Cross-Language Optimizations in Big Data Systems: A Case Study of SCOPE

Marija Selakovic
TU Darmstadt
Germany
m.selakovic89@gmail.com

Michael Barnett, Madan Musuvathi, Todd Mytkowicz
Microsoft Research
USA
mbarnett,madanm,toddm@microsoft.com

ABSTRACT

Building scalable big data programs currently requires programmers to combine relational (SQL) with non-relational code (Java, C#, Scala). Relational code is declarative — a program describes *what* the computation is and the compiler decides *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 6 case studies showing that triggering more generation of native code in these jobs yields significant performance improvement: optimizing just one portion resulted in as much as 25% improvement for an entire program.

ACM Reference Format:
Marija Selakovic and Michael Barnett, Madan Musuvathi, Todd Mytkowicz. 2018. Cross-Language Optimizations in Big Data Systems: A Case Study of SCOPE. In *Proceedings of 40th International Conference on Software Engineering: Software Engineering in Practice Track, Gothenburg, Sweden, May 27-June 3, 2018 (ICSE-SEIP '18),* 10 pages.
https://doi.org/10.1145/3183519.3183528

1 INTRODUCTION

Large-scale data-processing frameworks, such as MapReduce [10], SCOPE [4], Hadoop [12], Spark [34], have become an integral part of computing today. One reason for their immense popularity is that they provide a 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.

ICSE-SEIP '18, May 27-June 3, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5659-6/18/05…$15.00
https://doi.org/10.1145/3183519.3183528

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: e.g., programmers must write their non-relational logic to be deterministic and insensitive to the ordering of the input.

However, these systems are known to lag far behind traditional database systems in runtime efficiency [21, 28], 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 [27]. SCOPE, described more fully in Section 2, supports a hybrid native (C++) and C# runtime partly to alleviate this overhead. Like SCOPE, Hadoop Streaming lets programmers write programs in a mix of languages[2]. Our analysis shows that this cross-language interaction (in SCOPE, between the native and C# runtimes) is 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, e.g. [16].

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 several 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.

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 script, 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 across the network.
(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, i.e., implemented in C++. 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

```
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

does not involve creating any .NET objects or instantiating the *CLR*, i.e., the .NET virtual machine.

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 90% percent.

When facing a performance regression, a performance analyst first measures her system to understand where the bottlenecks are. Then, she can act on those data to make her system faster. While such an approach is well understood and easy to execute for desktop software, it becomes challenging in the context of a massive distributed system, like SCOPE. This paper first describes *how* we built a datacenter-wide profiler so we could even start to measure application bottlenecks. Our new profiling infrastructure is a combination of offline static analysis of the executed code in addition to low-overhead online measurements captured by SCOPE's runtime. Then, the paper describes the results of our profiling of over 3 million SCOPE programs across five data centers within Microsoft. We find programs with non-relational code take between 45-70% of data center CPU time.

Finally, we discuss the effects of cross-language optimization based on *method inlining*. By inlining a method call, the compiler/optimizer is now aware of the logic contained in the body of the method. We discuss the effectiveness of such optimizations in 6 case studies by optimizing jobs from 5 different teams at Microsoft.

2 SCOPE

SCOPE [4] is used internally within Microsoft and is now transitioning into an external offering as U-SQL [1]. Its relational part is very similar to SQL, enough so that we will ignore any differences. The non-relational part is C# [17]: all expressions in a 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 is implemented as 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 `WHERE` clause, *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. An execution of a script is called a *job*.

2.2 C++ vs. C#

Much like Hadoop streaming [2], SCOPE jobs consist of multiple runtimes and languages and while the details of this paper are about SCOPE, the general problem is shared among many big data systems. The SCOPE compiler attempts to generate 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 CLR.

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 CLR.)

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

Jobs run on a distributed computing platform, called Cosmos, designed for storing and analyzing massive data sets. Cosmos runs on five clusters consisting of thousands of commodity servers [4]. Cosmos is highly scalable and performant: 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 challenging problem. We can hope to find interesting conclusions only if our profiling infrastructure is scalable. 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.

3.1 Job Artifacts

After execution of a SCOPE job, the runtime produces several artifacts that contain code and runtime information for every job stage. Job artifacts are indefinitely stored within Cosmos itself in a *job repository.* This provides two benefits: 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#. We provide an overview of the subset of artifacts which we use to profile the data center.

Job Algebra. The 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 be marked with the *nativeOnly* flag, indicating that the entire vertex runs as native (C++). However, it does not distinguish between native and user-defined operators.

