Niijima: Sound and Automated Computation Consolidation for Efficient Multilingual Data-Parallel Pipelines

Guoqing Harry Xu
UCLA

Margus Veanes, Michael Barnett, Madan Musuvathi, Todd Mytkowicz, Ben Zorn
Microsoft Research

Huan He, Haibo Lin
Microsoft

Abstract

Multilingual data-parallel pipelines, such as Microsoft's Scope and Apache Spark, are widely used in real-world analytical tasks. While the involvement of multiple languages (often including both managed and native languages) provides much convenience in data manipulation and transformation, it comes at a performance cost — managed languages need a managed runtime, incurring much overhead. In addition, each switch from a managed to a native runtime (and vice versa) requires marshalling or unmarshalling of an ocean of data objects, taking a large fraction of the execution time. This paper presents Niijima, an optimizing compiler for Microsoft's Scope/Cosmos, which can *consolidate C#-based user-defined operators (UDOs) across SQL statements*, thereby reducing the number of dataflow vertices that require the managed runtime, and thus the amount of C# computations and the data marshalling cost. We demonstrate that Niijima has reduced job latency by an average of 24% and up to 3.3×, on a series of production jobs.

CCS Concepts • **Information systems → Data management systems**; • **Software and its engineering → Compilers**;

Keywords Big Data system, Scope/Cosmos, compiler optimization, SQL, user-defined operator

ACM Reference Format:
Guoqing Harry Xu, Margus Veanes, Michael Barnett, Madan Musuvathi, Todd Mytkowicz, Ben Zorn, and Huan He, Haibo Lin. 2019. Niijima: Sound and Automated Computation Consolidation for Efficient Multilingual Data-Parallel Pipelines. In *SOSP '19: Symposium on Operating Systems Principles, October 27–30, 2019, Huntsville, ON, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3341301.3359649

SOSP '19, October 27–30, 2019, Huntsville, ON, Canada

ACM ISBN 978-1-4503-6873-5/19/10…$15.00
https://doi.org/10.1145/3341301.3359649

1 Introduction

Modern data-parallel systems often involve components written in different programming languages for increased expressiveness and convenience. For example, systems in Google rely on interactions among Java, C++, and Go programs; Microsoft's Cosmos/Scope [5] jobs are written in SQL and C#; Apache Spark [44], which runs on the Java Virtual Machine (JVM), allows developers to define processing logic in four languages: Java, Scala, Python, or R. The execution of an analytical job often alternates between a native runtime and a managed runtime, requiring frequent format changes when data flows from one environment into another.

1.1 Problem Statement

Scope is a multilingual data-parallel system used widely at *Microsoft* for a range of data processing tasks. *Scope* consists of a scripting language that provides seamless integration of SQL and C# — SQL statements specify a dataflow graph, forming the backbone of a *Scope* pipeline, while C# methods can be freely invoked on table columns in each SQL statement to perform column transformations. User-defined operators (UDO) such as `PROCESS`, `COMBINE`, and `REDUCE` are table-valued functions that take a set of table rows as input and output another set. They are written in C# and attached to SQL statements [5]. The following snippet shows a simple *Scope* program that first generates a new table `data1` by selecting two columns *A* and *B* from `data` and turning all strings in *A* into lower case; it next performs a Map/Reduce style reduction based on column *B* using a UDO `MyReducer`.

```
data1 = SELECT A.ToLower() AS C, B FROM data;
data2 = REDUCE data1 ON B USING MyReducer;
```

A *Scope* script is compiled into a set of C++ and C# programs that mutually invoke each other — the relational logic is implemented in C++ for efficiency while the C# methods are compiled into C# programs. This compilation model brings two major challenges in efficiency. First, C# programs

```
1  DAT =  SELECT T.ToLower() AS T, ... FROM
2           EXTRACT T : string, ... FROM "SearchLog.txt"
3           USING TextExtractor;
4   ... /* 17 statements */
5  A  =   SELECT T, ... FROM DAT WHERE cond1
6   UNION SELECT T, ... FROM DAT WHERE cond2
7     ... /* 8 SELECT statements here */
8   UNION SELECT T, ... FROM DAT WHERE cond11;
9   ... /* 23 statements */
10 B = SELECT StringToTime(T) AS Time, ... FROM A WHERE ...;
11  ... /* 458 statements */
12 C = SELECT Time, ... FROM B HAVING Time.Date ==
       FromBinary(5248277762427387904
       /*2018-04-13T00:00:00.0000000Z*/).Date;
13  ...
```

(a) Code snippet from a Bing service.

```
1  DAT = SELECT StringtoTime(T.ToLower()) AS Time,... FROM
2          EXTRACT T : string, ... FROM "SearchLog.txt"
3          USING TextExtractor;
4   ... /* 17 statements */
5  A  =   SELECT Time, ... FROM DAT WHERE cond1 &&
         Time.Date == FromBinary(5248277762427387904).Date
6   UNION SELECT Time, ... FROM DAT WHERE cond2 &&
         Time.Date == FromBinary(5248277762427387904).Date
7     ... /* 8 SELECT statements here */
8   UNION SELECT Time, ... FROM DAT WHERE cond11 &&
         Time.Date == FromBinary(5248277762427387904).Date;
9   ... /* 23 statements */
10 B = SELECT Time, ... FROM A WHERE ...;
11  ... /* 458 statements */
12 C = SELECT Time, ... FROM B;
13  ...
```

(b) Consolidated version.

Figure 1. A real consolidation example; highlighted in red are C# computations.

are notoriously more expensive to execute than native code due to its need of a managed (.NET) runtime. While effort has been made to translate C# code into C++, translation is generally difficult in the presence of dynamically allocated objects since appropriately freeing these objects requires understanding of their liveness, which is undecidable. Character encoding is another major obstacle because C++ and .NET use different encoding schemes to represent strings.

The second challenge is that mixing C++ with C# incurs heavyweight data transfer costs between the native and .NET runtime. For example, to call `ToLower` in the above example, all (*e.g.*, billions of) rows from table `data` need to be *deserialized* from native bytes into .NET objects, which will, in turn, be *serialized* back to native bytes before they can be pushed to the next dataflow vertex. Furthermore, many of these C# computations are not strictly necessary at their current locations. For example, calling a C# method to transform all rows of a table from one format to another and then filtering out most rows is clearly an inefficiency. Delaying the method call until the filtering is done can significantly reduce the amount of computations if the filtering condition does not depend on the result of the call. However, this optimization, although simple, cannot be effectively performed by a SQL optimizer that does not understand the semantics of imperative code.

State of the Art. Optimization of dataflow pipelines is an extensively studied topic [5, 6, 13, 17, 18, 26, 30, 39, 42–46]. Most of these optimizations are system-level techniques that target increased parallelism or reduced I/O for various throughput and latency goals. Safely optimizing C# computations in SQL requires an uniform representation of dependence for SQL and C#, which none of the previous work has considered.

PeriScope [17] and Blitz [36] are two compiler-based techniques that can reduce, respectively, the shuffle-related I/O and number of compute stages by transforming/synthesizing (C#-based) UDO code. They are both designed to optimize for specific cases in *Scope* (*e.g.*, PeriScope for `REDUCE` and Blitz for `JOIN`). Furthermore, they are both *unsound* — they can potentially change the semantics of a program; hence, they can be used only for assisting manual tuning, *not* for performing automated optimization in a production system.

1.2 Our Contributions

Niijima[1] is a *sound and automated* program transformation technique that aims to minimize the amount of C# computations and/or their related serialization/deserialization costs by *consolidating* these computations in a SQL-based *Scope* pipeline. After consolidation, the C# computations, which used to be scattered all over the pipeline, are moved into a small number of SQL statements in a *semantics-preserving manner*. To create more consolidation opportunities, Niijima also attempts to *pull up filters*. Filter pullup leads to earlier filtering of data items, reducing the costs of both computation and shuffling. Another important goal of filter pullup is to get filters out of the way of computation consolidation, enabling more opportunities for moving C# computations.

Motivating Example. Figure 1 depicts a consolidation example, with (a) and (b) illustrating the original and the optimized program. The example in (a) is extracted from a production script — it loads a number of columns from a text file on the distributed storage using a C#-based extractor. The extracted data goes through a sequence of format transformation and filtering steps until it reaches Line 5 where the results from 11 `SELECT` clauses are combined using a `UNION` command. Later, Line 10 converts column T of each row from `String` into `Time` using a C# API `StringToTime`. Eventually, Line 12 filters out rows whose `Time` is not 00:00:00 of 2018-04-13.

