Stratosphere is a layered system that offers several programming abstractions to a user. We discuss them in top-down order, higher-level abstractions being more declarative and amenable to automatic optimization. The Meteor query language [37] offers a declarative abstraction for processing semi-structured data. The PACT programming model [8] is analogous to, and in fact a generalization of, the MapReduce model; PACTs offer a moderately low-level programming abstraction consisting of a fixed set of parallelization primitives and schema-less data interpreted by user-defined functions written in Java. This level of abstraction is especially useful for implementing complex operators that do not "fit" in a query language. Finally, the Nephele programming abstraction [67] allows a power user to specify custom parallelization schemes.

Over the last years, our research in Stratosphere has advanced the state of the art in data management in several aspects. We proposed a data programming model based on second-order functions to abstract parallelization [3,8], a method that uses static code analysis of user-defined functions to achieve goals similar to database query optimization in a UDF-heavy environment [41–43], abstractions to inte-

grate iterative processing in a dataflow system with good performance [25,26], an extensible query language and underlying operator model [37], techniques to infer cloud topologies and detect bottlenecks in distributed execution [9–11], as well as techniques to exploit dynamic resource allocation [68] and evaluate compression schemes [40]. We have, finally, evaluated Stratosphere on a variety of analytical use cases [14,51].

The contribution of this paper lies less in the individual research findings and more in placing these findings into a larger perspective. We, for the first time, present the architecture of the Stratosphere system as a whole and the interplay between various components that have appeared in individual publications. We discuss in detail the query optimization process in Stratosphere. In addition, we conduct an extensive experimental study against the open-source state of the art. Finally, we discuss lessons learned from building the system and offer our research outlook for the next years. Stratosphere is an open-source project available at `www.Stratosphere.eu` under the Apache license.

The rest of this paper describes the architecture, interfaces, applications, and internal workings of the system at its current stage, as well as highlights several research innovations that advance the state of the art in massively parallel data processing. Section 2 presents an overview of the Stratosphere system. Section 3 discusses Meteor, the Stratosphere query language, from an end-user perspective, exemplified via two use cases. Section 4 presents the underlying extensible Sopremo operator model. Section 5 presents the PACT programming model, the model of parallelization used in Stratosphere. Section 6 presents the optimization phases and techniques employed in Stratosphere. Section 7 discusses how programs are actually executed in Stratosphere by the Nephele distributed dataflow engine and Stratosphere's runtime operators. Section 8 experimentally compares Stratosphere with other open-source systems. Section 9 discusses ongoing work by the Stratosphere group. Finally, Sect. 10 discusses related work, and Sect. 11 concludes and offers a research outlook.

2 System architecture

The Stratosphere software stack consists of three layers, termed the *Sopremo*, *PACT*, and *Nephele* layers. Each layer is defined by its own programming model (the API that is used to program directly the layer or used by upper layers to interact with it) and a set of components that have certain responsibilities in the query processing pipeline. This section presents the overall Stratosphere architecture, briefly sketches the purpose and responsibilities of each layer, and highlights their interactions. In addition, the section establishes the terminology that is used in the rest of this paper.

The main motivation behind separating the Stratosphere system in three layers with different programming models is to provide users with a choice regarding the declarativity of their programs and to have different compilation targets when the "users" are language compilers. While the programming model of the highest layer, Sopremo, exhibits the highest degree of declarativity and is amenable to similar optimizations as in relational databases, the subjacent PACT and Nephele layers gradually trade declarativity for expressiveness. Through a series of compilation steps, Stratosphere can translate the higher-layer programs into lower-layer programs, thereby exploiting the richer semantics of the higher-level programming models for automatic optimizations in each compilation step.

Figure 1 sketches Stratosphere's architecture and illustrates the functionality each of the three layers provides. In the remainder of this section, we introduce each layer of the Stratosphere stack in top-down order.

Sopremo is the topmost layer of the Stratosphere stack. A Sopremo program consists of a set of logical operators connected in a directed acyclic graph (DAG), akin to a logical query plan in relational DBMSs. Programs for the Sopremo layer can be written in *Meteor*, an operator-oriented query language that uses a JSON-like data model to support the analysis of unstructured and semi-structured data. Meteor shares similar objectives as higher-level languages of other big data stacks, such as Pig [57] or Jaql [13] in the Hadoop ecosystem, but is highlighted by extensibility and the semantically rich operator model Sopremo, which also lends its name to the layer. Through Sopremo, domain specialists can easily integrate application-specific functions by extending Sopremo's set of operators, enabling automatic optimization at compile time for different domains.

Once a Meteor script has been submitted to Stratosphere, the Sopremo layer first translates the script into an operator plan. Moreover, the compiler within the Sopremo layer can derive several properties of the plan, which can later be exploited for the physical optimization of the program in the subjacent PACT layer. The Meteor language is presented by means of examples in Sect. 3. Details about the Sopremo layer and the optimization process are described in Sects. 4 and 6.

The output of the Sopremo layer and, at the same time, input to the PACT layer of the Stratosphere system is a *PACT program*. PACT programs[1] are based on the PACT programming model, an extension to the MapReduce programming model. Similar to MapReduce, the PACT programming model builds upon the idea of second-order functions, called PACTs. Each PACT provides a certain set of guarantees on what subsets of the input data will be processed together, and the first-order function is invoked at runtime for each of these

[1] PACT is a portmanteau for "parallelization contract."

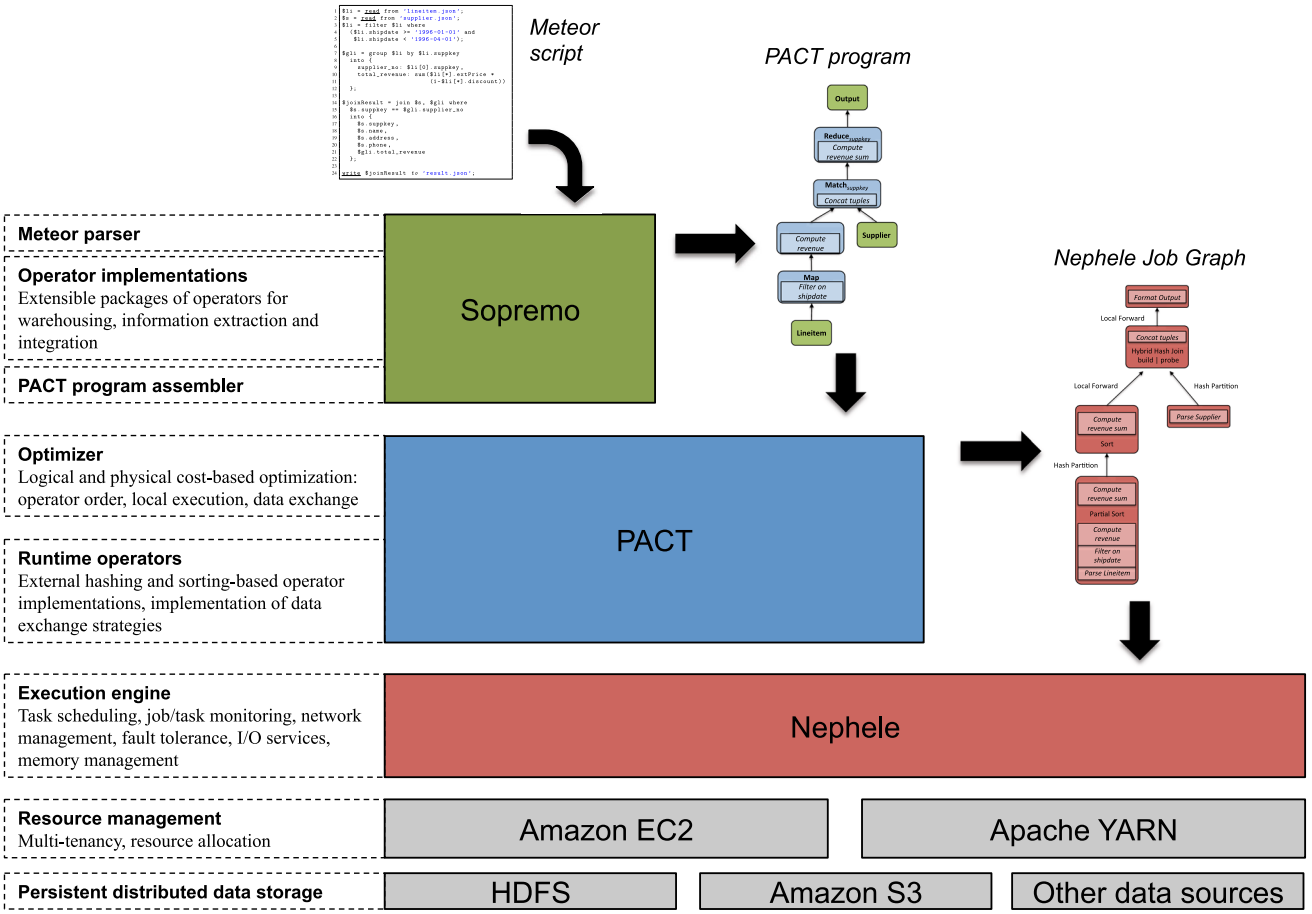

Fig. 1 The Stratosphere software stack. Functionality is distributed into three layers characterized by their distinct APIs (programming models). Stratosphere connects to popular open-source software for resource manager and data storage

subsets. That way, the first-order functions can be written (or generated from a Sopremo operator plan) independently of the concrete degree of parallelism or strategies for data shipping and reorganization. Apart from the Map and Reduce contracts, the PACT programming model also features additional contracts to support the efficient implementation of binary operators. Moreover, PACTs can be assembled to form arbitrarily complex DAGs, not just fixed pipelines of jobs as in MapReduce.

The first-order (user-defined) functions in PACT programs can be written in Java by the user, and their semantics are hidden from the system. This is more expressive than writing programs in the Sopremo programming model, as the language is not restricted to a specific set of operators. However, PACT programs still exhibit a certain level of declarativity as they do not define how the specific guarantees of the used second-order functions will be enforced at runtime. In particular, PACT programs do not contain information on data repartitioning, data shipping, or grouping. In fact, for several PACT input contracts, there exist different strategies to fulfill the provided guarantees with different implications on the required effort for data reorganization. Choosing the cheapest of those data reorganization strategies is the responsibility of a special cost-based optimizer, contained in the PACT layer. Similar to classic database optimizers, it computes alternative execution plans and eventually chooses the most preferable one. To this end, the optimizer can rely on various information sources, such as samples of the input data, code annotations (possibly generated by the Sopremo layer), information from the cluster's resource manager, or runtime statistics from previous job executions. Details about the optimizer's cost model as well as the actual optimization process are discussed in Sects. 5 and 6.

The introduction of a distinct layer that accepts arbitrarily complex DAGs of second-order functions that wrap arbitrary user code, and the ability to optimize such programs toward different physical execution strategies is one central aspect that differentiates the Stratosphere stack from other systems (e. g., Asterix [12]). The PACT layer separates the parallelization aspects from semantic aspects and provides a convenient intermediate programming model that sits between operators with known semantics and arbitrary parallel tasks.

The output of the PACT compiler is a parallel data flow program for Nephele, Stratosphere's parallel execution engine, and the third layer of the Stratosphere stack. Similar to PACT programs, Nephele data flow programs, also called *Job Graphs*, are also specified as DAGs with the vertices representing the individual tasks and the edges modeling the data flows between those. However, in contrast to PACT programs, Nephele Job Graphs contain a concrete execution strategy, chosen specifically for the given data sources and cluster environment. In particular, this execution strategy includes a suggested degree of parallelism for each vertex of the Job Graph, concrete instructions on data partitioning as well as hints on the co-location of vertices at runtime.

In comparison with Meteor and the PACT programming model, a Nephele Job Graph exhibits the highest level of expressiveness, but at the expense of programming simplicity. Stratosphere users, who choose to directly write their data analytics programs as Nephele Job Graphs, are no longer bound to a set of second-order functions but can freely implement the behavior of each vertex. When compiling the Job Graph from a PACT program, the PACT layer exploits this flexibility and injects additional code for data preparation together with the user's first-order function into a Nephele vertex. This PACT data preparation code is then in charge of reorganizing the incoming data (i. e. sorting, grouping, joining the data) such that it obeys the properties expected by the user's encapsulated first-order function.

Nephele itself executes the received Job Graph on a set of worker nodes. It is responsible for allocating the required hardware resources to run the job from a resource manager, scheduling the job's individual tasks among them, monitoring their execution, managing the data flows between the tasks, and recovering tasks in the event of execution failures. Moreover, Nephele provides a set of memory and I/O services that can be accessed by the user tasks submitted. At the moment, these services are primarily used by the PACT data preparation code mentioned above.

During the execution of a job, Nephele can collect various statistics on the runtime characteristics of each of its tasks, ranging from CPU and memory consumption to information on data distribution. The collected data are centrally stored inside Nephele's master node and can be accessed, for example, by the PACT compiler, to refine the physical optimization of subsequent executions of the same task. Further details on Nephele, especially on its scheduling, communication, and fault-tolerance strategies, are described in Sect. 7.

In order to increase the practical impact of our system, we take special care to make Stratosphere integrate well with existing, popular technologies. In particular, Stratosphere provides support for the popular Hadoop distributed file system and the cloud storage service Amazon S3, as well as for Eucalyptus. We plan to support multi-tenancy by integrating Stratosphere with resource management systems, such as Apache YARN. Moreover, Stratosphere can directly allocate hardware resources from infrastructure-as-a-service clouds, such as Amazon EC2.

3 Stratosphere by Meteor examples

Stratosphere has been designed to cover a wide variety of use cases, including the analysis of structured data (e.g., spreadsheets, relational data), semi-structured data (e.g., HTML websites), and unstructured, textual data. In this section, we present Meteor, one of Stratosphere's top-most programming interfaces by walking through two example programs: a TPC-H data warehousing query and an application that includes operators of two domain-specific packages for information extraction and data cleansing

Meteor organizes domain-specific operators in packages, and treats the former as first-class citizens, allowing users to freely combine existing operators and extend the language and runtime functionality with new operators. A main advantage of this approach is that the operator's semantics can be accessed at compile time and can be potentially used for optimization. To process a variety of different data types, Meteor builds upon a semi-structured data model that extends JSON [47]. The language syntax is inspired by Jaql [13]; however, we simplified many language features in order to provide mechanisms for a seamless integration of new operators and to support n-ary input and output operators.