Runtime Statistics. The runtime statistics file provides information on execution time for every job vertex and every operator inside the vertex. Among other statistics it includes CPU times, which we use as the primary metric of performance. Big data systems process significant amounts of data but often are CPU-bound[27] due to the large overheads behind serialization and de-serialization so we measure (in addition to bytes read and written) CPU time.

Generated C# and C++ Code. The SCOPE compiler generates both C# and C++ code for every job. An artifact containing the 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 containing the C# code includes implementations of non-native operators and user-written classes and functions defined inside the script. Both source and binary are available for the generated code.

3.2 Static Analysis

After collecting the artifacts described in Section 3.1, we perform 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, an operator can run as managed code due to only a single method call, or because of 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 at the granularity of a job vertex. The goal is to look for opportunities to run an entire vertex as C++ code, which would remove all steps of data serialization between user and native operators within the vertex.

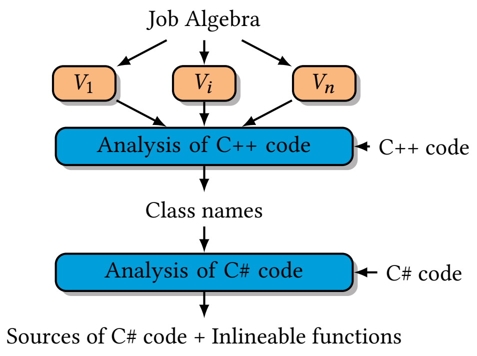

Figure 2: High level picture of static analysis

The first step of the analysis is to extract the names of each job vertex, which serves as a unique identifier for the vertex. Then for each vertex, the analysis parses the generated C++ to find the class containing the vertex implementation. As discussed in Section 3.1, for each vertex, the 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 the 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 the 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 for an operator to execute as C# just because of a single call to 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.

3.2.1 Detecting Sources of C# Code. To find .NET framework calls, it is enough to check whether a method definition comes from one of a small set of the core binaries in the .NET runtime. The analysis finds user-written functions by looking for their definition inside the script or in third-party binaries. Because the job 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 3.2.2), 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 the interface the class implements. We do not consider user operators for C++ translation: they generate quite complex code which would be non-trivial to translate into C++.

3.2.2 Analysis of User-Written Code. Inlining of a user-written function refers, per the standard definition, to replacing the call to the function in the script with the body of the function. We define *inlineable* methods as follows.

Definition 3.1 (Inlineable method). Method *m* is *inlineable* 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 assignment statements.
- It does not contain any references to the fields of an object.
- For all calls inside the method, arguments are passed by value (i.e., no *out* parameters or call-by-reference parameters).

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 EVALUATION

We analyze over 3,000,000 SCOPE jobs over a period of six days that run on five data centers at Microsoft. In summary, our experiments answer the following questions:

- *What is the proportion of time spent in native vs. non-native job vertices?* Between 43.70 % and 73.32 % of data center time is spent in job vertices that run managed code.
- *What proportion of time can be optimized by inlining UDFs using the current list of intrinsics?* Given the set of UDFs we found in our survey and the current list of intrinsics we can optimize up to 0.16 % of data center time.
- *What proportion of time can be optimized by extending the list of intrinsics? Which methods should be the most important for C++ implementation?* By increasing the list of intrinsics and optimizing all inlineable methods we can optimize up to 6.76 % of data center time. Furthermore, we conclude that *String* methods are the most important .NET framework methods amenable for C++ implementations.

4.1 Experimental Setup

To understand performance bottlenecks in SCOPE jobs we analyze over 3,000,000 jobs across 5 data centers. Table 1 lists, for each data center, the number of analyzed jobs along with their CPU time measured in hours. We observe that number of jobs and CPU time significantly vary between data centers. For example, data center *cosmos15* run the highest proportion of jobs we analyze, while *cosmos9* run the most expensive jobs. This is expected because different data centers are usually tailored for different types of jobs.

Data center	Number of jobs	CPU time (in hours)
cosmos8	375,974	28,559,063
cosmos9	171,203	40,714,052
cosmos11	851,222	23,312,271
cosmos14	474,911	21,299,039
cosmos15	1,200,026	31,324,407
Total:	3,073,336	145,208,834

Table 1: Analyzed jobs and their CPU time

The table shows that the jobs take a significant amount of resources. While we do not have access to the actual cost of these jobs, a very conservative estimate is to use 0.6 cents an hour, the cost of the cheapest Amazon EC2 instance [1] at the time of writing. Given that most of these jobs run recurrently every day (some every hour), extrapolating these costs amounts to over 650 million dollars per year. Thus, even a 1% performance improvement on these jobs will result in significant performance savings.