Niijima performs two optimizations on the example, as illustrated in Figure 1(b). First, the C# operation `StringToTime` is moved from Line 10 all the way up to the first statement — it gets consolidated with another C# computation `T.ToLower`. Second, predicate `Time.Date == FromBinary(...)` gets pulled from Line 12 up to each of the 11 `SELECT` clauses from Line 5 to Line 8 in the `UNION` statement. The type of the filter changes

[1] Niijima is a Japanese volcanic island that was merged with another island by lava in 2013.

Action	Benefit
Computation pushdown	Delayed compute; reduced computation and serialization
Computation merging	Increased #pure SQL stmts; reduced serialization
Filter pullup	Earlier filtering; reduced data and computation

Table 1. Three major benefits of Niijima's optimizations.

from HAVING to WHERE and it is merged with each of their original WHERE conditions into a new predicate of the conjunctive form. The filter type determines whether the filter is applied *before or after* the C# computation. In particular, WHERE is a pre-transformation filter that is applied before any C# computation in the same statement is executed while HAVING is a post-transformation filter that is applied after the execution of all the C# computations in the statement.

These optimizations do not change the semantics of the program because all of the C# methods involved are *pure methods without side effects*. The performance benefit here is two-fold. (1) Pulling the filter enables earlier filtering of 2/3 of all rows at the UNION statement so that only a very small number of rows that pass the time check flow to the downstream operators. This leads to significantly reduced computation and serialization. (2) Consolidating StringToTime from Line 10 into Line 1 makes Line 10 a pure SQL statement. Hence, the total number of C++ operators increases, leading to fewer dataflow vertices that require the .NET runtime and thus less data serialization/deserialization effort. As a matter of fact, Figure 1(b) runs **2.91**× faster than Figure 1(a) when processing a real-world dataset.

This example clearly shows how filter pulling and computation consolidation pave the way for each other. Without consolidating StringToTime into Line 1, there would be no way for us to pull the predicate at Line 12 because the predicate uses column Time, which is defined by the C# method call in Line 10. Hence, the improvement would be much less significant if these optimizations are enabled individually.

Table 1 summarizes Niijima's three major benefits. Filter pulling and computation consolidation mutually benefit each other — pulling a filter may remove dependences and enable consolidation of more C# computations, while consolidating C# computations may, similarly, make it possible for us to pull filters that could not be pulled otherwise. Niijima performs these two optimizations iteratively until a fixed point is reached, guarded by a static profit metric that guarantees the profitability of each move (§3.2.2).

Niijima v.s. Query Optimizations. These optimizations appear to be similar to *predicate push-down* and *operator fusion* performed by traditional database query optimizers. For example, operator fusion fuses together operators in a query plan to minimize the materialization overhead by passing tuples efficiently between them. On the contrary, Niijima falls into the category of dataflow-based compiler optimizations, which differ from the existing query optimizations in the following two ways.

First, as a compiler (front-end) technique, Niijima's scope is the whole program. It can consolidate computations in the first SQL statement with those in the last statement, even if these two statements are thousands of statements apart. Operator fusion, however, focuses on operators close to each other in a dependence neighborhood on a DAG. For a program with thousands of statements, for example, fusing the first and the last operator of the DAG would require fusing most of the operators in the DAG — a task impossible to do by any query optimizer.

Second, although we pull predicates, we do not claim that predicate pulling is a new contribution. Instead, our novelty is that by having a large scope for consolidating computations, Niijima enables more predicates to be pulled. Pulling these predicates would, in turn, enable more computations to be consolidated. In fact, none of the query optimizers could perform the optimizations as shown in Figure 1 since such optimizations require the optimizer to (1) have a global scope and (2) move filters and computations simultaneously.

Niijima v.s. PeriScope and Blitz. Although techniques such as PeriScope [17] and Blitz [36] can optimize *Scope* programs, Niijima is the first *fully automated and sound* technique implemented in *Scope*'s production compiler. PeriScope aims to break C# code in REDUCE UDOs into parts that can be executed before and after each shuffle (*i.e.*, "smart cut") — this is, in general, a daunting task to achieve in a semantics-preserving manner due to issues such as variable aliasing and heap-related dependences. Blitz synthesizes new UDOs to optimize joins. However, to use Blitz, a *Scope* program has to be first modeled *manually* using a meta language, creating practicality obstacles. The synthesizer is also unsound when handling loops. Note that while soundness is less important for static bug finders, compiler optimizations must be sound to be used in production systems.

The key insight leading to Niijima's success is that it operates at a sweetspot in a vast space of possible optimizations. Unlike traditional (back-end) query optimizations that focus on relational logic, Niijima is a *front-end* optimization for both SQL and C#, enabling and complementing the existing back-end optimizations. Furthermore, unlike PeriScope and Blitz that attempt to analyze and transform UDOs, Niijima *analyzes C# computations together with SQL statements but never splits and transforms UDOs.* Instead, Niijima moves C# calls around SQL statements based on the inputs/outputs of these calls.

UDOs, which are no different than any C# methods, often have complex code logic and pointer usage. It is nearly impossible to safely and automatically transform or synthesize UDOs because this would require safe and precise treatment of pointer-induced aliasing and thus a whole-program

pointer analysis to analyze the methods directly and transitively reachable from each UDO. How to develop a precise and scalable pointer analysis alone is an unsolved problem, not to mention other practical challenges such as how to create new UDOs holding the generated/split code, how to create SQL statements to attach these UDOs, and how to consistently change input/output schemas to accommodate the UDO changes.

On the contrary, moving C# method calls around SQL statements is a much more tractable task — this needs only reasoning about (1) column dependences between relational tables and (2) inputs/outputs of the involved C# methods. Pointer-induced aliasing does not exist in this context. Niijima neither changes the body of any C# method nor the schema of any table; it only moves filters and merges computations with *guaranteed safety*. In addition, as dictated by the *Scope* specifications, the C# methods embedded in SQL are all side-effect-free methods whose execution would not change the external state. Hence, when calls to these methods are moved around, the program's semantics would remain the same as long as certain constraints regarding column dependences are satisfied (§3.2).

Niijima v.s. Other Potential Alternatives. It is worth noting that the ultimate goal of this work is to integrate Niijima into the production system, offering the performance benefit to thousands of (new and legacy) jobs at Microsoft. As a result, practicality is our main concern throughout Niijima's development. A potential alternative to reducing the data marshalling cost is to define a common standard data format, such as the one used in Apache Arrow [1], which can make it easier to pass data across languages. However, data format is a fundamental contract in the system on which all components depend. Changing the data format dictates re-implementation of most interfaces in the system, creating significant practicality obstacles.

Another potential solution is to add compiler/runtime support inside the .NET runtime to allow C# computations to operate directly over native data, similarly to what was proposed in Gerenuk [25] or Apache Tungsten [12]. However, it was clear to us that *Scope*'s production runtime cannot use a modified .NET framework.

Compared to these alternatives, Niijima is a compiler-based technique that performs source-to-source translation directly on *Scope* scripts. It is much less intrusive than these other potential approaches because it does not need to change any runtime components.

This paper makes the following major contributions:

- a study demonstrating the opportunity for performance improvement in production pipelines;
- a sound and fully automated program analysis and transformation framework for UDO-heavy SQL pipelines;
- the first implementation of such a source-code-level transformation in *Scope*'s production runtime, demonstrating both ease of application and generality; by contrast, PeriScope and Blitz were both implemented as tuning tools that cannot be used without developers in the loop;
- a set of results indicating significant reduction in overall time on real workloads. With the Niijima-enabled runtime, we have optimized, on one of *Microsoft*'s production clusters, 21 distinct *Scope* programs executed between 4/3/18 and 4/13/18. Our optimization improved their running time by up to 3.3×; the overall (geometric means of) reductions in the total CPU and serialization/deserialization time are, respectively, 20% and 22%.

2 Background and Motivating Study

This section first provides a gentle introduction of the Scope language. Next, we present a study over a set of production scripts that motivates Niijima's development.

2.1 *Scope* Background

Scope is a SQL-like declarative language [5] designed to facilitate large-scale data analytics at Microsoft. Like SQL, data is modeled as sets of rows composed of typed columns. Every rowset has a well-defined schema. *Scope* programs are executed on top of *Cosmos*, a distributed runtime designed to run on large clusters consisting of thousands of commodity servers with support for fault tolerance, data partitioning, resource management, and parallelism. Since the SQL-like constructs in Scope all have standard semantics, we focus our discussion on UDOs.

Input and Output UDOs. A *Scope* program reads input data with command `EXTRACT` and writes output data with command `OUTPUT`. Both `EXTRACT` and `OUTPUT` are C# methods. Although *Scope* provides a set of built-in extractors and outputters, users can easily develop their customized `EXTRACT` or `OUTPUT` logic by implementing standard interfaces.

