Spotting Code Optimizations in Data-Parallel Pipelines through PeriSCOPE

Zhenyu Guo[†] Xuepeng Fan[†£] Rishan Chen[†‡] Jiaxing Zhang[†] Hucheng Zhou[†]
Sean McDirmid[†] Chang Liu [†§] Wei Lin[*] Jingren Zhou[*] Lidong Zhou[†]
[†]*Microsoft Research Asia* [*]*Microsoft Bing* [‡]*Peking University*
[£]*Huazhong University of Science and Technology* [§]*Shanghai Jiao Tong University*

ABSTRACT

To minimize the amount of data-shuffling I/O that occurs between the pipeline stages of a distributed data-parallel program, its procedural code must be optimized with full awareness of the pipeline that it executes in. Unfortunately, neither pipeline optimizers nor traditional compilers examine both the pipeline and procedural code of a data-parallel program so programmers must either hand-optimize their program across pipeline stages or live with poor performance. To resolve this tension between performance and programmability, this paper describes PeriSCOPE, which automatically optimizes a data-parallel program's procedural code in the context of data flow that is reconstructed from the program's pipeline topology. Such optimizations eliminate unnecessary code and data, perform early data filtering, and calculate small derived values (e.g., predicates) earlier in the pipeline, so that less data—sometimes much less data—is transferred between pipeline stages. We describe how PeriSCOPE is implemented and evaluate its effectiveness on real production jobs.

1 INTRODUCTION

The performance of big data computations improves dramatically when they are parallelized and distributed on a large number of machines to operate on partitioned data [5, 14]. Such data-parallel programs involve pipelines of computation stages where I/O intensive data shuffling between these stages can dominate program performance. Unfortunately, data-shuffling I/O is difficult to optimize automatically because computations at each pipeline stage are encoded as flexible procedural code; current pipeline optimizers treat this code as a black box while compilers treat pipelines as black boxes and so are unaware of the data flow between the procedural code at different computation stages. The programmer must manually perform optimizations that require examining both the program's pipeline and procedural code; e.g., to not propagate unused data or to move the computation of smaller derived values to an earlier stage so less data is transmitted during data shuffling. Performing these optimizations by hand is not only tedious, it also limits code reuse from generic libraries.

So that programmers can write data-parallel programs with reasonable performance without sacrificing programmability, automatic optimizations must examine both the pipeline and procedural code of a data-parallel program. Common logical optimizations [8, 28, 34, 40, 43] for data-parallel programs focus on a high-level pipeline topology that is subject to relational query-optimization techniques. Unfortunately, at best relational components are extracted from procedural code into a relational optimization framework [20], which is limited by the inability of the relational framework to match the expressiveness of procedural code. We instead observe that projecting well-understood declarative pipeline properties into more flexible procedural code is intrinsically simpler than extracting declarative properties from procedural code. Such projection can then be used to reconstruct program data flow, enabling automatic optimizations of procedural code across pipeline stages that can improve I/O performance.

This paper presents PeriSCOPE, which automatically optimizes the procedural code of programs that run on SCOPE [8, 42], a production data-parallel computation system. PeriSCOPE connects the data flow of a SCOPE program's procedural code together by examining a high-level declarative encoding of the program's pipeline topology. PeriSCOPE then applies three core compiler-like optimizations to the program. *Column reduction* suppresses unused data in the pipeline based on the program's reconstructed data flow. *Early filtering* moves filtering code earlier in the pipeline to reduce how much data is transmitted downstream. Finally, *smart cut* finds a better boundary between pipeline stages in the data flow graph to minimize cross-stage I/O; e.g., the code that computes a predicate from two string values could be moved to an earlier stage, so that only a boolean value, rather than two string values, needs to be transmitted. The result is faster program execution because less data needs to be transferred between pipeline stages.

We have implemented PeriSCOPE and evaluated its effectiveness on 28,838 real SCOPE jobs from a large production cluster. We also evaluate end-to-end performance comparisons on eight real jobs.

The rest of the paper is organized as follows. Section 2 presents a sample SCOPE program to show the po-

```
1  t1 = EXTRACT query:string,clicks:long,market:int,...
2       FROM    "/users/foo/click_0342342"
3       USING   DefaultTextExtractor("-s")
4       HAVING  IsValidUrl(url) AND clicks != 0;
5  t2 = REDUCE  t1 ON query
6       PRODUCE query, score, mvalue, cvalue
7       USING   PScoreReducer("clicks")
8       HAVING  GetLength(query) > 4;
9  t3 = PROCESS t2 PRODUCE query, cscore
10      USING   SigReportProcessor("cvalue")
11      OUTPUT  t3 TO "/users/foo/click/0342342";
```

Figure 1: Declarative code that defines the pipeline of a sample SCOPE program. Rows of typed columns (line 1) are first extracted from a log file (line 2) using a default text extractor (line 3) and filtered based on certain conditions (line 4). Next, the input rows are *reduced* with a user-defined function `PScoreReducer` (line 7) to produce a table with four columns (line 6) after being filtered (line 8). Finally, the user-defined function `SigReportProcessor` (line 10) is applied to the result as it is emitted (line 11).

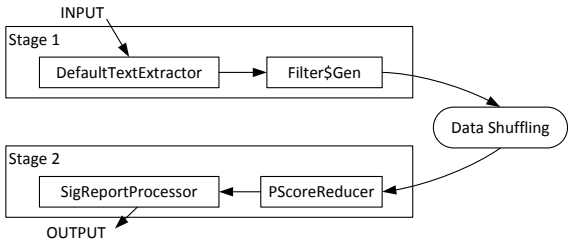

Figure 2: An illustration of the pipeline defined by the declarative code in Figure 1. The `Filter$Gen` operator is generated from the `HAVING` clauses on line 4 and 8 of Figure 1; other operators refer to user-defined functions. Each directed edge represents the data flow between operators.

tential benefits of PeriSCOPE's optimizations. The I/O-reduction optimizations in PeriSCOPE are described in Sections 3 and 4, with Section 3 covering column reduction and Section 4 discussing early filtering and smart cut, which are both forms of code motion. PeriSCOPE's implementation is covered in Section 5, followed by an evaluation in Section 6. We survey related work in Section 7 and conclude in Section 8.

2 A Motivating Example

We motivate PeriSCOPE by describing the pipeline-aware optimization opportunities that are found in a sample data-parallel program, which is adapted from a real SCOPE job. SCOPE is a distributed data-parallel computation system that employs a hybrid programming model where declarative SQL-like code describes a program's high-level pipeline structure, like other similar systems such as Hive [33], Pig [15], and DryadLINQ [40]. Figure 1 shows the declarative code of our sample job that is compiled into an execution pipeline, which we illustrate in Figure 2.