4.2 Native vs. Non-Native Time

The first goal of our analysis is to determine the amount of time spent between native vertices and vertices containing non-native code in SCOPE jobs. As mentioned in Section 2.2, SCOPE runs all relational logic efficiently in native code, while user-defined non-relational code is run in the CLR. Apart from avoiding the inherent overheads of running in non-native mode, the relational logic has the additional advantage of using all of the traditional optimizations modern databases typically perform. From prior analyses, it was known that around 80% of the SCOPE jobs use only relational constructs and thus run purely natively. Thus, at the outset, it was not obvious that non-relational optimizations would provide overall datacenter performance improvements.

Figure 3 shows for every data center the time spent executing native vertices versus vertices with non-native code. For this analysis, we combined the time taken by every job vertex from *Runtime Statistics* with the analysis of C++ code that determines whether each operator within a vertex is run in native or non-native mode. Our analysis of the C++ code is conservative and reports an operator as running in non-native mode only if the analysis is able to detect the source of the managed C# code or managed dll. Due to this conservative analysis, we tag some vertices as gray if the job metadata claims to include non-native operators but we are unable to detect the source. Modulo bugs in the SCOPE job metadata, these gray vertices are likely to be non-native vertices. But without improving our analysis we are unable to confirm this.

Figure 3 shows that the time spent in vertices with non-native operators represents a large fraction of data center time, ranging from 43.7% for cosmos15 to 73.3% for cosmos9. We can derive many conclusions from these results. First, these results could reflect the fact that the decades of work in optimizing relational code has borne fruit — purely relational components account for a smaller percentage of datacenter runtime. Second, it could very well be the case that jobs with inherently expensive computations require

[1]as of August 2017

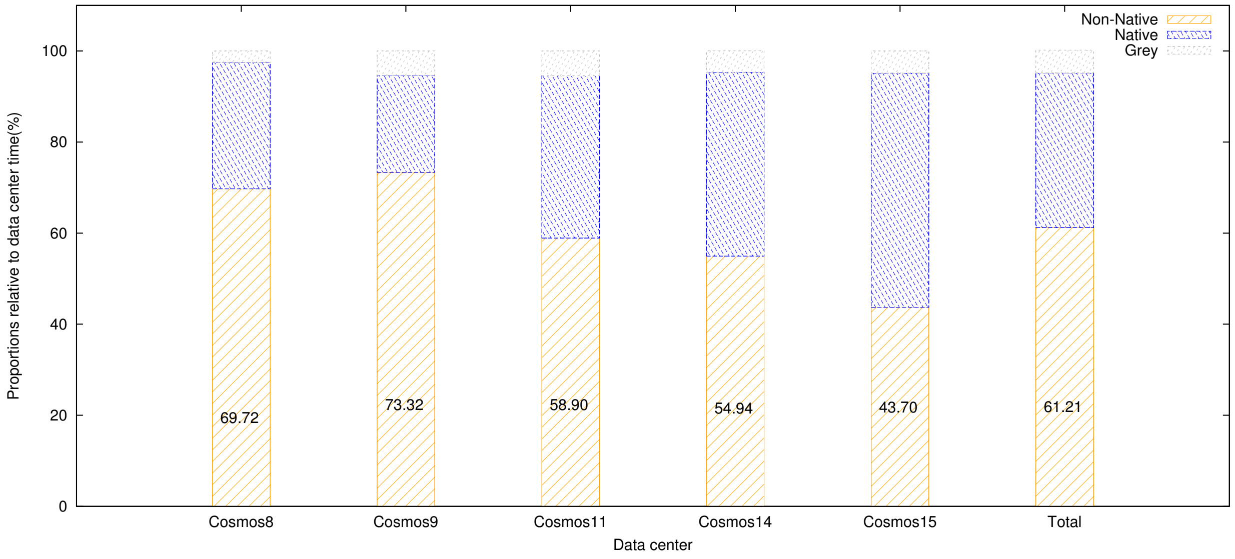

Figure 3: Time spent in native vs. non-native vertices

logic that does not fit within the relational subset of SCOPE, and thus requires the use of non-native code. Finally, these results could reflect the inherent overheads of running non-native code in the context of big-data processing.

