Static Analysis for Optimizing Big Data Queries

Diego Garbervetsky
Universidad de Buenos Aires, FCEyN, DC
ICC, CONICET. Argentina

Zvonimir Pavlinovic
New York University. USA

Michael Barnett, Madanlan Musuvathi, Todd Mytkowicz
Microsoft Research. USA

Edgardo Zoppi
Universidad de Buenos Aires, FCEyN, DC
ICC, CONICET. Argentina

ABSTRACT

Query languages for big data analysis provide user extensibility through a mechanism of user-defined operators (UDOs). These operators allow programmers to write proprietary functionalities on top of a relational query skeleton. However, achieving effective query optimization for such languages is extremely challenging since the optimizer needs to understand data dependencies induced by UDOs. SCOPE, the query language from Microsoft, allows for hand coded declarations of UDO data dependencies. Unfortunately, most programmers avoid using this facility since writing and maintaining the declarations is tedious and error-prone. In this work, we designed and implemented two sound and robust static analyses for computing UDO data dependencies. The analyses can detect what columns of an input table are never used or *pass-through* a UDO unchanged. This information can be used to significantly improve execution of SCOPE scripts. We evaluate our analyses on thousands of real-world queries and show we can catch many unused and pass-through columns automatically without relying on any manually provided declarations.

CCS CONCEPTS

• **Information systems → Query optimization**; *Parallel and distributed DBMSs*; • **Software and its engineering → Automated static analysis**;

KEYWORDS

Static analysis, Query optimization, Big Data, UDOs

ACM Reference format:
Diego Garbervetsky, Zvonimir Pavlinovic, Michael Barnett, Madanlan Musuvathi, Todd Mytkowicz, and Edgardo Zoppi. 2017. Static Analysis for Optimizing Big Data Queries. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 6 pages.
https://doi.org/10.1145/3106237.3117774

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3117774

1 INTRODUCTION

Programming big data applications is often done using data processing languages that combine relational-style constructs with imperative user-defined operators. Examples of systems relying on this paradigm are, for instance, Spark [14], SCOPE [5, 15], and U-SQL [1]. An important component of such systems are *query optimizers* that work only over the relational skeleton of a program. The user-defined operators (UDOs) are opaque and not analyzed during optimization. Hence, query optimizers often miss opportunities to significantly improve resource savings for big data applications.

The goal of this work is to automatically infer useful information about UDOs during compile-time that can be used to optimize query processing. In particular, we focus on big data programs written in SCOPE, a query processing language developed at Microsoft. SCOPE scripts receive, analyze, and return tables of data, similar to SQL. In practice, user-defined operators can operate on tables that can have several hundreds of columns and be hundreds of GB large. According to SCOPE team experts, most of the network and computational resources spent during execution of a SCOPE query are in fact irrelevant for the query result. For instance, the runtime will pass all of the table columns to a UDO although the operator might use only a small fraction of input columns to produce the output table. This happens since the runtime does not have detailed knowledge on UDO inner-workings and hence must conservatively assume that all input columns are used. The techniques we develop in this work automatically analyze UDOs and provide the query optimizer with such valuable pieces of information.

We use static analysis techniques to detect specific column access patterns induced by a UDO: which columns it reads from and the dependencies between columns. This information can be used to drastically optimize query execution:

1. Columns in input tables not read by a UDO can be *pruned* away, i.e., filtered out earlier in the query plan. This can result in less data transferred between nodes, often measured in hundreds of GB.
2. Columns that are passed unmodified through a UDO are *pass-through columns*. The runtime can directly copy values of such columns from the input table to the output table without expensive data marshalling through the UDO. Knowledge about pass-through columns can also help the query optimizer understand the distributed partitioning of the output table.

Additionally, unused and pass-through input columns can be used to validate user-provided optimization declarations.

The primary constraint for the analyses we present is total soundness: we cannot produce incorrect result because that can lead to invalid query results.

The first static analysis we introduce aims at quickly providing a conservative approximation of the input columns accessed by a UDO. The analysis performs a simple, yet effective escape and constant propagation alike analyses. The second analysis is more ambitious, but also more costly. It computes precise data and control dependencies in a UDO and relies on range and points-to analysis. We require both of the analyses to graciously handle complex .NET constructs such as loop iterators and closures. In general, we give great attention to making our analyses sound and robust.

