Enabling Operator Reordering in Data Flow Programs Through Static Code Analysis

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

Aljoscha Krettek
Technische Universität Berlin, Germany
aljoscha.krettek@campus.tu-berlin.de

Kostas Tzoumas
Technische Universität Berlin, Germany
kostas.tzoumas@tu-berlin.de

Abstract
In many massively parallel data management platforms, programs are represented as small imperative pieces of code connected in a data flow. This popular abstraction makes it hard to apply algebraic reordering techniques employed by relational DBMSs and other systems that use an algebraic programming abstraction. We present a code analysis technique based on reverse data and control flow analysis that discovers a set of properties from user code, which can be used to emulate algebraic optimizations in this setting.

1. Introduction
Motivated by the recent “Big Data” trend, a new breed of massively parallel data processing systems has emerged. Examples of these systems include MapReduce [8] and its open-source implementation Hadoop [1], Dryad [11], Hyracks [6], and our own Stratosphere system [5]. These systems typically expose to the programmer a data flow programming model. Programs are composed as directed acyclic graphs (DAGs) of operators, some of the latter typically being written in a general-purpose imperative programming language. This model restricts control flow only within the limits of operators, and permits only dataflow-based communication between operators. Since operators can only communicate with each other by passing sets of records in a pre-defined hardwired manner, set-oriented execution and data parallelism can be achieved.

Contrary to these systems, relational DBMSs, the traditional workhorses for managing data at scale, are able to optimize queries because they adopt an algebraic programming model based on relational algebra. For example, a query optimizer is able to transform the expression \( \sigma_{(x < 3)}(\alpha x \cdot (S \times T)) \) to the expression \( (\sigma_{R.X < 3}(R) \times S) \times T \), exploiting the associativity and commutativity properties of selections and joins.

While algebraic reordering can lead to orders of magnitude faster execution, it is not fully supported by modern parallel processing systems, due to their non-algebraic programming models. Operators are typically written in a general-purpose imperative language, and their semantics are therefore hidden from the system. In our previous work [10], we bridged this gap by showing that exposure of a handful of operator properties to the system can enable reorderings that can simulate most algebraic reorderings used by modern query optimizers. We discovered these properties using a custom, shallow code analysis pass over the operators’ code. Here we describe this code analysis in detail, which we believe is of interest by itself as an non-traditional use case of code analysis techniques. We note that our techniques are applicable in the context of many data processing systems which support MapReduce-style UDFs such as parallel programming models [5, 8], higher-level languages [6, 3], and database systems [9, 8].

Related work: In our previous work [10], we describe and formally prove the conditions to reorder user-defined operators. That paper also contains a more complete treatment of related work. Here, we focus on more directly related research. Manimal [12] uses static code analysis of MapReduce programs for the purpose of recommending possible indexes. Our code analysis can be seen as an example of peephole optimization [4], and some of the concepts may bear similarity to techniques for loop optimization. However, we are not aware of code analysis being used before for the purpose of swapping imperative blocks of code to improve performance of data-intensive programs.

The rest of this paper is organized as follows. Section 2 describes the programming model of our system, and introduces the reordering technology. Section 3 discusses our code analysis algorithm in detail. Finally, Section 4 concludes and offers research directions.

2. Data Flow Program Reordering
In our PACT programming model [5], a program \( P \) is a DAG of sources, sinks, and operators which are connected by data channels. A source generates records and passes them to connected operators. A sink receives records from operators and serializes them into an output format. Records consist of fields of arbitrary types. To define an operator \( O \), the programmer must specify (i) a second-order function (SOF) signature, picked from a pre-defined set of system second-order functions (currently Map, Reduce, Match, Cross, and CoGroup), and (ii) a first-order function (called user-defined function, UDF) that is used as the parameter of the SOF. The model is strictly second-order, in that a UDF is not allowed to call SOFs. The intuition of this model is that the SOF defines a logical mapping of the operator’s input records into groups, and the UDF is invoked once for each group. These UDF invocations are independent and can be thus scheduled on different nodes of a computing cluster.