Data Processing UDOs. In many cases, it is difficult or even impossible to express a complex operation declaratively with SQL commands. Examples include special data transformation (*e.g.*, turning all strings in a column from upper to lower cases) or customized aggregates. To overcome this challenge, *Scope* provides three highly-extensible commands: `PROCESS`, `REDUCE`, and `COMBINE`, all of which can be customized by the user in C#. `PROCESS` takes a rowset as input, processes each row in turn, and outputs a different rowset which may or may not have the same schema as the input. `REDUCE` takes as input a grouped rowset, processes each group, and outputs another sequence of rows per group. `COMBINE` is a binary operator that takes as input two rowsets, combining them and outputting a sequence of rows. These user-defined commands can be

used in a similar manner to traditional SQL commands such as SELECT, but their semantics can be much richer than the relational semantics of SQL commands because they are defined by imperative code in C#.

Scope *Compilation and Optimizations.* The *Scope* compiler parses the script, checks the syntax, resolves names, and generates a parse tree as the output. The parse tree is passed down to the *Scope* optimizer that performs an extensive set of query optimizations based on the Cascades framework [16], which also powers the qeury optimizer in Microsoft's SQL Server. At a high level, the optimizer enumerates all possible rewritings of a query expression and chooses the one with the lowest estimated cost. Finally, the optimizer generates an *execution plan*, which is submitted to the *Cosmos* execution environment for execution.

A physical execution plan is represented as a dataflow DAG. Each vertex on the DAG is a program and each edge represents a data channel. A vertex program is composed from *Scope* physical operators, which are implemented in C++. These operators may or may not call C#-based UDOs, depending on whether the program involves customized commands and/or C# methods. In cases they do call UDOs, data serialization/deserialization will occur. Otherwise, the vertex program is a pure C++ program running natively. Operators within a vertex program are executed in a pipelined manner. A vertex becomes runnable when its inputs are ready. The *Cosmos* runtime tracks the state of vertices and channels, schedules runnable vertices for execution, decides where to run a vertex, sets up the containers to run a vertex, and finally starts the program.

2.2 Motivating Study

Scope is being used daily for a variety of data analysis and data mining applications inside Microsoft. To understand how pervasively consolidation opportunities exist, we manually studied the 20 most expensive *Scope* programs executed between 9/1/17 and 9/10/17 on each of the three production clusters X, Y, and Z at *Microsoft*, each consisting of many thousands of machines. Millions of jobs are processed on these clusters every week. We sorted all jobs in this period on their cumulative CPU time (*i.e.*, the sum of the CPU time for each vertex on the dataflow graph). The 60 highest ranked jobs were obtained for manual inspection.

Statistics. Table 2 reports the statistics of the 60 programs we manually inspected. In terms of functionality, they can be roughly classified into three categories: search engine related services (31), machine learning tasks (14), and repository mining (15). These jobs all have very large cumulative CPU time. Each job has an average of 22 SQL statements, of which 13 (59%) have embedded C# computations. Of these 60 programs, 44 (77.3%) were automatically generated by tools and extremely difficult to understand by human (*e.g.*, a SQL statement can span hundreds of lines and process hundreds of columns).

CL	Time	LoC	SQL	C#	AG
X	2415/34381 (5668)	167/3578 (926)	3/124 (19)	2/97 (11)	14
Y	1319/19196 (3167)	59/14527 (1457)	3/3792 (27)	0/2114 (21)	18
Z	837/2401 (1409)	48/2116 (431)	3/160 (9)	0/142 (6)	12

Table 2. For the 20 most expensive programs in each cluster (**CL**), **Time** is cumulative CPU time (in hours), **LoC** is lines of source code, **SQL** is the number of SQL statements, and **C#** is the number of SQL statements that have embedded C# computations, **AG** is the number of programs that were automatically generated. The first four columns are presented as *min*/*max* (*geometric mean*).

To demonstrate why consolidation is a task best left for a compiler, we tried by hand to manually optimize these programs. Due to their complexity, it took one developer a full month to determine which programs and which statements in a program are optimizable. Overall, we found that 56 of the 60 programs can be consolidated. The 4 non-consolidable jobs are very simple programs with only 3 SQL statements. Based on statement types, we further classify the consolidation opportunities into three patterns: `EXTRACT-PROCESS (EP)`, `PROCESS-PROCESS (PP)`, and `PROCESS-OUTPUT (PO)`.

An EP pattern exists if one C# computation on a process-operator (*e.g.*, `SELECT`, `PROCESS`, or `REDUCE`) can be consolidated with another one in a previous data-loading extract-operator. A PP pattern exists if the C# computations in multiple process-operators can be merged together, while a PO pattern is such that a C# computation in a data-dumping output-operator can be consolidated with another one in a previous process- or extract-operator. In Figure 1, both consolidating the C# computation and pulling the predicate are PP instances.

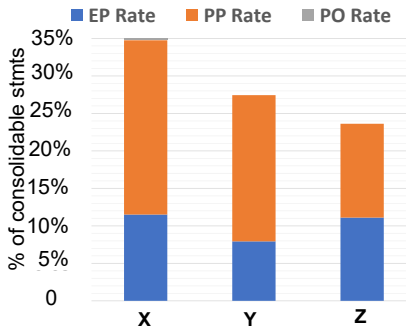


Figure 2. Percentages of consolidable *Scope* statements from *Microsoft*'s three production clusters.

Figure 2 reports, for programs in each cluster, the average percentage of consolidable statements in the three categories. For the 60 inspected jobs, an overall of 28.6% SQL statements

```
1 file = SELECT JobGUID.ToLower() AS JobId,
2                Content, ManifestType, Filename
3        FROM (SSTREAM input_file_JobFileExt)
4        HAVING ManifestType IN(...) AND
5               Filename IN(...) AND ...;
6 Algebra = SELECT JobId, ... WHERE !Content.IsEmpty();
7 Nebula = SELECT JobId, ... WHERE !Content.IsEmpty();
```

(a) Original program.

```
1 file = SELECT JobGUID AS JobId,
2                Content, ManifestType, Filename
3        FROM (SSTREAM input_file_JobFileExt)
4        HAVING ManifestType IN(...) AND
5               Filename IN(...) AND ...;
6 Algebra = SELECT JobId.ToLower(), ...
7           WHERE !Content.IsEmpty();
8 Nebula = SELECT JobId.ToLower(), ...
9          WHERE !Content.IsEmpty();
```

(b) Consolidated program.

Figure 3. A simple consolidation resulted in a 22.9% reduction in end-to-end CPU time.

are amenable to consolidation. The majority (19.4%) are instances of the PP pattern, while EP and PO take, respectively, 9.1% and 0.2%.

Performance. We performed manual consolidation for four programs. The consolidation led to 5% – 38% reductions in cumulative CPU time. Figure 3 provides a closer examination for one program that mines *Scope* job repositories. By pushing the `ToLower` call from Line 1 down to the two downstream statements at Line 6 and 8, we reduced the CPU time of this job from 1544 hours to 1190 hours over a 8TB dataset.

This is because (1) the computation is pushed across the `HAVING` filter, which filters out 42% of all data rows; this reduction is significant as these statements are very close to the data source and there is a large amount of data processed by them; and (2) the first statement becomes a pure SQL statement and hence no data serialization is needed there. Note that moving the `ToLower()` call would not incur any extra serialization costs because these two statements already involve C# computations — data needs to be deserialized anyways to invoke `Content.IsEmpty()` in these statements.

Summary. We made four important observations in the study. First, consolidation opportunities exist pervasively in production scripts. Second, simple optimizations can lead to large performance benefits. Third, an absolute requirement is that all optimizations must be sound — the production system would not accept any optimization that can potentially alter program semantics even in a corner case. Fourth, a practical solution must be fully automatic as many programs are auto-generated and not human-readable. Neither PeriScope nor Blitz can satisfy all of these requirements.

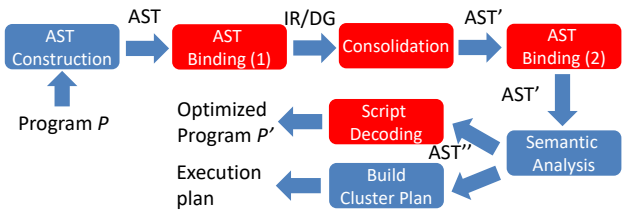


Figure 4. An overview of Niijima; red boxes represent newly added phases.

3 Niijima Design and Implementation

Overview. Figure 4 shows an overview of Niijima. Niijima's transformation is done via rewriting of a program's abstract syntax tree (AST). Niijima contains two binding phases, one before (1) and one after (2) the Niijima transformation. These two phases bind types with nodes (*e.g.*, representing different expressions and constructs) of the AST of the original and optimized program, respectively.

