Peeking into the Optimization of Data Flow Programs with MapReduce-style UDFs

Fabian Hueske [*1], Mathias Peters [†2], Aljoscha Krettek [*3], Matthias Ringwald [*4], Kostas Tzoumas [*5], Volker Markl [*6], Johann-Christoph Freytag [†7]

[*]*Technische Universität Berlin, Germany*
[1,5,6]firstname.lastname@tu-berlin.de
[3,4]firstname.lastname@campus.tu-berlin.de

[†]*Humboldt-Universität zu Berlin, Germany*
[2]firstname.lastname@informatik.hu-berlin.de
[7]lastname@dbis.informatik.hu-berlin.de

***Abstract*—Data flows are a popular abstraction to define data-intensive processing tasks. In order to support a wide range of use cases, many data processing systems feature MapReduce-style user-defined functions (UDFs). In contrast to UDFs as known from relational DBMS, MapReduce-style UDFs have less strict templates. These templates do not alone provide all the information needed to decide whether they can be reordered with relational operators and other UDFs. However, it is well-known that reordering operators such as filters, joins, and aggregations can yield runtime improvements by orders of magnitude.**

We demonstrate an optimizer for data flows that is able to reorder operators with MapReduce-style UDFs written in an imperative language. Our approach leverages static code analysis to extract information from UDFs which is used to reason about the reorderbility of UDF operators. This information is sufficient to enumerate a large fraction of the search space covered by conventional RDBMS optimizers including filter and aggregation push-down, bushy join orders, and choice of physical execution strategies based on interesting properties.

We demonstrate our optimizer and a job submission client that allows users to peek step-by-step into each phase of the optimization process: the static code analysis of UDFs, the enumeration of reordered candidate data flows, the generation of physical execution plans, and their parallel execution. For the demonstration, we provide a selection of relational and non-relational data flow programs which highlight the salient features of our approach.

I. Introduction

The collected amount of data, as well as the complexity of data analysis tasks performed are rapidly increasing. Over the last years, a large body of research has been devoted to enable complex analysis on huge amounts of data. Many systems that have been proposed for big data analytics are based on the concept of parallel data flows, including parallel relational DBMS and MapReduce. In recent years, several of higher-level languages and programming interfaces have been designed to ease the definition of complex processing tasks [1, 2, 3, 4, 5]. These languages are based on algebraic operators such as filters, joins, (un-)nesting, and aggregation. However, many of these abstractions also provide interfaces for MapReduce-style user-defined functions (UDFs) [1, 2, 3, 4, 6, 7], which are written in an imperative programming language such as Java or C++ and obey certain restrictions on the function signature.

Query optimization is a key technology for data processing systems. A reasonable choice of operator order and physical execution strategies often yields runtime improvements by orders of magnitude over arbitrary alternatives. In the context of algebraic expressions as for example relational queries, optimization is well-researched. Due to their strict templates, scalar, aggregation, and table-generation UDFs can be, in principle included into the optimization process [8, 9]. However, data flows that include UDFs with more general templates such as MapReduce-style UDFs pose a big challenge for optimization.

Recently, we have shown how static code analysis can be used to change the order of MapReduce-style UDFs in data flow programs [10, 11]. Here, we demonstrate the optimization of data flows specified as *Pact programs* [12]. Our prototype is based on the Stratosphere system [13]. It features a static code analysis component, and a cost-based optimizer, which supports a large subset of the possible transformations supported by traditional relational optimizers. The set of transformations includes selection and join reordering, and invariant group transformations [14], as well as reasoning about interesting properties such as order and data partitioning [15]. Our demonstrator shows step-by-step the optimization of Pact programs, starting with the static code analysis, the optimization process, and finally the parallel execution of a chosen execution plan. We demonstrate the features of our approach using a set of analytical data flow programs from relational and non-relational domains.

The remainder of this paper is structured as follows. Section II introduces the Pact programming model, discusses reordering conditions for Pact operators, and presents an overview of the optimization process. Section III presents our demonstration setup, and Section IV concludes.

II. Background

A. The Pact Programming Model

The Pact programming model [12] is a generalization and extension of the MapReduce programming model [16]. Pact programs are specified as DAGs where data sources, operators, and data sinks are connected by data transfer channels. Data is represented as sets of records with arbitrarily typed fields.

