Exploring Cross-Language Optimizations in Big Data System: An Empirical Study of SCOPE

Anonymous Author(s)

ABSTRACT

Building scalable big data programs requires programmers combine relational (SQL) with non-relational code (Java, C♯, Scala). Relational code is declarative — a programme describes *what* the computation is and the compiler decides *how* how to distribute the program. SQL query optimization has enjoyed a rich and fruitful history, however, most research and commercial optimization engines treat non-relational code as a black-box and thus are unable to optimize it.

This paper empirically studies over 3 million SCOPE programs across five data centers within Microsoft and finds programs with non-relational code take between 45-70% of data center CPU time. We further explore the potential for SCOPE optimization by generating more native code from the non-relational part. Finally, we present X case studies showing that triggering more generation of native code in these jobs yields significant performance improvement that ranges between Y and Z%.

ACM Reference Format:
Anonymous Author(s). 2017. Exploring Cross-Language Optimizations in Big Data System: An Empirical Study of SCOPE. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 5 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Large-scale data-processing frameworks, such as MapReduce [**?**], SCOPE [1], Hadoop [**?**], Spark [**?**], have become an integral part of computing today. One reason for their immense popularity is that they provide a simplified programming model that greatly simplifies the distribution and fault-tolerance of big-data processing. For instance, frameworks like SCOPE and Spark provide a SQL-like declarative interface for specifying the relational skeleton of data-processing jobs while providing extensibility by supporting expressions and functions written in general-purpose languages like C#, Java, or Scala.

The relational aspect is crucial: it is what enables the automatic parallelization for efficiently scaling out to arbitrary amounts of data. Big data systems assume that the non-relational part is written carefully enough so that it does not violate the assumptions needed for automatic parallelization: i.e., programmers must write their non-relational logic to be deterministic and insensitive to the ordering of the input.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

However, these systems are known to lag far behind traditional database systems in runtime efficiency [2, 3], primarily because of the flexibility of the programming model they support. For instance, a key bottleneck in Spark is neither the disk nor the network, but the time spent by the CPU on compression/decompression of data, serialization/deserialization of the input into/from Java objects, and the JVM garbage collection [**?**]. SCOPE, described more fully in Section 2, supports a hybrid native (C++) and C# runtime partly to alleviate this overhead. However, our analysis shows that the interaction between the native and C# runtimes still remains a significant cost in the overall system. Equally importantly, the presence of non-relational code blocks the powerful relational optimizations implemented in these data-processing runtimes [**?**].

The goal of this work is to study and better understand the key performance bottlenecks in modern data-processing systems, and demonstrate the potential for cross-language optimizations. While this paper is primarily about SCOPE, we believe our results and optimizations generalize to other data-processing systems. SCOPE is the key data-processing system used at Microsoft running at least half a million jobs daily on serveral Microsoft data centers. Figure 1 shows a simple example of a SCOPE program (hereafter referred to as a *script*) that interleaves relational logic with C# expressions.

```
data = SELECT *
  FROM inputStream
  WHERE !String.IsNullOrEmpty(A) AND B == "Key1";
```

(a) Predicate visible to optimizer.

```
data = SELECT *
  FROM inputStream
  WHERE M(A, B);

#CS
bool M(string x, string y) {
   return !String.IsNullOrEmpty(x) && y == "Key1";
}
#ENDCS
```

(b) Predicate invisible to optimizer.

Figure 1: Script Examples

In Figure 1a, the predicate in the `WHERE` clause is subject to two potential optimizations:

(1) The optimizer may choose to *promote* one (or both) of the conjuncts to an earlier part of the program, especially if either `A` or `B` are columns used for partitioning the data. This can dramatically reduce the amount of data needed to be transferred from one vertex to another.
(2) The SCOPE compiler has a set of methods that it considers to be *intrinsics*. An intrinsic is a .NET method for which

the SCOPE runtime has a semantically equivalent native function. For instance, the method `String.isNullOrEmpty` checks whether its argument is either null or else the empty string. The corresponding native method is able to execute on the native data encoding which does not involve creating any .NET objects or instantiating the .NET virtual machine (CLR).