We did preliminary experimentation on a small subset of the jobs locally to study the performance bottlenecks of SCOPE jobs with managed operators. Our profiles show that the presence non-relational components reduce the throughput of a job by a factor of 10× or more, with the performance bottleneck being the serialization/deserialization overhead of converting data into and from C# objects. Note that we are unable to run most of the jobs locally as accesses to the data they process is severely restricted due to privacy concerns. Thus, it is quite possible that the results from our preliminary experimentation might not be representative of the jobs that run on the datacenters. Nevertheless, conversations with the SCOPE team validated these experiments, and Figure 3 shows the potential performance improvements possible by optimizing the interaction between native and non-native parts of SCOPE.

4.3 Optimizable Job Vertices

We say a job vertex is *optimizable* if its only source of managed code comes from inlineable methods that in turn have only calls to existing intrinsics. An example of such a job is shown in Figure 1. This is an extremely conservative definition, but it allows us to quantify how much data center time we can optimize given the current list of intrinsics. Moreover, by inlining method calls, we expect an entire vertex to run as native code, which should significantly improve the vertex execution time.

Figure 4 shows the proportion of CPU time of *optimizable* vertices relative to data center time and to time spent in vertices with non-native code. We observe that with the current list of intrinsics we can optimize a relatively small proportion of data center time. For example, in cosmos9 that runs the most expensive jobs, we can optimize at most 0.01% of data center time. The situation is slightly better in cosmos14 and cosmos15, but in these data centers, the proportion of non-native time is relatively lower compared to cosmos9.

The crucial observation is that given results illustrate only the time in data centers that can be affected by inlining method calls in optimizable vertices. To measure the actual performance improvement it is necessary to rerun every optimized job. Further details on performance improvements for several jobs we optimize are given in Section 5.

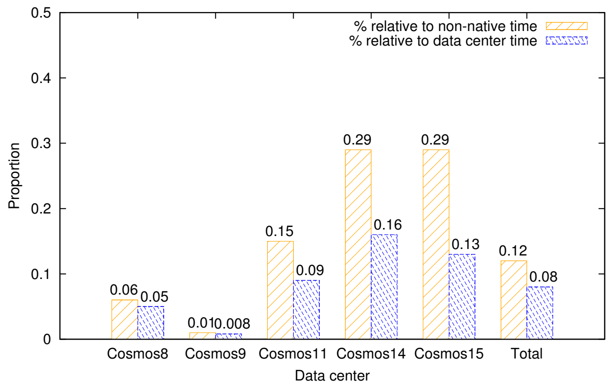

Figure 4: Optimizable job vertices

4.4 Potential for C++ Translation

To motivate the importance of providing the C++ implementation for more framework methods, we measure how much time is spent in the following type of vertices:

- Vertices with .NET framework calls as the only source of managed code
- Vertices with .NET framework calls or calls to inlineable methods as the only source of managed code

We call these vertices *potentially optimizable*, because they can run as native by increasing the list of intrinsics.

Figure 5 shows the proportion of time spent in potentially optimizable vertices relative to data center time. We measure the proportions by assuming that all .NET framework methods have C++ implementation. Results illustrate that we can optimize between 1.01 and 6.76 % of data center time by just increasing the list of intrinsics. Even though 1.01 % of time spent in cosmos9 looks relatively low, it counts for almost 407,140 CPU hours for a period of several days. Knowing this type of impact motivates the future work on enabling more C++ translation of framework methods.

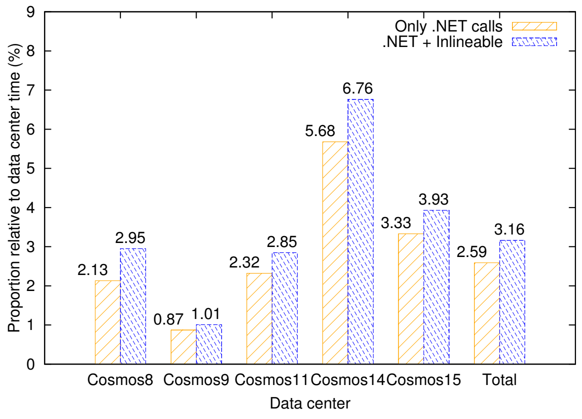

Figure 5: Potentially optimizable job vertices

4.4.1 Most Relevant .NET Framework Methods. Assuming that all .NET framework methods have C++ implementation is unrealistic. To provide more insights on the framework methods that actually matter, we perform two types of study: the study of the most relevant methods considering the execution time of a vertex and the study of the most important method types.