3.1 Structured data analysis

We introduce Meteor's language features using a modified version of TPC-H Query 15. The Meteor script that implements the query is shown in Listing 1. The script starts with reading the `lineitem` table from a file (line 1). It subsequently selects a three-month time interval (lines 3–5) and computes the total revenue for each supplier by grouping on `suppkey` (lines 7–12). Finally, the script joins the grouped records with the `supplier` table on the attribute `suppkey` (lines 14–16), assembles the output format (lines 17–23), and writes the result to a file (line 25).

Meteor statements can assign the result of an operator invocation to a variable, which either refers to a materialized data set or to a logical intermediate data set, i. e., the result of an operator. Variables start with a dollar sign ($) to easily distinguish data sets from operator definitions. For example, the variable `$li` in line 1 refers to a logical data set, the result of the `read` operator.

Each operator invocation starts with the unique name of the operator (underscored in all listings) and is typically followed by a list of inputs and a set of operator properties (displayed in italics), which are configured with a list of name/-expression pairs. Consider the `filter` operator in lines 3–4: the operator has input `$li` and is configured with a *where*

```
1  $li = read from 'lineitem.json';
2
3  $li = filter $li where
4    ($li.shipdate >= '1996-01-01' and
5     $li.shipdate < '1996-04-01');
6
7  $gli = group $li by $li.suppkey
8    into {
9      supplier_no: $li[0].suppkey,
10     total_revenue: sum($li[*].extPrice *
11                        (1-$li[*].discount)
                          )
12   };
13
14 $s = read from 'supplier.json';
15 $joinResult = join $s, $gli where
16   $s.suppkey == $gli.supplier_no
17   into {
18     $s.suppkey,
19     $s.name,
20     $s.address,
21     $s.phone,
22     $gli.total_revenue
23   };
24
25 write $joinResult to 'result.json';
```

Listing 1 TPC-H Query 15 variant as Meteor script

```
1  script::= (statement ';')*
2  statement::= variable '=' operator
3  operator::= name+ inputs? properties? ';'
4  inputs::= (variable 'in')? variable (',' inputs)?
5  variable::= '$'name
6  properties::= property properties?
7  property::= name+ expression
8  expression::= literal|array|object|...
```

Listing 2 Excerpt of Meteor's EBNF grammar

property, which specifies the selection condition. Property expressions are often only literals but may be as complex as the property expression *into* of the join operator (e. g., lines 15–23), which specifies the schema of the resulting data set.

Listing 2 summarizes the general Meteor syntax in extended Backus-Naur Form.

The relational package of Meteor offers a wide variety of data transformation and matching operators on the JSON data model, such as filter, transform (which allows for arbitrary field modifications), pivot (nesting and unnesting), split (array de-normalization), group, join, union, set intersection, and difference. We refer the reader to Table 1 and reference [37] for details.

3.2 Queries with domain-specific operators

Operators from different Meteor packages can be jointly used to build complex analytical queries. Suppose a research institute wants to find out who of its past, and present affiliated researchers have appeared in recent news articles for a PR campaign. Given a set of employee records from the past five years and a corpus of news articles, we would like to find news articles that mention at least one former or present employee.

```
1  using ie;
2  using cleansing;
3
4  $articles = read from 'news.json';
5  $articles = annotate sentences $articles
6    use algorithm 'morphAdorner';
7  $articles = annotate entities $articles
8    use algorithm 'regex' and type 'person';
9  $peopleInNews = pivot $articles around
10     $person = $article.annotations[*].
           entity
11   into {
12     name: $person,
13     articles: $articles
14   };
15
16 $persons = read from 'person.json';
17 $persons = remove duplicates
18     where average(levenshtein(name),
19       dateSim(birthDay)) > 0.95
20     retain longest(name);
21 $personsInNews = join $refPerson in
       $persons,
22     $newsPerson in $peopleInNews
23   where $refPerson.name == $newsPerson.
         name
24   into {
25     $refPerson.*,
26     articles: $newsPerson.articles[*].url
27   };
28
29 write $personsInNews to 'result.json';
```

Listing 3 Meteor query combining information extraction, data cleansing, and relational operators

Listing 3 displays the corresponding Meteor script. After importing the necessary Sopremo packages (lines 1–2), the news articles corpus is read from a file, and information extraction (IE) operators are applied (lines 4–14) to annotate sentence boundaries and the names of people mentioned in the articles. Subsequently, the data set is restructured with the `pivot` operator to group news articles by person names (lines 9–14). In lines 16–20, the employee records are read and duplicate records are removed using the `remove duplicates` operator, configured with similarity measure, threshold, and a conflict resolution function. The data sets are then joined on the person name (lines 21–27). The `into` clause specifies the output format, which contains all employee attributes and the URLs of news articles mentioning a certain person.

Table 1 Overview of available Sopremo operators

Operator	Meteor keyword	Description
Selection	`filter`	Filters the input by only retaining those elements where the given predicate evaluates to true
Projection	`transform`	Transforms each element of the input according to a given expression
(Un)nesting	`nest`	Flattens or nests incoming records according to a given output schema
	`unnest`	
Join	`join`	Joins two or more input sets into one result-set according to a join condition. A self-join can be realized by specifying the same data source as both inputs. Provides algorithms for anti-, equi-, natural-, left-outer-, right-outer-, full-outer-, semi-, and theta-joins
Grouping	`group`	Groups the elements of one or more inputs on a grouping key into one output, such that the result contains one item per group. Aggregate functions, such as count() or sum(), can be applied
Set/bag union	`union`	Calculates the union of two or more input streams under set or bag semantics
	`union all`	
Set difference	`subtract`	Calculates the set-based difference of two or more input streams
Set intersection	`intersect`	Calculates the set-based intersection of two or more input streams
Pivot	`pivot`	Restructures the data around a pivot element, such that each unique pivot value results in exactly one record retaining all the information of the original records
Replace (All)	`replace`	Replaces atomic values with with a defined replacement expression
	`replace all`	
Sorting	`sort`	Sorts the input stream globally
Splitting	`split`	Splits an array, an object, or a value into multiple tuples and provides a means to emit more than one output record
Unique	`unique`	Turns a bag of values into a set of values
Sentence annotation / splitting	`annotate sentences`	Annotates sentence boundaries in the given input text and optionally splits the text into separate records holding one sentence each
	`split sentences`	
Token annotation	`annotate tokens`	Annotates token boundaries in the given input sentencewise. Requires sentence boundary annotation
Part-of-speech annotation	`annotate pos`	Annotates part-of-speech tags in the given input sentencewise. Requires sentence and token boundary annotations
Parse tree annotation	`annotate structure`	Annotates the syntactic structure of the input sentencewise. Requires sentence boundary annotations
Stopword annotation / removal	`annotate stopwords`	Annotates stopwords occurring in the given input and optionally replaces stopword occurrences with a user-defined string
	`remove stopwords`	
Ngram annotation / splitting	`annotate ngrams`	Annotates token or character ngrams with user-defined length n in the given input. Optionally, the input can be split into ngrams
	`split ngrams`	
Entity annotation / extraction	`annotate entities`	Annotates entities in the given input and optionally extracts recognized entity occurrences. Supports general-purpose entities(e.g., persons, organizations, places, dates), biomedical entities (e.g., genes, drugs, species, diseases), and user-defined regular expressions and dictionaries. Requires sentence and token boundary annotations
	`extract entities`	
Relation annotation / extraction	`annotate relations`	Annotates relations sentencewise in the given input and optionally extracts recognized relationships using co-occurrence- or pattern-based algorithms. Requires sentence, part-of-speech and entity annotations
	`extract relations`	
Merge records	`merge`	Merges existing annotations of records which share the same ID
Data scrubbing	`scrub`	Enforces declaratively specified rules for (nested) attributes and filters invalid records
Entity mapping	`map entities`	Uses a set of schema mappings to restructure multiple data sources into multiple sinks. Usually used to adjust the data model of a new data source to a global data schema
Duplicate detection	`detect duplicates`	Efficiently finds fuzzy duplicates within a data set
Record linkage	`link records`	Efficiently finds fuzzy duplicates across multiple (clean) data sources
Data fusion	`fuse`	Fuses duplicate representations to one consistent entry with declaratively specified rules
Duplicate removal	`remove duplicates`	Performs duplicate detection, subsequent fusion, and retains nonduplicates

Top Base, *Middle* Information Extraction, *Bottom* Data Cleansing

4 Extensibility in Stratosphere's operator model

The previous examples in Sect. 3 have shown how Meteor can be used to perform structured data analysis, extract information from text, and cleanse data. This flexibility stems from the underlying, semantically rich Sopremo operator model. All operators (including relational ones) are organized in packages and dynamically loaded during the parsing process of a Meteor script. Meteor can be seen as a textual interface for Sopremo, and Meteor scripts are translated one-to-one into Sopremo plans. Besides the textual Meteor interface, query plans of Sopremo operators could also be composed with graphical interfaces or other query languages.

Figure 2a depicts the Sopremo plan (which, in general can be a directed acyclic graph) generated from the Meteor script of Listing 3 by the Meteor parser. Each operator invocation in the Meteor script corresponds to a Sopremo operator in the plan. Meteor variables are translated to edges in the plan, signifying the data flow between Sopremo operators. Sopremo operators are configured with the corresponding properties in the Meteor script (we omit those from the figure). In the remainder of this section, we briefly discuss the notions of extensibility and operator composition in Sopremo. A more in-depth discussion of the concepts can be found in reference [37].

To seamlessly integrate domain-specific Sopremo packages, these must satisfy some constraints. Each package and its operators must be self-contained in three ways. First, operators are self-contained in that the Sopremo programmer provides a parallel implementation of new operators in addition to their semantics. An operator can be defined either as a composition of other operators or as an elementary operator with a corresponding PACT program (the directly lower programming layer of Stratosphere) implementation. As Sopremo does not allow recursive compositions, all operators can be reduced to a (possibly large) set of interconnected elementary operators, which are backed by PACT programs.

Second, operators expose their properties through a reflective API. The properties, such as the condition of a join, are transparently managed and validated by the operator itself. Operators may use their properties to choose an appropriate implementation. Thus, no additional knowledge outside of the packages is required to properly configure the operators.

Third, the package developer may optionally provide relevant metadata to aid the optimizer in plan transformation and cost estimation.

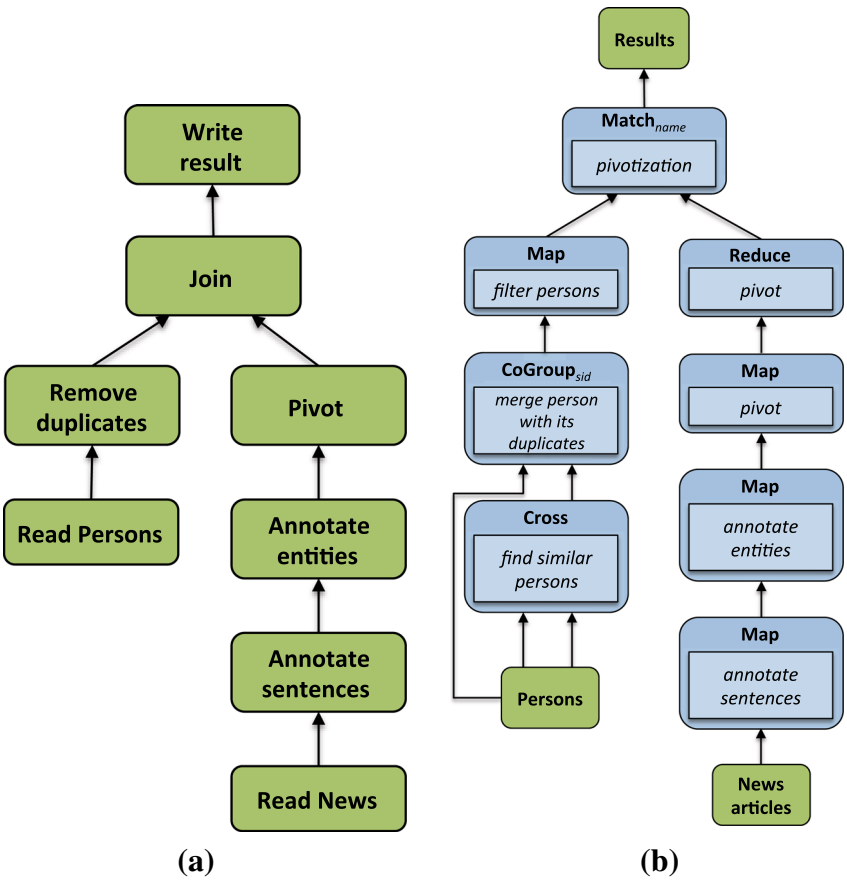

Fig. 2 Sopremo plan and PACT program corresponding to the Meteor query in Listing 3. **a** Sopremo plan **b** PACT program

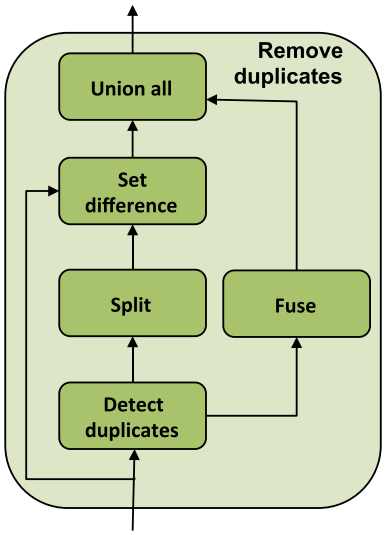

Fig. 3 The "Remove duplicates" operator defined as a composite operator

To facilitate extensibility, we introduced the concept of operator composition in Sopremo. Following the good practices of modularization and information hiding in software engineering, developers can define complex operators using simpler ones. This enables code reuse and allows complex operators to immediately benefit from more efficient re-implementations of simpler operators. Composition can also improve optimization. Transformation rules that cannot be applied to a composite operator might be valid for its building block operators.