The first step towards a safe optimization is to explicitly expose the dependences between program variables. Such dependences provide a basis for determining the consolidation scope — for example, we must not pull a filter up across a computation if the filter *uses* a value that is *defined* by the computation. Although def-use-based dependences are widely used in the literature of program analysis, a unique challenge here is how to model dependences for the two involved languages in a *unified manner* so that constructs in different languages can be analyzed together.

To overcome this challenge, we propose an *intermediate representation* (IR) that captures an important set of properties of SQL and C# that are necessary for consolidation, while abstracting away irrelevant details. Niijima takes a *Scope* program as input and generates the IR for the program on which consolidation is performed. Next, we turn the IR into a dependence graph (DG) where control and data dependences are explicitly modeled. The generation of IR/DG is done within the first binding phase. The DG is then fed to the consolidation phase, which outputs a new (optimized) AST (*i.e.*, AST′ in Figure 4). This AST goes to the semantic analysis that performs type checking and dead code elimination. This phase outputs another version of AST (*i.e.*, AST′′).

At this point, there are two ways to proceed. We can either (1) decode AST′′ back to the source code or (2) build an execution plan for it and submit the plan to a cluster for execution. The first way is primarily for the purpose of program understanding — developers can read the decoded program to see what optimizations have been performed, while the second is the normal way leading to the execution of the program.

In the rest of this section, we will first discuss our design of the IR and DG. Then we will proceed to presenting our consolidation algorithms based on the DG.

3.1 Dependence Modeling

IR. Despite the existence of various query representations (*e.g.*, used in logic plan generators), these representations are primarily syntax trees constructed on a per-query basis. To enable Niijima to move UDO calls across queries, we are interested in *cross-query dependences*, which cannot be provided by existing representations.

We developed a new IR, which consists of a set of *micro-transformers*. Each micro-transformer represents a transformation that turns a set of input columns into a single target column. Our goal here is to expose three key pieces of information necessary for our optimizations: *columns* involved in a transformation, *C# computations* that perform the transformation, and *filters/shuffles* that guard the transformation.

$$\langle T_{in}.C_1, \ldots, T_{in}.C_n \rangle \xrightarrow{\lambda} T_{out}.D$$
FILTER[PRE/POST, ϕ]*
SHUFFLE[PRE/POST, $\langle T_i.C_i, \ldots, T_j.C_j \rangle$]*

Figure 5. A rule-based language for modeling micro-transformers.

We designed a rule-based language, as shown in Figure 5, to model micro-transformers. $\langle T_{in}.C_1, \ldots, T_{in}.C_n \rangle$ are the input columns, where T_{in} is the name of the input table and $C_1, \ldots, C_n$ are column names. These columns serve as arguments for the C# computation λ, which produces a new column $T_{out}.D$. FILTER represents a filter condition such as `WHERE` and `HAVING`, which takes two parameters. The first one specifies whether it is a *pre-transformation* (PRE) or a *post-transformation* (POST) filter.

For example, `WHERE` is a pre-transformation filter that is applied *before* any embedded C# computation is executed in a SQL statement while `HAVING` is a post-transformation filter that is not applied until all C# computations in the statement are executed. This information is important for us to determine the target location when moving filters or computations. The second parameter is a boolean formula ϕ, which is the condition defining the filter.

A number of relational operations, such as `JOIN`, `GROUP-BY`, `SORT-BY`, `PARTITION-BY`, or `REDUCE`, need to shuffle data. A shuffle operation needs to be explicitly modeled because (1) it may lead to row reductions and (2) the columns on which the shuffle is performed may induce data dependences. We use SHUFFLE to expose such information. Each SHUFFLE clause also takes two parameters, one specifying whether it is a pre- or post-transformation shuffle, and a second exposing the set of columns on which the shuffle is performed.

►**Example.** For the example statement in Figure 6(a), its Niijima IR is illustrated in Figure 6(b). Since the statement selects two columns and invokes C# methods on them, Niijima generates two micro-transformers, each for a column transformation. These micro-transformers have the same SHUFFLE and FILTER clauses because they belong to the same query statement. The two SHUFFLE clauses represent, respectively, the `JOIN` operator (*i.e.*, pre-transformation shuffle) and the `GROUP-BY` operator (*i.e.*, post-transformation shuffle) with the involved columns exposed. The two FILTER clauses correspond to the `WHERE` (*i.e.*, pre-transformation) filter and `HAVING` (*i.e.*, post-transformation) filter, respectively. ◄

From IR to DG. The micro-transformers expose necessary information that enables fine-grained modeling of dependences. As the next step, Niijima computes a dependence graph from the IR to explicitly model column-based dependences. A DG has a set of *data nodes* and a set of *control nodes*. Each data node in the DG represents one of the three possible states of a table column:

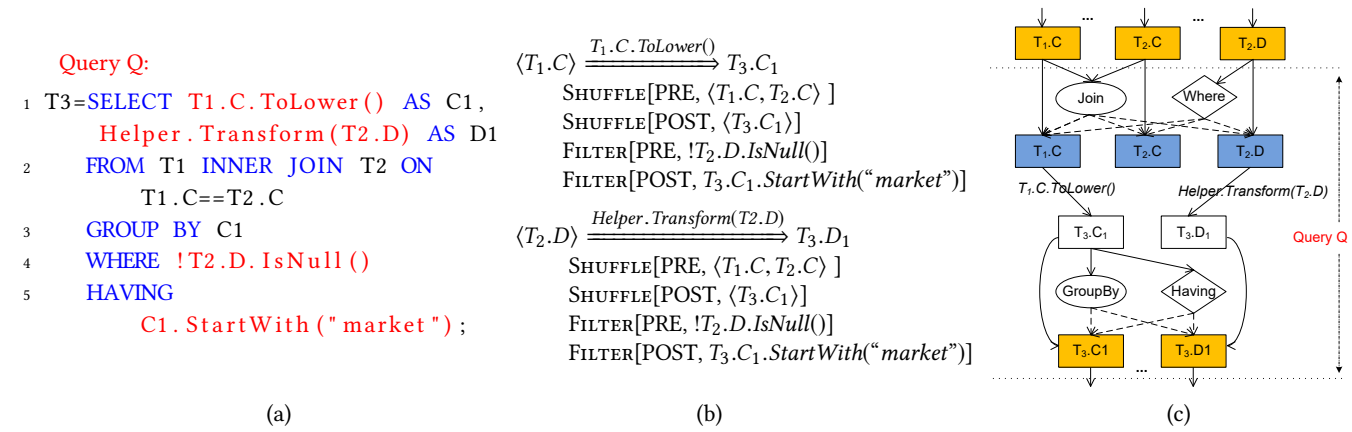
- *Post-having column (PHC)*: each PHC node of a column C represents the state of C right after a *post-transformation* filter and/or shuffle is applied.
- *Post-where column (PWC)*: each PWC node of a column C represents the state of C right after a *pre-transformation* filter and/or shuffle is applied.
- *Transformed column (TC)*: each TC node of a column C represents the state of C right after C is generated by transforming other columns.

These three states of a column are the static abstractions of the *locations* of the column data when they flow through the pipeline. Such location information determines where a filter or a C# computation can be moved (see §3.2). In addition to data nodes, there are two kinds of control nodes: *filter nodes* and *shuffle nodes*. Each filter/shuffle node represents a FILTER/SHUFFLE clause in the IR.

These data and control nodes give rise to a set of data and control dependence edges. A *data dependence* edge exists between two data nodes or from a data node to a control node if data flows from the first node to the second. If a C# computation (λ) is needed to transform the first into the second, the edge is annotated with λ. A *control dependence* edge exists from a control node to a data node if the filter/shuffle represented by the control node can change *the volume of data* flowing to the data node.

►**Example.** Figure 6(c) shows the DG generated from the IR in (b). Orange, blue and white boxes represent, respectively, PHC, PWC, and TC nodes. Filter and shuffle nodes are represented by diamonds and ovals, respectively. Solid and dashed edges represent data and control dependences, respectively.