We evaluated our analyses[1] on thousands of SCOPE query scripts that are executed internally at Microsoft on a daily basis. We corroborate that our analyses implementation achieve the expected robustness and soundness requirement while still being effective. This work was done as a collaboration between researchers at Microsoft, University of Buenos Aires, and New York University under patronage of Microsoft SCOPE product group.

2 BACKGROUND

Before presenting the analyses we prove more information about the SCOPE language, UDOs, and its ecosystem in general.

2.1 Cosmos

Cosmos is a distributed computing platform developed at Microsoft for storing and analyzing massive datasets [15]. Designed to run on large clusters consisting of thousands of commodity servers, Cosmos main platform objectives are availability, reliability, scalability, performance, and reduced cost. The main components of Cosmos are storage, execution environment, and SCOPE, a high-level programming language for big data analytics. We now describe the last component in more detail as it is the focus of this work.

2.2 SCOPE

SCOPE is the programming language used to write *scripts* that are executed in Cosmos. It is primarily a version of SQL with several extensions. The computation model of SCOPE is defined in terms of a directed acyclic graph. Data exchanged between nodes in the graph are in the form of strongly-typed *tables*. A table comprises a set of *columns*, each column containing values of some particular type. The data in a table is organized as a set of *rows*: each row has a field for every column.

The code that executes within a node is either generated by the system or is user code, authored in C#, called a user-defined operator (UDO). A UDO can be any combination of table filters, projections, and joins that are either impossible (or difficult) to express in the SQL-ish subset of the language.

SCOPE API for UDOs: A SCOPE UDO is implemented as a C# class which subclasses one of three base classes. The simplest, Processor, implements a method that takes a row from the input table as a parameter and returns zero or more rows [15]. A Reducer implements a method that takes a *rowset* (a set of rows from the input table that all have the same values in a specified set of columns) and returns zero or more rows. Finally, a Combiner is like a Reducer, but receives two rowsets as its input [15].

All three must override a method in their respective base classes that returns the *schema* of their output table. This method is executed during query compilation. Optionally, the method may also indicate that column pruning is allowed and to also attach information to each (output) column indicating which input columns that column depends upon. Without this information, the optimizer must make the conservative assumption that all input columns are read and that no information is available about which columns the output table might be partitioned on. Not only do many UDOs fail to add this optional information, but there is no check to make sure that any declarations are in fact correct.

UDO Example: Figure 1 shows an illustrative example of a UDO.

```
1  IEnumerable<Row> Process(RowSet input_rowset, Row output_row,
       string[] args) {
2    foreach (Row input_row in input_rowset.Rows) {
3      input_row.CopyTo(output_row);
4      string market = input_row[0].String;
5      output_row[2].Set("FOO" + market);
6      yield return output_row;
7    }}
```

Figure 1: Example SCOPE UDO

The operator returns an output table that is essentially a copy of the input table where value of the output column indexed by 2 is created using the value of the input column indexed by 0. Columns can either be accessed by integer indices, or more commonly, using string indices. The above example exhibits a structure common to almost all real-world UDOs: it is written as an *iterator*, a C# idiom for defining a lazy, cooperative state machine that must be polled for each element in the sequence it returns. There is a foreach loop iterating over the collection of input rows, code that creates an output row, and a yield instruction that returns that row.

2.3 UDO Representation

While the source code in Figure 1 is simple, it is compiled into a much more complicated representation in the resulting bytecode. Our analysis, as most static analyses, operates on the bytecode, not the source code. As depicted in Figure 2, the foreach-yield loop is implemented using a closure class (`<Process>_d__d<>3`) whose method `GetEnumerator` essentially populates the compiler generated fields that (1) model the parameters of the original `Process` method and fields that (2) represent the state of the loop iteration.

The `MoveNext` method is a state machine that, depending on the state, invokes the actual enumerator of the input row set and performs one iteration, possibly computing an output row. The analysis on UDOs must be aware of this internal organization, look for these particular methods, associate the internal fields with the original method parameters, and simulate the potentially multiple invocations of `MoveNext`. Additionally, it must also understand how SCOPE operations are represented in MSIL.