Figure 3 shows the implementation of the duplicate removal operator as a composite operator. Here, the duplicate detection is performed by another Sopremo operator. To remove the found duplicate records, we need to fuse the duplicates (right hand side) and merge the result with all nonduplicate records (left hand side). The example demonstrates nested compositions. Although duplicate detection may be naïvely implemented as a theta join, most times it is a complex composition that implements a multi-pass sorted neighborhood or other advanced algorithms.

To illustrate the current analytical capabilities of the Sopremo libraries that are shipped with the system, Table 1 lists selected operators and their functionality.

Sopremo plans are compiled to PACT programs by a program assembler. Figure 2b shows a translated PACT program for the Sopremo plan of Fig. 2a. The PACT programming model is discussed in detail in the next section. The Sopremo to PACT assembler translates each Sopremo operator into one or more PACT operators. Before the PACT program is assembled, composite Sopremo operators are recursively decomposed into their individual operators until only elementary operators remain. These elementary operators can be directly translated into the second-order functions that PACT provides, such as Map and Reduce. Furthermore, the assembler infers from all attribute fields that are referenced in a Meteor script a compact data representation scheme to quickly access these important fields in the tree-structured Sopremo values. The PACT program is assembled by instantiating the PACT implementations of all Sopremo operators and connecting their inputs and outputs. The properties of the Sopremo operators are embedded into the PACT program by adding this information to the configuration of the respective PACT operators.

5 Model for parallel programming

Stratosphere provides an explicit programming model, called the PACT programming model, that abstracts parallelization, hiding the complexity of writing parallel code. This section discusses the data model of the PACT model (Sect. 5.1), the individual operators and the composition of one-pass (acyclic) PACT programs from operators (Sect. 5.2), and finally the composition of iterative (cyclic) PACT programs (Sect. 5.3).

5.1 Data model

PACT operators operate on a flat record data model. A data set, an intermediate result produced by one PACT operator and consumed by another, is an unordered collection of records. A record is an ordered tuple of values, each having a well-defined data type. The semantics and interpretation of the values in a record, including their types, are opaque to the parallel runtime operators; they are manipulated solely by the UDFs that process them.

Certain functions require to form groups of records by attribute equality or by other types of associations. For such operations, a subset of the record's fields is defined as a *key*. The definition of the key must include the types of the values in these fields to allow the runtime operators to access the relevant fields (for sorting and partitioning) from the otherwise schema-free records.

Nested Sopremo JSON objects are converted to records during the compilation of Sopremo plans to PACT programs. JSON nodes that act as keys are translated to individual record fields.

5.2 PACT operators and acyclic PACT programs

A PACT is a *second-order function* that takes as argument a data set and a first-order user-defined function (UDF). A PACT *operator* or simply operator consists of a PACT second-order function and a concrete instantiation of the UDF. PACTs specify how the input data are partitioned into independent subsets called *parallelization units (PUs)*. The

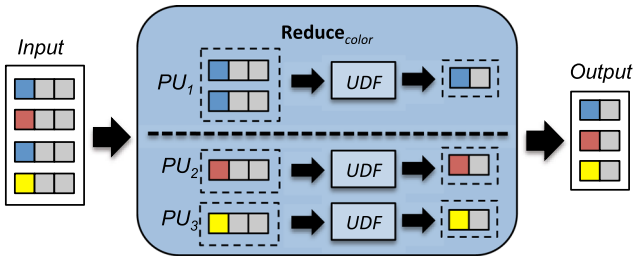

Fig. 4 A PACT operator using a Reduce PACT

actual semantics of data manipulation is encapsulated in the user-defined functions (UDFs). The PACT programming model is declarative enough to abstract away parallelism, but does not directly model semantic information as the Sopremo layer; this is encapsulated within the UDF logic and largely hidden from the system. While this may seem limiting, it enables the specification of a wider variety of data analysis programs (e. g., reduce functions that are not simple aggregates [27]).

Figure 4 shows the structure of a PACT operator that uses the Reduce function as its PACT. The input data set is logically grouped using the key attribute ("color" in the figure, which corresponds to the first attribute). Each group of records with a certain key value forms a parallelization unit. The UDF is applied to each PU independently. By specifying the operator using the Reduce PACT, the programmer makes a "pact" with the system; all records that belong to the same PU will be processed together by a single invocation of the UDF. The logical division of a data set into PUs can be satisfied by several physical data-partitioning schemes. For example, in Fig. 4, PUs can be physically partitioned into two nodes as indicated by the thick horizontal dotted line: PU_1 resides in node 1, and PU_2 and PU_3 reside together in node 2. The logical output of the PACT operator is the concatenation of the outputs of all UDF invocations. In the example, the UDF collapses records into a single record per group (e. g., computing an aggregate) and returns the key value together with the computed aggregate.

Currently, five second-order functions (shown in Fig. 5) are implemented in the system. In addition, we have developed two higher-order functions used for iterative processing (we discuss those later). The *Map* creates a PU from every record in the input. The *Reduce* function forms a PU with all records that have the same value for a user-defined key attribute.[2]

The *Match*, *Cross*, and *CoGroup* PACTs operate on two input data sets. The parallelization units of the Match function are all pairs of records that have the same key attribute value.

[2] We follow the definitions from the original MapReduce paper [22] but exclude execution-specific assumptions (such as the presence of sorted reduce inputs).

```
1  class DupElim extends CoGroupStub {
2    void cogroup (
3      Iterator<PactRecord> persons,
4      Iterator<PactRecord> duplicates,
5      Collector<PactRecord> output) {
6      if (!duplicates.hasNext())
7        // No duplicates
8        out.collect(persons.next());
9      else {
10       PactRecord cleanPerson =
11         merge (persons.next(),duplicates);
12       out.collect (cleanPerson);
13     }
14   }
15 }
16 ...
17 int SID = 0;
18 CoGroupContract =
19   CoGroupContract.build (DupElim.class,
   PactLong.class, SID,  SID);
```

Listing 4 An example of UDF code for a CoGroup operator

Match therefore performs an inner equi-join and applies the UDF to each resulting record pair. The Cross function dictates that every record of the first input together with every record of the second input forms a PU, performing a Cartesian product. CoGroup generalizes Reduce to two dimensions; each PU contains the records of both input data sets with a given key. The source of records (left or right input) is available to the UDF programmer. Compared with Match (*record-at-a-time* join), CoGroup is a *set-at-a-time* join. As such, CoGroup subsumes Match with respect to expressiveness, but it has stricter conditions on how the PUs are formed and hence fewer degrees of freedom for parallelization. For a formal definition of the five PACTs, we refer the reader to reference [43].

Listing 4 shows a possible PACT implementation code of the "Duplicate Removal" operator from Sect. 3.2 (see also Table 1). The Java class inherited from (`CoGroupStub`) indicates the type of PACT (CoGroup). User code is encapsulated in the `cogroup` method. The inputs (persons and possible duplicates) are grouped on the first field, `personId`. This is specified in the code that instantiates the operator (lines 17–19). The UDF is called for each person together with its (zero or more) duplicates. If duplicates are found, they are merged to form a cleaned version of the person record.

We defer the discussion of programs that make repeated passes over the input until the next section. For now, a PACT program is a directed acyclic graph with PACT operators, data sources, and data sinks as nodes. Operators with multiple successors forward the same data to each successor and thus behave similar as, for example, common subexpressions in SQL. Figure 2b shows the composite Meteor/Sopremo example of Sect. 3.2 transformed to a PACT program. The program has two data sources, Persons and News. For exam-

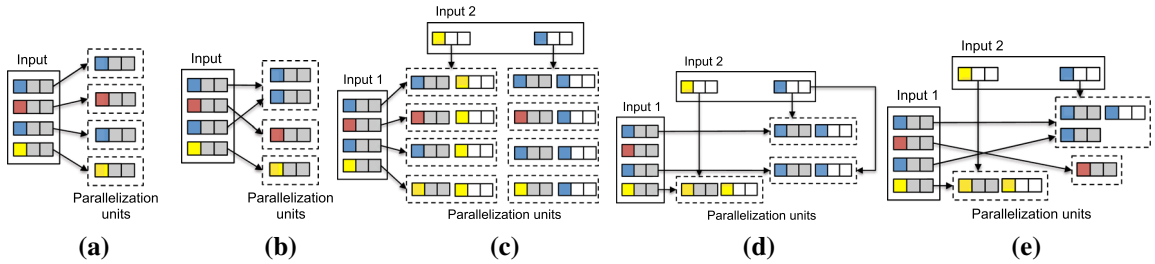

Fig. 5 The five second-order functions (PACTs) currently implemented in Stratosphere. The parallelization units implied by the PACTs are enclosed in *dotted boxes*. **a** Map **b** Reduce **c** Cross **d** Match **e** CoGroup

ple, duplicate removal is implemented as CoGroup over the person's input and the output of the preceding Cross. The UDF is invoked on a list containing exactly one person and a list of possible duplicates. If the duplicate list is not empty, it is merged into one "clean" person record with the person list. The program has one data sink, Results, which writes every record delivered from the preceding Match operator into the underlying file system. PACT programs can be generated by the Sopremo compiler or specified by hand.

5.3 Iterative PACT programs

Many data analysis tasks cannot be implemented as algorithms that make a single pass over the data. Rather, they are of iterative nature, repeating a certain computation to refine their solution until a convergence criterion is reached. The PACT programming model supports the expression of such programs through higher-order fixpoint operators [26].

To achieve good performance without exposing explicit mutable state, PACT offers two different declarative fixpoint operators: one for *Bulk-* and one for *Incremental Iterations*. Both are defined by means of a *step function* that is evaluated repeatedly over a data set called the partial- or intermediate solution (see Fig. 6). The step function is an acyclic PACT program. One parallel application of the step function to all partitions of the partial solution is called a superstep [65].

Bulk Iterations execute the PACT program that serves as the step function in each superstep, consuming the entire partial solution (the result of the previous superstep or the initial data set) and recompute the next version of the partial solution, which will be consumed at the next iteration. The iteration stops when a user-defined termination criterion is satisfied.

In *Incremental Iterations*, the user is asked to split the representation of the partial solution into two data sets: a *solution set* (S in Fig. 5b) and a *workset* (W in Fig. 5b). At each superstep, an incremental iteration consumes only the working set and selectively modifies elements of the solution set, hence incrementally evolving the partial solution rather than fully recomputing it. Specifically, using S and W, the step function computes the next workset and a *delta set* (D in Fig. 5b), which contains the items to be updated in the solution set. The new working set holds the data that drive the next superstep, while the solution set holds the actual state of the partial solution. Elements of the solution set (termed "cold") that are not contained in D need not be updated. To facilitate the efficient merge between the current solution set and the delta set, each element in the solution set must be uniquely addressable by a key.

When applicable, incremental iterations typically lead to more efficient algorithms, because not every element in the intermediate solution needs to be examined in each superstep. The *sparse computational dependencies* present in many problems, and data sets allow a superstep to focus on the "hot" parts of the intermediate solution and leave the "cold" parts untouched. Frequently, the majority of the intermediate solution cools down comparatively fast, and the later supersteps operate only on a small subset of the data. Note that the intermediate solution is implicitly forwarded to the next superstep, not requiring the algorithm to recreate it.

Programmers implement iterative algorithms by defining the step function as a regular acyclic PACT program that uses the partial solution as a data source and next partial solution as a sink. This step function is then embedded into a fixpoint operator that takes an initial partial solution and invokes the step function repeatedly on the next version of the intermediate solution until a certain termination condition is reached. While bulk iterations require an explicit termination condition (either a convergence criterion or a fixed number of supersteps), incremental iterations terminate when they produce an empty working set.

Figure 6 shows a step function for the bulk and an incremental version of a graph algorithm. This generic example algorithm associates an ID with each vertex and propagates the ID from each vertex to its neighbors, where each neighbor adopts the ID if it is smaller than its own current ID. This algorithm eventually distributes IDs according to connected components, but is in a slightly modified version applicable to many other graph problems, such as shortest paths and maximum flow. In the bulk version, the step function joins the vertex state with the edges to create candidates for each vertex's neighbors (Match) and then selects the minimum

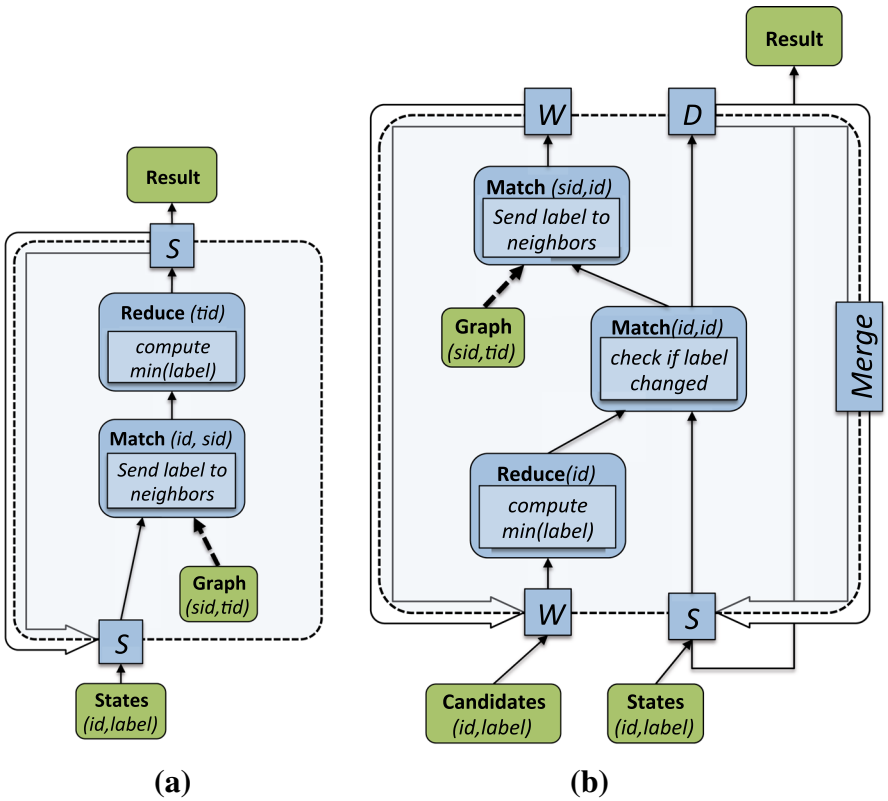

Fig. 6 An algorithm that finds the connected components of a graph as a bulk iteration and an incremental Stratosphere iteration. **a** Bulk iteration **b** Incremental iteration

ID from the candidates for each vertex (Reduce). The incremental version holds the candidate IDs in the workset and the vertex state as the solution set. In addition to the aforementioned operations, it joins the minimal candidate with the solution set and checks whether the selected ID is actually new. Only in that case, it returns a new value for the vertex, which goes into the delta set and into the Match that creates the workset for the next superstep. By selectively returning or not returning values from the join between the workset and solution set, the algorithm realizes the dynamic computation that excludes unchanged parts of the model from participating in the next superstep.