The three PHC nodes on the top (in orange) represent the three input columns $T_1.C$, $T_2.C$, and $T_2.D$ at the moment they are produced from the previous SQL query and about to flow into query Q. Upon entering Q, data is subject to



(a) (b) (c)

Figure 6. A *Scope* query (a), and its IR (b) and DG (c); in the DG, boxes, ovals, and diamonds represent column, shuffle, and filter nodes, respectively; boxes with orange, blue, and white background represent post-having column (PHC) nodes, post-where column (PWC) nodes, and transformed column (TC) nodes.

the first (pre-transformation) round of shuffling (by `JOIN`) and filtering (by `WHERE`). Since `JOIN` is performed on $T_1.C$ and $T_2.C$, the control node representing `JOIN` is *data-dependent* on the PHC nodes $T_1.C$ and $T_2.C$; a similar dependence exists for `WHERE`. After the shuffling and filtering, the three columns are in their PWC states (shown in blue). These PWC nodes are *control-dependent* on both `JOIN` and `WHERE`, while *data-dependent* on their corresponding PHC nodes.

Next, $T_1.C$ gets transformed by the C# computation $T_1.C.ToLower$ into $T_3.C_1$ while $T_2.D$ gets transformed by $Helper.Transform$ into $T_3.D_1$. Two TC nodes (in white) exist to represent the generated columns $T_3.C_1$ and $T_3.D_1$. There is a data dependence edge from the PWC node $T_1.C$ (*i.e.*, input of the transformation) to the TC node $T_3.C_1$ (*i.e.*, output of the transformation), and another one from the PWC node $T_2.D$ (input) to the TC node $T_3.D_1$ (output). These edges are annotated, respectively, with the two C# methods performing the transformations.

At this point, data is subject to another (post-transformation) round of shuffling (by `GROUP-BY`) and filtering (by `HAVING`). After this round, the columns $T_3.C_1$ and $T_3.D_1$ are in their PHC states (shown in orange). These two PHC nodes are *control-dependent* on the shuffle and filter nodes, and *data-dependent* on their corresponding TC nodes. They are the output of query Q and will be connected to pre-transformation filter/shuffle nodes of the successor SQL queries.

The DG exits at each data destination node (*i.e.*, a special column node), representing the output file to which the processed data is written. For simplicity of algorithm design, we always create PWC and PHC nodes for each query regardless of whether the statement has filter/shuffle operations. In cases where a statement does not have any filter/shuffle, we still create data dependence edges to connect these nodes, but no control dependence exists. The IR-to-DG algorithm is straightforward and omitted in the paper. ◀

Handling of UDOs. There are two ways to handle a UDO. In *Scope*, a UDO needs to explicitly declare its input and output schema. A conservative approach is to treat a UDO as a blackbox — one can create a PWC node a for each input column and a TC node b for each output column, and link each pair (a, b) with a data dependence edge annotated with the name of the UDO, assuming that any output column b depends on any input column a.

A more precise handling, as we used in Niijima, is to analyze the UDO to create precise, fine-grained input-output dependences. We used a sound and precise dependence analysis [15] to extract dependences in the C# code and add these dependences as edges into the graph we build.

3.2 Code Motion

The dependence graph can be used for a variety of optimization tasks, including dead column elimination, common computation substitution, invariant code hoisting, *etc.* While this paper focuses on computation consolidation, future work can easily develop other optimizations based on the proposed IR and dependence graph.

Both the optimizations of pulling up filters and consolidating C# computations can be formulated as a graph traversal problem over the dependence graph. Niijima iteratively performs filter pullup and computation consolidation until no new opportunity can be found (*i.e.*, a fixed point is reached). In the rest of this section, we assume that the C# computations embedded in SQL are all side-effect-free.

3.2.1 Filter Pullup

While filter optimizations have been performed in different contexts [17, 34], most of these techniques work at the *back-end* — *e.g.*, targeting the query plans or even the imperative

Algorithm 1: The filter pullup algorithm that implements a breadth-first search of the DG.

```
   Input: DG G = (V, E)
1  foreach Filter node n ∈ V do
2      I ← InEdges(n)
3      do
4          NewEdges ← ∅
5          foreach Edge e ∈ I do
6              /*Case 1: e has a lambda*/
7              if ContainsLambda(e) then
8                  RecordAndContinue()
9              /*Case 2: e.src has another edge going to a
               different statement*/
10             foreach Edge e' ∈OutEdges(e.src) AND e ≠ e'
               do
11                 if e.stmt ≠ e'.stmt then
12                     RecordAndContinue()
13             /*Case 3: pulling a filter may lose an argument*/
14             if n's in-neighbors belong to different statements
               and they contribute different sets of columns to n
               then
15                 RecordAndContinue()
16             NewEdges ← NewEdges ∪ InEdges(e.src)
17     while I ← NewEdges
```

code generated. On the contrary, Niijima operates at the *frontend* and moves filters by directly modifying the AST of a *Scope* program.

Although our predicate-pulling algorithm is similar in spirit to existing algorithms, Niijima can optimize filters and computations together across the entire program. By contrast, prior techniques are designed primarily for relational algebra and do not work well in the presence of large numbers of UDOs. We demonstrate, empirically, in §4 that many additional opportunities were found by Niijima even when jobs were executed on the production runtime that has mature query optimizations enabled.

Where to Move. Starting at each (`WHERE` or `HAVING`) filter node, we traverse the DG *backwards* to check whether the filter can be pulled to an upstream query. Algorithm 1 shows our BFS-based traversal algorithm. The backward graph traversal stops at an edge when any of the three conditions (*i.e.*, Line 7, 10, and 14) holds. These conditions represent three constraints we need to respect when moving code; they are illustrated by the three examples in Figure 7.

Case 1 represents a *dataflow define-use* constraint, illustrated by Figure 7(a). The traversal stops upon reaching an edge with a C# computation, because the computation generates a value that is later used by the filter.

Case 2 represents a *control flow* constraint, illustrated by Figure 7(b). The traversal stops when seeing an edge whose

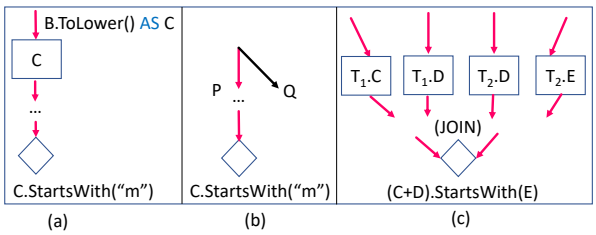


Figure 7. The three conditions under which the traversal needs to stop.

source node has another edge going to a different statement (*e.g.*, Q). Pulling the filter would inappropriately filter out data that should flow to Q.

Case 3 represents an *argument* constraint, illustrated by Figure 7(c). This case is concerned specifically about joins. A join operation combines the columns from multiple input tables into an output table T. If the filter requires data from a set of T's columns and these columns come from different incoming tables, we cannot pull up the filter across the join as each incoming table itself is not sufficient to invoke the filter. In Figure 7(c), for example, the filter requires data from the columns C, D, and E, while C and D exist in table T_1, and D and E exist in table T_2. Neither T_1 nor T_2 can alone satisfy the filter if we pull the filter into the two upstream statements.

When the graph traversal stops at one of these conditions, the RecordAndContinue function records the current location (*i.e.*, node) and then continues the loop at Line 5. In cases where there are multiple paths leading to the filter node in the DG (such as the scenario shown in Figure 8(a)), this function would direct the algorithm to stop the search in one path and continue to search in the other paths. When the algorithm finishes, it returns a set of target locations to which the filter can be moved.

How to Move. For each target location returned by the algorithm, we create a clone of the predicate and insert the clone at the location. Finally, we remove the original filter from the DG. The AST of the program is updated accordingly.

There are four major challenges in conducting the actual move. The *first* challenge is *what the filter type should be*, that is, whether the target filter should be a `HAVING` or a `WHERE` filter. This is determined by the type of the node at the target location — if the target location contains a PWC node, the moved filter should be a `WHERE` filter; otherwise, it should be a `HAVING` filter.

The *second* challenge is *how to deal with aliasing*. Since a `SELECT` operator may contain many `AS` clauses (*e.g.*, *A* `AS` *B*), these clauses give rise to aliasing (*e.g.*, *A* and *B* become aliases). To overcome this challenge, we maintain an alias map and add aliasing information into the map as we encounter `AS` clauses during graph traversal. Upon inserting

the filter to the target location l, we replace the variables involved with their corresponding aliases at l by consulting with the alias map.

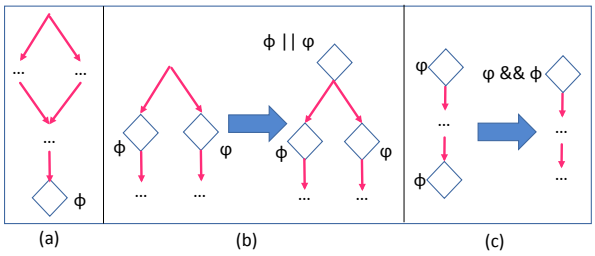


Figure 8. Filter pullup examples: (a) the challenge in a multi-path scenario, (b) merging filters from multiple paths (*i.e.*, disjunction); (c) merging filters if the target location has an existing one (*i.e.*, conjunction).