For the first study, we take all .NET framework methods called in potentially optimizable vertices and rank them based on the vertex execution time. Table 2 shows for every data center ten most important framework methods. The last row further illustrates how much data center time can be optimized if all methods in the list become intrinsics. We further highlight methods that appear to be relevant across many data centers. For example, *System.String.ToLower* and *System.String.Concat* are among the most relevant methods across all data centers. Furthermore, if the native implementation is provided for the first ten methods in cosmos14, it would be enough to optimize more than 5% of data center time.

Figure 6 illustrates the most important method types for data center cosmos11. The results are comparable for other data centers. *String* methods dominate and they count as the only source of non-native code in 1.49% of the time spent in potentially optimizable vertices. Other method types are significantly less relevant,

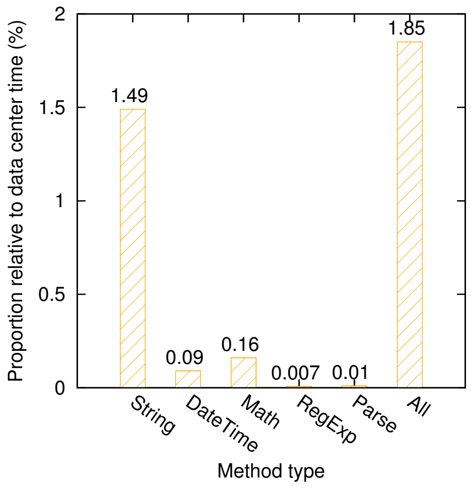

Figure 6: Relevance of .NET framework method types (cosmos11)

but when combined they influence 1.85% of the data center time. These studies show the potential for improving data center performance by providing more intrinsics and thus enabling more C++ translation.

5 CASE STUDIES

In order to quantify the effects of optimizing the scripts, we performed several case studies. 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. We then manually looked at the top jobs from a ranked list (by CPU time) of jobs containing an optimizable vertex.

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 6 jobs re-run by their owners. We roughly categorize the jobs by their total CPU time: short, medium, and long.

5.1 Optimizations With Effects On Job Algebra

As explained in Section 1, the optimizer may choose to modify the job algebra given the new information available to it. For example, predicates might be pushed deeper into the DAG which can result in dramatic data reduction. However, none of the case studies ended up causing this kind of optimization.

Cosmos8	Cosmos9	Cosmos11	Cosmos14	Cosmos15
Convert.ToInt64	**String.Equals**	String.Replace	**DateTime.ToString**	**String.ToLower**
Int32.Parse	**String.ToLower**	**String.ToLower**	String.IndexOf	String.LastIndexOf
String.ToLower	Int32.Parse	String.ToUpper	DateTime.ToLocalTime	**DateTime.ToString**
String.Concat	String.Replace	**String.Concat**	**String.ToLower**	**String.Concat**
String.Replace	Convert.ToDateTime	String.Trim	String.ToUpper	Convert.ToUInt64
Double.Parse	Regex.isMatch	Math.Max	Regex.IsMatch	Enumerable.SelectMany
Math.Round	DateTime.ToUniversalTime	**String.Equals**	**String.Equals**	Enumerable.Distinct
Char.NewArr	**String.Concat**	TimeSpan.Days	**String.Concat**	String.Format
String.ToUpper	TimeSpan.Days	**DateTime.ToString**	String.Trim	**String.Equals**
String.Upper	DateTime.Subtract	String.ToCharArray	String.Split	String.IndexOf
1.27%[2]	0.63%	1.61%	5.15%	1.8%

Table 2: Most relevant .NET Framework methods per data center. All methods are within the `System` **namespace. Methods in bold are those that appear in the top 10 in at least 3 of the five data centers.**

5.2 Optimizations Without Effects On Job Algebra

Even if the physical plan does not change, the resulting program might be more efficient if it avoids the native to managed transition. 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#.

In total, we looked at 6 re-run jobs, summarized in Figure 7. For one job (D), the optimization did not trigger C++ translation of an inlined operator because the operator called to a non-intrinsicable method that we mistakenly thought was an intrinsic. After detecting this problem, we fix the set of intrinsics and use the new set to obtain data presented in Section 4.

For jobs A and B, we were able to perfom the historical study over a period of 18 days. Both jobs are medium-expensive jobs, run daily and contain exactly one optimizable vertex due to a UDF. In both cases, inlining that UDF resulted in the entire vertex being executed in C++. Figure 8 shows the improvements in CPU time and throughput for an optimized vertex in Job A over an 18 day period, the last 5 of which were with the inlined UDF. The values are normalized by the average of the unoptimized execution times; the optimized version saves approximately 60% of the execution time. However, the normalized vertex CPU time in Job B does not show any consistent improvement for the last five jobs. Closer analysis of the vertex shows that the operator which had been in C# accounted for a very tiny percentage of the execution time for the vertex. This is consistent with our results for Job A, where the operator had essentially been 100% of the execution time of the vertex.