The operators of a SCOPE pipeline manipulate a data model of *rows* and *columns* and can be encoded as *user-defined functions* of procedural code that are either defined by the user or reused from generic libraries. A *computation stage* consists of one or more chained operators, and runs on a group of machines independently with partitioned data stored locally; *data-shuffling phases* then connect computation stages together by transmitting requisite data between machines. The pipeline in Figure 2 contains two *computation stages* that are separated by one data-shuffling phase according to the reduce call on line 5 in Figure 1. SCOPE applies logical optimizations, such as early selection, to programs according to the declarative structure of their pipeline. For example, the filtering clause on line 8 of Figure 1 can be applied before data shuffling; and so the `Filter$Gen` operator in the first stage of Figure 2 therefore includes the conditions from line 8 as well as line 4. Such logical optimizations apply only to the declarative code defined in Figure 1, treating the procedural code of the `DefaultTextExtractor`, `PScoreReducer`, and `SigReportProcessor` as black boxes.

The SCOPE program of Figure 1 is easily written by reusing two functions (`DefaultTextExtractor` and `SigReportProcessor`) from generic libraries while the encoding of the custom `PScoreReducer` function, shown in Figure 3, is straightforward. However, the program contains three serious I/O inefficiencies that need to be eliminated before it is "fast enough." First, the `if` statement on line 7 of Figure 3 is actually a procedural filter that discards rows. Such rows can be filtered out early so that they are not transmitted during the data-shuffling phase, which can be accomplished by splitting `PScoreReducer` into two parts as encoded in Figure 5: a `PScoreReducerPre` function that executes the computations of lines 5–7 in Figure 3 before data-shuffling; and a `PScoreReducerPost` that executes the rest of the computations from the original `PScoreReducer` function after data-shuffling. Our sample program's declarative SCOPE code is updated in Figure 4 to reflect this split, whose pipeline is illustrated in Figure 6.

Next, the `alteredQuery` column is transmitted only for computing a simple predicate on line 9 of Figure 3; the predicate computation can be done before the shuffling phase so that smaller boolean values are transmitted instead of strings. This is accomplished by computing the predicate in `PScoreReducerPre` on line 16 of Figure 5 and propagating its result as a column to

```
1  public class PScoreReducer : Reducer {
2  List<Row> Reduce(List<Row> input, string[] args){
3    maxImpr = 0; score = 0; mvalue = 0; cvalue = 0;
4    foreach (Row row in input) {
5     int impr = SmoothImpr(row[args[0]].Long());
6     bool incl = row["ctrls"].Contains(args[0]);
7     if (!incl && impr < 0) continue;
8     string[] keys = row["query"].Split(',');
9     bool p = row["alteredQuery"].ContainsAny(keys);
10     if (p)
11       score += ...;
12     if (impr > maxImpr)
13       maxImpr = impr;
14     if (impr * IMPR_RATIO > maxImpr) continue;
15     ... cvalue += ...
16     ... mvalue += ... row["market"] ...
17   }
18   outRow[1] = Normalize(score, ...);
19   outRow["mvalue"] = mvalue;
20   outRow["cvalue"] = cvalue;
21   ...
22   yield return outRow;
23 }}
```

Figure 3: The procedural code of the **PScoreReducer** user-defined function. Because **PScoreReducer** is a reduce operator, the preceding data shuffling ensures that rows having the same shuffling key are grouped together. For each group (**input**) of rows sharing the same shuffling key (line 2), this reduce operator iterates on each row in that group using a loop (lines 4-17) and outputs a single row as **outRow** for that group (line 22). The **impr** variable of line 5 represents an "improvement" that regulates accumulation of **mvalue** and **cvalue**.

PScoreReducerPost where it is used on line 29. An analogous transformation can be applied to **clicks**, which is used for computing **impr**, converting it from a long to an int.

Finally, the **SigReportProcessor** function called on line 10 of Figure 1 uses only the **cvalue** column, bound to its input parameter, that is computed by the **PScoreReducer** function; in contrast the **mvalue** column computed on lines 16 and 19 of Figure 3 is unused and therefore does not need to be computed and propagated in the **PScoreReducerPost** function of Figure 5. More importantly, if **mvalue** is eliminated, the **market** column does not need to be extracted in Figure 1 by **DefaultTextExtractor** and transmitted during the data-shuffling that is illustrated in Figure 2. Instead, the programmer can define their own specialized **MyTextExtractor** function (top of Figure 5) that does not extract and propagate the **market** column. The market column is also eliminated from Figure 4's declarative code.

The PeriSCOPE approach

The optimized sample program in Figures 4 and 5 executes more efficiently at the expense of programmability: it can no longer reuse the **DefaultTextExtractor**

```
1  t1 = EXTRACT query:string,clicks:long,~~market:int,~~...
2       FROM    "/users/foo/click_0342342"
3       USING   MyTextExtractor
4       HAVING  IsValidUrl(url) AND clicks != 0
5                               AND GetLength(query) > 4;
6  t2 = PROCESS t1 PRODUCE query, impr, ...
7       USING   PScoreReducerPre;
8  t3 = REDUCE  t2 ON query
9       PRODUCE query, score, ~~mvalue,~~ cvalue
10      USING   PScoreReducerPost
11 t4 = PROCESS t3 PRODUCE query, cscore
12      USING   SigReportProcessor("cvalue")
13      OUTPUT  t4 TO "/users/foo/click/0342342";
```

Figure 4: Optimized declarative code of our sample program; strike-through text is original code that is eliminated.

```
1  public class MyTextExtractor : Extractor {
2  List<Row> Extract(StreamReader reader, string[] args){
3    ...
4    string[] columns = line.Split(',');
5    ~~int market = int.Parse(columns[2]);~~
6    ~~outRow[2].Set(market);~~
7    ...
8  }}
9  public class PScoreReducerPre : Processor {
10 List<Row> Process(List<Row> input, string[] args){
11   foreach (Row row in input) {
12     int impr = SmoothImpr(row["clicks"].Long());
13     bool incl = row["ctrls"].Contains("clicks");
14     if (!incl && impr < 0) continue;
15     string[] keys = row["query"].Split(',');
16     outRow["p"] =
17       row["alteredQuery"].ContainsAny(keys);
18     outRow["impr"] = impr;
19     ...
20     yield return outRow;
21 }}
22 public class PScoreReducerPost : Reducer {
23 List<Row> Reduce(List<Row> input, string[] args){
24   maxImpr = 0; score = 0; mvalue = 0; cvalue = 0;
25   foreach (Row row in input) {
26     int impr = row["impr"].Int();
27     ~~bool incl = row["ctrls"].Contains("clicks");~~
28     ~~if (!incl && impr < 0) continue;~~
29     if (row["p"].Boolean()) score += ...;
30     if (impr > maxImpr) maxImpr = impr;
31     if (impr * IMPR_RATIO > maxImpr) continue;
32     ... cvalue += ...
33     ~~... mvalue += ... row["market"] ...~~
34   }
35   outRow["score"] = Normalize(score, ...);
36   ~~outRow["mvalue"] = mvalue;~~
37   outRow["cvalue"] = cvalue;
38   ...
39   yield return outRow;
40 }}
```

Figure 5: Optimized procedural code of our sample program.

function, the logic for the **PScoreReducer** function is now distributed into two sections that execute in different pipeline stages, while the optimizations are tedious as the programmer must carefully move code across pipeline stages. These optimizations should be automated but cannot be performed by either logical pipeline optimizers that treat user-defined functions as black boxes or by

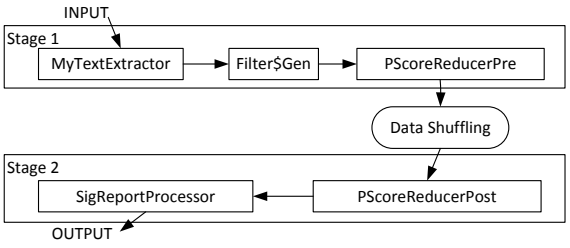

Figure 6: Resulting pipeline for the optimized job.