On the other hand, Figure 1b shows a slight variation where the user implemented the predicate in a separate C# method. Unfortunately, the SCOPE compiler treats the call to user-defined functions as a black box. As a result, both optimizations are disabled and the predicate is executed in a C# virtual machine. The resulting serialization and data-copying costs can reduce the throughput of the job by as much as XXX percent.

First, we buld a profiling infrastructure based on static anlaysis of job artifacts.

Second, we do a comprehensive survey.

Then, we use the results of the survey to propose optimizations.

Finally, we evaluate.

Despite knowing the cause of inefficiencies, still the little is known about how much time SCOPE jobs spend in non-relational code and what are the opportunities to trigger more generation of native code. To answer these questions, we propose a new profiling infrastructure based on static analysis of job artifacts. By doing this, we are able to post-analyse large amount of jobs without introducing any additional overhead. Except the time spent in non-relational code, our profiling approach also surveys different sources of non-relational code in SCOPE jobs. It returns a list of most commonly used framework methods, for which having C++ implementation would trigger more generation of native code. Finally, we discuss the effects of cross-language optimization based on *method inlining*. The idea behind method inlining is to instead calling a method, move the logic of that method to the operator. By doing this, the compiler is able generate native code from inlined logic. We prove the effectiveness of such optimizations in Z case studies by optimizing K jobs from U different teams at Microsoft.

Our experimental evaluation shows that non-native code takes between 45-75% of data center CPU time. By further increasing the list of framework functions that have native implementation we can optimize up to Z% of data center time. Finally, we illustrate that performance impact of inlining jobs to run as native is statistically significant and range between A and B%.

As a consequence, our findings motivate future research in at least two ways. First, on tools and techniques that help developers write more efficient big data jobs by avoiding unnecessary generation of non-native code. Second, on compiler optimizations that automatically generate native code from the non-relational logic.

In summary, this work contributes the following:

- *Profiling infrastructure for analyzing SCOPE jobs.* We present the approach for profiling data centers based on static analysis of job artifacts. The approach reports time spent in relational vs non-relational code and detects different sources of non-relational code for every jobs stage.
- *Reporting opporunities for cross-language optimizations.* We discuss two possible ways to enable further generation of native code. First, we survey the most relevant framework functions relevant for native implementation. Second, we present an analysis to find opportunities that trigger the generation of native code by *inlining* user-written functions.
- *Empirical evidence.* We illustrate by X case studies that enabling job stage to run as native code signifcantly improve the job performance by factors Y-K.

2 SCOPE

SCOPE [] is used internally within Microsoft and is now transitioning into an external offering as U-SQL []. Its relational part is very similar to SQL, enough so that we will ignore any differences. The non-relational part is C# []: all expressions in a SCOPE program *script* are written as C# expressions. In addition, SCOPE allows user-defined functions, *UDFs*, and user-defined operators, *UDOs*.

2.1 Execution of a Script

A script defines a directed acyclic graph (DAG) where each vertex is a set of operators implemented on the same physical (or virtual) machine. We use the term *node* for the physical or virtual machine that a vertex is implemented on. The edges of the DAG are communication channels that use a high-speed communication network between nodes. The operators within a vertex are the end product of a very sophisticated optimizer []; expressions written within a certain construct in the script may end up being executed in vertices that do not correspond to the construct in a simple manner. For instance, a sub-expression from a filter may be *promoted* into a vertex which extracts an input table from a data source, whereas the rest of the filter may be in a vertex that is many edges distant from the input layer.

2.2 C++ vs. C#

The SCOPE compiler generates both C++ and C# operators for the same source-level construct. Each operator, however, must execute either entirely in C# or C++: mixed code is not provided for. Thus, when possible, the C++ operator is preferred because the data layout in stored data uses C++ data structures. Thus, for example, a simple projection of a subset of the columns can be done entirely without using the CLR. But when a script contains a C# expression that cannot be converted to a C++ function, such as in Figure 1b, the CLR must be started, each row in the input table must be converted to a C# representation, i.e., a C# object representing the row must be created, and then the C# expression can be evaluated in the .NET Runtime.