We also optimized Job F, a very low cost job. It only runs a few times a month, so we were able to obtain timing information for only a few executions. The vertex containing the optimized operator accounted for over 99% of the overall CPU time for the entire job. We found the CPU time to be highly variable; perhaps this is because the job runs so quickly so it is more sensitive to the batch environment in which it runs. However, we found the throughput measurements to be consistent: the optimized version provided twice the throughput for the entire job (again, compared to the average of the unoptimized version).

Finally, for jobs C and E we were not able to perform the same kind of historical study: instead we have just one execution of the optimized scripts. For this execution we found improvements in both vertex and job CPU times.

6 THREATS TO VALIDITY

Underapproximation of performance impact. The amount of time that can be optimized by either increasing the list of intrinsics or method inlining is an underapproximation of the total optimizable time. We do not consider effects of inlining 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 [9]. Understanding the impact of inlining on other 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 non-native 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.

Job Name	*C++ translation*	*Job Cost*	*CPU time*		*Throughput*
			Vertex Change	*Job Change*	
A	yes	medium	59.63%	23.00%	30%
B	yes	medium	no change	no change	no change
C	yes	low	41.98%	25.00%	38%
D	no	-	-	-	-
E	yes	high	7.22%	4.79%	5%
F	yes	low	no change	no change	115%

Figure 7: Summary of case studies. The reported changes are percent improvements in CPU time and throughput.

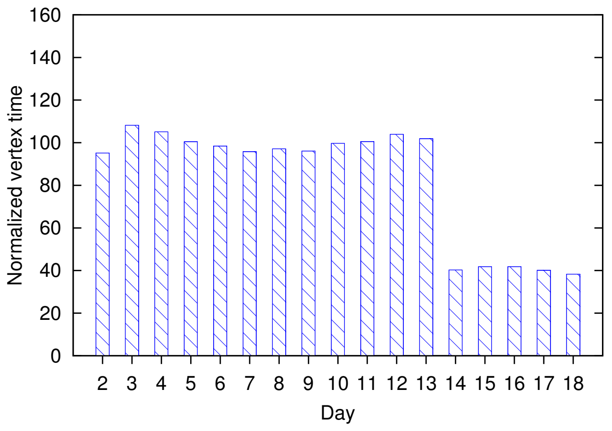

Figure 8: Case Study A

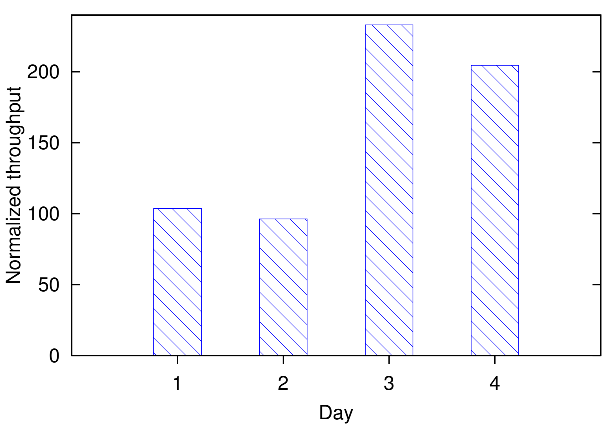

Figure 9: Case Study F

7 RELATED WORK

7.1 MapReduce Optimizations

Since MapReduce [11] is the de facto programming model for big data analytics [15], the performance of MapReduce is crucial to ensure optimal resource utilization and efficiency. Various approaches have been proposed for the optimization of MapReduce jobs. They include techniques to improve the scheduling of task execution [20, 35], to efficiently perform joins and indexing [12, 14], an analysis for automatic work-sharing across multiple jobs [25] and an extension to MapReduce model for efficiently merging the data computed by map and reduce modules [33] . Although some of the standard database optimizations, such as filter pushdown are implemented in Pig [26], the recent work by Jahani et al. [21] suggests that many traditional query optimizations are not applicable for MapReduce because of the user-written operators. It further proposes several optimizations of *map()* functions by targeting data-centric programming idioms [21]. Our work shows that time spent in non-relational code takes a large fraction of data center time and cross-language optimizations demonstrate a potential to significantly improve the performance of MapReduce jobs.