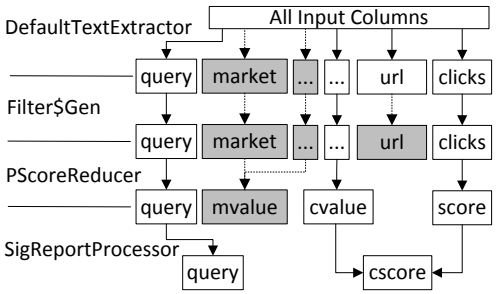

Figure 7: A simplified column-dependency graph for column reduction. Columns and computation in the shaded areas are removed by the column reduction optimization.

compilers that are unaware of pipelines.

PeriSCOPE automates such optimizations by considering both user-defined functions and the pipeline of a data-parallel program. In particular, PeriSCOPE reconstructs the data flow across the user-defined functions according to the pipeline topology and applies *column reduction* to remove unused columns along with the computations to compute them from user-defined functions; e.g., PeriSCOPE can eliminate the unused `mvalue` and `market` columns of our sample program. PeriSCOPE next identifies filtering conditions in a user-defined function and moves them earlier in the pipeline through *early filtering*; e.g., the `if` condition on line 7 of Figure 3 is moved earlier to reduce row propagation in the pipeline. Finally, PeriSCOPE applies *smart cut* that finds better boundaries between two stages to minimize data-shuffling I/O by moving size-reducing transformation upstream and size-enlarging transformation downstream in the pipeline. We describe how PeriSCOPE automates these optimizations in Sections 3 and 4.

3 Column Reduction

A user-defined function might not use a particular input column that is available to it in a calling pipeline. For example, the `SigReportProcessor` function of Section 2's sample program does not use the `mvalue` column of the pipeline encoded in Figure 1. In distributed data-parallel programs, transferring unused columns during data-shuffling can consume a significant amount of network I/O. As we discuss in Section 6, this problem commonly arises from the reuse of user-defined functions that we observe in production SCOPE jobs.

Column reduction is an optimization in PeriSCOPE that leverages information about how operators are connected together in a pipeline to eliminate unused columns from the program, removing their associated computation and I/O costs. The optimization analyzes the dependency information between the input and output columns of all operators in the pipeline; Figure 7 shows part of the column dependency graph for the example in Figure 1. An input or output column of an operator is represented as a vertex while an edge from a source column s to a destination column d indicates that d has either a data or control dependency on s. Only data dependency edges are shown in Figure 7 as control dependency edges are too numerous to illustrate clearly. Because SCOPE allows a column to be accessed by name (e.g., line 6 in Figure 3) or index (e.g., line 18), a column read or write operation may be *unresolved* during compilation, where PeriSCOPE considers that this column could be any column that is visible to the user-defined function, leaving no opportunity for column reduction. Fortunately, as we discuss in Section 5, column accesses that cannot be resolved through simple optimization techniques are relatively rare—at only a 1.9% occurrence in our survey of real SCOPE jobs.

PeriSCOPE applies column reduction by computing a set of "used" output columns for each operator that are used as the input columns of succeeding operators in the pipeline topology. If the operator immediately precedes a data-shuffling phase, the shuffling-key columns are also required as used output columns. Any unused output columns of an operator are removed and, if the operator is a user-defined function, PeriSCOPE also rewrites it to remove all code that only contributes to computing the removed output columns. If any columns were removed, PeriSCOPE removes any input columns that are no longer used because of removed code and repeats column reduction again.

For example, thecolumn `mvalue` is removed from the `PScoreReducer` function because it is not used by the `SigReportProcessor` function listed in Figure 3.

This causes the code that computes `mvalue` to be removed (lines 36 and 33 in Figure 5), which further causes the output column `market` to be removed from the `DefaultTextExtractor` function. Finally, PeriSCOPE creates, through specialization that eliminates code, function whose code is semantically similar to the `MyTextExtractor` function of Figure 5.

4 CODE MOTION

Code motion moves code from user-defined functions across pipeline stages using two techniques: *early filtering* and *smart cut* that respectively reduce the number of rows and the size of each row transmitted during data shuffling. Because such code motion can be done only if safe, i.e., the result of the program is unchanged, we describe the correctness conditions of code motion using the example in Figure 3, with a focus on identifying the domain knowledge that is needed to define correctness.

Moving a statement across a data-shuffling phase is not always safe. For the statements on lines 12–14 in Figure 3, the value of `maxImpr` depends on the processing order of the input rows. Because data shuffling re-orders rows based on a shuffling key, computing `maxImpr` before and after data shuffling would yield different results. PeriSCOPE makes the following two observations on data shuffling, each leading to a safety rule for code motion. First, the data-shuffling phase reads the shuffling-key columns of each row, leaving other columns untouched; i.e.,

RULE 1. *PeriSCOPE must not move a statement after data shuffling if it generates shuffling-key columns.*

Second, the data-shuffling phase can change transparently the number and order of the rows processed on each machine through re-partitioning and grouping. Any computation that relies on the number or order of the rows, which we say is *stateful*, cannot be moved across the data-shuffling phase; i.e.,

RULE 2. *PeriSCOPE must not move a stateful statement across the data shuffling phase.*

PeriSCOPE applies loop dependency analysis [4] to the body of the main loop for each user-defined function to identify stateful statements as those that have loop-carried dependencies. A loop-carried dependency indicates that the destination statement relies on the execution of the source statement from an earlier iteration. For example, the statement on line 12 of Figure 3 relies on its reassignment on line 13 from previous iterations and is therefore stateful; by similar reasoning, the statements on lines 11, and 13–15 are also stateful.