Both of the analyses we develop in this work take as input a SCOPE *job*, a compilation of a SCOPE script. Each job has a set of processors (UDOs) that, as mentioned earlier, implement Processor, Reducer, or Combiner APIs. For each UDO in the job, we find the

[1]Available at https://github.com/Microsoft/rudder

```
IEnumerable<Row> Process(RowSet input_rowset, …)
{
  <Process>d__d <Process>d__d = new <Process>d__d(-2);
  <Process>d__d.<>4__this = this;
  <Process>d__d.<>3__input_rowset = input_rowset;
  <Process>d__d.<>3__output_row = output_row;
  <Process>d__d.<>3__args = args;
  return <Process>d__d;
}
```

```
IEnumerator <Row> IEnumerable<Row>.GetEnumerator()
{
 <Process>d__d <Process>d__d;
 if (… <>1__state == -2)  {
      this.<>1__state = 0;
      <Process>d__d = this;
 }
 else
      …

 <Process>d__d.input_rowset = this.<>3__input_rowset;
 <Process>d__d.output_row = this.<>3__output_row;
 <Process>d__d.args = this.<>3__args;
 return <Process>d__d;
}
```

```
bool IEnumerator.MoveNext(){
   ...
   switch (this.<>1__state)
   {
     case 0:
        this.<>1__state = -1;
        this.<>7__wrap10 = this.input_rowset.get_Rows().GetEnumerator();
        this.<>1__state = 1;
        goto IL_CB;
     case 2:
           ...
   }
           ...
   if (this.<>7__wrap10.MoveNext())
   {
      ...
      this.<market>5__f = this.<input_row>5__e.get_Item(0).get_String();
      this.output_row.get_Item(2).Set("FOO" + this.<market>5__f);
      this.<>2__current = this.output_row;
      this.<>1__state = 2;
      result = true;
      return result;
   }
      ...
   return result;
}
```

parameters & local state

read input table

get column

set column

Figure 2: MSIL closure representation of the UDO from Figure 1. Top-left: `Process` method. Bottom-left: the enumerator generated for `yield return`. Right: `MoveNext` method of the enumerator.

corresponding closure class and run our analyses on the `MoveNext` method assuming the above closure representation.

3 ACCESSED COLUMNS ANALYSIS

The first analysis statically analyzes the code of a UDO to overapproximate the input columns that are being used by the operator. Using this information, the SCOPE distributed runtime environment can ship over the network only the values of inferred columns instead of values of all of the table columns while executing the UDO, without compromising the correctness of the results.

3.1 Approach

We decided to design the analysis to be as simple as possible. First, the analysis is intra-procedural, analyzing method bodies in isolation. Second, the analysis does not distinguish input from output columns, which would otherwise require potentially detailed aliasing information. Lastly, the analysis (soundly) answers that all input columns are read if the UDO has exceptional control flow, which would otherwise also require more complicated treatment. These design decisions helped us validate that our analysis is sound with no exceptions, in contrast to *soundy* analyses [8].

Clearly, the above mentioned simplifications may make our analysis not effective. The reason these decisions make sense for computing accessed input columns is based on an empirical observation: while UDOs can become quite complex in terms of the functionality they embody, the way the columns are accessed is typically straightforward. That is, columns are typically accessed directly by string or integer indices, as in Figure 1, or by variables that can be resolved as constants at compile-time.

Figure 3 is a high-level illustration of our analysis. The analysis takes UDO in previously described MSIL format, performs classical control flow graph transformations and optimizations described in Section 5, and then proceeds to three core subanalyses: escape analysis, constant-set propagation, and used columns analysis.

3.2 Escape Analysis

A valid platform assumption is that upon entering the `MoveNext` method no other method or object has an access to the input row

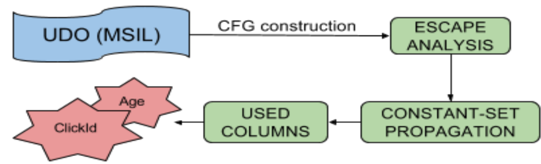

Figure 3: Subanalyses chain for inferring input columns

objects. Our analysis first checks whether this invariant also holds upon exiting the method via *escape* subanalysis. To see why this check is important, consider the following excerpt of code taken from a real-world SCOPE script.

```
1  IEnumerable<Row> Process(RowSet input, ...) {
2    ...
3    foreach (Row current in input.Rows) {
4      outputRow[0].Set(this.CreateBondEntity(current));
5      ...
6      outputRow[1].Set(this.RandomCurrentBondEntity());
7      yield return outputRow;
8    }}
```

The method call on line 4 can potentially save a reference to the input row `current`. Since our analysis is intra-procedural, there is no way of knowing whether some other object has an access to an input row after line 4. Hence, the method call on line 6 can potentially read, i.e., access some column of `current`. Our analysis would hence miss such accesses which would result in unsoundness.