We refer the reader to reference [26] for a complete treatment of iterations in Stratosphere. At the time of this writing, the iteration's feature is in an experimental stage and has not been integrated with Sopremo and Meteor. Iterative programs are compiled to regular DAG-shaped Nephele Job Graphs that send upstream messages to coordinate superstep execution.

6 Optimization in Stratosphere

This section discusses Stratosphere's optimizer. The optimizer compiles PACT programs into Nephele Job Graphs. Sopremo plans are translated into PACT programs prior to optimization as discussed in Sect. 4. The overall architecture of the optimizer is presented in Sect. 6.1. Sections 6.2 and 6.3 discuss the reordering of PACT operators and the generation of physical plans.

6.1 Optimizer overview

The Stratosphere optimizer builds on technology from parallel database systems, such as logical plan equivalence, cost models, and interesting property reasoning. However, there are also aspects that clearly distinguish it from prior work.

Many of the distinguishing features of Stratosphere's optimizer compared with conventional query optimizers originate from differences in program specification. Most relational database systems provide a declarative SQL interface. Queries specified in SQL are translated into expression trees of relational algebra. These expression trees are rewritten using transformation rules, which are based on commutativity and associativity properties of relational operators and finally compiled into physical execution plans.

In contrast to a relational query, PACT programs are directed acyclic graphs (DAGs) of PACT operators. Since DAGs are more general than trees, traditional plan enumeration techniques need to be adapted. Operators differ as well; while relational operators have fully specified semantics, PACT operators are parallelizable second-order functions

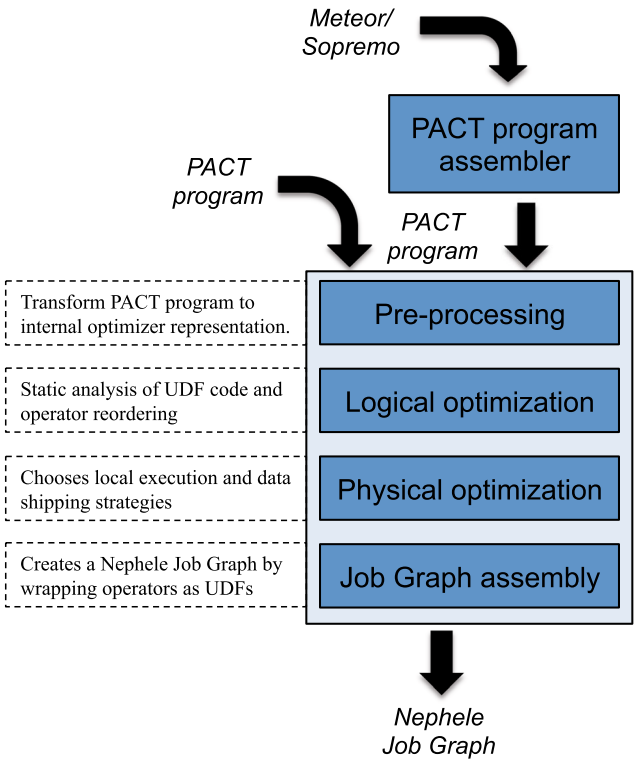

Fig. 7 The different program transformation phases of the Stratosphere optimizer

that encapsulate user-defined first-order functions. Due to the presence of arbitrary user code, the semantics of a PACT operator are not, in general, known by the optimizer. Therefore, plan rewriting rules as known from relational optimizers do not apply in the context of PACT programs. In addition, the lack of semantics hinders the computation of reliable size estimates for intermediate results, which are important for cost-based optimization. Finally, relational optimizers can leverage their knowledge of data schema. In contrast, PACT's data model is based on records of arbitrary types in order to support a wide variety of use cases. Data is only interpreted by user code and hence opaque to the optimizer.

Figure 7 shows the architecture stages of the Stratosphere optimizer, and Fig. 8 shows the different representations of a program as it passes through the different optimization stages. The optimizer compiles PACT programs into Nephele Job Graphs. Data processing tasks specified as Sopremo plans are translated into PACT programs prior to optimization and compilation. This process was described in Sect. 4. The optimizer itself consists of four phases. Similar to many relational optimizers, the optimization process is separated into a logical rewriting and a physical optimization phase. The separation between logical and physical optimization is a result of the bottom-up historical evolution of the Stratosphere system (the PACT layer and physical optimization predate the

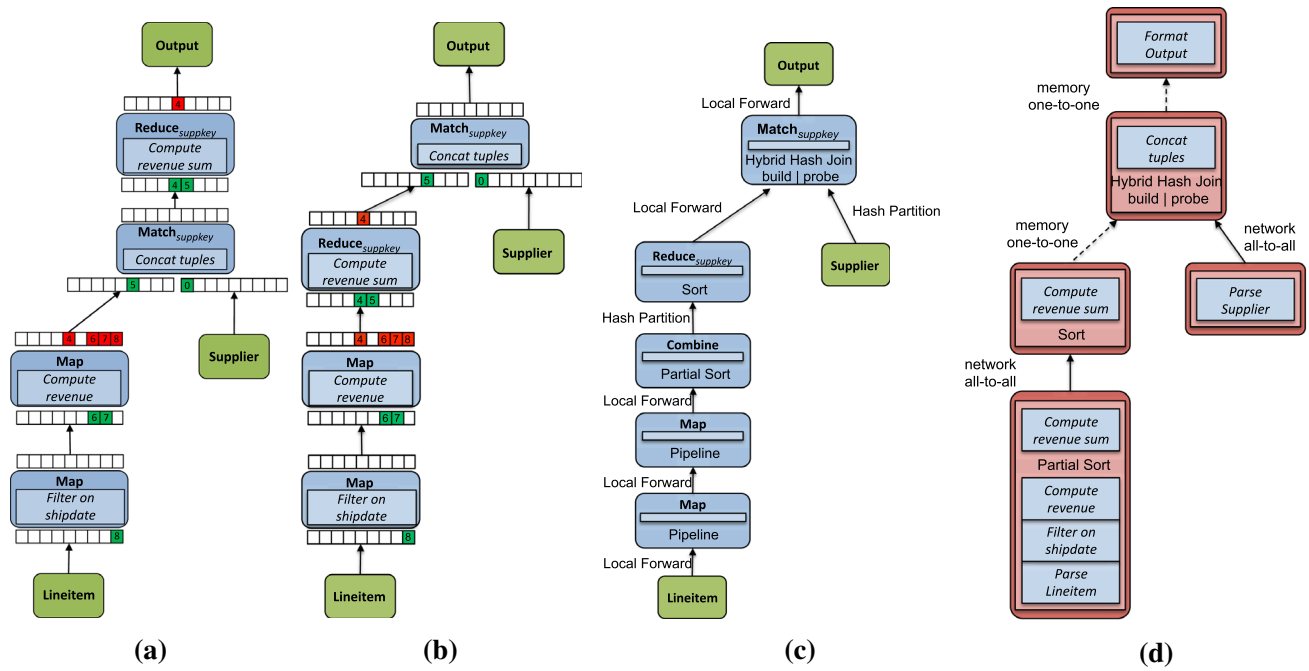

Fig. 8 Plan transformation through the different phases of the optimizer. A submitted PACT program is modified by logical optimization and produces an equivalent program after operator reordering. Read and write sets are shown as records with *green* and *red* fields. Then, a physical plan with annotated local execution and data shipping strategies is obtained, and finally a Nephele Job Graph is emitted by the optimizer. **a** Original PACT program **b** Modified PACT program **c** Physical plan **d** Nephele Job Graph (color figure online)

logical optimization and the Sopremo layer); we are currently designing an optimizer that unifies the optimization of Sopremo and PACT operators and chooses the order of operators and their physical execution strategies in a single pass. See Sect. 9 for details.

Prior to optimization, the optimizer transforms a PACT program into an internal representation. This representation is a DAG consisting of operator nodes that represent data sources, data sinks, PACT operators, and internal operations, such as "Combine" (if applicable for a Reduce) or "Temp" (materialization) operators. Internal operators do not change the semantics of a data flow; however, they can improve its execution, and they are sometimes required to prevent deadlocks (see Sect. 6.3 for details).

In the next phase, the optimizer generates semantically equivalent plans by reordering operators. Rewriting rules as known from relational optimizers do not directly apply in our context as the optimizer is not aware of the semantics of the UDF operators. Section 6.2 presents the operator reordering techniques of the optimizer, which are based on the detection of attribute access conflicts, static code analysis, and switching of consecutive operators.

Physical optimization comes after operator reordering. The second-order function of an operator defines its logical parallelization. For a given second-order function, there can be multiple physical data shipping strategies (such as hash- or range partitioning or broadcasting), that provide the parallelization requirements, as well as several local physical execution strategies (such as sort- or hash-based techniques). Similar to database systems, interesting properties [60] can be leveraged. The optimizer generates a physical execution plan bottom-up by choosing execution strategies and considering interesting properties. The optimization process is explained in detail in Sect. 6.3. Finally, the resulting execution plan is translated into a Nephele Job Graph and submitted for execution.

6.2 Operator reordering

In the first optimization phase, the Stratosphere optimizer reorders operators similarly to the logical optimization phase in relational optimizers [43]. However, as mentioned in Sect. 6.1, traditional transformation rules cannot be directly applied due to the unknown semantics of the UDFs inside of the PACT operators. Instead, we defined and proved two sufficient conditions to reorder two successive PACT operators without changing the program semantics. These conditions are based on the detection of conflicting attribute accesses and the preservation of group cardinalities. We use the notion of read and write field sets to hold the information of all fields that an UDF reads and writes. Thereby, a write access to a record may add or remove an attribute or modify the value of an existing attribute.

The first reordering condition compares the read and write sets of two successive PACT operators and checks for overlapping access patterns. In order to evaluate to true, only the read sets of the operators may intersect. Otherwise, the operators have conflicting read-write or write-write accesses. This reasoning is similar to conflict detection in optimistic concurrency control methods [50] and compiler techniques to optimize loops. The second condition only applies for group-based PACT operators, such as Reduce and CoGroup. Since the semantics of a grouping operator might depend on the size of its input group, this condition ensures that input groups are preserved if the operators are reordered. We showed that our conditions are applicable for all combinations of the set of currently supported second-order functions (see reference [43]). Figures 8a, b depict two semantically equivalent PACT programs for the TPC-H example query discussed in Sect. 3.1. Assuming that the left plan was given as input to the optimizer, our reordering techniques allow to switch the Reduce and Match operators, which yields the plan on the right hand side. The Match and Reduce operators perform the join and grouped aggregation, respectively. This transformation is possible because both operators have nonconflicting read and write sets, which are indicated in the figures by green and red-colored record fields above and below each operator. Match also fulfills the group-preservation condition. Since it is a primary-key foreign-key join on the grouping attribute, Match does not change the cardinality of the individual reduce groups. Therefore, this transformation is an invariant group transformation as known from relational query optimization [20].

In order to evaluate the conditions, the optimizer requires read and write sets and bounds on the output cardinality $(0, 1, n)$ of the operator UDFs. We employ static code analysis (SCA) techniques to automatically derive this information [41]. Our approach leverages our knowledge of and control over the API that the user uses to access record fields. Hence, we can safely identify all record accesses of an UDF by tracking the corresponding API calls, e.g., `r.getField(1)` to read field 1 from record r or `r.setField(2,v)` to write value v to r's field 2. The extraction algorithm uses control flow, Def-Use, and Use-Def data structures obtained from an SCA framework to trace the effects of record accesses through the UDF. Our approach guarantees correctness through conservatism. Field accesses can always be added to the corresponding read or write sets without loss of correctness (but with loss of optimization potential). Supersets of the actual read and write sets might only imply additional access conflicts; therefore, the optimizer might miss valid optimization choices but will never produce semantically incorrect transformations.

Based on the conditions to identify semantic-preserving operator reorderings, we designed a novel algorithm to enumerate all valid transformations of the input PACT pro-

gram [43]. In contrast to relational optimizers, where plans are built by subsequently adding operators, our algorithm is based on recursive top-down descent and binary switches of successive operators. We enumerate operator orders only on programs where the data flow resembles a tree. For data flows that are DAGs but not trees, i. e., where some operators have multiple successors, the problem becomes similar to that of relational optimization with common subexpressions. As a simple solution, we split the data flow after each such operator, thereby decomposing it to a set of trees. These trees are then individually optimized and afterward recomposed to a DAG. A limitation of this method is that operators can never be moved across operators with multiple successors.

Given the reordering conditions and the plan enumeration algorithm, the optimizer can emulate many transformations that are known from the relational domain such as selection and projection push down, join-order enumeration, and invariant grouping transformations. However, also nonrelational operators are included into the optimization process. A full adaption of the reordering algorithm to DAGs is part of our future work.

6.3 Physical optimization

After the logical rewriting component has picked an equivalent PACT program, the latter is further optimized to produce a physical execution plan with concrete implementations of data shipping and local operator execution strategies using estimated execution costs [8].