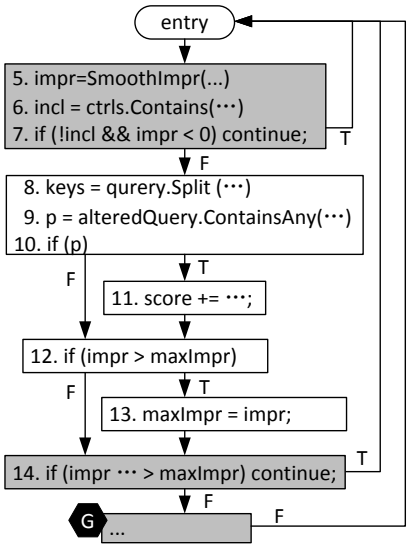

Figure 8: Control flow graph for the loop body in `PScoreReducer` in Figure 3. Edges marked T and F are branches that are taken when the last predicate in the source basic block evaluates to true and false, respectively. The vertices in gray are the basic blocks that contain filtering statements.

Care must be taken in identifying stateful statements. A statement might be stateful if it calls a routine that transparently accesses data; such as by reading and writing a global variable. Also, statefulness is only determined by dependencies on rows that are iterated by the main loop of a user-defined function; PeriSCOPE treats inner loops as single statements.

Early Filtering

Early filtering is applied to the first user-defined function in a computation stage that executes after a data shuffling phase. PeriSCOPE first identifies filtering statements in the user-defined function's main loop, which are statements that branch back to the beginning of the main loop. Figure 8 shows the control flow graph for the loop body of `PScoreReducer` in Figure 3; statements 7, 14 and the end of basic block G are identified as filtering statements. Because earlier filtering will reduce the number of rows that are iterated on later, PeriSCOPE must ensure that moving a filtering statement does not change the cumulative effect of any downstream stateful statements; i.e.,

RULE 3. *PeriSCOPE must not move a filtering statement before a data shuffling phase if the statement is, or is reachable from, a stateful statement.*

This rule excludes statement 14 in Figure 8 because it is stateful, and excludes the last statement in basic block G because it is reachable from statement 14. PeriSCOPE next identifies code that computes the filtering condition by applying backward slicing [35], which starts from the identified filtering statement and collects, as its *backward slice*, the statements that can affect it. The backward slice of statement 7 in Figure 8 includes statements 5–7. PeriSCOPE then copies the entire backward slice upward causing rows to be filtered out before data shuffling occurs. Given rule 3, the slice cannot contain any stateful statements and so this copy is always safe. Finally, the conditions of the moved filter can now be assumed in the original user-defined function, enabling the removal of code through dead code elimination. For the code in Figure 8, statement 7 is removed because `(!incl && impr < 0)` is always false; no row otherwise is permitted past the data-shuffling phase due to early filtering. Statement 6 is then removed because `incl` is not used anymore, causing `ctrls` to become unused in the user-defined function. As a result, early filtering not only reduces the number of rows that are transferred across a data shuffling phase, but can also trigger column reduction (e.g., on `ctrls`).

Smart Cut

The cross-stage flow of data across the network in a data-parallel program is significantly more expensive than a traditional program whose data flows only through memory. PeriSCOPE therefore aims at re-partitioning the code by finding *smart cuts* as shuffling I/O boundaries that minimize cross-stage data flow. Finding smart-cuts can be formulated as a compiler-like instruction scheduling problem [2, 25]. However, while a compiler usually rearranges instructions to improve instruction-level parallelism on a specific CPU architecture, smart cut reorders statements to reduce the amount data transmitted across the network.

Smart cut is applied to user-defined functions that are immediately adjacent to data-shuffling phases. PeriSCOPE first applies if-conversion [3] to the body of the main loop for a given user-defined function so that the loop body becomes a single basic block, which is necessary because instruction scheduling can only be applied to blocks of non-branching instructions. Figure 9 shows the simplified result for the code segment on lines 5–15 of Figure 3, after lines 6 and 7 are removed according to early filtering. Every statement is now guarded with a predicate that specifies the path condition of its execution; e.g., the statement on line 13 is guarded with predicate `p1` because it is executed only when `p1` is true.

```
5   (T)  impr = SmoothImpr(row["clicks"].Long());

8   (T)  keys = row["query"].String().Split(',');
9   (T)  p = row["alteredQuery"].ContainsAny(keys);

11  (p)  score += ...;
12  (T)  p1 = impr > maxImpr;
13  (p1) maxImpr = impr;
14  (T)  p2 = !(impr * IMPR_RATIO > maxImpr);
15  (p2) ... cvalue += ...
```

Figure 9: Simplified if-conversion result for lines 5–15 in Figure 3. `T` stands for `True` which means that the statement always executes.

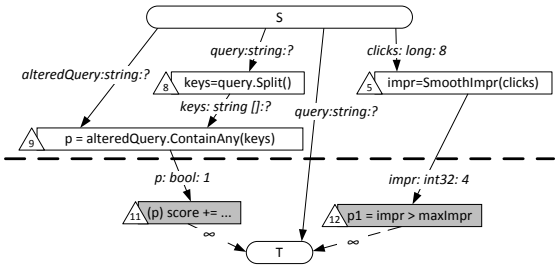

Figure 10: Labeled data dependency graph with a smart cut. Statements 13–15 are omitted. Statements in gray are stateful.

PeriSCOPE then builds a data dependency graph [2, 25] for this basic block using the SSA [13] format. Vertices in the data dependency graph are instructions, while directed edges represent read-after-write (RAW) data dependencies where sink instructions use variables defined in the source instructions. PeriSCOPE labels the edges with the name and byte size of the dependent variables, which are either columns or local variables. Figure 10 shows part of the labeled data dependency graph for our example; PeriSCOPE further adds two vertices `S` and `T` to represent the overall input and output of this code snippet, respectively. PeriSCOPE also adds an edge labeled `query` from `S` to `T` as `query` is used as the shuffling key and should always be transmitted. To ensure that rules 1 and 2 are not violated, PeriSCOPE adds directed edges from `S` to any statement that is either stateful or generates shuffling keys before the data-shuffling phase, and adds directed edges from any stateful statement after the data-shuffling phase to `T`; all of these edges have an infinite weight to ensure that those statements are never moved across the data-shuffling phase.

The smart-cut problem is now reduced to one of finding an edge cut between `S` and `T` in the data dependency graph that minimizes the total byte size of all dependent variables on edges across the cut. The problem appears

similar to the minimum cut problem [10] of a directed graph. However, there are two subtle differences between our problem and the standard minimum cut problem:

- Data flows across the data-shuffling phase in only one direction, so all edges must have the same direction across the cut.
- When multiple edges on a cut are labeled with the same variable name, the byte size of that variable is only counted once as it only needs to be transferred once.