Figure 1(a) shows an example of a PACT program. The data flow starts with two data sources \( \text{Src}_1 \) and \( \text{Src}_2 \) that provide records which have the fields [0, 1] and [3, 4] set respectively (the numbering is arbitrary). \( \text{Src}_1 \) feeds its data into a Map operator with a UDF \( f_1 \). The Map SOF creates an independent group for each input record, and \( f_1 \) is itself written in Java. UDF \( f_1 \) reads both fields of its input record (0 and 1), appends the sum of both fields as field 2, and emits the record. Similarly, the records of \( \text{Src}_2 \) are forwarded to a Map operator with UDF \( f_2 \) which sums the fields 3
Figure 1. Example Data Flows: (a) original order, (b) first reordered alternative, (c) second reordered alternative

and 4, appends the sum as field 5 and emits the record. The outputs of both Map operators are forwarded as inputs to a Match operator with a UDF \( f_3 \) and the key field \([0]\) for the first and \([3]\) for the second input. The Match SOF creates a group for each pair of records from both inputs that match on their key fields. \( f_3 \) merges the fields of both input records and emits a result. We give the pseudocode of all three user functions in the form of 3-address code [4] below.

30: \( f_3(\text{InRec } \$r_1, \text{InRec } \$r_2) \)
31: \( \$or:=\text{copy}(\$r_1) \)
32: union(\$or, \$r_2)
33: emit(\$or)

The pseudocode shows the UDF API to process PACT records. The user-functions \( f_1, f_2, \) and \( f_3 \) receive as input one or two input records of type InRec. The only way that a user function can emit an output record of type OutRec is by calling the emit(OutRec create()) function. Output records can be either initialized as empty (OutRec create()), or by copying an input record (OutRec copy(InRec)). Records can be combined via the function void union(OutRec, InRec). Fields can be read to a variable via Object.getField(InRec,int) addressed by their position in the input record. The value of a field can be set via void setField(OutRec, int, Object). Note that our record API is based on basic operations and similar to other systems’ APIs such as Apache Pig [2].

Figures 1(b) and (c) show potential reorderings of the original data flow (a) where either Map\( (f_1) \) or Map\( (f_2) \) has been reordered with Match\( (f_3, [0], [3]) \). While data flow (b) is a valid reordering, alternative (c) does not produce the same result as (a). In previous work, we presented conditions for valid reorderings of data flow operators centered around conflicts of operators on fields [11]. For example, since we know that \( f_1 \) reads fields 0 and 1, and \( f_2 \) reads fields 0 and 2, while \( f_3 \) reads fields 0 and 3, we can conclude that \( f_1 \) and \( f_2 \) only have a read conflict on field 0, and can thus be safely reordered. UDFs that have write conflicts cannot be reordered. This would be true if \( f_1 \) did not append the sum as field 2, but overwritten field 0 with the sum. Additional complications arise from the way output records are formed. Although on the first sight, \( f_1 \) and \( f_2 \) perform a very similar operation, i.e., summing two fields and appending the result, there is a fundamental difference. While \( f_1 \) creates its output record by copying the input record (line 14), \( f_2 \) creates an empty output record (line 24) and explicitly copies the fields of the input record (lines 25, 26). The side effect of creating an empty output record is that all fields of an input record are implicitly removed from the output. By reordering Map\( (f_2) \) with Match\( (f_3, [0], [3]) \), the fields 0, 1, and 2 will get lost since Map\( (f_2) \) does not explicitly copy them into the newly created output record.