Data sources have no incoming channels and generate records usually by deserializing them from input files. Operators receive data, process it, and forward their result to the upstream operator(s) in the DAG. Data sinks have no outgoing channels; they usually transform their input into a suitable format for presentation or further processing.

A Pact operator consists of a second-order system function (SOF) and a first-order user-defined function (UDF). The SOF maps the operator's input data into subsets which are independently processed by passing them to the UDF. Hence, the parallelization opportunities for an operator are determined by the SOF alone. Currently, our system supports five SOFs which are shown in Figure 1. The two unary SOFs, Map and Reduce, have the same semantics as in MapReduce. The remaining SOFs process data from two inputs. Cross forms the Cartesian product of both inputs and calls its UDF for each pair of records. Match calls the UDF for each pair of records from both inputs where their key fields are the same. Hence, it resembles an equi-join. CoGroup forms a group for each key, and calls the UDF with all records that share the same key in both inputs. In this work, we only consider tree-shaped Pact programs, i. e., operators forward their results to exactly one successor.

B. Reordering Conditions for Pact Operators

UDFs are written in imperative code; hence, data processing systems are not aware of their exact semantics. In general, two Pact operators cannot be reordered if they have conflicting read-write or write-write accesses on any record field. Therefore, the optimizer depends, for each operator, on information about the fields that its UDF reads and writes in order to reason about valid transformations. This information is provided by so-called read and write sets. Transformations that involve grouping operators such as Reduce and CoGroup require in addition that the cardinality of input groups is not affected by the rewrite. For example, a Reduce may only be reordered with a Map operator, if the Map operator does not change the size of Reduce's input groups or the Map operator filters entire key groups, i. e., filters on the Reduce key. Therefore, the optimizer also requires information about lower and upper bounds of an operator's output cardinality in addition to read and write sets. We refer to our prior work for a detailed discussion and proofs of the reordering conditions [10].

C. Optimization of Pact Programs

Relational DBMS optimizers compile queries posed in a declarative language into physical execution plans. Often, the compilation process is divided into two stages. First, logical rewriting applies algebraic rewrite rules such as filter push-down, and resolving of nested subqueries to an algebraic representation of the query. Second, a physical execution plan is generated by replacing the logical operators of the rewritten algebraic query with physical operators[1]. Often, the selection of physical operators is based on execution cost estimates.

Pact programs differ significantly from declarative queries. Instead of algebraic expressions, Pact programs are trees of operators whose semantics are mostly unknown due to their UDFs. Hence, many techniques from traditional query optimization cannot be applied. Our optimizer for Pact programs operates in three steps. First, a static code analysis component peeks into the byte code of each UDF and extracts information that the optimizer requires to reason about valid transformations. Second, the optimizer enumerates all valid reordered data flow alternatives. Finally, a cost-based optimizer computes a physical execution plan for each alternative, and returns the plan with the least estimated execution costs. In the following, we discuss the aforementioned three steps in more detail.

Static Code Analysis: Our optimizer derives read and write sets as well as output cardinality bounds from UDFs by statically analyzing their imperative code. In a nutshell, the analysis exploits the fact that records can only be accessed, modified, and emitted via a restricted API[2]. Therefore, we can easily identify statements that read from, write to, or emit a record. The analysis algorithm uses control and data flow graphs provided by a static code analysis framework. Our approach guarantees safe, albeit conservative estimations of the actual read and write sets and output cardinality bounds, such that only valid reorderings will be considered by the optimizer. We refer the reader to reference [11] for a detailed description of the algorithm.

Order Enumeration: Using the information which was derived by static code analysis, the optimizer enumerates all valid reorderings for the given program. In contrast to traditional query optimization, the program is represented as a specific operator tree and not as an algebraic expression. Our enumeration algorithm is based on transformations that switch the order of two neighboring operators and uses top-down recursive descent. The algorithm computes all valid reorderings of a (sub)plan p in two steps. First, it computes all candidate root operators of p by enumerating all alternatives for p's subplans. Second, it recursively enumerates all alternative subplans for each root operator. We presented a detailed description of the enumeration algorithm in prior work [10].