Computing an optimal edge cut statically is difficult because the precise weights of some edges depend on dynamic data. For example, it is hard to statically estimate the weights of string-typed columns and variables as their length is unknown. In practice, PeriSCOPE resorts to a simple heuristic-based technique to identify opportunities to move code across data-shuffling phases. Specifically, PeriSCOPE looks for a simple pattern with a variable computed from one or more input columns. If the total size of the input columns that are used only for computing this variable is larger than the size of this variable, this computation should be moved to an earlier stage. Similarly, PeriSCOPE also looks for a reverse pattern where a variable is used to generate one or more output columns. In Figure 10, the input columns `alteredQuery` and `query` from Figure 3 are used to compute variable `p` in the optimize function of Figure 5. Although the `alteredQuery` column is never used elsewhere, the `query` column is used in a later stage. Because the byte size of a string type (`alteredQuery`) is always larger than that of a boolean variable (`p`), the cut should cross the edges labeled with `p`, instead of those labeled with `alteredQuery`. The same reasoning applies to the computation of `impr` from `clicks`. In the end, edges between statements 9 and 11, and between statements 5 and 12, are selected for the smart cut.

Finally, PeriSCOPE applies instruction scheduling according to the selected cut. In our example (Figure 9), statements 5, 8 and 9 are moved before data shuffling. The recorded schema across the data-shuffling phase is changed accordingly where two new columns are added: the boolean-typed `p`, and the integer-typed `impr`, and two old columns (the string-typed `alteredQuery` and the long-typed `clicks`) are deleted. The result is similar to the code shown in Figures 4 and 5 in Section 2.

5 IMPLEMENTATION

PeriSCOPE examines a SCOPE program's operators, the definition of the rows used by the operators, and the program's pipeline topology represented as a directed acyclic graph (DAG) in the program's execution plan. The operators and row definitions are extracted from .NET binary executables, while the pipeline topology is represented as an XML file. PeriSCOPE extends ILSpy [32], a de-compiler from .NET byte-code to C# code, and Cecil [36], a library to generate and inspect .NET assemblies, to implement PeriSCOPE as two components. PeriSCOPE's *optimizer* is built on top of ILSpy to specialize all operators in the input execution plan, applying all PeriSCOPE's optimizations to operators (as user-defined functions) as found at the intermediate representation (IR) level. The *generator* emits new bytecode for user-defined functions and generates all utility code for the program, such as new row schemas and their related serialization routines, as well as the new SCOPE description file for the execution plan.

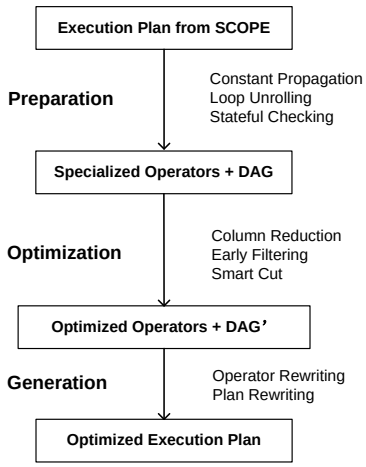

Figure 11: Optimization flow in PeriSCOPE.

The optimizer and generator components are both implemented in C# with 7,334 and 2,350 lines of code, respectively. Figure 11 illustrates PeriSCOPE's optimization flow with three major tasks, each containing several steps, where the optimizer performs the first two tasks, while the generator performs the last. Plan rewriting updates the original DAG XML file that describes pipeline topology because some original operators are now split into different computation stages.

SCOPE user-defined functions use a column index or name to access a column. When such index is a variable, program analysis can only make conservative assumption on what column is being manipulated, significantly reducing the opportunity for PeriSCOPE's optimizations. In our survey of SCOPE programs, we found

that 11.4% of the column accesses use variable indices. Fortunately, our investigation shows that many column index variables are determined by the input arguments to the user-defined functions, which is the case on lines 7 and 10 of Figure 1, and constant propagation can be applied to resolve their concrete value. However, not every column index can be resolved by constant propagation only. A user-defined function might enumerate columns using a variable in a loop, which we addressed by standard loop unrolling [2, 25] when the schema describing the data is known. A column index could also be determined by the value of another column, which PeriSCOPE cannot deal with very well. Fortunately, we have found that among the 11.4% of the jobs that use a variable as a column index, PeriSCOPE is able to resolve 83.3% of them, leaving just 1.9% of the jobs containing unresolved column access.

Instead of directly rewriting operator code, PeriSCOPE copies operator code when it needs to be written because a user-defined function can be reused multiple times in a job, each reuse requiring different code transformations. Likewise, row type schema definitions and serialization code are copied and transformed as columns are eliminated from different points in the pipeline.

6 EVALUATION

We use a real trace of 28,838 jobs from a 10,000 machine SCOPE production cluster to evaluate PeriSCOPE's core I/O reduction optimizations of column reduction, early filtering, and smart cut. Our evaluation focuses on first assessing the overall potential for these optimizations and second evaluating in detail the effectiveness of these optimizations on the end-to-end performance of several real production jobs. With an average analysis time of 3.9 seconds for each job, our current implementation successfully analyzes 19,914 (69%) of the 28,838 jobs. PeriSCOPE fails on the rest of these jobs given limitations in our implementation primarily relating to inconsistent SCOPE versions (18.9%) or outright ILSpy decompilation failures (8.5%), but a minority involve code that cannot be analyzed in general due to unresolved column indices (1.9%) or for reasons that we have yet been unable to determine (1.7%). Table 1 shows that 14.05% of the jobs are eligible for column reduction optimization, 10.47% for early filtering, and 5.35% for smart cut. Some jobs are eligible for multiple types of optimizations, and so the total percentage (22.18%) of jobs that are eligible for those optimizations is lower than the sum of the three.

We next examine the user-defined functions of these jobs. We found that these jobs used only 2,108 unique user-defined functions, meaning many jobs are encoded purely in declarative code that leverages pre-existing user-defined functions. About 16.4% of the user-defined functions are reused more than ten times, where the most popular user-defined function is reused 4,076 times. We suppose that the heavy reuse of user-defined functions creates more opportunities for PeriSCOPE's optimizations. And in fact, about 80.2% of the user-defined functions in jobs eligible for column reduction were reused at least 13 times, confirming our speculation that generic library functions contain a lot of redundancies that can be optimized away. On the other hand, no such correlation is observed for early filtering or smart cut, whose eligibility appear to be unrelated to reuse. Finally, 637 (30.2%) unique user-defined functions used in these jobs have arguments in their function bodies that are used as branch conditions or column names, while 79.1% of the user-defined function invocations in the job scripts contain constant parameters. Specialization of such user-defined functions is a necessary pre-processing step to resolve columns and apply PeriSCOPE's optimizations.

optimization	eligible jobs
column reduction	4,052 (14.05%)
early filtering	3,020 (10.47%)
smart cut	1,544 (5.35%)
Total	6,397 (22.18%)

Table 1: Optimization coverage statistics which lists the number and the percentage of the jobs that are eligible for the given optimization.