7.2 Profiling MapReduce Jobs

Many MapReduce systems, such as Hadoop [2], provide facilities for monitoring cluster performance. However, the collected metrics represent cluster-level information from which the regular user does not benefit. Enabling users to optimize their jobs by tuning job parameters, Herodotou et al. [19] propose a dynamic binary instrumentation of the MapReduce framework to capture dataflows and costs during job execution at the task level or the phase level. In contrast to the dynamic analysis, our approach to profiling big data jobs is purely static and based on the analysis of job artifacts. Doing this allows us to analyze a large number of jobs without introducing any additional overhead.

7.3 Query Optimizations

There has been extensive work in optimizing relational queries since the early '70s [5]. A large class of optimizations include exploiting commutativity among operators [7, 32],reducing multi-block queries to single block [22, 24], using semijoin techniques to optimize multi-block queries [23, 31], query materialization [6, 29], and query indexing [3, 13, 30]. Several approaches address optimizations of stored procedures (also called user-defined predicates) in relational systems [8, 18]. While these aforementioned techniques are powerful in optimizing relational queries, traditional query optimizers treat non-relational code as a black-box.

8 FUTURE WORK

We plan to pursue several avenues in the near future. First, we would like to do a broader set of experiments. Even in our limited case studies we found that it is not always beneficial to inline UDFs. It would be necessary to have an analysis that can predict when to perform the optimization.

Furthermore, we would like to explore ways to relax the restrictions on which methods we can inline. In particular, user-defined operators (UDOs) consume an inordinate amount of time in each data center: being able to optimize them could provide a very large benefit.

Finally, exploiting other optimizations in the context of big data jobs at Microsoft is also left for future work.

9 CONCLUSIONS

Big data systems are not as performant as they could be. While this is true of all systems, the scale that such systems operate at means that any performance gap can translate into a large amount of money. At the same time, large distributed systems are not as easy to measure and instrument as single-box systems. Thus, we believe it crucial for new tools and processes to be developed for monitoring their performance.

We also believe having an expressive general-purpose language like C# or Java integrated into a big data query language is a good thing: programmers should be able to re-use existing components in languages that they are already comfortable with. However, such multi-language paradigms break the barriers that current program analysis and optimization tools are based on.

We have shown some techniques for measuring data center peformance and the utility of even small, simple optimizations. Clearly, we have just scratched the surface of two very promising research areas.

ACKNOWLEDGEMENTS

We want to especially thank the SCOPE team for answering our endless questions and also the product groups that helped us experiment with their scripts. This work was partially funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project.

REFERENCES

[1] [n. d.]. 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/. ([n. d.]). Accessed: 2017-05-09.

[2] Apache [n. d.]. *Hadoop Streaming*. Apache, https://hadoop.apache.org/docs/current/hadoop-streaming/HadoopStreaming.html.

[3] E. Bertino and W. Kim. 1989. Indexing Techniques for Queries on Nested Objects. *IEEE Trans. on Knowl. and Data Eng.* 1, 2 (June 1989), 196–214. https://doi.org/10.1109/69.87960

[4] 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

[5] Surajit Chaudhuri. 1998. An Overview of Query Optimization in Relational Systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '98)*. ACM, New York, NY, USA, 34–43. https://doi.org/10.1145/275487.275492

[6] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. 1995. Optimizing Queries with Materialized Views. In *Proceedings of the Eleventh International Conference on Data Engineering (ICDE '95)*. IEEE Computer Society, Washington, DC, USA, 190–200. http://dl.acm.org/citation.cfm?id=645480.655434

[7] Surajit Chaudhuri and Kyuseok Shim. 1994. Including Group-By in Query Optimization. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 354–366. http://dl.acm.org/citation.cfm?id=645920.672834

[8] D. Chimenti, R. Gamboa, and R. Krishnamurthy. 1989. Towards an Open Architecture for LDL. In *Proceedings of the 15th International Conference on Very Large Data Bases (VLDB '89)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 195–203. http://dl.acm.org/citation.cfm?id=88830.88851

[9] John Darlington. 1978. A Synthesis of Several Sorting Algorithms. *Acta Inf.* 11, 1 (March 1978), 1–30. https://doi.org/10.1007/BF00264597

[10] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[11] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[12] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. 2010. Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 515–529. https://doi.org/10.14778/1920841.1920908

[13] Yi Fang, Marc Friedman, Giri Nair, Michael Rys, and Ana-Elisa Schmid. 2008. Spatial Indexing in Microsoft SQL Server 2008. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*. ACM, New York, NY, USA, 1207–1216. https://doi.org/10.1145/1376616.1376737

[14] Avrilia Floratou, Jignesh M. Patel, Eugene J. Shekita, and Sandeep Tata. 2011. Column-oriented Storage Techniques for MapReduce. *Proc. VLDB Endow.* 4, 7 (April 2011), 419–429. https://doi.org/10.14778/1988776.1988778

[15] Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. 2009. Building a High-level Dataflow System on Top of Map-Reduce: The Pig Experience. *Proc. VLDB Endow.* 2, 2 (Aug. 2009), 1414–1425. https://doi.org/10.14778/1687553.1687568

[16] Zhenyu Guo, Xuepeng Fan, Rishan Chen, Jiaxing Zhang, Hucheng Zhou, Sean McDirmid, Chang Liu, Wei Lin, Jingren Zhou, and Lidong Zhou. 2012. Spotting Code Optimizations in Data-Parallel Pipelines through PeriSCOPE.. In *OSDI*. 121–133.