Execution Strategy Selection: After all reordered alternatives of the Pact program have been enumerated, a cost-based optimizer is called for each alternative to compute an execution plan by choosing physical execution strategies. The set of considered execution strategies is well-known from traditional parallel databases and includes partitioning, broadcasting, external sorts, and merge- and hash-joins. The optimizer's cost model is a combination of the estimated network I/O, disk I/O, and CPU costs of UDFs. The Pact programming model provides an interface to specify compiler hints such as selectivity and CPU costs of a UDF to improve cost estimates. These hints can be manually annotated, set by

[1]Although join reordering is conceptually a logical rewrite, join orders are enumerated during physical optimization, since the optimal order depends on the selected execution strategies.

[2]Note that our API consists of basic operations which are also supported by other programming interfaces as for example Pig [1].

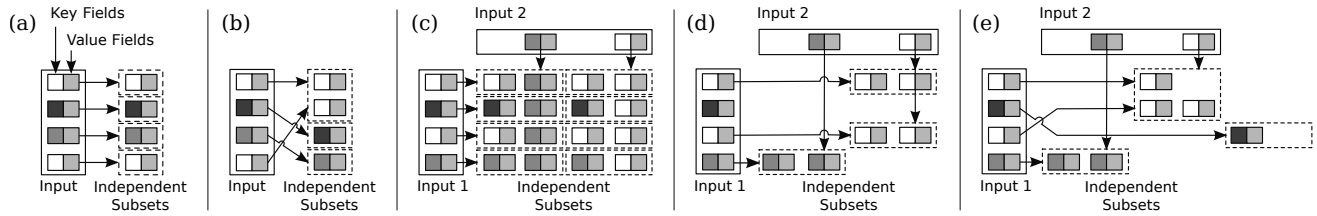

Fig. 1. (a) Map, (b) Reduce, (c) Cross, (d) Match, and (e) CoGroup second-order functions.

a higher-level language compiler, or possibly observed during runtime. During plan enumeration, the optimizer exploits the write set information derived by the static code analysis to reason about the propagation of interesting properties. A physical data property such as a partitioning on a field f is still present after an operator o was applied, if o did not modify f, i. e., f must not be in o's write set. Finally, the plan with minimum estimated costs is selected, and submitted for execution. Details on the selection of physical execution strategies can be found in prior work [12]. Note that it is possible to merge top-down order enumeration and cost-based execution strategy selection into a single step. We plan to integrate them in a later system release.

III. Demonstration

We demonstrate our optimizer for UDF data flows. The optimizer transforms data flows specified as Pact programs into physical execution plans, and executes them in parallel on the Nephele execution engine [17]. Our demonstration focuses on the visualization of the optimization process, i. e., we show in detail how the static code analysis component derives optimization information from UDF code, how reordered alternatives are enumerated, and show the physical execution plans that result from cost-based optimization. We provide a selection of Pact programs from relational and non-relational domains which demonstrate the salient features of our optimization approach. In the following, we present the Pact programs we provide for the demonstration and describe in detail how the optimization process is visualized.

A. Demonstrated Data Flows

We implemented a selection of relational analytical queries as Pact programs. Such tasks are commonly executed on massively parallel systems, as indicated by the popularity of higher-level languages for structured data analysis [4, 3, 5]. The provided programs resemble queries of the TPC-H benchmark and demonstrate that our approach is able to perform many rewrites which are known from relational optimizers such as filter reordering, bushy join-order enumeration, and limited forms of aggregation push-down. In addition to relational queries, we provide tasks from non-relational domains such as clickstream processing and data transformation as common in ETL workloads. These tasks show that our approach is able to reorder non-relational operators in data flows, a feature that we believe is unique among current systems.

B. Demonstration of the Optimization Process

We demonstrate a job submission client that visualizes the optimization process step-by-step. The demonstration starts by choosing a Pact program, providing program parameters, and submitting it to the optimizer. In the following we describe how the submission client visualizes static code analysis, operator order enumeration, the physical execution plans, and the parallel execution of the program.

Static Code Analyzer: The submission client lists all UDFs of the submitted program and shows the code of selected UDFs in typed 3-address code [18], a representation that is very suitable for code analysis. The client displays the information that was extracted from the UDF, i. e., read and write sets and bounds on the output cardinality, and visualizes how it was derived from the 3-address code by highlighting the relevant paths along the control and data flows. Figure 2 a) shows a screenshot of the static code analysis visualization.