Stratosphere's runtime supports several execution strategies known from parallel database systems. Among these strategies are repartition and broadcast data transfer strategies and local execution strategies, such as sort-based grouping and multiple join algorithms. In addition to execution strategies, the optimizer uses the concept of interesting properties [60]. Given a PACT operator, the optimizer keeps track of all physical data properties, such as sorting, grouping, and partitioning, that can improve the operator's execution [73]. Figure 8c shows a possible physical execution plan for the PACT program of Fig. 8b. Here, the Match operator benefits from the data of the left input being partitioned on `lineitem.suppkey` due to the preceding Reduce operator. This property is leveraged by locally forwarding the data between Reduce and Match and hence avoiding data transfer over the network. However, in contrast to the relational setting, it is not obvious whether an operator's UDF preserves or destroys physical data properties, i. e., a physical property can be destroyed by an UDF that modifies the corresponding record field. The optimizer uses the notion of constant record fields, which can be derived from an UDF's write set (see Sect. 6.2) to reason about interesting property preservation. In this plan, attribute `lineitem.suppkey` is held in record field 5. Since the Reduce operator in our example does not modify field 5 as indicated in Fig. 8b, it preserves the partitioning on that field.

The optimizer uses a cost-based approach to choose the best plan from multiple semantically equivalent plan alternatives. The cost model is based on estimated network I/O and disk I/O as these are the factors that dominate most jobs in large clusters (we are currently using CPU cost for some optimization decision in a prototypical stage). Therefore, the size of intermediate results must be estimated. While this is a challenging task in relational database systems, it is even more difficult in the context of the PACT programming model due to its focus on UDFs. In the current state, the optimizer follows a pragmatic approach and relies on the specification of hints, such as UDF selectivity. These hints can be either set manually by a user, derived from upwards layers, or in the future be retrieved from a planned metadata collection component (see Sect. 9).

The algorithm to enumerate physical plans is based on a depth-first graph traversal starting at the sink nodes of the program. While descending toward the sources, the optimizer tracks interesting properties. These properties originate from the specified keys of the PACT operators and are traced as long as they are preserved by the UDFs. When the enumeration algorithm reaches a data source, it starts generating physical plan alternatives on its way back toward the sinks. For each subflow, it remembers the cheapest plan and all plan alternatives that provide interesting properties. Finally, the best plan is found, after the algorithm reached the data sinks.

To correctly enumerate plans for arbitrary DAG data flows, we analyze the program DAGs to identify where the data flow "branches" (i. e., operators with multiple outgoing edges) and which binary operators "join" these branches back together. For these joining operators, the subplans rooted at the branching operator are treated like common subexpressions and the plan candidates for that operator's inputs must have the same subplan for the common subexpression. Furthermore, data flows may deadlock if some, but not all, paths between a branching operator and the respective joining operators are fully pipelined. When we find such a situation, we place artificial pipeline breakers on the pipelined paths. This is done as part of the candidate plan enumeration and included in the cost assessment of subplans.

The compilation of iterative PACT programs features some specific optimization techniques [26]. Prior to physical plan enumeration, the optimizer classifies the edges of the data flow program as part of the dynamic or constant data path. In Fig. 6, constant data paths are indicated by thick dotted arrows (they consist of the scan operator of the graph structure in both plans). The dynamic data path comprises all edges that transfer varying data in each iteration. Edges that transfer the same data in each iteration belong to the constant data path. During plan enumeration, the costs of all operators

on the dynamic data path are weighted with an user-specified number of iterations (if this is unknown, we have found that a magic number equal to a few iterations is sufficient in most cases). Therefore, the optimizer favors plans that perform much work within the constant data path. Subsequent to plan enumeration, the optimizer places a "Cache" operator at the intersection of constant and the dynamic data path. When executing the program, the data of the constant path are read from this Cache operator after it has been created during the first iteration. Note that the cached result may be stored in a variety of forms depending on the requirements of the dynamic data path, e. g., in sorted order or in a hash table data structure.

Further optimizations apply for incremental iterative programs where the solution set is updated with the delta set after each iteration. The elements of the solution set are identified by a key, and the set itself is stored partitioned and indexed by that key. An update of the solution set is realized by an equi- or outer-join with the delta set on the key. For this join, the partitioned index of the solution set can be exploited. Furthermore, the optimizer can decide that the delta set is not materialized, and instead, the solution set is immediately updated. However, this option only applies if it is guaranteed that elements from the solution set are accessed only once per iteration and that only local index partitions are updated.

7 Parallel dataflow execution

After a program has been submitted to Stratosphere (either in the form of a Meteor query, a Sopremo plan, a PACT program, or a Nephele Job Graph), and after it has passed all the necessary compilation and transformation steps, it is submitted for execution to *Nephele*,[3] Stratosphere's distributed execution engine.

The Nephele execution engine implements a classic master/worker pattern (Fig. 9). The master (called *Job Manager*) coordinates the execution while the workers (called *Task Managers*) execute the tasks and exchange intermediate results among themselves. The Job Manager pushes work to the Task Managers and receives a number of control messages from them, such as task status changes, execution profiling data, and heartbeats for failure detection. To reduce latency, messages for tasks are bundled and pushed eagerly, rather than in periodic intervals.

The execution of a program starts with the Nephele Job Manager receiving the program's *Job Graph*. The Job Graph is a compact description of the executable parallel data flow. Each vertex represents a unit of sequential code, which is one or more pipelined data flow operators and/or UDFs. The

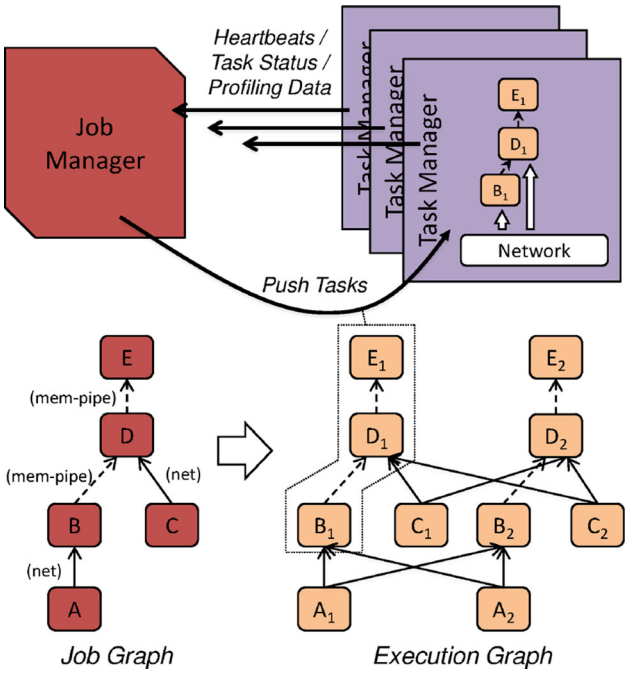

Fig. 9 Nephele's process model and the transformation of a Job Graph into an Execution Graph

channels represent the passing of records between the operators and describe the pattern by which the parallel instances of a pair of vertices connect to each other. Example of these patterns is *all-to-all*, which is used to re-partition data, or *pointwise*, for simple forward passing in a local pipeline. In order to track the status of the parallel vertex and channel instances individually, the Job Manager spans the Job Graph to the *Execution Graph*, as shown in Fig. 9. The Execution Graph contains a node for each parallel instance of a vertex, which we refer to as a task.

7.1 Tasks, channels, and scheduling

Tasks go though a life cycle of *scheduling*, *deploying*, *running*, and *finished*. Initially, all tasks are in the scheduling phase. When a task becomes active (deploying), the Job Manager selects a suitable computing instance (effectively a share of resources) to deploy the task upon. This instance may be requested from a cloud service, or a resource manager of a local cluster. Having found a suitable instance, the Job Manager pushes to that instance a deployment description of the task including optionally required libraries, if those are not yet cached on the target instance. To reduce the number of deployment messages, a task is deployed together with all tasks that it communicates with through local pipelines. When deployed, each task spawns a thread for its code, consuming its input streams and producing output streams.

The Job Manager deploys initially all input tasks with their local pipelines. All other tasks are deployed *lazily*; when a task first tries to send data to another task via network, it will

[3] Nephele was a cloud nymph in ancient Greek mythology. The name comes from Greek "$\nu\epsilon\phi o\varsigma$," meaning "cloud." The name tips a hat to Dryad [44] (a tree nymph) that influenced Nephele's design.

request the target address of that task from the Job Manager. If that target task is still in the scheduling phase, the Job Manager will commence its deployment. The channels through which tasks exchange data are typically pipelined through main memory or TCP streams, to reduce latency and I/O load. They may be materialized as files, if a pipeline breaker is explicitly required. Besides transferring data buffer from the source to the target task, channels may also transfer custom *events*. An event is a metadata message that can be sent both in the same direction as the data flows (in order with the records) or in the opposite direction. Internally, Stratosphere uses those events among other things for channel startup and teardown, and to signal superstep transitions for iterative algorithms [26].

In practice, the source and target tasks of several different network channels may be co-located on the same Task Manager. This is due to intra-node parallelism or different tasks co-partitioning their data. Instead of naïvely creating one TCP connection for each network channel, Stratosphere multiplexes network channels through TCP connections, such that every Task Manager has at most one physical TCP connection to each other Task Manager.

Because network bandwidth is often a scarce resource; reducing the number of transferred bytes is frequently a suitable means of increasing performance. Compressing the buffers with a general-purpose compression algorithm (zip, lzo, etc) before shipping them trades extra CPU cycles for network bandwidth savings. Different compression algorithms have different tradeoffs, where more CPU intensive compression algorithms typically yield higher compression rates and thus save more network bandwidth. The best compression algorithm for a certain program depends on the type of data shipped (e. g., text or media) and other program characteristics, for example, on how many spare CPU cycles, it has before being CPU bound. Nephele supports various compression algorithms plus a mechanism that dynamically adjusts the compression algorithm to find the algorithm best suitable for a certain channel [40].

7.2 Fault tolerance

Fault-tolerance techniques allow systems to recover the program execution in the presence of failures. Details about the fault-tolerance mechanisms used by Stratosphere are described in a previous publication [39]; this section gives a brief overview of these techniques.

Stratosphere's fault tolerance is predicated on log-based rollback recovery; The system materializes intermediate task results and, in the presence of a failure, resets the affected tasks and replays their input data from the materialization points. If the input to a task has not been materialized, the predecessor tasks are re-executed as well, tracking back through the data flow graph to the latest materialized result (possibly the original input). The propagation of restarts is similar to rollback propagation in uncoordinated checkpoint environments [24,66] and has been adopted, in one form or another, by many data flow engines that execute analytical queries: Hadoop [5] (where the data flow is the simple fixed Map-Shuffle-Reduce pipeline), Dryad [44], or Spark [70].

The aforementioned systems all implement the *blocking operator model*, i. e., each operator produces its complete result before any downstream operator may start consuming the result. While this model often increases the execution latency, it simplifies the fault tolerance mechanism, as it ensures that a task consumes only intermediate results that are completely available. It prevents situations where a downstream task consumed a portion of its predecessor's output, but the remainder became unavailable due to a failure. In contrast, Stratosphere supports pipelined tasks and materializes checkpoints "to the side" without breaking the pipeline. During task execution, the system copies the buffers that contain the task's result data into a memory cache. If the cache is full, the buffers are gradually moved to disk. Once the last result of a task was produced, a checkpoint is fully materialized like in the blocking model. However, in Stratosphere, subsequent tasks do not have to wait until the checkpoint was written to disk to process data. In case of a task failure, any downstream tasks that have already consumed parts of this task's result data are restarted as well. If the failed task is known to produce deterministic results, we keep the downstream tasks running and they deduplicate incoming buffers using sequence numbers (similar to package deduplication in TCP). If writing a checkpoint to disk fails but all result data have been forwarded, the system discards the checkpoint and continues processing. In case of a task failure, the system has to recover from an earlier checkpoint or the original input data.

Since Stratosphere's runtime is generally pipelined, the system can decide which task results to materialize and which to stream. Some task results may be very large and force the checkpoint to disk. In many cases, these are not worth materializing, because reading them from disk is not significantly faster then recomputing them from a smaller previous checkpoint. The current prototype of our fault tolerance implementation employs a heuristic, we call *ephemeral checkpointing,* to decide at runtime whether to create a checkpoint or not; We start materializing a task's output by keeping the result buffers. When running low on memory resources, we discard materialization points where certain conditions are met; (1) The task's produced data volume is large, as determined by the ratio of consumed input versus produced output buffers up to that point, and (2) the average processing time per buffer is below a certain threshold, indicating a fast task that can efficiently recompute its output (as opposed to a CPU-intensive UDF, for example). A preselection of interesting checkpointing positions can be done at optimization time

based on estimated result sizes. We present evaluation results of our ephemeral checkpoint technique in Sect. 8.7.

7.3 Runtime operators

Next to UDF drivers, Stratosphere's runtime provides operators for external sorting, hybrid hash join, merge join, (block) nested loops join, grouping, co-grouping, as well as shipping strategies for hash partitioning, balanced range partitioning, and broadcasting. In principle, their implementation follows descriptions in the database literature (e. g., [30,32]). We modified the algorithms slightly to be suitable for a language without explicit memory control, as explained in the next paragraphs.

Stratosphere is implemented in Java. For a framework that is designed to execute to a large extent user-defined functions, Java is a good match,[4] as it offers an easier programming abstraction than, for example, C or C++. At the same time, Java still allows a good level of control about the execution process and offers good performance, if used well. The implementation follows the requirements to implement operators for data intensive processing in Java.

One central aspect is the handling of memory, because Java, in its core, does not give a programmer explicit control over memory via pointers. Instead, data items are typically represented as objects, to which references are passed around. A typical 64bit JVM implementation adds to each object a header of 2 pointers (of which one is compressed, 12 bytes total) and pads the object to have a size, which is a multiple of 8 bytes [46]. Consider the example of a tuple containing 4 fields (integers or floating point numbers with 4 bytes each), having a net memory requirement of 16 bytes. A generic object-oriented representation of that tuple would consequently require up to 64 bytes for the 4 objects representing the fields, plus 32 bytes for an array object holding the pointers to those objects. The memory overhead is hence more than 80 %. A single custom code-generated object for the record still consumes 32 bytes—an overhead of 50 %.

An additional downside of the classic object-oriented approach is the overhead of the automatic object deallocation through the garbage collector. Java's garbage collection works well for a massive creation and destruction of objects, if its memory pools are large with respect to the number of objects that are alive at a certain point in time. That way, fewer garbage collections clear large amounts of objects in bulk. However, all memory dedicated to the efficiency of the garbage collector is not available to the system for sorting, hash tables, caching of intermediate results, or other forms of buffering.