[17] Anders Hejlsberg, Mads Torgersen, Scott Wiltamuth, and Peter Golde. 2010. *C# Programming Language* (4th ed.). Addison-Wesley Professional.

[18] Joseph M. Hellerstein and Michael Stonebraker. 1993. Predicate Migration: Optimizing Queries with Expensive Predicates. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD '93)*. ACM, New York, NY, USA, 267–276. https://doi.org/10.1145/170035.170078

[19] Herodotos Herodotou and Shivnath Babu. 2011. Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs. 4 (01 2011), 1111–1122.

[20] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 261–276. https://doi.org/10.1145/1629575.1629601

[21] 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

[22] Won Kim. 1982. On Optimizing an SQL-like Nested Query. *ACM Trans. Database Syst.* 7, 3 (Sept. 1982), 443–469. https://doi.org/10.1145/319732.319745

[23] Inderpal Singh Mumick and Hamid Pirahesh. 1994. Implementation of Magic-sets in a Relational Database System. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD '94)*. ACM, New York, NY, USA, 103–114. https://doi.org/10.1145/191839.191860

[24] M. Muralikrishna. 1992. Improved Unnesting Algorithms for Join Aggregate SQL Queries. In *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB '92)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 91–102. http://dl.acm.org/citation.cfm?id=645918.756653

[25] Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. 2010. MRShare: Sharing Across Multiple Queries in MapReduce. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 494–505. https://doi.org/10.14778/1920841.1920906

[26] Christopher Olston, Benjamin Reed, Adam Silberstein, and Utkarsh Srivastava. 2008. Automatic Optimization of Parallel Dataflow Programs. In *USENIX 2008 Annual Technical Conference (ATC'08)*. USENIX Association, Berkeley, CA, USA, 267–273. http://dl.acm.org/citation.cfm?id=1404014.1404035

[27] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 293–307. https://www.usenix.org/conference/

nsdi15/technical-sessions/presentation/ousterhout

[28] 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

[29] Thomas Phan and Wen-Syan Li. 2008. Dynamic Materialization of Query Views for Data Warehouse Workloads. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering (ICDE '08)*. IEEE Computer Society, Washington, DC, USA, 436–445. https://doi.org/10.1109/ICDE.2008.4497452

[30] Timos K. Sellis. 1988. Intelligent caching and indexing techniques for relational database systems. *Information Systems* 13, 2 (1988), 175 – 185. https://doi.org/10.1016/0306-4379(88)90014-2

[31] Praveen Seshadri, Joseph M. Hellerstein, Hamid Pirahesh, T. Y. Cliff Leung, Raghu Ramakrishnan, Divesh Srivastava, Peter J. Stuckey, and S. Sudarshan. 1996. Cost-based Optimization for Magic: Algebra and Implementation. *SIGMOD Rec.* 25, 2 (June 1996), 435–446. https://doi.org/10.1145/235968.233360

[32] Weipeng P. Yan and Perake Larson. 1995. Eager Aggregation and Lazy Aggregation. In *Proceedings of the 21th International Conference on Very Large Data Bases (VLDB '95)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 345–357. http://dl.acm.org/citation.cfm?id=645921.673154

[33] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. 2007. Map-reduce-merge: Simplified Relational Data Processing on Large Clusters. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD '07)*. ACM, New York, NY, USA, 1029–1040. https://doi.org/10.1145/1247480.1247602

[34] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. *HotCloud* 10, 10-10 (2010), 95.

[35] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. 2008. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 29–42. http://dl.acm.org/citation.cfm?id=1855741.1855744