Data Flow Optimizer: During the enumeration of reordered data flow alternatives, the optimizer memorizes all enumerated subplans. This information is read by the submission client and displayed in a browsable fashion such that the space of enumerated reordered alternatives can be explored by replacing any subplan with an equivalent alternative. For each operator its read and write set information is displayed to help reasoning about reordering conflicts. In addition, the client features a step-by-step visualization of the enumeration process, i. e., it shows all alternative plans in the order in which they were enumerated. Figure 2 b) is a screenshot of the plan enumeration visualization.

Physical Execution Plans: For each reordered data flow alternative, the cost-based optimizer computes the physical execution plan with least estimated costs by choosing data shipping and local processing strategies. The submission client shows the optimized physical execution plans for all alternatives ordered by their estimated cost. The visualization shows the chosen operator execution strategies, data properties (sorted, grouped, partitioned), and estimates such as data size, cardinality, number of UDF calls, and costs. Figure 2 c) shows a visualization of a physical execution plan.

Program Execution: Finally, a selected execution plan is submitted to the Nephele execution engine [17]. Nephele executes acyclic data flows consisting of sequential processing tasks which are connected via data transfer channels. Nephele splits the execution of a task into multiple subtasks which are distributed among compute nodes and executed in parallel. The submission client visualizes the execution as shown in Figure 2 d). It displays the parallel data flow graph, consisting of subtasks (colored rectangles) and their connections (lines). The status of a subtask (waiting, running, finished, failed) is indicated by its color. In addition, information about the aggregated resources consumption (CPU load, Memory usage, and