To overcome these problems, we designed the runtime to work on serialized data in large byte arrays, rather than on objects. The working memory for the runtime operators is a collection of byte arrays resembling memory pages (typically of size 32KiBytes). Each record is a sequence of bytes, potentially spanning multiple memory pages. Records are referenced via byte offsets, which are used internally in a similar way as memory pointers. Whenever records need to be moved, for example, from a sort buffer into the buffers of a partitioner, the move operation corresponds to a simple byte copy operation. When certain fields of the record need to be accessed, such as in a UDF, the fields are lazily deserialized into objects. The runtime caches and reuses those objects as far as possible to reduce pressure on the garbage collector.

Toward the runtime operators, the records are described through serializers (record layout, length, copying) and comparators (comparisons, hashing). For sorting and hashing operators, every comparison still incurs an invocation of a virtual (non inline-able) function on the comparator, potentially interpreting a record header in the case of variable length data types. To reduce that overhead, we generate normalized keys and cache hash codes, as described by Graefe et al. [31]. This technique allows the operators to work to a large extent with byte-wise comparisons agnostic to specific record layouts.

An additional advantage of working with serialized data and a paged memory layout is that for many algorithms, the pages containing the records can be directly written to disk in case of memory pressure, yielding implementations that destage efficiently to secondary storage. The result of these implementation techniques is a memory efficient and robust behavior of the runtime operators, which is essential for data intensive applications.

One can naïvely map the DAG of operators from the execution plan (cf. Sect. 6) to a Job Graph by making each operator its own vertex. However, recall that this way, each operator runs its own thread and the vertices communicate with each other using the stream model. If matching the number of threads per instance to the number of CPU cores, this easily leads to under-utilization of the CPU, if the operators' work is not balanced. In case of having multiple operators per core, it incurs unnecessary context switches and synchronization at thread-to-thread handover points. For that reason, we put multiple operators into one vertex, if they form a local pipeline. An example would be data source with a Map UDF, a sort operator, and a preaggregation. We use a combination of *push-* and *pull-chaining*. Pull-chaining corresponds to nesting iterators, and is typically referred to as the "Volcano Execution Model" [33]. However, certain UDFs produce multiple records per invocation, such as unnesting operations. To keep the programming abstraction simple, we do not force the programmer to write such UDFs in the form of

[4] When referring to Java, we refer also to other languages built on top of Java and the JVM, for example, *Scala* or *Groovy*.

an iterator, as that usually results in more complex code[5]. In such cases, we chain the successor tasks using an abstraction of *collectors*, which implement an *accept()* function. This function is the symmetric *push* counterpart to the iterators' pull function (typically called *next()*).

8 Experimental evaluation

We experimentally evaluate the current version of Stratosphere against other open-source systems for large-scale data processing. To that purpose, we conducted a series of experiments comparing Stratosphere against version 1.0.4 of the vanilla MapReduce engine that ships with Apache Hadoop[5], version 0.10.0 of Apache Hive [6]—a declarative language and relational algebra runtime running on top of Hadoop's MapReduce, as well as version 0.2 of Apache Giraph [4]—an open-source implementation of Pregel's vertex-centric graph computation model [54] that uses a Hadoop map-only job for distributed scheduling. This section presents the obtained experimental results and highlights key aspects of the observed system behavior.

8.1 Experimental setup

We ran our experiments on a cluster of 26 machines, using a dedicated master and 25 slaves connected through a Cisco 2960S switch. Each slave node was equipped with two AMD Opteron 6128 CPUs (a total of 16 cores running at 2.0 GHz), 32 GB of RAM, and an Intel 82576 gigabit Ethernet adapter. All evaluated systems run in a Java Virtual Machine (JVM), making their runtimes and memory consumption easy to compare. We used 29 GB of operating memory per slave, leaving 3 GB for the operating system, distributed filesystem caches, and other JVM memory pools, such as native buffers for network I/O. Consequently, for each system under test, the cluster had a total amount of 400 hardware contexts and an aggregate Java heap of 725 GB.

For all systems, job input and output were stored in a common HDFS instance using plain ASCII format. Each datanode was configured to use four SATA drives for data storage, resulting in approximately 500 MB/s read and write speed per datanode, and total disk capacity of 80 TB for the entire HDFS. Each test was run in isolation, since both Stratosphere and the newer Hadoop versions (based on YARN) share no resources between queries and realize multi-tenancy through exclusive resource containers allocated to each job.

[5] Some language compilers can transform functions that return a sequence of values automatically into an iterator. Java, however, offers no such mechanism.

In all reported experiments, we range the number of slaves from 5 to 25. For both MapReduce and Stratosphere, the configured degree of parallelism *(DOP)* was 8 parallel tasks per slave, yielding a total *DOP* between 40 and 200 tasks, and full CPU utilization for two overlapping data-parallel phases. To reduce the effect of system noise and outliers (mostly through lags in the HDFS response time), we report the median execution time of three job executions.

8.2 TeraSort

To measure the efficiency of Hadoop's and Stratosphere's execution engines in an isolated way, we performed a simple experiment comparing the TeraSort job that ships as part of the example programs package with the two systems. Both TeraSort implementations are expressed trivially using a pair of an identity map and reduce UDFs and a custom range-partitioning function for the shuffle phase. For our experiment, we generated TeraGen input data with scaling factor *(SF)* ranging from 10 to 50 for the corresponding *DOP* from 5 to 25 (a *SF* of 1 corresponds to 10^9 bytes). In order to factor out the impact of file system access and isolate the sorting operator, we also executed a variant of the TeraSort job that generates input data on the fly and does not write out the result. The observed runtimes for both variants (Fig. 10a) indicate that the distributed sort operators of Stratosphere and Hadoop have similar performance and scale linearly with the *DOP* parameter.

8.3 Word count

In our second experiment, we compared Stratosphere and Hadoop using a simple "word count" job that counts the word frequencies and is often used as a standard example for MapReduce programs. A standard optimization of the naïve Word Count implementation that we implemented for both systems is to exploit the algebraic nature of the applied aggregate function and use a combiner UDF in order to reduce the volume of the shuffled data. As an input, we used synthetically generated text data with words sampled with skewed frequency from a dictionary with 100000 entries. The dictionary entries and their occurrence frequencies were obtained by analyzing the Gutenberg English language corpus [59]. As before, a *SF* of 1 corresponds to 1 GB of plain text data.

The results are presented in Fig. 10b. As in the previous experiment, both systems exhibit linear scale-out behavior. This is not surprising, given that TeraSort and Word Count have essentially the same second-order task structure consisting of a map, a reduce, and an intermediate data shuffle. In contrast to TeraSort, however, the Word Count job is approximately 20 % faster in Stratosphere than in Hadoop.

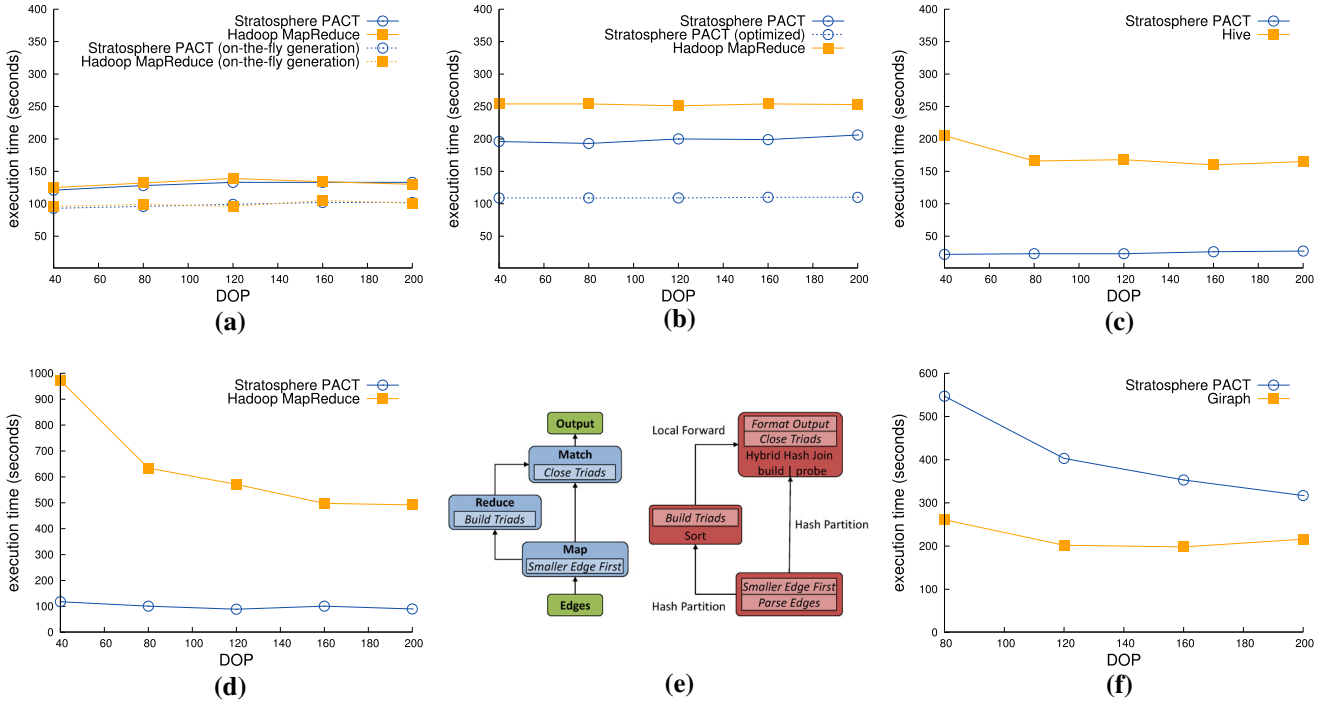

Fig. 10 Scale-out experiments with *DOP* ranging from 40 to 200. **a** TeraSort **b** Word count **c** TPCH Q3 **d** Triangle enumeration **e** PACT and Nephele DAGs for Triangle Enumeration **f** Connected Components

The reasons for that are twofold. First, Stratosphere's push-based shuffle enables better pipelined parallelism between the map and the shuffle operators. The combiner function is applied "batch-wise" to sorted parts of the mapper's output, which then are eagerly pushed downstream. In contrast, Hadoop always writes and sorts the mapper output to disk, applying the (optional) combiner UDF multiple times during this process, and serving the constructed sorted data partitions upon request. This approach produces less data when the cardinality of the grouping key is low, but comes at the price of fixed costs for a two-phase sort on the sender side. The second reason for Stratosphere's superior performance is the use of a sort optimization that reduces the amount of type-specific comparisons at the cost of replicating a key prefix in a binary format that can be compared in a bit-wise fashion [31]. This optimization is especially useful for keys with complex data types, such as strings.

We also note that a substantial amount of the processing time in the Word Count map phase goes into tokenizing the input text into separate words. An optimized string tokenizer implementation that works directly with Stratosphere's string types and a simplified code page translation (indicated with a dotted line on Fig. 10b) yields 50 % performance benefit compared with the version that uses a native JDK *StringTokenizer*. The same optimization can be done for the Hadoop implementation and is likely to result in similar performance improvement.

8.4 Relational query

To evaluate the impact of Stratosphere's cost-based optimization and powerful data flow runtime for the execution of more complex tasks, we ran a modified version of Query #3 from the TPC-H benchmark. We omitted the *order by* and *limit* clauses, because the current Stratosphere version has no implementation of *top-k* operators. We compare Stratosphere against Hive—a data warehousing solution that compiles and executes SQL-like queries as sequences of Hadoop MapReduce jobs.

```
SELECT l_orderkey,
       SUM(l_extendedprice *
       (1 - l_discount)),
       o_orderdate, o_shippriority
FROM customer c
     JOIN orders o ON
     (c_custkey = o_custkey)
     JOIN lineitem l ON
     (l_orderkey = o_orderkey)
WHERE c_mktsegment = 'HOUSEHOLD'
      AND o_orderdate < '1995-03-15'
      AND l_shipdate > 1995-03-15'
GROUP BY l_orderkey,
         o_orderdate,
         o_shippriority
```

The results on Fig. 10c illustrate the benefits of Stratosphere's more general approach for specification and optimization of complex data processing programs. Hive's optimizer is bound to the fixed MapReduce execution pipeline and has to split complex HiveQL expressions into multiple MapReduce jobs, which introduces unnecessary I/O overhead due to the fact that between each map and reduce phase all the data has to be spilled to disk. Stratosphere on the other hand can optimize and execute arbitrary complex jobs as a whole, using the DAG-oriented program representation described in Sect. 6. For the evaluated query, the optimizer makes full use of this, selecting hash-based execution strategies for the two join operators (which are realized using a Match contract) such that the larger input is always pipelined. As the build sides of the two Match operators and the input of the reducer handling the "revenue" aggregate computation both fit into memory, no data are written to disk until the final output is produced.

8.5 Triangle enumeration

Another example that illustrates the benefits of PACT compositionality is the algorithm that enumerates graph triangles, described by Cohen in [21] as a prerequisite for deeper graph analysis like identification of dense subgraphs. The algorithm is formulated in [21] as a sequence of two MapReduce jobs, where the first reducer is responsible for building all triads (pairs of connected edges), while the second reducer simulates a join between triads and edges to filter out all triads that cannot be closed by a matching edge.

In contrast to the cumbersome MapReduce realization presented above, Stratosphere's PACT model offers a native way to express triangle enumeration as an single, simple dataflow using a map, a reduce, and a match. Moreover, as with the TPC-H query, Stratosphere's optimizer compares the cost of alternative execution strategies and picks a plan that reduces that cost via pipelining of the data-heavy "triads" path, as indicated in Fig. 10e. Such an optimization can have substantial impact on the execution time, as the output of the "build triads" operator is asymptotically quadratic in the number of edges.

Figure 10d shows the results of an experiment enumerating all triangles of a symmetric version of the Pokec social network graph [62]. The results highlight the benefit of the pipelined execution strategy compared with a naïve Hadoop implementation that uses two MapReduce jobs and materializes the intermediate results after each step. For small degrees of parallelism, the difference is close to an order of magnitude. Due to the skewed vertex degree distribution, increasing the *DOP* does not have an impact for the pipelined Stratosphere version, whereas Hadoop benefits from the reduced read and write times for result materialization. However, this effect wears off for $DOP > 160$ and can be dampened further through the use of a better scheduling policy that takes into account the data skew.

