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

Anonymous Author(s)

ABSTRACT

Systems for big data analytics are programmed in languages that combine relational (SQL) with non-relational code (Java, C♯, Scala). By writing big data jobs in a declarative manner, the decision on how to distribute the job over processors is managed automatically by the runtime. Even though SQL optimizations have been extensively researched over past several decades, still little is known on why and how to optimize such cross-language systems. In this paper, we illustrate that job stages with non-relational code take between 45-70% of data center CPU time by empirically studying over 3 million SCOPE jobs across five data centers within Microsoft. 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),* 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Large-scale data-processing frameworks, such as MapReduce [?], SCOPE [?], 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.

However, these systems are known to lag far behind traditional database systems in runtime efficiency [? ?], primarily because of the flexibility of the programming model they support. For instance, recent work [?] shows that the key bottlenecks in Spark are not the disk or 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. While SCOPE, as described later in Section 2, supports a hybrid native (C++) and C# runtime partly to alleviate this overhead, our analyses of these jobs 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-SQL code in these jobs blocks the powerfulness of relational optimizations these data-processing runtimes implement [?].

The goal of this work is to study and better understand the key performance bottlenecks in modern data-processing systems, and propose potential for cross-language optimizations. While this paper primarily focusses on 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 script that interleaves relational logic with C# expressions. The SCOPE compiler treats certain C# expressions as *intrisics.* In other words, it understands that the expression `String.isNullOrEmpty(A)` is checking whether the field `A` is null or empty and accordingly executes the entire query natively with performance akin to an equivalent SQL query. Also, if this query is part of a larger query, then the SCOPE optimizer performs the relational optimization of moving the filter `String.isNullOrEmpty(A)` to earlier portions of the query to significantly reduce the data processed during the execution.

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

(a) Job running as native.

```
data = SELECT *
       FROM inputStream
       WHERE CheckIfNullOrEmpty(A);

#CS
bool CheckIfNullOrEmpty(string input) {
   return !String.IsNullOrEmpty(A);
}
```

(b) Job running as non-native.

Figure 1: Examples of SCOPE scripts that run as native and non-native

On the other hand, the script in Figure 1b is a slight variation of the query above where the user implemented the filter in a separate C# function. Unfortunately, the SCOPE compiler treats the call to the user-defined function as a black box. As a result, the job runs in a hybrid mode where parts of the input are run natively and the rest are run 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.

Conference'17, July 2017, Washington, DC, USA
© 2017 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 BACKGROUND

Many (all?) "Big Data" systems are actually a mixture of languages []. In general, we categorize them as comprising a *relational* part and a *non-relational* part. The former is some variant of SQL [], i.e., a declarative data-flow language that expresses how data flows from inputs to outputs (both usually being relational tables) through a series of data-parallel operators such as filters, projections, and various types of joins. The non-relational part is expressed in an imperative language, like C#, Java, Scala, or Python [].

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

2.1 Scope Language

SCOPE [] is a Big Data system used internally within Microsoft. However it is 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#[]. This means that all expressions in a SCOPE program (aka *script*) are written as C# expressions. In addition, SCOPE allows user-defined functions (written as C# methods) and user-defined operators (written as C# classes).

2.1.1 Execution of a Scope Script. A SCOPE 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 Running C++ vs. C♯ Vertex

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 .NET Runtime. But when a script contains a C# expression, such as in Figure **??**, then each row in the input table must be converted to a C# representation, i.e., a C# object representing the row must be created. 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.)

3 PROFILING INFRASTRUCTURE FOR DATA CENTERS

3.1 Scale of The Problem

3.2 Job Artifacts

3.3 Static Analysis

Figure 2: High level picture of main steps during static analysis

3.3.1 *Sources of Non-Native Code.*

- .NET Framework Calls
- User written functions
- Custom processors, reducers, combiners, extractors,etc.

3.3.2 *Analysis of User-Written Code.*

4 CROSS-LANGUAGE OPTIMIZATIONS

4.1 Optimizations Without Effects On Job Algebra

4.2 Optimizations With Effects On Job Algebra

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

- 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 LIMITATIONS AND THREADS TO VALIDITY

Underapproximation of performance impact.

Granularity of the analysis.

Challenges for implementing more intrinsics.

8 RELATED WORK

8.1 Query Optimizations

....

9 CONCLUSIONS