Because this can be inefficient, the SCOPE runtime contains C++ functions that are semantically equivalent to a subset of the .NET Framework methods that are frequently used; these are called *intrinsics*. The SCOPE compiler then emit calls to the (C++) intrinsics in the C++ generated operator, which is then used at runtime in preference to the C# generated operator. (As opposed to using *interop* to execute native code from within the .NET Runtime.)

2.3 Compiler/Optimizer Communication

In general, the C# code is compiled as a black box: no analysis/optimization is peformed at this level. One consequence is that any calls to a UDF within a SCOPE expression (filter predicate, projection

function) require the operator containing the call to be implemented in C#.

3 PROFILING INFRASTRUCTURE FOR DATA CENTERS

This section provides details on our profiling infrastructure for data centers that run big-data jobs. Because SCOPE compiler does not provide any instrumentation facility, we chose to collect profiling data by analyzing job artifacts. The benefit of doing this is at least twofold: we can derive data for a relatively large number of jobs since we do not require re-running them, and we can also answer more complex, but interesting questions, such as which job stages run as C++ vs. C#.

3.1 Scale of The Problem

SCOPE jobs run on a distributed computing platform, called Cosmos, designed for storing and analyzing massive data sets. Cosmos run on five clusters consisting of thousands of commodity servers [1]. Important properties of Cosmos is scalability and performance. Nowadays, it stores exabytes of data across hundreds of thousands of physical machines. Cosmos runs millions of big-data jobs every week and almost half million jobs every day. It is used by more than 10,000 developers at Microsoft running very diverse workloads and scenarios.

Finding optimization opportunities that are applicable to such a large number of diverse jobs is a very challenging problem. To bring interesting conclusions we must ensure the scalability of our profiling infrastructure. To achieve this, the important aspect to consider is *what type of information we should analyze*. In the following sections, we describe our major decisions when building infrastructure for profiling big data jobs.

3.2 Job Artifacts

After execution of a SCOPE job finishes, the runtime produces several artifacts that contain code and runtime information for every job stage. Job artifacts are indefinitely stored in Cosmos repository and this section gives an overview of each artifact we use to profile data center.

Job Algebra. Job algebra is a graph representation of the job execution plan. Job vertices are presented as outer-most nodes in a graph. Each job vertex contains all operators that run inside that vertex and an operator can be either user-defined or native. Optionally, if all operators are native, the vertex can contain *nativeOnly* flag indicating that entire vertex runs as native (C++). However, it does not distinguish between native and user-defined operators.

Runtime Statistics. Runtime statistics file provides information on execution time for every job vertex and every operator inside the vertex. However, it includes only CPU times, which we use as a primary metric of a job execution time.

Generated C# and C++ Code. SCOPE runtime generates C# and C++ code for every job. An artifact with C++ code has for every vertex a code region containing a C++ implementation of the vertex and another code region that provides class names for every operator that runs as C#. An artifact with C# code includes implementations of non-native operators and user-written classes and functions defined inside the script.

3.3 Static Analysis

After collecting artifacts described in Section 3.2, we perfom static analysis to detect different sources of C# code in every vertex of a job. This is important to understand the opportunities for optimizing job vertices through C# to C++ translation. For instance, a vertex can run as managed code due to a single method, or more complex C# code.

Figure 2 gives an overview of our analysis. It has two main components: *Analysis of C++ code* and *Analysis of C# code*. Each analysis performs on the granularity of a job vertex. The goal is to look for opportunities to run entire vertex as C++ code, which would remove all steps of data serialization between user and native operators within the vertex.

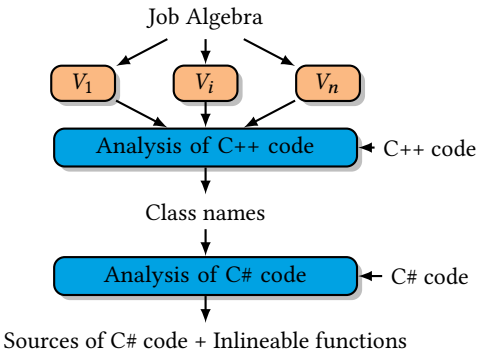

Figure 2: High level picture of static analysis

The first step of the analysis is to extract names of every job vertex, which serve as unique identifiers for vertices. Then for each vertex, the analysis parses generated C++ to find the class containing the vertex implementation. As discussed in Section 3.2, for each vertex, C++ implementation contains two code regions: one that indicates which part of a vertex runs as C++ code and another region listing class names of operators that run as C# code. If the latter region is empty, we conclude that entire vertex runs as C++ code. The output of the first stage of our analysis is the collection of class names that contain C# operators.