Escape subanalysis checks if an object of type `Row` potentially *escapes* the method body. That is, escape analysis checks if some command in the UDO body passes an object of type related to `Row` as a parameter to some method call or saves it to some field. If so, the analysis claims that a row has *escaped* and answers that all table input columns are read. Otherwise, no other object or method has access to the input rows, i.e., all column accesses are contained within `MoveNext` body. Our analysis then proceeds to the next subanalysis.

3.3 Constant-set Propagation

The next subanalysis closely resembles widely known constant propagation. In fact, the major point where our subanalysis and

constant propagation differ is in the way they handle join points. Consider the following code excerpt.

```
1   if (...) { column = "Age" }
2   else { column = "age" }
```

Such code fragments are frequent in UDO programs as programmers often consult table schemas to make sure they got column name capitalizations right. After the *if-then-else* statement, at the join point, a typical constant propagation implementation would not consider `column` variable to be constant. Since our goal is to actually infer the columns being accessed, we can soundly save the information that column can take values in the set {"Age", "age"} instead of saying that `column` can take on any value. This is why we called this subanalysis *constant-set* propagation, since for each non-reference variable we save the set of constant values the variable can take. Speaking in terms of abstract interpretation, we simply perform disjunctive completion of the abstract domain for constant propagation [6].

3.4 Used Columns Analysis

The last subanalysis we undertake is named *used-columns* analysis. In the similar spirit as previous subanalyses, its main characteristic is simplicity. For each method call on an object of type `Row`, we check whether the method being called is known at compile time and is `get_Item`, `get_Schema`, or `Reset`. Only the mentioned methods can truly be trusted, in the sense they definitely do not access any columns of the calling `Row` object. Then, we check if for each `Row.get_Item(var)`, our constant-set propagation inferred a set of constants for `var`. If both checks pass, we take the union of these sets of constants as our overraproximation of accessed input columns. If either of the checks fail, the analysis answers that all input columns are being accessed.

The returned set of columns overapproximates both input and output columns being accessed by a UDO, due to the lack of aliasing information. Thus, all input columns that are not in this set are not being used by the UDO. We note that column information for an input table is available at compile time since SCOPE needs table schemas for compilation.

4 COMPUTING INPUT/OUTPUT DEPENDENCIES

The second analysis we introduce is more sophisticated. It computes column input/output relationships induced by the UDO and pass-through columns. Precise dependency information allows for more aggressive, but still conservative, optimization of query plans. Identifying pass-through columns enables significant savings in network/computation bandwidth.

Example: Consider again the UDO presented in Figure 1 and the input table schema {JobGuid(0), SubmitTime(1)} where integer index for the column name is given in parentheses. The outcome of our second analysis looks like:

- Inputs = {JobGuid(0), SubmitTime(1)}
- Outputs = {JobGuid(0), SubmitTime(1), NewColumn(2)}
- Pass-through = {JobGuid(0), SubmitTime(1)}
- OtherDeps = {NewColumn(2) ← literal + JobGuid(0)}

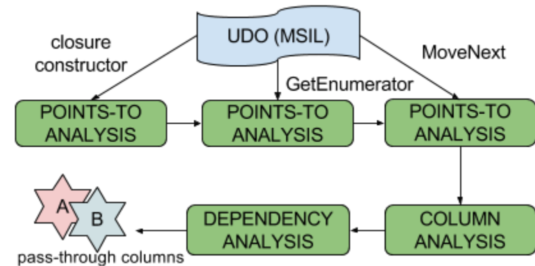

Figure 4: Subanalyses chain for dependency analysis

Inputs (resp. Outputs) is the set of input (resp output) columns observed by the analysis. The indices represent the column index associated with the column name. Pass-through is the set of output columns that were computed using one single input column. OtherDeps refers to other dependencies observed by the analysis. In this case, NewColumn(2) refers to a new column that depends on a literal ("FOO") and the input column JobGuid.