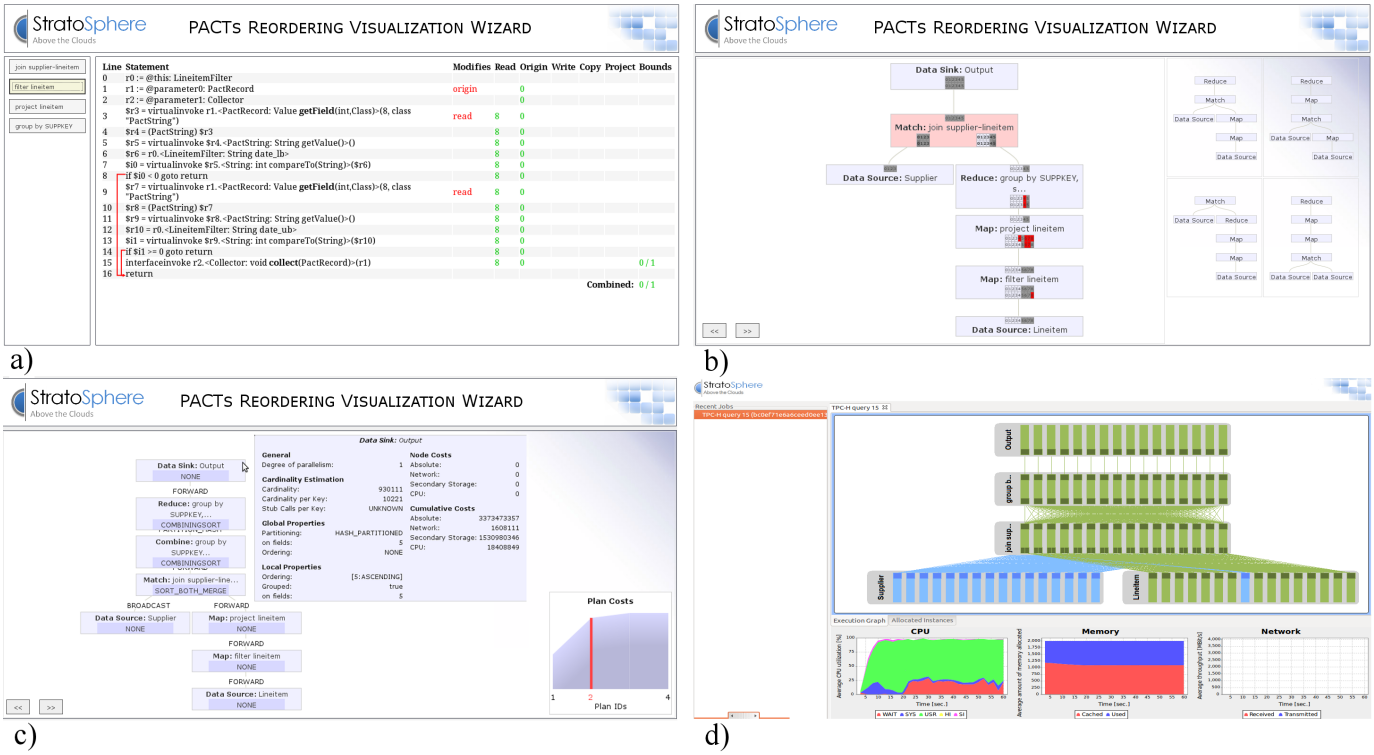

Fig. 2. Screenshots: (a) static code analysis, (b) operator order enumeration, (c) physical execution plans, (d) execution.

network traffic) of the cluster's compute nodes is displayed.

IV. Conclusion

We demonstrate the optimization of parallel data flows that contain MapReduce-style user-defined operators, an abstraction which is very popular and supported by several of today's parallel data processing frameworks. Our optimizer employs static code analysis to extract information from UDF code which is necessary to reason about operator reordering. We present a job submission client that visualizes all steps of the optimization process, i. e., code analysis, operator order enumeration, choice of physical execution plans, and finally the parallel execution. Providing a selection of relational and non-relational data flow programs, we demonstrate that our approach is able to resemble many optimizations which are known from relational optimizers but also includes non-relational operators into optimization.

Acknowledgments

This research was funded by the German Research Foundation under grant FOR 1036. We thank our coauthors from previous work [10], and the Stratosphere team.

References

[1] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *SIGMOD Conference*, 2008, pp. 1099–1110.

[2] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Y. Eltabakh, C.-C. Kanne, F. Özcan, and E. J. Shekita, "Jaql: A scripting language for large scale semistructured data analysis," *PVLDB*, vol. 4, no. 12, pp. 1272–1283, 2011.

[3] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "Scope: easy and efficient parallel processing of massive data sets," *PVLDB*, vol. 1, no. 2, pp. 1265–1276, 2008.

[4] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive - a warehousing solution over a map-reduce framework," *PVLDB*, vol. 2, no. 2, pp. 1626–1629, 2009.

[5] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong, "Tenzing a sql implementation on the mapreduce framework," *PVLDB*, vol. 4, no. 12, pp. 1318–1327, 2011.

[6] E. Friedman, P. M. Pawlowski, and J. Cieslewicz, "Sql/mapreduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions," *PVLDB*, vol. 2, no. 2, pp. 1402–1413, 2009.

[7] http://www.greenplum.com/technology/mapreduce.

[8] J. M. Hellerstein, "Optimization techniques for queries with expensive methods," *ACM Trans. Database Syst.*, vol. 23, no. 2, pp. 113–157, 1998.

[9] S. Chaudhuri and K. Shim, "Optimization of queries with user-defined predicates," *ACM Trans. Database Syst.*, vol. 24, no. 2, pp. 177–228, 1999.

[10] F. Hueske, M. Peters, M. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas, "Opening the black boxes in data flow optimization," *PVLDB*, vol. 5, no. 11, pp. 1256–1267, 2012.

[11] F. Hueske, A. Krettek, and K. Tzoumas, "Enabling rewrite optimizations of data flow programs through static code analysis," in *XLDI Workshop, affiliated with ICFP*, 2012.

[12] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke, "Nephele/pacts: a programming model and execution framework for web-scale analytical processing," in *SoCC*, 2010, pp. 119–130.

[13] http://stratosphere.eu.

[14] S. Chaudhuri and K. Shim, "Including group-by in query optimization," in *VLDB*, 1994, pp. 354–366.

[15] J. Zhou, P.-Å. Larson, and R. Chaiken, "Incorporating partitioning and parallel plans into the scope optimizer," in *ICDE*, 2010, pp. 1060–1071.

[16] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI*, 2004, pp. 137–150.

[17] D. Warneke and O. Kao, "Nephele: efficient parallel data processing in the cloud," in *SC-MTAGS*, 2009.

[18] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*. Pearson, 2006.