Case Study

To understand the overall effectiveness of PeriSCOPE's optimizations, we compare the performance of the jobs before and after our optimization in terms of both execution time and the amount of I/O used during the data-shuffling phase. Ideally, we would carry out this experiment with representative benchmarks, which unfortunately do not exist. We therefore select eight real and typical SCOPE jobs that are eligible for at least one of PeriSCOPE's optimizations and whose input data is still available on the cluster. The selected jobs are mostly related to web-search scenarios that process crawler traces, search query histories, search clicks, user information, and product bidding logs. Our experiment executes these real production jobs (cases 1–8 in Figure 12) on various number of machines. Specifically, cases 1, 2, and 4 use 1,000 machines, case 3 uses 10 machines, cases 5–7 use 192 machines, while case 8 uses 100 machines.

Figure 12 shows the performance-gain breakdown for

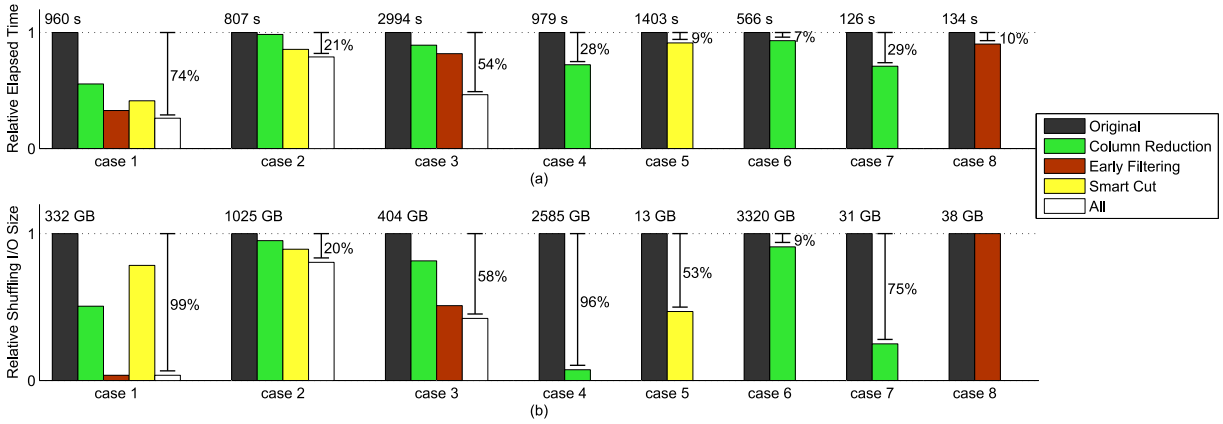

Figure 12: Performance gains with PeriSCOPE's column reduction, early filtering, and smart cut optimizations; chart (a) labels unoptimized job time in seconds while chart (b) labels total unoptimized job shuffling I/O size in GB; the bars in each case represent the effectiveness of each optimization relative to unoptimized execution time (a) or shuffling I/O (b); shorter bars indicate more reduction; the "All" bar is only shown for cases that are eligible for more than one PeriSCOPE's optimization; final case reduction of time or I/O is presented as a percentage next to an `I`-bar; both the execution time and the shuffling I/O are average values with a relative standard deviation (RSD) ranging from 7.3% to 23.0% due to the nature of our shared computing environment.

our chosen 8 production jobs in terms of a reduction in both execution time and data-shuffling I/O. The unoptimized and optimized versions of each job are executed three times; we report the average. Due to the nature of our shared computing environment we are using, we see high relative standard deviations (7.3% to 23.0%) in our latency experiments, while the reduction numbers in data-shuffling I/O is a more reliable indicator. In particular, highest standard deviations are seen for cases 5 (23.0% and 22.6%) and 6 (18.0% and 14.9%), indicating that the reductions are insignificant statistically in those cases. The execution time reduction for case 8 (10%) is also statistically insignificant with standard deviations of 13.4% and 7.3%. Case 1 benefits from all three of PeriSCOPE's optimizations, cases 2–3 are eligible for two, while cases 4–8 are only eligible for one each. PeriSCOPE reduces data-shuffling I/O in all cases but the last by between 9% and 99%; the last case incurs no benefit for reasons discussed below. Execution time is reduced by between 7% to 74%, which, beyond data-shuffling I/O, includes other tasks such as executing data-processing code, and reading and writing data to and from storage. Case 4 is particularly sensitive to storage overhead as this job extracts data from a 2.26 TB log file.

Column reduction can be applied six of the eight jobs, yielding I/O reductions ranging from 4.8% up to 96% that depend on how many columns are removed compared to the total byte size of all columns. Column reduction on case 4 removes 18 columns out of 22; the reducer that executes immediately after an extractor uses only 4 of the columns extracted. For case 7, only 2 out of 31 columns are used by its reducer; other columns are consumed by other operators and are not transmitted across the data-shuffling phase.

The effectiveness of early filtering depends highly on the goal of filtering. We have found that filtering conditions simply exclude rows whose columns have invalid values. While such case is rare, early filtering leads only to a negligible I/O reduction; case 8 is exactly this case. The execution time of case 8 is still reduced because PeriSCOPE moved the filtering computation to before the data-shuffling phase, improving the parallelism because more resource (136 CPU cores) are allocated to the stage before shuffling than after (42). When the filtering does not check for invalid values, they usually exclude a large number of rows and early filtering is quite effective. As an extreme case, data-shuffling I/O is reduced by 99% in case 1 because the vast majority of the rows in this job are filtered out and so do not need to be transmitted in the pipeline. The opportunity for early filtering discovered by PeriSCOPE was not obvious: *7* `if` conditions, some of them deeply nested, select desired rows for various computations, and manually writing a single filtering condition to replicate these `if` conditions is not trivial for a developer.

In contrast to early filtering, smart cut will always deliver I/O reductions when it can be applied. Compu-

tations that trigger smart cut typically involve one column that is mapped to a column of a smaller size, usually via the conversion from string to some arithmetic types, or size-reduction operations such as *Trim* and *Substring*. Binary operations (e.g., $+, *, ==, >$) between two input columns can also trigger smart cut. For example, case 5 contains two string-typed columns as start and end event timestamps; the job parses the two as integer timestamps and computes their delta for the elapsed time of the event, where smart cut causes the delta to be precomputed.

Discussion

Overall, we found that column reduction and smart cut are always effective in reducing data-shuffling I/O while the effectiveness of early filtering highly depends on the purpose of the filtering. Our experiments have also demonstrated that programmers often write inefficient data-parallel programs; we speculate that they are either unaware of how to optimize these programs or are valuing programmability over performance. In this context, PeriSCOPE's optimizations are valuable as the programmer can be less concerned about I/O performance.

Even experienced programmers who value performance could eventually rely on PeriSCOPE's optimizations to avoid hand-optimizing their code and allow them to reuse more existing code in their programs. In this case, the reliability and predictability of PeriSCOPE's optimizations are as important as the optimizations' effectiveness; we leave an exploration of this topic to future work.