4.1 Approach

Our solution is inspired by work of Xia et al. [13] that proposes a data dependency analysis for SCOPE programs using an abstract interpretation engine Clousot [9]. The analysis computes dependencies over *traceables*: tables, columns, and row counters. For each output column of a UDO, the analysis reports traceables upon which that column depends on.

The hard constraint of having a sound and robust implementation prevented us from using their work. Clousot makes the optimistic assumption that any references that are not *must aliases* are distinct. As pointed out in [13], there is no aliasing for the input tables and the fields in the UDO closure classes. However, considering objects reachable in method calls forces us to be more conservative in three ways:

- use a conservative points-to analysis to support a may-alias and escape analysis
- use range analysis (intervals) and constant propagation for tracking column indices
- add support for exceptional control flow

4.2 Analysis Sketch

Figure 4 shows the analysis sketch. Given a UDO, the analysis first makes sure it has a proper representation of the heap effects. Therefore, we first compute a points-to graph (PTG), an alias heap abstraction explained in more detail shortly, of the closure constructor. Then, we compute the PTG for the `GetEnumerator` method (taking as input the constructor PTG) and finally, the PTG of `MoveNext`. This ensures we reach this method with all closure fields properly initialized and updated. With the proper PTG we can now run the column analysis to discover columns indices and map column names to indices and, finally, the dependency analysis.

Points-to analysis. We based our points-to analysis on a well-known flow-sensitive analysis by Salcianu and Rinard [11]. It is essentially a forward dataflow analysis that builds points-to graphs. A PTG is graph where each node (identified by a program location) represents the set of all objects that might be allocated at that location, and edges stand for potential references between those

objects. Given a PTG, we can determine whether two access paths (in the form of v, $v.f$, $v.f.g$, etc.) may alias by simply traversing the PTG and checking if they both can reach the same node.

Our points-to analysis can handle complex .NET programs constructs such as delegates, lambdas, and predicates (appearing frequently in LINQ queries). It can run intra- and inter-procedurally. For the sake of performance, we decided to run it intra-procedurally with some exceptions. We analyze invocations of closure auxiliary methods and lambda expression appearing in parameters. To handle collections, we use conservative summaries in the spirit of [3].

Column Analysis. The goal of this analysis is to determine the set of potential indices (columns) used to access a row. We compute an interval analysis to determine the possible integer values that a column index may have, which is more sophisticated than constant-set propagation. In addition, we perform a basic string analysis to discover columns accessed by name and map them to their corresponding indices.

4.3 Dependency Analysis

As mentioned earlier, we based the analysis on [13]. Given a UDO u, the analysis identifies a set of *traceables* and computes the following mappings:

- $DepVar_u(v)$ - tracks traceables flowing to variables
- $DepHeap_u(loc(o.f))$ - tracks traceables flowing to fields
- $DepOutput_u(o)$ - track traceables associated to output columns
- Esc_u - tracks traceables escaping through a non-pure method (in the case of inter-procedural analysis)

The analysis is essentially a forward dataflow analysis that propagates traceables through variables. Here we show some of the most interesting rules (→ means propagate):

SCOPE API based row rules: detect and propagate rows.

- `a = Rows(b)`: $DepVar(b) \rightarrow DepVar(a)$
- `a = Current(b)`: $DepVar(b) \rightarrow DepVar(a)$

SCOPE API based column rules: detect and propagate columns

- `a = Item(b,i)`: let $C = \{ T.\text{colum(i)}, T \in DepVar(b)\} \rightarrow DepVar(a)$ (reads column i from b)
- `Column.Set(o,b)`: $DepVar(b) \rightarrow DepOutput(o)$ (writes column o)
- `Row.Copy(a,o)`: propagates all traceable input columns from a into output columns in o (similar to multiple applications of the previous 2 rules)

Heap rules: propagate traceables from heap locations to variables, and vice versa. Use points-to analysis to determine heap locations.

- `a = b.f`: $DepHeap(b.f) \rightarrow DepVar(a)$
- `a.f = b`: $DepVar(b) \rightarrow DepHeap(a.f)$

If the method under analysis invokes another method, we check if the traceables of interest (i.e., input-output rows) can be reachable from the parameters. If so, in the intra-procedural case we give up (mark them as escaping) as we cannot tell what the non-analyzed callee is going to do. In the inter-procedural case, we apply the analysis on the callee.