The *third* challenge is *how to deal with filters in disjoint paths*. Case 2 in Algorithm 1 is often too strict. Many real *Scope* programs have a diamond-shape dataflow graph, as illustrated in Figure 8(a). Applying Algorithm 1 would move ϕ up to the top of each of the two paths, but not be able to pull the filter further out of these paths due to the conservative handling of Case 2.

As an optimization, we add an additional step after Algorithm 1 is done. This step relaxes the handling of Case 2 to pull multiple filters together from disjoint paths if they are *at the top of these paths*. Instead of stopping at the branching point (as done in Case 2), this step pulls filters from all these paths and merges them by creating a *disjunctive predicate*.

Figure 8(b) shows such an example — after pulling, the new filter has a predicate $\phi||\varphi$ where ϕ and φ were originally the top nodes of the two disjoint paths. In this case, however, we cannot remove the original filters ϕ and φ *unless they are the same filter* (*e.g.*, pulled up from the same origin, as illustrated in Figure 8(a)).

The *final* challenge is *what to do if the target location contains another filter*. In this case, we need to turn them into a *conjunctive predicate*, as illustrated in Figure 8(c).

3.2.2 Computation Consolidation

Unlike filters that can only be pulled up, C# computations can be moved in either direction for consolidation. Niijima consists of two phases for moving C# computations: a *forward* and a *backward* phase. For each data dependence edge annotated with a non-UDO C# computation, the forward phase traverses the DG forward, attempting to push the computation to a downstream statement. If this phase cannot find any consolidation opportunities, the second phase is started to traverse the DG backwards.

The forward phase is potentially more beneficial than the backward phase — if we can push a computation across a filter (not data-dependent on the computation), less data would flow to the computation and, hence, both computation and serialization can be reduced. Based on this insight, we run the forward phase first and do *not* run the backward phase unless the forward phase cannot find any consolidation opportunities.

Algorithm 2: The forward phase for consolidation.

```
Input: DG G = (V, E)
Output: A set S of target locations
1  foreach Edge e ∈ E with a λ annotation do
2      O ← OutEdges(e.tgt)
3      do
4          foreach Edge f ∈ O do
5              /*Case 1: Cannot push across another lambda*/
6              if ContainsLambda(f) then
7                  RecordAndContinue()
8              /*Case 2: Cannot push across a filter/shuffle*/
9              if f.tgt is a filter/shuffle node then
10                 RecordAndContinue()
11             /*Case 3: Cannot lose more than one argument*/
12             if the statement containing f.tgt does not preserve
               at least n -1 arguments of λ then
13                 RecordAndContinue()
14             NewEdges ← NewEdges ∪ {f.tgt}
15     while O ← NewEdges
```

Forward Phase. Algorithm 2 shows our forward traversal algorithm. Similarly to Algorithm 1, there are three stop-conditions for the traversal. The first and second conditions represent *dataflow define-use* constraints. For example, we cannot push down a computation λ across another computation λ' if λ' uses the value defined by λ (as indicated by the reachability on the DG). Neither can we push λ down across a filter or shuffle node that uses a column defined by λ.

The third condition represents an *argument* constraint. We cannot push λ to a statement where more than one of the arguments of λ are not preserved. The following code snippet shows such an example:

```
d1 = SELECT A+B AS C FROM d;
...
d2 = SELECT C.ToLower() FROM d1;
```

Since neither argument A nor B is preserved in the second statement, we cannot push the string concatenation computation $A+B$ down and consolidate it with `C.ToLower`. Although we could modify the code to select additional columns to preserve these arguments, such modification would increase the amount of data flowing between these statements, causing an unpredictable performance impact. To be conservative, we stop the traversal upon encountering such cases.

Note that we can still perform consolidation if only one argument (B) is missing, such as the following case:

```
d1 = SELECT A, A+B AS C FROM d;
...
d2 = SELECT C.ToLower() FROM d1;
```

Since the computation (*e.g.*, `A+B`) will be pushed to the target statement, column C is freed up and can thus be used to pass the missing argument B. The optimized program becomes

```
d1 = SELECT A, B AS C FROM d;
...
d2 = SELECT (A+C).ToLower() FROM d1;
```

Similarly to the handling of filters, when the traversal stops at an edge, it records the current location and continues the loop at Line 15 to search for opportunities in other paths. Our merging algorithm will be discussed shortly.

Backward Phase. The backward phase has four stop-conditions. The first three are exactly the same as the three cases in Algorithm 1. An additional condition here is that the traversal stops upon encountering a control-dependence edge. The following code snippet illustrates such a case:

```
d1 = SELECT B, A+B AS C FROM d HAVING B != "m";
d2 = SELECT C.ToLower() AS D FROM d1;
```

Due to the `HAVING` filter, there are control dependences from the `HAVING` node to the PHC nodes representing the resulting columns B and C of d_1. The backward traversal starting at `C.ToLower()` has to stop at the `HAVING` node due to the control dependences although there does not exist a data dependence between them. This is because pulling the computation up across the filter would potentially cause the computation to be performed on more data. Hence, we could not consolidate `C.ToLower` into `A+B`.

However, the backward consolidation would be possible if we change the filter type from `HAVING` to `WHERE` — the control dependences would be "lifted" before the computation `A+B`, removing the barrier for the backward traversal to reach `A+B`. Note that it is always possible (and more profitable) to push `A+B` down and consolidate it forward into `C.ToLower()`. This is the reason why we always run the forward phase first.

Computation Merging. We run these two phases for each edge annotated with a C# computation based on their topological order in the DG, excluding those annotated with UDOs. UDOs are special computations that bind with specific types of SQL statements and thus cannot be safely moved.

The traversal algorithm returns a set of target locations to which the computation can be moved. Unlike filter pullup, which is always beneficial, moving C# computations may or may not produce runtime benefit. To guarantee that moving a computation is profitable, we define a static *profit metric* guarding each move. The metric considers two criteria: (1) if the move is a forward move, it needs to cross one or multiple filters/shuffles, implying reduced computations; or (2) the target location contains another C# computation so that the two computations can be merged; this implies reduced serialization. We do not conduct a move unless at least one criterion is met.

If a target location already contains a C# computation, the two computations need to be merged. Merging can be done by expression substitution. Formally, the following rule explains our substitution algorithm for a forward consolidation. Suppose the statements that contain the source and target computations are P and Q, respectively. Above the line are the pre-merge source and target computations while their post-merge counterparts are shown below the line. Alias is a map that takes as input a column name Y and a statement Q, and returns another column name that is the alias of Y at Q.

$$\begin{array}{l} \text{Source Query } P: \ f(X_1, X_2, \ldots, X_i, \ldots) \ \textbf{AS} \ Y, \\ \text{Target Query } Q: \ g(Y_1, Y_2, \ldots, Y_j, \ldots) \ \textbf{AS} \ Z, \\ X_i \text{ is the non-preserved argument, } \textsc{Alias}(Y, Q) = Y_j \end{array}$$

$$\begin{array}{l} \text{Source Query } P': \ X_i \ \textbf{AS} \ Y \\ \text{Target Query } Q': g(Y_1, Y_2, \ldots, f(\textsc{Alias}(X_1, Q), \textsc{Alias}(X_2, Q), \\ \qquad\qquad \ldots, Y_j, \ldots), \ldots) \ \textbf{AS} \ Z \end{array}$$

Suppose f and g are two C# methods and the call of f needs to be merged into that of g. We first find the argument (*i.e.*, X_i) from the source statement P that does not get preserved in the target statement Q. Note that if multiple arguments are not preserved in Q, we cannot perform the consolidation as prevented by Case 3 in Algorithm 2. Our idea here is that we can change the source to preserve X_i while pushing f to Q. Hence, we modify the source computation to X_i **AS** Y, using column Y to preserve X_i. *Method composition* is done at the target statement Q. In particular, the original argument Y_j (which is the alias of Y) is replaced with the call to f with all its arguments replaced with their corresponding aliases at Q.

Although this rule deals only with forward consolidation, backward consolidation can be done in a similar manner and is not shown due to space constraints.

4 Evaluation

We implemented Niijima in the *Scope*'s production runtime. Our implementation, which supports the full-blown *Scope* language with more than 100 relational and imperative constructs, has approximately 7500 lines of C# code: 3.5K lines for the IR generation and dependence graph construction and 4K lines for the graph traversal, code motion, as well as AST rewriting. The integration into the production runtime made it possible for us to run legacy scripts without any modification of their source code. We evaluated Niijima using a production A/B testing tool — the Niijima-enabled runtime was uploaded onto a cluster; the tool automatically ran each selected job using this new runtime and then compared its performance with its logged performance from a

previous run (on a standard runtime). Due to their unsoundness, neither PeriScope nor Blitz was implemented in the production runtime and thus neither could automatically run jobs without any user intervention.