The information that needs to be extracted from the user code in order to reason about reordering of operators is as follows. The read set \( R_f \) of a UDF \( f \) is the set of fields from its input data sets that might influence the UDF’s output, i.e., fields that are read and evaluated by \( f \). The write set \( W_f \) is the set of fields of the output data set that have different values from the corresponding input field. The emit cardinality bounds \([EC_f]\) and \([EC_f']\) are lower and upper bounds for the number of records emitted per invocation of \( f \). Reference [10] defines these properties more formally, and provides conditions for reordering operators with various SOFs given knowledge of these properties. In addition to change the order of operators, the optimizer can leverage these properties to avoid expensive data processing operations, e.g., a previously partitioned data set is still partitioned after a UDF was applied, if the partitioning fields were not modified by the UDF. Moreover, field projections can be pushed down based on read set information.

While it is very difficult to statically derive the exact properties by UDF code analysis in the general case, it is possible to conservatively approximate them. In reference [11] we discussed this static code analysis pass for the simple case of unary operators. In the next section, we provide the full algorithm that deals with the additional complexity due to binary operators, and provide detailed pseudocode.

3. Code Analysis Algorithm

Our algorithm relies on a static code analysis (SCA) framework to get the bytecode of the analyzed UDF, for example as typed three-address code [4]. The framework must provide a control flow graph (CFG) abstraction, in which each code statement is represented by one node along with a function \( \text{Preds}(s) \) that returns the statements in the CFG that are “true” predecessors of state- ment \( s \), i.e., they are not both predecessors and successors. Finally, the framework must provide two methods \( \text{DefUse}(s, \$v) \) and \( \text{UseDef}(s, \$v) \) that represent the Definition-Use chain of the variable \$v at statement \( s \), and the Use-Definition chain of variable \$v at statement \( s \) respectively. Any SCA framework that provides these abstraction can be used.

The algorithm visits each UDF in a topological order implied by the program DAG starting from the data sources. For each UDF \( f \), the function \( \text{VisitUDF} \) of Algorithm 1 is invoked. First, we compute the read set \( R_f \) of the UDF (lines 7-10). For each statement of the form \( \text{Stmt} := \text{getField}(\$r, n) \) that results in a valid use of variable \( \$t \) (DefUse($g, $t) \neq 0) we add field \( n \) to \( R_f \).

Approximating the write set \( W_f \) is more involved. We compute four sets of integers that we eventually use to compute an approximation of \( W_f \). The origin set \( O_f \) of UDF \( f \) is a set of input ids. An integer \( o \in O_f \) means that all fields of the \( o \)-th input record of \( f \) are copied verbatim to the output. The explicit modification set \( E_f \) contains fields that are modified and then included in the output. We generally assume that fields are uniquely numbered within the program (as in Figure 1). The copy set \( C_f \) contains fields that are copied verbatim from one input record to the output. Finally, the projection set \( P_f \) contains fields that are projected from the output, by explicitly being set to \( null \). The write set is computed from these sets using the function ComputeWriteSet (lines 15).

All fields in \( E_f \) and \( P_f \) are explicitly modified or set to \( null \) and therefore in \( W_f \). For inputs that are not in the origin set \( O_f \), we add all fields of that input which are not in \( C_f \), i.e., not explicitly copied.

To derive the four sets, function \( \text{VisitUDF} \) finds all statements of the form \( \text{emit}(\$or) \), which include the output record \$or in
algorithm 1 Code analysis algorithm

1: function Compute-Write-Set(f, O, E, C, F, P)
2:     \( W_f = E_f \cup P_f \)
3:     for \( i \in \text{Inputs}(f) \) do
4:         if \( i \notin O \) then \( W_f = W_f \cup (\text{Input-Fields}(f, i) \setminus C_f) \)
5:     return \( W_f \)
6: end function

7: function Visit-UDF(f)
8:     \( R_f = \emptyset \)
9:     \( G = \Phi \) : all statements of the form \( g:s=t\) getField(\( \$ir,n \))
10:    for \( g \) in \( G \) do
11:        if \( \text{def-use}(g, \$t) \neq \emptyset \) then \( R_f = R_f \cup \{n\} \)
12:    \( E = \Phi \) : all statements of the form \( e: \text{emit}(\$or) \)
13:    for \( e \in E \) do
14:        \( (O_e, E_e, C_e, P_e) = \text{Visit-Stmt}(e, \$or) \)
15:    return \( (R_f, O_e, E_e, C_e, P_e) \)
16: end function