Aliasing. We use the points-to graphs for detecting aliasing pairs (variables and fields) and to resolve method invocations. Also, every time we obtain traceables for a variable v we make sure we also have the traceable from its aliases. Similarly, for $DepHeap(v.f)$ we use PTGs nodes for referring to objects. For instance, if $PT(v) = \{A, B\}$ the analysis will produce two locations: $\{A.f, B.f\}$ for $v.f$.

4.4 Computing Pass-through Columns

Pass-through analysis is a byproduct of the dependency analysis. A pass-through column is an output column whose value is taken directly from one input column.

In order to determine a column is pass-through during the dependency analysis we check that *only* one input column is used for its computation and no other value.

5 EVALUATION

We implement our analyses on top of a dataflow analysis framework capable of analyzing .NET programs written in MSIL. We rely on the analysis framework [2] developed by one of the authors. This framework provides three address code and SSA representations [2] of the MSIL, well-known control flow and dataflow analyses, as well as a general engine for dataflow fixpoint computation.

Our implementation works over the SSA form of MSIL. We first utilize existing framework facilities to model implicit MSIL stack operations with explicit top-of-the-stack variable operations. We also make use of the classical copy-propagation and live-variables analyses [2]. We directly implement the escape, constant-set propagation, points-to, range, and dependency analysis on top of the framework.

We aim at answering following questions: i) what is the ratio between used and total number of input columns?; ii) how many pass-through columns are discovered?; iii) are analysis running times within an acceptable bound?; iv) is the information obtained by the analyses useful?

We run the analyses on about 4000 real-world SCOPE projects extracted from Cosmos' top jobs (in terms of resource usage) executed on 4/30/2016. Table 1 shows the results for both analyses. There were 1151 UDOs in total[3]. The analysis times ranged between 100ms to a couple of seconds. We note the our analyses as well as the underlying framework are not yet fully optimized for performance. We believe a mature implementation would experience running times measured in few hundred milliseconds. The total number of columns involved in the UDOs, according to the declared schemas, were 25014 for input and 24941 for output columns.

The first analysis is about 6-8 times faster than the second analysis but more imprecise. In many cases it cannot conclude a precise answer and, for the sake of soundness, abruptly returns that all columns are potentially used. This imprecision is mainly due to exceptional control flow, escaping rows, untrusted row methods, and non-constant variables used for column accesses. Nevertheless, for about 25% of the UDOs the analysis obtained results, discovering that at least 37% of the columns were not accessed. We feel these are very good results that justify the simplicity of the analysis.

[2]Available at http://github.com/edgardozoppi/analysis-net

[3]We ignore jobs not using UDOs or using only compiled generated UDOs.

Table 1: Statistics for 1151 UDOs from 4000 SCOPE jobs.

	UDOs w/ results	Unused Cols		Pass-through Outputs
		Inputs	Outputs	
Analysis 1	25%	37%		N/A
Analysis 2	76%	54%	50%	74%

The second analysis is slower but more precise, as expected, and handles more UDOs. It discovers that about 50% of the columns are unused. In addition, it discovers that about 74% of columns are in fact pass-through. The imprecision mainly comes from dealing with complex index computation for column accesses, traceable escaping through method invocations (needs inter-procedural analysis), and complex data structures like SCOPE maps (see future work).

We could not measure actual query times since we did not have access to the Cosmos databases, but according to the SCOPE team experts, a reduction in the number of columns passed to a UDO and the savings in marshaling induced by pass-through columns can be drastic. It is important to emphasize that even a small percentage of resource savings yields huge savings in total as the analyzed scripts run daily at Microsoft and operate on hundreds of GB of data.

6 RELATED WORK

There are two previous efforts directly related to our work. PeriSCOPE is a static analysis tool that optimizes SCOPE execution plans by analyzing UDOs [7]. The authors present three analyses that compute UDO information useful for optimizing the query execution. One of the analysis is in fact used columns analysis. Unfortunately, it is not clear how general soundness of their algorithm can be argued. For instance, the authors do not explain how they deal with the cases where a `Row` object can escape a method body. In our work, on the other hand, soundness is an imperative.

As we discussed in Section 4.4, Xia et al. present a static analysis that infers column dependencies in SCOPE UDOs [13]. Their approach relies on an optimistic must aliasing assumption which violates our soundness principle. Such assumption could prevent building different sound analyses on top of their infrastructure.