7 Related Work

PeriSCOPE is closely related to a large body of research in the areas of data-parallel computation, distributed database systems [16] and query optimizations, and compiler optimizations [2, 4, 25]. Instead of attempting to cover those areas thoroughly, we focus on the most related research that lies in the intersection of those three areas.

Distributed data-parallel systems

MapReduce [14] has inspired a lot of follow-up research on large-scale distributed data-parallel computation, including Hadoop [5] and Dryad [18]. The MapReduce model has been extended [38] with Merge to support joins and adapted [11] to support pipelining. High-level languages for data-parallel computation have also been proposed for ease of programming. Examples include Sawzall [31], Pig Latin [29, 15], SCOPE [8], Hive [33, 34], and DryadLINQ [40]. In addition, FlumeJava [9] is a Java library for programming and managing MapReduce pipelines that proposes new parallel-collection abstractions, does deferred evaluation, and optimizes the data flow graph of an execution plan internally before executing. Nova [27] is a work-flow manager with support for stateful incremental processing which pushes continually arriving data through graphs of Pig programs executing on Hadoop clusters. Cascading [7] is a Java library built on top of Hadoop for defining and executing complex, scale-free, and fault tolerant data processing work-flows. Bu et al. [6] shows how recursive SQL queries may be translated into iterative Hadoop jobs. Programs in those systems go through a compilation and optimization process to generate code for a low-level execution engine, such as MapReduce and Dryad. All of them support user-defined functions that are treated as black boxes during optimization of the program's pipeline.

PeriSCOPE's optimizations work at the level of bytecode operators and pipeline descriptions, which are typically the result of the existing compilation and optimization process. Conceptually, the approaches taken by the PeriSCOPE's optimizations can be applied to data-parallel systems other than SCOPE, because almost all systems produce a pipeline with operators that call user-defined functions. The coverage and the effectiveness of the concrete optimizations, however, vary due to their different programming models and language runtime implementation. We show two cases where the differences in those systems matter. First, the data models differ, ranging from a relational data model (e.g., SCOPE) or its variations (e.g., Hive, Pig), to the object model (e.g., FlumeJava and DryadLINQ), which introduces different opportunities and difficulties for PeriSCOPE's optimizations. For example, with an object model, PeriSCOPE does not need to resolve the column access index any more, because all fields are accessed explicitly. Also, in an object model, declaring a new schema requires explicit class/object definitions. The resulting inconvenience often cause developers to reuse existing object definitions that contains unneeded fields, offering more opportunities for column reduction. Developers sometimes write custom (de-)serialization functions for an object to achieve better performance, which would pose challenges to PeriSCOPE's optimizations that cause schema changes: those functions must be modified accordingly.

Second, different systems might define different interfaces to their user-defined functions; those interfaces represent different trade offs between expressiveness and ease of analysis. For example, SCOPE exposes a collection of records to a mapper while others usually take a single record as the input to a mapper (e.g., in the

MapReduce framework in Hadoop). Other examples include the reducer interface in SCOPE versus the UDAF (user-defined aggregation function) interface in Hive, where the former exposes the record collection and the latter only receives a single value, and is usually applied to a single column. The more restricted the interface and the less expressive the language, the easier it is to analyze. The interface definition also influences where the optimization opportunities lie. For example, if a user-defined function is defined to take a single column as its input, cross-column relationships are now explicitly expressed, reducing the need for program analysis and optimizations.

Database optimizations

Most of the data-parallel systems adopt a hybrid programming model that combines declarative relational operators with procedural user-defined functions, and are heavily influenced by database systems. The support of relational operators in those systems allows jobs to be specified easily, while at the same time facilitates database optimizations based on relational algebra.

There are interesting similarities between some of the PeriSCOPE's optimizations and the classic database optimizations. Early filtering in PeriSCOPE corresponds naturally to early selection in database optimizations. The counterpart to column reduction in database optimization is early projection that drops unused columns as early as possible. Such logical optimizations [28] have already been proposed for data-parallel programs, but they cannot be readily applied when user-defined functions are involved because they rely on relational operators.

A line of related research focuses on extracting relational queries from user-defined functions. HadoopToSQL [19] transforms MapReduce queries to use the indexing, aggregation, and grouping features provided by SQL databases, taking advantage of advanced storage engines by employing symbolic execution to extract selection and projection conditions. Manimal [20] similarly extracts relational operations such as selection, projection, and data compression from user-defined functions through static data flow analysis. Early filtering and column reduction are possible in Manimal because those optimizations have clear relational interpretations. But PeriSCOPE can also remove unnecessary code from the user-defined functions, while Manimal never rewrites user-defined functions as it can only optimize the relational layer. For example, lines 27, 28, 33, 36 in Figure 5, which are eliminated by PeriSCOPE, would not be removed by Manimal. As a result, PeriSCOPE further removes columns `ctrls` and `market`, as well as the code on lines 5 and 6 in Figure 5 through column reduction, which Manimal cannot do.

Neither Manimal nor HadoopToSQL support smart cut because neither system rewrites any user-defined functions in its optimizations. For smart cut, the closest concept in database optimization that we are aware of is the notion of "virtual columns" from Oracle [30], where a computed column is lazily evaluated when it is used, similar in spirit to moving a computation to a later place in the code. Such lazy evaluation is limited to special cases and cannot be performed on user-defined functions in general.

Program analysis and optimizations

The need to analyze user-defined functions, by means of techniques such as data flow analysis [2, 4, 25], abstract interpretation [12], and symbolic execution [17], has already been recognized. Ke et al. [21] focuses on data statistics and computational complexity of user-defined functions to cope with data skew. Sameer et al. [1] concludes that certain data and code properties can improve performance of data-parallel jobs, and presents the RoPE system that adaptively re-optimizes jobs by collecting statistics on such code and data properties in a distributed context. Scooby [37] analyzes the data flow relationships of SCOPE's user-defined functions between input and output tables, such as column independence and column equality, by extending the Clousot analysis infrastructure [22]. Yuan et al. [39] define the *associative-decomposable* property of a reducer function to enable partial aggregation automatically after analysis on the reducer functions. Sudo [41] identifies a set of interesting user-defined functions, such as pass-through, one-to-one, and monotonicity, and develops a framework to reason about data-partition properties, functional properties, and data shuffling in order to eliminate unnecessary data shuffling. Sudo analyzes user-defined functions to infer their properties, but never rewrites any user-defined functions.

Compilation of declarative language has huge impact on the efficiency of a high-performance and high-throughput environment. Steno [26] can translate code for declarative LINQ [24, 23] queries both in serial C# programs and DryadLINQ programs to type-specialized, inlined, and loop-based procedural code that is as efficient as hand-optimized code. PeriSCOPE similarly applies those optimizations in program specialization as a preparation step, although differences in the language designs between SCOPE and LINQ lead to different challenges and approaches. Steno can automatically generate

code for operators expressed in LINQ, but has to treat external functions called inside operators as black boxes. PeriSCOPE instead works with compiled user-defined functions, which include such external functions.