17: function Visit-Stmt(\( e, \$or \))
18:     if \( \text{visited}(e, \$or) \) then
19:         return \( \text{Memo-Sets}(e, \$or) \)
20: end function

21: if \( s \) of \( \text{form} \$or = \text{create}() \) then \( \text{return} \left\{0, 0, 0, 0\right\} \)
22: if \( s \) of \( \text{form} \$or = \text{copy}(\$ir) \) then
23:     \( \text{return} \left(\text{Input-Id}(\$ir), 0, 0, 0\right) \)
24: end function

25: \( P_f = \text{PreDef}(s) \)
26: \( (O_e, E_e, C_e, P_e) = \text{Visit-Stmt}(e, \$or) \)
27: \( \text{for} \ p \in P_f \) do
28:     \( (O_e, E_e, C_e, P_e) = \text{Merge}(O_e, E_e, C_e, P_e), (O_e, E_e, C_e, P_e), (O_e, E_e, C_e, P_e) \)
29: end function

30: if \( s \) of \( \text{union}(\$or, \$ir) \) then
31:     \( \text{return} \left(\text{Input-Id}(\$ir), E_e, C_e, P_e\right) \)
32: end function

33: \( T = \text{UseDef}(e, \$t) \)
34: \( \text{for} \ t \in T \) do
35:     \( \text{return} \left(\text{Input-Id}(\$ir), E_e, C_e, P_e\right) \)
36: \( \text{return} \left(\text{Input-Id}(\$ir), E_e, C_e, P_e\right) \)
37: end function

38: if \( s \) of \( \text{setField}(\$or, n, \$t) \) then
39:     \( \text{return} \left(\text{Input-Id}(\$ir), E_e, C_e, P_e\right) \)
40: end function

41: \( C = \{E_1 \cup C_1, \varnothing \} \cup \{i \in C_1, \text{input-Id}(i) \in O_1\} \)
42: \( \cup \{i \in C_2, \text{Input-Id}(i) \in O_1\} \)
43: \( \text{return} \left(\text{Input-Id}(\$ir), E_e, C_e, P_e\right) \)

4. Conclusions and Future Work

We presented a shallow code analysis technique that operates on dataflow programs composed of imperative building blocks (“operators”). The analysis is a hybrid of reverse data flow and control flow analysis, and determines sets of record fields that express the data conflicts of operators. These sets can be used to “emulate” algebraic reorderings in the dataflow program. Our techniques guarantee safety through conservatism and are applicable to many data processing systems that support UDFs. Future work includes research on intrusive user-code optimizations, i.e., modifying the code of UDFs, and on the effects that the use of functional programming languages to specify UDFs has on our approach and possible optimizations.

Acknowledgments

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

References

[1] http://hadoop.apache.org/
[2] http://pig.apache.org/
[3] http://www.greenplum.com/technology/mapreduce
[4] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. Compilers: Principles, Techniques and Tools. Pearson, 2006.
[5] 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, pages 119–130, 2010.
[6] V. R. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In ICDE, pages 1151–1162, 2011.
[7] 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, 1(2):1265–1276, 2008.
[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In OSDI, pages 137–150, 2004.
[9] E. Friedman, P. M. Pawlowski, and J. Cieslewicz. SQL/mapreduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. PVLDB, 2(2):1402–1413, 2009.
[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, 5(11):1256–1267, 2012.
[11] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. pages 59–72, 2007.
[12] E. Jahani, M. J. Cafarella, and C. Ré. Automatic optimization for mapreduce programs. PVLDB, 4(6):385–396, 2011.