4.1 Experiment Setup

Although our goal was to evaluate Niijima with a large number of jobs, we faced several practical obstacles that limited what we could do. First, we were granted access to only one virtual cluster (*i.e.*, a permission group), and thus, we were not able to access any datasets for jobs in other virtual clusters. Second, our evaluation was done when the product team was running extensive testing for their next release. Due to our low scheduling priority, each job submitted by us was often queued for a long time (*e.g.*, normally an hour and up to dozens of hours) before it was executed.

We started with 58 distinct jobs available to us in the week from 4/6/18 to 4/13/18 and excluded those that have very short ($\leq$ 5 minutes) and very long ($\geq$ 2.5 hours) running times — the performance of short-running jobs may often be impacted by noise while long-running jobs would take too long to finish. This gave us a set of 21 jobs for our performance evaluation.

Note that although these jobs have relatively small running times, they were executed on thousands of machines and their CPU times can be quite long (*e.g.*, more than 1000 hours). They were submitted by various groups for different purposes including repository analysis (4), machine learning (4), and data cooking and recooking (13). The added compilation time due to Niijima's optimizations is negligible (*e.g.*, $\leq$ 5 seconds).

4.2 Performance

Overall Performance. Table 3 shows the statistics of these jobs and their running times. We ran each job at least *three times* to minimize the noise from the distributed environment; reported here are the medians. We took care to guarantee that each run had the same scheduling priority and used the same resource allocation from the cluster: 35 containers were allocated for each job and each container had a maximum of 6GB memory. The running time of each job is referred to as the *latency* of the job in *Scope*'s terminology. The latency improvements after Niijima's optimizations are shown in column **SP**.

For the four jobs (job3, job8, job9, and job15), the post-optimization latencies are longer than their pre-optimization counterparts. We inspected these programs and found that it was impossible for our optimizations to hurt performance. Hence, we consider them to be normal variations, which show that Niijima's optimizations are ineffective for these four programs. Our inspection also found that the optimized filters/computations in these programs are all close to the data destination (*i.e.*, the `OUTPUT` statements) — the amount of data flowing through the optimized filters/computations is rather small, making the optimizations less effective.

Job	LoC	F	C	Data	LPr	LPo	SP	Variation
job1	606	0	9	2636.6	58.1	43.4	1.34	(-2%, +3.1%)
job2	179	0	4	10225.1	156.6	139.9	1.12	(-4.5%, +1.1%)
job3	68	1	18	0.12	23.4	23.9	0.98	(-2.1%, +1.2%)
job4	128	0	3	7938.5	74.9	70.0	1.07	(-5.0%, +4.3%)
job5	61	0	12	0.44	9.4	5.6	1.68	(-1.9%, +1.5%)
job6	395	1	7	442.8	43.1	40.9	1.05	(-1.8%, +6.1%)
job7	237	1	6	2056.1	50.2	43.8	1.15	(-0.3%, +4.0%)
job8	246	2	0	1.04	15.5	16.1	0.96	(-0.4%, +5.1%)
job9	233	4	9	1035.4	24.4	25.5	0.96	(-3.3%, +3.1%)
job10	167	3	8	1.16	8.2	6.6	1.24	(-3.0%, +2.4%)
job11	1195	7	19	51.2	42.5	14.6	2.91	(-2.9%, +2.8%)
job12	280	5	0	1832.9	28.5	24.7	1.15	(-6.0%, +1.7%)
job13	243	1	13	6327.1	79.3	71.3	1.11	(-2.5%, +3.0%)
job14	89	2	6	24.7	39.2	36.8	1.07	(-7.2%, +2.3%)
job15	89	3	8	45.6	37.4	38.0	0.98	(-1.5%, +0.4%)
job16	413	1	23	1.0	6.0	1.8	3.33	(-2.5%, +3.8%)
job17	167	0	4	1.2	7.8	7.5	1.04	(-3.4%, +0.2%)
job18	1524	8	16	8024.6	80.5	68.6	1.17	(-1.1%, +5.0%)
job19	310	4	5	288.5	28.6	23.6	1.21	(-1.3%, +4.2%)
job20	108	4	8	3258.4	75.6	67.1	1.13	(-2.0%, +2.1%)
job21	694	3	12	3250.8	77.7	65.0	1.20	(-1.0%, +4.6%)
GM	-	-	-	-	-	-	1.24	

Table 3. Programs evaluated: reported are their numbers of lines of code (**LoC**), numbers of filters pulled (**F**), numbers of computations moved (**C**), data processed (**Data** in GB), pre-optimization latencies (**LPr** in minutes), post-optimization latencies (**LPo**), speedups (**SP**), and time variations (**Variation**).

Overall, Niijima improves latency by 24% over the production runtime whose query optimizer was based on the Cascades framework [16] and has been continuously optimized for more than a decade. The runtime also selectively replaced C# methods with their C++ counterparts (known as *intrinsics*) if they exist. However, many commonly-used C# methods — especially those handling strings, such as `ToLower`, `StartWith`, or `IndexOf` that we use as examples in this paper — cannot be easily implemented in C++ as they are by default *culture-variant*. Due to the culture-variant semantics as well as different encodings used in C# (UTF16) and C++ (UTF8), it is difficult even to correctly calculate the index for a given character.

Of these programs, 13 see a gain of more than 10%, which is significant enough to overcome the detrimental impact of noise. Noticeably, job5, job11, and job16 are sped up by 1.68×, 2.91×, and 3.33×. Figure 1 is actually a simplified version of job11 where Niijima pulls the predicate up to each of the eleven `SELECT` clauses in the `UNION` statement. This is only possible because a computation that used to define `Time` in each such `SELECT` is pulled and merged with another computation in an earlier statement. This example clearly demonstrates the benefit of the iterative process of filter pullup and computation consolidation.

CPU, I/O, and Serialization. To better understand Niijima's benefit, we profiled, for each job, its cumulative CPU time, the amount of data read/written, and its vertex initialization time. The deltas for these three metrics between the

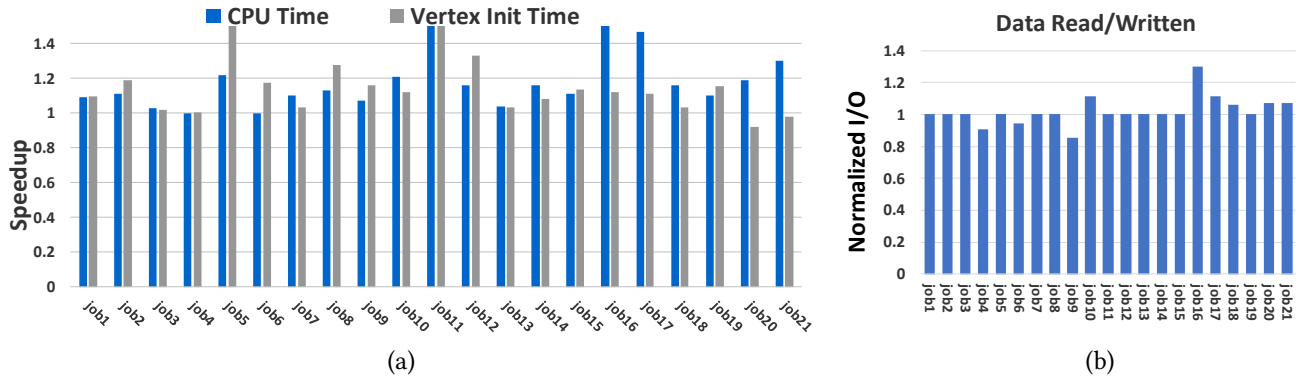


Figure 9. (a) Speedups (the higher the better) achieved by Niijima in cumulative CPU time and vertex initialization time; (b) sizes of data read/written normalized to those of original programs (the lower the better).

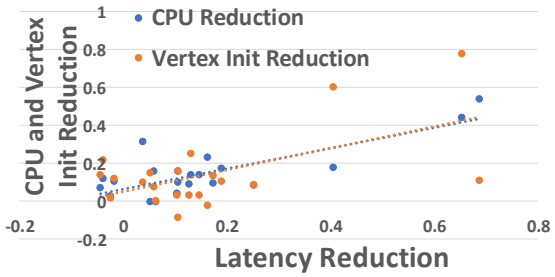


Figure 10. Correlations between latency and CPU as well as latency and vertex initialization reductions.