Based on the output of the previous stage, the analysis parses generated C# code to find definitions and implementation of every class in the list. The implementation of managed operators contains two sources of C# code: generated code, which we whitelist and skip in our analysis and the user-written code. After analyzing user code, the analysis outputs the following sources of method calls:

- .NET framework calls
- User written functions
- User written operators

We are particularly interested in the first two categories. It is not unusual case when a vertex runs as C# execution just because of a framework method. We want to know what are the most important framework methods to optimize to enable C++ translation for a

large number of vertices. Furthermore, we observe the cases when a logic of user-written function is relatively simple, and inlining the function logic as demonstrated in Figure 1 would enable generation of C++ instead of C# code.

The details on how we detect different sources of managed code and inlineable functions are described as follows.

3.3.1 Detecting Sources of C# Code. To find .NET framework calls, it is enough to check whether a method definition comes from .NET runtime. The analysis finds user-written functions by looking for their definition inside the script or in third-party projects. Because Cosmos repository keeps only binaries of third-party projects, we further analyze only user functions for which the source code is available. It is easier to optimize these functions through inlining (as described in Section 3.3.2), because we can manually confirm the correctness of inlined code. Finally, in SCOPE, users can easily implement their own operators: extractors (for parsing and constructing rows from a file), processors and reducers (for row processing), and combiners (for processing rows from two input tables). The analysis finds user operators by checking whether a class that implements an operator inherits from one of the predefined base classes.TODO: reformulate this We do not consider user operators for C++ translation because of TODO: elaborate on this.

3.3.2 Analysis of User-Written Code. Inlining of a user-written function refers to moving a logic of the function to the script operator. We define *inlineable* methods as follows.

Definition 3.1 (Inlineable method). Method *m* is *inlineable* method if it has the following properties:

- It contains only .NET framework calls
- It does not contain loops and try-catch blocks
- It does not contain any read or write to a variable or objects
- For all calls inside the method, arguments are passed by value

Furthermore, we distinguish among inlineable methods those that allow for *instant* C++ generation, because all called .NET framework methods are *intrinsics*. However, the analysis of other inlineable functions is important, because it provides the intuition on how many vertices are potentially optimizable in this way.

4 CROSS-LANGUAGE OPTIMIZATIONS

Cross-language optimizations can result in two different changes to the generated program that executes a query.

4.1 Optimizations With Effects On Job Algebra

The first changes the physical plan, i.e., the job algebra, because more information is made available to the query optimizer. For example, predicates might be pushed deeper into the DAG which can result in dramatic data reduction.

4.2 Optimizations Without Effects On Job Algebra

The second is that, for SCOPE, the set of intrinsics means that by lifting more non-relational code into the parts of the program where such things are visible to the optimizer, more code can be executed in C++ instead of in C#. So even though the physical plan has not changed, the resulting program might be might more efficient since it avoids the native to managed transition.

5 EVALUATION

Before evaluation we should introduce the notion of inlining... RQ:

- *What is the proportion of time spent in native vs. non-native job vertices?*
- *What proportion of time can be optimized having the current list of methods with C++ implementation?*
- *What proportion of time can be optimized by extending the list of methods with C++ implementation? Which methods should be the most important for C++ implementation?*

5.1 Experimental Setup

5 data centers, time span, mention that many jobs are periodic (run usually daily or every couple of days)

Figure 3: Table with number of jobs and total cpu time for every data center

5.2 Native vs. Non-Native Time

This figure illustrates how much time is spent in vertices (job stages) that run as native vs. non-native code. x axis represents each data center, while y axis gives the percentage...