We mention few other related works. Our dependency analysis closely resembles data and control dependence in compiler optimizations [2, 10]. Also, points-to and alias analysis are a classical topic in static analysis community [3, 4, 12]. However, these efforts are orthogonal to the work presented in this paper.

7 CONCLUSIONS

We implemented two static analyses aimed at obtaining unused column information and input/output dependencies in SCOPE UDOs. We put a special focus on being sound and robust while designing and implementing the analyses. Our implementation successfully analyzed thousands of SCOPE scripts and found many input columns that are never used and a significant amount of pass-through columns in real-world UDOs. The inferred information can be used to drastically optimize execution of SCOPE scripts. This works shows how static analysis techniques can be used to improve performance of industrial-strength applications.

Lessons learned. One of the most important lessons we learned is that in an industrial setting it is much more important to be robust and sound than to be precise. Even the most sophisticated analysis will not make its way into production if its soundness cannot be guaranteed and clearly argued. Our recommendation is to start with the simplest analysis possible. Then if needed, more sophisticated analysis can be built on top, keeping the same principles in mind.

Another important lesson learned is that static analyses techniques can be successfully applied for real-world programs at large. The trick behind our success was to incorporate domain specific knowledge into the analysis to get sensible results. SCOPE scripts, while potentially arbitrarily complex, typically follow a simple structure and manipulate certain data structures in a predictable way. We believe there are other problem domains where such tailored static analyses can prove themselves extremely valuable.

Future work. Our next steps are (1) the evaluation of the actual impact of the analyses on execution performance of real-world SCOPE scripts and (2) implementation of an IDE plugin that automatically generates data-dependency annotations for the UDOs, thus hinting the developer where optimizations can be gained and validating her assumptions. At the same time, we plan to enhance our analyses. For instance, the analysis for computing input columns accessed by a UDO can be improved by making it more inter-procedural by using function inlining. In this way, we can gain more precision during escape analysis. Likewise, we plan to support other features of SCOPE such as SCOPE maps, JSON, and other structured column types that are used in scripts to encode sparse columns.

ACKNOWLEDGMENTS

This work was partially supported by the projects ANPCYT PICT 2013-2341, 2014-1656 and 2015-1718, UBACYT 20020130100384BA, CONICET PIP 112 201301 00688CO, 112 201501 00931CO.

REFERENCES

[1] U-sql, the new big data language for azure data lake. https://azure.microsoft.com/en-us/blog/u-sql-the-new-big-data-language-for-azure-data-lake/. Accessed: 2017-05-09.
[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.
[3] M. Barnett, M. Fähndrich, F. Logozzo, and D. Garbervetsky. Annotations for (more) precise points-to analysis. *In IWACO*, pages 11–18, 2007.
[4] B. Blanchet. Escape analysis: correctness proof, implementation and experimental results. In *In POPL*, pages 25–37. ACM, 1998.
[5] 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. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, 2008.
[6] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of logic and computation*, 2(4):511–547, 1992.
[7] Z. Guo, X. Fan, R. Chen, J. Zhang, H. Zhou, S. McDirmid, C. Liu, W. Lin, J. Zhou, and L. Zhou. Spotting code optimizations in data-parallel pipelines through periscope. In *OSDI*, pages 121–133, 2012.
[8] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundiness: a manifesto. *Commun. ACM*, 58(2):44–46, 2015.
[9] F. Logozzo. Clousot: Static contract checking with abstract interpretation. *Formal Verification of Object-Oriented Software*, page 5.
[10] S. S. Muchnick. *Advanced compiler design implementation.* Morgan Kaufmann, 1997.
[11] A. Sălcianu and M. Rinard. Purity and side effect analysis for java programs. In *In VMCAI*, pages 199–215. Springer, 2005.
[12] B. Steensgaard. Points-to analysis in almost linear time. In *In POPL*, pages 32–41. ACM, 1996.
[13] S. Xia, M. Fähndrich, and F. Logozzo. Inferring dataflow properties of user defined table processors. In *SAS*, pages 19–35, 2009.
[14] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.
[15] J. Zhou, N. Bruno, M. Wu, P. Larson, R. Chaiken, and D. Shakib. SCOPE: parallel databases meet mapreduce. *VLDB J.*, 21(5):611–636, 2012.