8.6 Connected components

In our next experiment, we used Giraph—an open-source implementation of Pregel [54]—as a reference point for the evaluation of Stratosphere's iterative dataflow constructs. For our experiments, we ran the connected components algorithm proposed by Kang et al. in [49] on the crawled Twitter graph used by Cha et al. in [17], using *DOP* from 80 to 200.

Figure 10f displays the observed execution times. For low node counts, Giraph currently offers twice the performance compared with Stratosphere. We attribute this mostly to a better tuned implementation, since both approaches essentially are based on the same execution model—the bulk synchronous iterations presented in Sect. 5.3. However, while the benefits of scaling out are clear in Stratosphere, increasing the *DOP* for Giraph has hardly any performance gains. Further analysis of the execution time of each superstep in the two systems revealed that although the time required to execute the first supersteps drops for higher *DOP* values, the latter supersteps actually take more time to finish (Fig. 11a, b). For Giraph, this effect is stronger, and the total sum of all supersteps cancels out the improvement of the early stages. We attribute this behavior to inefficiencies in the implementation of Giraph's worker node communication model. We also note that the linear scale-out behavior in both systems is dampened by the skew in the vertex distribution across the graph nodes, and further scale-out will ultimately be prohibited as the data-partitioning approach used for parallel execution will not be able to handle the skew in the input data and the associated computations.

8.7 Fault tolerance

In this section, we give preliminary results on the overhead and efficiency of the ephemeral checkpointing and recovery mechanism as briefly described in Sect. 7.2. The numbers are based on the experiments of a previous publication [39]. We used the "Triangle Enumeration" program (including the preprocessing steps originally suggested by Cohen [21]) and a variant of the "Relational Query" (with only one join between the "Lineitem" and "Orders" tables). For each program, we measured the failure-free runtime both with and without checkpointing, as well as the total runtime including recovery after a failure. For both programs, the failure occurs in the join operator after roughly 50 % of the failure-free runtime. For the relational query, the fault-tolerance mechanism checkpoints the results before the join (the filtered and projected tables), and for the triangle enumeration, it saves both the data before the candidate-creating reducer (which inflates

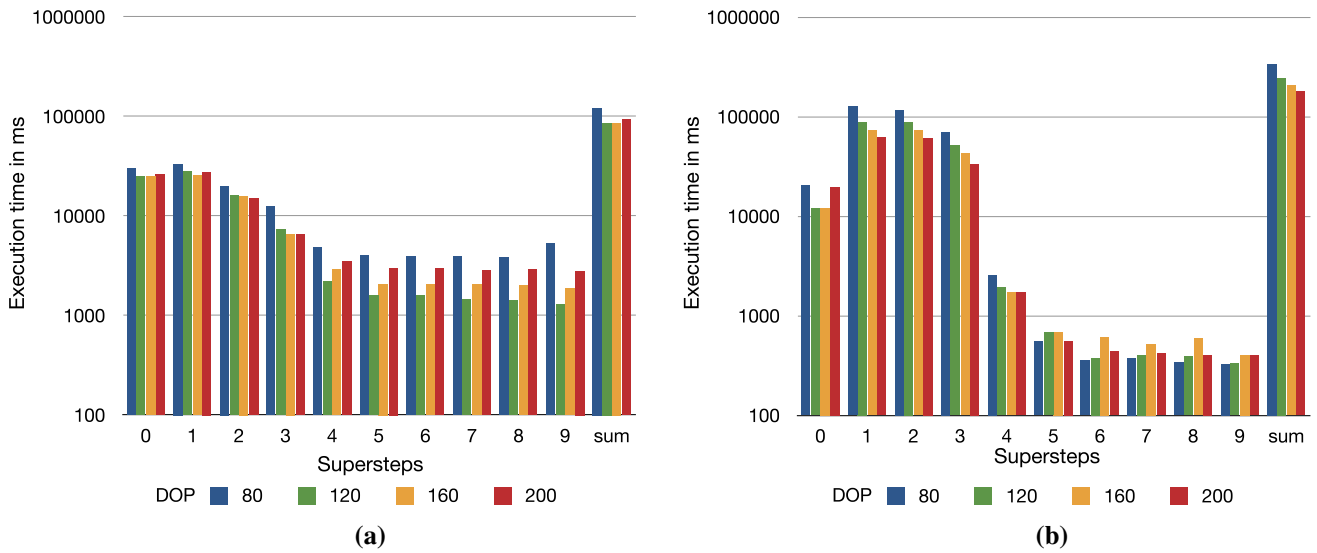

Fig. 11 Execution time per superstep of the *Connected Components* fixpoint algorithm. **a** CC Supersteps (Giraph) **b** CC Supersteps (Stratosphere)

Table 2 Runtime (secs) of jobs in the fault tolerance experiments

Experiment	Rel. query	Triangle enum.
Failure free (no checkp.)	1,026	646
Failure free (w/ checkp.)	1,039	678
Checkpointing Overhead	1.2 %	5 %
Failure & recovery	1,131	747

the data volume) and after the data-reducing join operator. The experiments were run on 8 machines with 4 cores each and 16 GB RAM each, which is enough to keep checkpoints entirely in main memory (Table 2).

We observe that the checkpointing itself adds very little overhead, because it simply keeps a copy of the buffers in memory. The mechanism of ephemeral checkpoints selectively materializes small intermediate results, keeping the required memory small compared with the size of data sets and certain intermediate results. For both jobs, the runtime with a failure is only moderately higher than the failure-free runtime. In the case of the relational query, a big part of the execution time is spent in scanning, filtering, and projecting the inputs. Because the result is checkpointed, these costly operations are not repeated as part of the recovery. In the case of the triangle enumeration program, the join operator repeats its work. It can, however, skip shipping roughly the first half of its result data to the sinks, thereby avoiding a good part of the result writing costs.

8.8 Conclusions

The results in this section indicate that Stratosphere offers comparable or better performance against alternative open-source systems that provide general-purpose (Hadoop, Hive) or domain-specific (Giraph) functionality. Most of the gain can be attributed to execution layer features (e.g., eager push communication, different PACT implementations) and the ability of Stratosphere's optimizer to consider these features during plan enumeration in a cost-based manner. In contrast, most of the other open-source systems make hard-coded decisions and fix some of the physical aspects in the used execution plans, which may cause inefficiencies depending on the size and the distributions of the input data.

9 Ongoing work

We are currently working on several research and development threads in the context of Stratosphere. In this section, we describe work that is currently ongoing, deferring a broader research outlook until Sect. 11.

A major direction of our work pertains to query optimization. First, we are currently working on consolidating and unifying the PACT and Sopremo layers of Stratosphere into a single operator model that includes operators with known semantics as well as user-defined functions. Our goal was to arrive at an one-pass optimizer that considers operator reordering and parallelization transformations in the same pass. The optimizer will fall back to static code analysis techniques (an extension of the current prototypical techniques) only when operator semantics are not known. The optimizer will provide pay-as-you-go implementation and optimization of new, domain-specific operators, enabling developers to rapidly add new operators, i.e., by implementing basic algorithms, which can be extended and improved over time. Second, we are designing a module that injects monitoring

operators in the plan that collect runtime statistics and report to a metadata store to use during optimization. Third, we are working on refining the cost model of the optimizer toward supporting the goal of *robustness*, in particular, finding plans that are optimized for a variance in addition to an expected cost metric. Fourth, we are working toward adapting plans at runtime, as uncertainty about the intermediate result sizes and the characteristics of the execution environment (especially in the cloud setting) can quickly render static optimization decisions meaningless.

A second major direction is related to strengthening the system's fault-tolerant execution capabilities. We are working on an adaptive algorithm that selectively picks which intermediate results to materialize, taking into account failure probabilities, the execution graph structure, the size of the task results, the cost of the tasks, measured and adapted at runtime.

Iterative jobs present novel opportunities for handling their fault-tolerant execution. We are currently investigating to which extent algorithmic compensation techniques can alleviate the need for checkpointing intermediary algorithm state to stable storage (e.g., by exploiting the robust nature of fixed point algorithms commonly used in data mining). By this, we intend to enable novel optimistic approaches to fault tolerance in distributed iterative data processing.

A third direction relates to the scalability and efficiency of the Nephele execution engine through connection multiplexing, application-level flow control, multicasting support, and selective pipeline breaking to avoid deadlocks or reduce stalls caused by head-of-the-line waiting effects. In addition, we are working on better memory management during data shuffling in very large clusters.

Finally, we are experimenting with porting additional high-level query or scripting languages on top of the Stratosphere query processor. We have, in initial stages, a prototype of Pig [48], and a Scala dialect of the PACT programming model [36].

10 Related work

10.1 End-to-end big data systems

There are currently a few systems under development in academia and industry that seek to advance the state of the art in distributed data management. The Hadoop ecosystem (including higher-level languages such as Hive, Pig, libraries such as Mahout and other tooling) is the most popular. Compared with Hadoop, Stratosphere offers more efficient execution and more sophisticated optimization due to the extended set of primitives. In addition, the PACT model encourages more modular code and component reuse [2]. The Hadoop ecosystem does not currently support DAG-structured plans or iterative jobs and is therefore very inefficient in use cases that require or benefit from those.

Asterix [12] is a research effort by several campuses at the University of California. Like Stratosphere, Asterix offers a complete stack including a higher-level language AQL, a query optimizer, and a distributed runtime system [15]. While both Stratosphere and Asterix aim at bridging the gap between MapReduce and parallel DBMSs, they start at the opposite ends of the spectrum (and often meet in the middle). Asterix starts with a semi-structured data model and language, while Stratosphere follows a UDF-centric model. Asterix includes data storage based on LSM-trees, paying the price of a data loading phase for efficient execution, while Stratosphere connects to external data sources (e. g., HDFS), converting data to optimized binary representations only after the initial scans. Nevertheless, we have arrived at similar lessons with the Asterix team in several aspects pertaining the development of distributed data management systems.

Scope [74] is a system used by Microsoft Bing for several analytic tasks.[6] Scope integrates with Microsoft's LINQ interface, allowing the specification of rich UDFs similar to Stratosphere. It features a sophisticated query optimizer [16,61,73] and runtime system based on a rewrite of Dryad [44,72]. Scope is perhaps the system most similar to Stratosphere, perhaps aiming at scalability more than efficiency. To the best of our knowledge, Scope does not efficiently support incrementally iterative queries.

The Spark [70] system from UC Berkeley is a distributed system that operates on memory-resident data. Spark provides functionality equivalent to Stratosphere's bulk iterations, but not incremental iterations. In addition, while Spark is a system that processes batches of data, Stratosphere features an execution engine that pipelines data. Pipelining is better suited to use cases that incrementally process data as found in many machine learning applications.

10.2 Query languages and models for parallel data management

It is conceptually easy to parallelize the basic operators of relational algebra, and parallel analytical databases have existed for decades. The MapReduce paradigm widens the scope of parallel programs to include more generalized user-define aggregation functions [22]. Our PACT model is a generalization of MapReduce that in addition enables the parallelization of generalized joins. MapReduce has been the compilation target of SQL subsets [19,64], as well as other languages inspired by scripting languages [57,58] or XQuery [13]. The Meteor language borrows its syntax from Jaql [13]. Stratosphere is a better compilation target for the

[6] At the time of writing, Scope is not offered as a product or service by Microsoft.

aforementioned languages than MapReduce, as the system's optimizer can be reused, and a richer set of primitives is available as the compilation target language. While these query languages are based on the relational or a semi-structured model, other efforts [1,18] aim to embed domain-specific languages in functional general programming languages. An integration of a functional language with Stratosphere constitutes a major part of our future work.

10.3 Query optimization

Query optimization is one of the most researched topics in the context of data management. While the physical optimization as described in Sect. 6.3 is closely related to traditional optimization of relational queries as, for example, in references [33,60], the rewriting of data flows consisting of user-defined operators has gained more interest recently. Similar to our approach, Microsoft's Scope compiler leverages information derived by static code analysis to reason about the preservation of interesting physical data properties [35,71]. Manimal [45] applies static code analysis on Hadoop map and reduce UDFs. In contrast to our work, operators are not reordered but map functions that include filter conditions are modified in order to access an index instead of performing a full scan of the input file. Stubby [52] optimizes workflows consisting of multiple MapReduce jobs by merging consecutive map and reduce operators to reduce the overhead of running multiple MapReduce jobs. Stubby gains the information for its optimizations from manual code annotations.

10.4 Distributed dataflow execution

While the principles behind parallel databases have been known since the 80s [23,28] and industrial-strength parallel databases have existed for as long [63], the last decade brought a new wave of "massively parallel" relational query processors [7,34] as well as truly scalable implementations of more restricted models, notable MapReduce [22]. Nephele is a scalable implementation of the DAG dataflow model, similar to Microsoft's Dryad [44]. A different approach is followed in the Asterix project, where the Hyracks engine [15] executes physical operators (e. g., physical join implementations) rather than user-defined functions wrapped in glue code that parallelizes execution. All these projects aim to scale query processing to clusters of 100s or even 1000s and beyond nodes.

10.5 Distributed iterative algorithms

Over the last years, a number of stand-alone graph processing systems or approaches to integrate iterative processing in dataflow engines have been proposed. Spark [70] handles state as resilient distributed data sets and provides constructs for efficiently executing iterative data flows that recompute the state as a whole. GraphLab [53] is a specialized framework for parallel machine learning, where programs model a graph expressing the computational dependencies of the input. Programs are expressed as update functions on vertices, which can read neighboring vertices' state through a shared memory abstraction. Furthermore, GraphLab provides configurable consistency levels and asynchronous scheduling of the updates. Pregel [54] is a graph processing adoption of bulk synchronous parallel processing [65]. Programs directly model a graph, where vertices hold state and send messages to other vertices along the edges. By receiving messages, vertices update their state. Rex [56] is a parallel shared-nothing query processing platform that provides programmable deltas for expressing incrementally iterative computations. Naiad [55] unifies incrementally iterative computations with continuous data ingestion into a new technique called differential computation. In this approach, intermediate results from different iterations and ingestion periods are represented as partially ordered deltas, and final results are reconstructed lazily using computationally efficient combinations of their lineage deltas.