Figure 4: graph that shows how much time is spent in native vs. non-native job stages

5.3 Optimizable Job Stages

This figure illustrates how much time is spent in job stages that we can optimize by inlining the user-written logic. X axis represents data center, Y axis represents precentage of time spent in inlineable job stages relative to total data center time and total time spent in non-native code.

Figure 5: Optimizable job stages

5.4 Potential for C++ Translation

The following figure shows how much time can be affected by extending the list of functions that have C++ implementations. X axis represents data center, Y axis precentage of time spent in job stages that can be optimized, relative to total non-native and data center time.

Figure 6: Potentially optimizable job stages

5.4.1 Most Relevant .NET Framework Methods. Ranking criteria: cost of job stage

The table below shows for every data center what are the most important .NET framework methods taking into account our ranking criteria...Furthermore, it illustrates how much time would be affected by providing C++ implementation of these methods, relative to total non-native time and total data center time.

Figure 7: Most relevant .NET methods per data center

6 CASE STUDIES

In order to quantify the effects of optimizing the scripts, we performed a case study. We say that a C# method is *intrinsicable* if it is a .NET Framework method for which the SCOPE compiler has a semantically equivalent C++ function. The jobs were chosen based on a static analysis that found *optimizable vertices*. An optimizable vertex is one that is implemented in C#, but the C# code calls only intrinsicable methods or user-defined functions, *UDFs*, where the UDF, in turn, calls only intrinsicable methods, and does not call any other UDFs, i.e., our inlining depth is one. Furthermore, we identified a job as being of interest if it contains an optimizable vertex was responsible for X% of the total job's time. **mb:[Marija, is that true? Or did we just care about the cost of the job?]** We categorized the jobs by how long they ran for: short, medium, and long. **mb:[Marija, do you have any stats about the distribution of PnHours? Are there really just three buckets? [0,100), [100,1000), and [1000,...)?]**

Because the input data for each job is not available, we needed to contact the job owners and ask them to re-run the job with a manually-inlined version of their script. We were able to have 12 jobs re-run by their owners.

- Description of every optimized job
- Performance evaluation (methodology)
- At least one job that changes algebra
- At least one job that does not change algebra

7 THREADS TO VALIDITY

Underapproximation of performance impact. Amount of time that can be optimized by either increasing the list of intrinsics or method inlining is the underapproximation of the total optimizable time. We do not consider effects of optimizations on another compiler optimizations. For example, if a method is inlined on a column used to partition data, the inlining would not only trigger more C++ generation, it would also enable filter promotion []. Understanding impact of C++ translation on another compiler optimizations is left for future work.

Assumptions for static analysis. Our static analysis detects sources of C# code based on several assumptions. For example, we use naming conventions when pruning generated methods in C# implementation of managed operators. A user can potentially call some of these methods in the script, meaning that we would skip a valuable source of user-written C# code. However, in practice, such methods are not used in the context of big-data jobs and our manual exploration of many SCOPE scripts illustrates that our assumptions hold.

Challenges for implementing more intrinsics. We discuss the relevance of providing C++ implementation for more .NET Framework methods. However, providing C++ translation for some of these methods poses several challenges. For example, memory management in C# is very different because it has a garbage collector, while C++ does not. Another challenge is related to different string encodings in C# and C++ runtimes, and for some corner cases, there is no clear one-to-one mapping. However, increasing the list of intrinsics would certainly bring significant performance benefits in SCOPE jobs, and there is a clear motivation for future work to address this problem.

8 RELATED WORK

8.1 Query Optimizations

....

9 CONCLUSIONS

REFERENCES

[1] Ronnie Chaiken, Bob Jenkins, Perake Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. 2008. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1265–1276. https://doi.org/10.14778/1454159.1454166

[2] Eaman Jahani, Michael J. Cafarella, and Christopher Ré. 2011. Automatic Optimization for MapReduce Programs. *Proc. VLDB Endow.* 4, 6 (March 2011), 385–396. https://doi.org/10.14778/1978665.1978670

[3] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. 2009. A Comparison of Approaches to Large-scale Data Analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD '09)*. ACM, New York, NY, USA, 165–178. https://doi.org/10.1145/1559845.1559865