8 Concluding Remarks

Optimizing distributed data-parallel computation benefits from an inter-disciplinary approach that involves database systems, distributed systems, and program languages. In particular, PeriSCOPE has demonstrated performance gains on real production jobs by applying program analysis and compiler optimizations in the context of the pipelines that these jobs execute in. Much more can be done. We can explore how to enhance the reliability and predictability of PeriSCOPE's optimizations so programmers can reuse existing code as well as write straightforward code without much guilt that performance is being sacrificed. Going further, we can explore how the programming model itself can be enhanced with more guarantees about program behavior, allowing for even more aggressive optimizations that further improve performance.

Acknowledgement

We thank the anonymous reviewers for their valuable comments. We are particularly grateful to our shepherd, John Wilkes, for his insightful feedback.

References

[1] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Reoptimizing data parallel computing. In *Proceedings of the 7th Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, USA, 2012. USENIX Association.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th SIGACT-SIGPLAN symposium on Principles of programming languages (POPL)*, pages 177–189, Austin, Texas, 1983. ACM.

[4] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001.

[5] Apache. Hadoop: Open-source implementation of MapReduce. `http://hadoop.apache.org`, March 2012.

[6] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient iterative data processing on large clusters. *The Proceedings of the VLDB Endowment (PVLDB)*, 3:285–296, 2010.

[7] Cascading. `http://www.cascading.org/`, March 2012.

[8] 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. *The Proceedings of the VLDB Endowment (PVLDB)*, 1:1265–1276, 2008.

[9] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *Proceedings of the SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 363–375, Toronto, Canada, 2010. ACM.

[10] C. S. Chekuri, A. V. Goldberg, D. R. Karger, M. S. Levine, and C. Stein. Experimental study of minimum cut algorithms. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 324–333. SIAM, 1997.

[11] T. Condie, N. Conway, P. Alvaro, and J. M. Hellerstein. MapReduce online. In *Proceedings of the 7th Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, USA, 2010. USENIX Association.

[12] P. Cousot. Abstract interpretation. *ACM Computing Surveys*, 28, June 1996.

[13] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *Transactions on Programming Languages and Systems (TOPLAS)*, 13:451–490, 1991. ACM.

[14] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 107–113, San Fancisco, CA, USA, 2004. USENIX Association.

[15] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a highlevel dataflow system on top of MapReduce: The Pig experience. *The Proceedings of the VLDB Endowment (PVLDB)*, 2:1414–1425, 2009.

[16] G. Graefe. The cascades framework for query optimization. *IEEE Data Engineering Bulletin*, 18(3), 1995.

[17] R. H. Halstead. Multilisp: A language for concurrent symbolic computation. *Transactions on Programming Languages and Systems (TOPLAS)*, pages 501–538, 1985. ACM.

[18] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd European conference on Computer systems (EuroSys)*, pages 59–72, Lisbon, Portugal, 2007. ACM.

[19] M.-Y. Iu and W. Zwaenepoel. HadoopToSQL: a mapReduce query optimizer. In *Proceedings of the 5th European conference on Computer systems (EuroSys)*, pages 251–264, Paris, France, 2010. ACM.

[20] E. Jahani, M. J. Cafarella, and C. Ré. Automatic optimization for MapReduce programs. *The Proceedings of the VLDB Endowment (PVLDB)*, 4:385–396, 2011.

[21] Q. Ke, V. Prabhakaran, Y. Xie, Y. Yu, J. Wu, and J. Yang. Optimizing data partitioning for data-parallel computing. In *Proceedings of the 13th Workshop on Hot Topics in Operating System (HotOS)*, Napa, CA, USA, 2011. USENIX Association.

[22] F. Logozzo and M. Fähndrich. On the relative completeness of bytecode analysis versus source code analysis. In *Compiler Construction*, pages 197–212, Budapest, Hungary, 2008. Springer-Verlag.

[23] E. Meijer, B. Beckman, and G. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *Proceedings of the SIGMOD International Conference on Management of Data (SIGMOD)*, pages 706–706. ACM, 2006.

[24] Microsoft. LINQ. `http://msdn.microsoft.com/en-us/library/bb308959.aspx`, February 2007.

[25] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[26] D. G. Murray, M. Isard, and Y. Yu. Steno: automatic optimization of declarative queries. In *Proceedings of the SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 121–131, San Jose, CA, USA, 2011. ACM.

[27] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V. B. N. Rao, V. Sankarasubramanian, S. Seth, C. Tian, T. ZiCornell, and X. Wang. Nova: continuous Pig/Hadoop workflows. In *Proceedings of the SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1081–1090, Athens, Greece, 2011. ACM.

[28] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic optimization of parallel dataflow programs. In *Annual Technical Conference (ATC)*. USENIX Association, 2008.

[29] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *Proceedings of the SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1099–1110, Vancouver, Canada, 2008. ACM.

[30] Oracle. Virtual column. `http://www.oracle-base.com/articles/11g/virtual-columns-11gr1.php`, March 2011.

[31] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 2005.

[32] SharpDevelop. ILSpy. `http://wiki.sharpdevelop.net/ilspy.ashx`, June 2012.

[33] 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 MapReduce framework. *The Proceedings of the VLDB Endowment (PVLDB)*, 2:1626–1629, 2009.

[34] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using Hadoop. In *Proceedings of the 26th International Conference on Data Engineering (ICDE)*, pages 996–1005, Long Beach, CA, USA, 2010. IEEE.

[35] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE)*, pages 439–449, San Diego, CA, USA, 1981. IEEE Press.

[36] Xamarin. Mono Cecil. `http://www.mono-project.com/Cecil`.

[37] S. Xia, M. Fähndrich, and F. Logozzo. Inferring dataflow properties of user defined table processors. In *Proceedings of the 16th International Static Analysis Symposium (SAS)*, pages 19–35, Los Angeles, CA, USA, 2009. Springer-Verlag.

[38] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1029–1040, Beiing, China, 2007. ACM.

[39] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *Proceedings of the SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, pages 247–260. ACM, 2009.

[40] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, USA, 2008. USENIX Association.

[41] J. Zhang, H. Zhou, R. Chen, X. Fan, Z. Guo, H. Lin, J. Y.Li, W. Lin, J. Zhou, and L. Zhou. Optimizing data shuffling in data-parallel computation by understanding user-defined functions. In *Proceedings of the 7th Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, USA, 2012. USENIX Association.

[42] J. Zhou, N. Bruno, M. chuan Wu, P.-Å. Larson, R. Chaiken, and D. Shakib. SCOPE: parallel databases meet MapReduce. In *The VLDB Journal*. Springer-Verlag, 2012.

[43] J. Zhou, P.-Å. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *Proceedings of the 26th International Conference on Data Engineering (ICDE)*, pages 1060–1071. IEEE, 2010.