11 Conclusions and research outlook

We presented Stratosphere, a deep software stack for analyzing Big Data. Stratosphere features a high-level scripting language, Meteor, which focuses on providing extensibility. Using Meteor and the underlying Sopremo operator model, domain-specific experts can extend the system's functionality with new operators, in addition to operator packages for data warehousing, information extraction, and information integration already provided. Stratosphere features an intermediate UDF-centric programming model based on second-order functions and higher-order abstractions for iterative queries. These programs are optimized using a cost-based optimizer inspired by relational databases and adapted to a schema-less and UDF-heavy programming and data model. Finally, Nephele, Stratosphere's distributed execution engine provides scalable execution, scheduling, network data transfers, and fault tolerance. Stratosphere occupies a sweet spot between MapReduce and relational databases. It offers declarative program specification; it covers a wide variety of data analysis tasks including iterative or recursive tasks; it operates directly on distributed file systems without requiring data loading; and it offers scalable execution on large clusters and in the cloud.

The lessons learned while building Stratosphere have opened several directions for research. First, we see a lot of potential in the design, compilation, and optimization of high-level declarative languages for various analytical domains, in particular machine learning. There, it is chal-

lenging to divide the labor between the language compiler, database optimizer, and runtime system, and define the right abstractions between these components. Second, we believe that distributed data management systems should be efficient in addition to scalable and thus should adapt their algorithms and architecture to the ever-evolving landscape of hardware, including multi-core processors, NUMA architectures, co-processors, such as GPUs and FPGAs, flash and phase-change memory, as well as data center networking infrastructure.

Finally, we see our next major research thread revolving around use cases that move beyond batch data processing and require fast data ingestion and low-latency data analysis. Such systems, in order to provide rich functionality, must effectively manage mutable state, manifested in the form of stateful user-defined operators operating on data streams, or in the context of incremental iterative algorithms. Such state must be embedded in a declarative language via proper abstractions and efficiently managed by the system. In addition, workloads of (long-running) programs need to be optimized together.

Acknowledgments We would like to thank the Master students that worked on the Stratosphere project and implemented many components of the system: Thomas Bodner, Christoph Brücke, Erik Nijkamp, Max Heimel, Moritz Kaufmann, Aljoscha Krettek, Matthias Ringwald, Tommy Neubert, Fabian Tschirschnitz, Tobias Heintz, Erik Diessler, Thomas Stolltmann.

References

1. Ackermann, S., Jovanovic, V., Rompf, T., Odersky, M.: Jet: an embedded dsl for high performance big data processing. In: BigData Workshop at VLDB (2012)
2. Alexandrov, A., Ewen, S., Heimel, M., Hueske, F., Kao, O., Markl, V., Nijkamp, E., Warneke, D.: Mapreduce and pact - comparing data parallel programming models. In: BTW, pp. 25–44 (2011)
3. Alexandrov, A., Battré, D., Ewen, S., Heimel, M., Hueske, F., Kao, O., Markl, V., Nijkamp, E., Warneke, D.: Massively parallel data analysis with pacts on nephele. PVLDB **3**(2), 1625–1628 (2010)
4. Apache Giraph. http://incubator.apache.org/giraph/
5. Apache Hadoop. http://hadoop.apache.org/
6. Apache Hive. http://sortbenchmark.org/
7. Aster Data. http://www.asterdata.com/
8. Battré, D., Ewen, S., Hueske, F., Kao, O., Markl, V., Warneke, D.: Nephele/pacts: a programming model and execution framework for web-scale analytical processing. In: SoCC, pp. 119–130 (2010)
9. Battré, D., Frejnik, N., Goel, S., Kao, O., Warneke, D.: Evaluation of network topology inference in opaque compute clouds through end-to-end measurements. In: IEEE CLOUD, pp. 17–24 (2011)
10. Battré, D., Frejnik, N., Goel, S., Kao, O., Warneke, D.: Inferring network topologies in infrastructure as a service cloud. In: CCGRID, pp. 604–605 (2011)
11. Battré, D., Hovestadt, M., Lohrmann, B., Stanik, A., Warneke, D.: Detecting bottlenecks in parallel dag-based data flow programs. In: MTAGS (2010)
12. Behm, A., Borkar, V.R., Carey, M.J., Grover, R., Li, C., Onose, N., Vernica, R., Deutsch, A., Papakonstantinou, Y., Tsotras, V.J.: Asterix: towards a scalable, semistructured data platform for evolving-world models. Distrib. Parallel Databases **29**(3), 185–216 (2011)
13. Beyer, K.S., Ercegovac, V., Gemulla, R., Balmin, A., Eltabakh, M.Y., Kanne, C.C., Özcan, F., Shekita, E.J.: Jaql: a scripting language for large scale semistructured data analysis. PVLDB **4**(12), 1272–1283 (2011)
14. Boden, C., Karnstedt, M., Fernandez, M., Markl, V.: Large-scale social media analytics on stratosphere. In: WWW (2013)
15. Borkar, V.R., Carey, M.J., Grover, R., Onose, N., Vernica, R.: Hyracks: a flexible and extensible foundation for data-intensive computing. In: ICDE, pp. 1151–1162 (2011)
16. Bruno, N., Agarwal, S., Kandula, S., Shi, B., Wu, M.C., Zhou, J.: Recurring job optimization in scope. In: SIGMOD Conference, pp. 805–806 (2012)
17. Cha, M., Haddadi, H., Benevenuto, F., Gummadi, P.K.: Measuring user influence in twitter: the million follower fallacy. In: ICWSM (2010)
18. Chafi, H., DeVito, Z., Moors, A., Rompf, T., Sujeeth, A.K., Hanrahan, P., Odersky, M., Olukotun, K.: Language virtualization for heterogeneous parallel computing. In: OOPSLA, pp. 835–847 (2010)
19. Chattopadhyay, B., Lin, L., Liu, W., Mittal, S., Aragonda, P., Lychagina, V., Kwon, Y., Wong, M.: Tenzing a sql implementation on the mapreduce framework. PVLDB **4**(12), 1318–1327 (2011)
20. Chaudhuri, S., Shim, K.: Including group-by in query optimization. In: VLDB, pp. 354–366 (1994)
21. Cohen, J.: Graph twiddling in a mapreduce world. Comput. Sci. Eng. **11**(4), 29–41 (2009)
22. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. In: OSDI, pp. 137–150 (2004)
23. DeWitt, D.J., Gerber, R.H., Graefe, G., Heytens, M.L., Kumar, K.B., Muralikrishna, M.: Gamma—a high performance dataflow database machine. In: VLDB, pp. 228–237 (1986)
24. Elnozahy, E.N.M., Alvisi, L., Wang, Y.M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. **34**(3), 375–408 (2002)
25. Ewen, S., Schelter, S., Tzoumas, K., Warneke, D., Markl, V.: Iterative parallel data processing with stratosphere: an inside look. In: SIGMOD (2013)
26. Ewen, S., Tzoumas, K., Kaufmann, M., Markl, V.: Spinning fast iterative data flows. PVLDB **5**(11), 1268–1279 (2012)
27. Fegaras, L., Li, C., Gupta, U.: An optimization framework for map-reduce queries. In: EDBT, pp. 26–37 (2012)
28. Fushimi, S., Kitsuregawa, M., Tanaka, H.: An overview of the system software of a parallel relational database machine grace. In: VLDB, pp. 209–219 (1986)
29. Ghemawat, S., Gobioff, H., Leung, S.T.: The google file system. In: SOSP, pp. 29–43 (2003)
30. Graefe, G., Bunker, R., Cooper, S.: Hash joins and hash teams in microsoft sql server. In: VLDB, pp. 86–97 (1998)
31. Graefe, G.: Implementing sorting in database systems. ACM Comput. Surv. **38**(3), Article ID 10 (2006)
32. Graefe, G.: Parallel query execution algorithms. In: Encyclopedia of Database Systems, pp. 2030–2035 (2009)
33. Graefe, G.: Volcano—an extensible and parallel query evaluation system. IEEE Trans. Knowl. Data Eng. **6**(1), 120–135 (1994)
34. Greenplum. http://www.greenplum.com/
35. Guo, Z., Fan, X., Chen, R., Zhang, J., Zhou, H., McDirmid, S., Liu, C., Lin, W., Zhou, J., Zhou, L.: Spotting code optimizations in data-parallel pipelines through periscope. In: OSDI, pp. 121–133 (2012)
36. Harjung, J.J.: Reducing formal noise in pact programs. Master's thesis, Technische Universität Berlin, Faculty of EECS (2013)
37. Heise, A., Rheinländer, A., Leich, M., Leser, U., Naumann, F.: Meteor/sopremo: an extensible query language and operator model. In: BigData Workshop at VLDB (2012)

38. Heise, A., Naumann, F.: Integrating open government data with stratosphere for more transparency. Web Semant.: Sci. Serv. Agents World Wide Web **14**, 45–56 (2012)
39. Höger, M., Kao, O., Richter, P., Warneke, D.: Ephemeral materialization points in stratosphere data management on the cloud. Adv. Parallel Comput. **23**, 163–181 (2013)
40. Hovestadt, M., Kao, O., Kliem, A., Warneke, D.: Evaluating adaptive compression to mitigate the effects of shared i/o in clouds. In: IPDPS Workshops, pp. 1042–1051 (2011)
41. Hueske, F., Krettek, A., Tzoumas, K.: Enabling operator reordering in data flow programs through static code analysis. CoRR abs/1301.4200 (2013)
42. Hueske, F., Peters, M., Krettek, A., Ringwald, M., Tzoumas, K., Markl, V., Freytag, J.C.: Peeking into the optimization of data flow programs with mapreduce-style udfs. In: ICDE (2013)
43. Hueske, F., Peters, M., Sax, M., Rheinländer, A., Bergmann, R., Krettek, A., Tzoumas, K.: Opening the black boxes in data flow optimization. PVLDB **5**(11), 1256–1267 (2012)
44. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. In: EuroSys, pp. 59–72 (2007)
45. Jahani, E., Cafarella, M.J., Ré, C.: Automatic optimization for mapreduce programs. PVLDB **4**(6), 385–396 (2011)
46. Java HotSpot VM Whitepaper. http://www.oracle.com/technetwork/java/whitepaper-135217.html
47. JavaScript Object Notation. http://json.org/
48. Kalavri, V.: Integrating pig and stratosphere. Master's thesis, KTH, School of Information and Communication Technology (ICT) (2012)
49. Kang, U., Tsourakakis, C.E., Faloutsos, C.: Pegasus: a peta-scale graph mining system. In: ICDM, pp. 229–238 (2009)
50. Kung, H.T., Robinson, J.T.: On optimistic methods for concurrency control. ACM Trans. Database Syst. **6**(2), 213–226 (1981)
51. Leich, M., Adamek, J., Schubotz, M., Heise, A., Rheinländer, A., Markl, V.: Applying stratosphere for big data analytics. In: BTW, pp. 507–510 (2013)
52. Lim, H., Herodotou, H., Babu, S.: Stubby: a transformation-based optimizer for mapreduce workflows. PVLDB **5**(11), 1196–1207 (2012)
53. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: Distributed graphlab: a framework for machine learning in the cloud. PVLDB **5**(8), 716–727 (2012)
54. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: SIGMOD Conference, pp. 135–146 (2010)
55. McSherry, F., Murray, D., Isaacs, R., Isard, M.: Differential dataflow. In: CIDR (2013)
56. Mihaylov, S.R., Ives, Z.G., Guha, S.: Rex: recursive, delta-based data-centric computation. PVLDB **5**(11), 1280–1291 (2012)
57. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: a not-so-foreign language for data processing. In: SIGMOD Conference, pp. 1099–1110 (2008)
58. Pike, R., Dorward, S., Griesemer, R., Quinlan, S.: Interpreting the data: parallel analysis with sawzall. Sci. Program. **13**(4), 277–298 (2005)
59. Project Gutenberg. http://www.gutenberg.org/
60. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: SIGMOD Conference, pp. 23–34 (1979)
61. Silva, Y.N., Larson, P.A., Zhou, J.: Exploiting common subexpressions for cloud query processing. In: ICDE, pp. 1337–1348 (2012)
62. Stanford Network Analysis Project. http://snap.stanford.edu/
63. Teradata. http://www.teradata.com/
64. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., Murthy, R.: Hive—a warehousing solution over a map-reduce framework. PVLDB **2**(2), 1626–1629 (2009)
65. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990)
66. Wang, Y.M., Fuchs, W.K.: Lazy checkpoint coordination for bounding rollback propagation. In: Reliable Distributed Systems, 1993. Proceedings., 12th Symposium on, pp. 78–85 (1993)
67. Warneke, D., Kao, O.: Nephele: efficient parallel data processing in the cloud. In: SC-MTAGS (2009)
68. Warneke, D., Kao, O.: Exploiting dynamic resource allocation for efficient parallel data processing in the cloud. IEEE Trans. Parallel Distrib. Syst. **22**(6), 985–997 (2011)
69. Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, Ú., Gunda, P.K., Currey, J.: Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language. In: OSDI, pp. 1–14 (2008)
70. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: NSDI (2012)
71. Zhang, J., Zhou, H., Chen, R., Fan, X., Guo, Z., Lin, H., Li, J.Y., Lin, W., Zhou, J., Zhou, L.: Optimizing data shuffling in data-parallel computation by understanding user-defined functions. In: NSDI (2012)
72. Zhou, J., Bruno, N., Lin, W.: Advanced partitioning techniques for massively distributed computation. In: SIGMOD Conference, pp. 13–24 (2012)
73. Zhou, J., Larson, P.Å., Chaiken, R.: Incorporating partitioning and parallel plans into the scope optimizer. In: ICDE, pp. 1060–1071 (2010)
74. Zhou, J., Bruno, N., Wu, M.C., Larson, P.Å., Chaiken, R., Shakib, D.: Scope: parallel databases meet mapreduce. VLDB J. **21**(5), 611–636 (2012)