Job	F	FGain	C	CGain
job1	0	0	9	25.3%
job5	0	0	12	40.0%
job10	1	2.2%	0	0
job11	2	15.3%	8	5.4%
job16	0	0	21	68.6%

Table 4. Contributions of filter pullup and computation consolidation: reported are the numbers of filters pulled without computation consolidation (**F**), the performance improvements from filter pullup alone (**FGain**), the numbers of C# computations moved (**C**), and the performance improvements from computation consolidation alone (**CGain**).

pre- and post-optimization executions are shown in Figure 9. These metrics measure, respectively, the amounts of computation (Figure 9(a)), serialization/deserialization (Figure 9(a)), and I/O (Figure 9(b)). Note that we could not obtain the actual time spent on I/O and serialization/deserialization, as these efforts overlap with the computation. To overcome this challenge, we had to use the amount of data read/written to approximate I/O. Furthermore, since a major component of the initialization of each dataflow vertex is to set up the .NET runtime and deserialize native bytes into .NET objects, we use the cumulative cost of vertex initialization to approximate the serialization/deserialization effort.

The geometric means of the improvements in these three aspects are, respectively, 1.20, 1.02, and 1.22. The vertex initialization time is sped up by 1.22, which confirms with our hypothesis that consolidation can lead to fewer runtime switches and thus reductions in serialization/deserialization. The latency reductions align well with the reductions in the CPU times as well as the serialization/deserialization costs, as illustrated in Figure 10.

The reduction in the size of data read/written is marginal — this is expected as our optimizations are not designed for reducing input/output. For job4, job6, and job9, the optimized programs read/wrote more data than their original versions. We inspected these optimized programs and confirmed that this could not be due to our optimizations. According to the product team, the extra data accessed might have come from failure recoveries during the execution.

Individual Contributions. To understand the individual contributions of filter pullup and computation consolidation, we picked the five jobs with the largest gains (*i.e.*, job1, job5, job10, job11, and job16). We ran each job with either filter-pullup or computation moving alone and measured its latency. The comparisons are reported in Table 4. As no filter was pulled for job1 and job5, their performance gains were entirely due to computation consolidation. This is also the case for job16. For the other two jobs, it is clear to see the close relationship between filter pullup and computation consolidation. For example, for job11, when these two optimizations were done together, 7 filters and 19 computations were moved. On the contrary, when they were done individually, only 2 filters or 8 computations could be moved. This clearly demonstrates the importance of the co-existence of these two optimizations — existing work that focuses only on moving filters would not be effective in the presence of C# code unless these computations are moved as well.

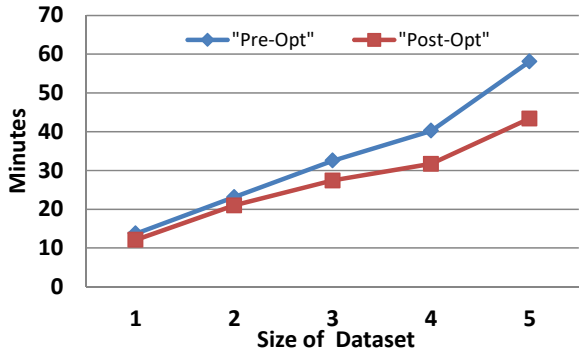


Figure 11. Effectiveness of our optimizations on different datasets: on the X-axis are the five datasets while the Y-axis shows job1's running times.

4.3 Scalability

We studied the effectiveness of our optimizations on different sizes of data processed. Since job1 mines the *Scope* job repository, we ran it on five different datasets, consisting of the execution data of the jobs executed in five different time ranges: 4/3-4/13, 4/3-4/11, 4/3-4/9, 4/3-4/7, and 4/3-4/5. Figure 11 shows the comparison results on these different datasets. It is expected that our optimizations are more effective on large datasets than small datasets. There are two reasons — early filtering has greater impact when a large amount of data can be filtered out. Furthermore, serialization/deserialization has a much higher cost; hence, larger savings in serialization can be expected when more data needs to be frequently converted between native bytes and .NET objects.

Evaluation Summary. C# computations are pervasive in *Scope* programs. Our evaluation demonstrates that computation consolidation is a powerful optimization for most of the *Scope* programs we studied, especially when they process large datasets. The majority of the performance gains come from reduced CPU time as well as reduced serialization and deserialization costs. Our results also suggest that filter pullup and computation consolidation should be done simultaneously to fully unleash the power of the optimizations. Doing either individually can significantly limit their usefulness.

5 Related Work

Scope Optimizations. Two existing works most closely related to Niijima are PeriScope [17] and Blitz [36], which are both designed to optimize the *Scope* pipelines. As discussed earlier in §1, they both attempt to modify UDOs, which often invoke directly or transitively a sea of other library methods. Hence, neither could guarantee soundness in program transformation. For example, PeriScope breaks apart code in a REDUCE UDO by performing "smart cut" without considering side effects and pointer aliasing. To use Blitz, each program needs to be manually modeled using a meta language and the synthesizer is also unsound. It is clear that unsound compiler techniques are *unacceptable* in production settings.

Optimizations of Database-backed Applications. There exists a body work [7–10, 37], focusing on SQL management and optimizations in the context of general-purpose languages. For example, Sqlcache [37] analyzes SQL queries issued by web applications to cache their results. Pyxis [8] is a program analysis based approach that partitions a database-backed application and turns part of it into stored procedures executed on the database server. QBS [10] synthesizes SQL queries from their imperative language implementations to take advantage of effective optimizations performed by query optimizers. Sloth [9] is a compiler-based technique that identifies and exploits batching opportunities across multiple queries.

Although these techniques all need to model dependences, they focus primarily on better integration of SQL and imperative code rather than moving imperative calls across multiple SQL queries. As a result, their dependence modeling differs significantly from that of ours.

Dataflow Systems and Optimizations. In the past decade, a variety of data computation models and processing systems have been developed [2, 3, 5, 11, 13, 20, 33, 39–42, 44]. Most of these systems were developed in managed languages such as Java, C#, and Scala, and suffered from performance penalties coming from the managed runtime systems. There is a line of work [4, 14, 23, 24, 26–29] that attempts to analyze and optimize the runtime system to improve the performance and scalability of Big Data systems. In particular, ITask [14] is a programming model that allows the runtime system to systematically interrupt processing logic in the presence of high memory pressure. Yak [28] is a garbage collector that incorporates region-based memory management into generational garbage collection for more efficient object collection in big data systems. Skyway [26] is a JVM-based technique that aims to reduce the cost of serializing/deserializing Java objects in systems such as Spark. Despite these efforts, none of them are designed for multilingual pipelines with relational and imperative code.

Database Query Optimizations. Query optimization [21] has been extensively studied in the database community. However, traditional database query optimizations are not designed for multilingual data pipelines — for example, early selection/filtering of columns has been widely adopted in SQL engines; however, it is unclear how a filter can be pulled if the filter contains a C# operation or the SQL statements to which the filter is attached contain C# operations, unless fine-grained dependence information can be computed as

done in this work. In addition, as demonstrated in §4, the effectiveness of filter pullup alone can be very limited without computation consolidation.

There exists a body of work [19, 35] on optimizing queries with user-defined operators. For example, Hueske et al. [19] propose an analysis-based approach that analyzes user-defined operators to extract a set of conditions to perform common rewriting such as reordering and bushy join-order enumeration. Sofa [35] is a logical optimizer that can optimize data flows with user-defined functions based on a declarative dataflow language called Meteor. Unlike Niijima that focuses on optimization of imperative calls embedded in a SQL pipeline, these techniques attempt to *recover* logical query optimizations that were disabled due to the unknown semantics of user-defined operators.

Niijima can be thought of as a multiple query optimization [38] technique. However, Niijima differs from the existing techniques, most of which target optimization of subqueries such as common subquery sharing. Niijima is a compiler optimization that optimizes C# calls rather than relational subqueries; hence, it is orthogonal to the past multi-query optimization techniques.

Code Analysis for Big Data. Code analysis has made its way into Big Data systems. FlumeJava [6] is a library that contains optimizations for data-parallel pipelines. Deca [22] is an analysis-based technique that aims to optimize memory management for Spark. Facade [29] and Generuk [25] provide compiler and runtime system support for a managed Big Data system to use native memory. Weld [31, 32] is a common IR and runtime that aims to optimize across machine learning functions and libraries for the entire workflow. While Niijima and Weld share similar goals, they focus on different optimization domains. Furthermore, Weld requires developers to provide annotations while Niijima's optimizations are completely automated.

6 Conclusion

Data processing systems become increasingly multilingual and heterogeneous. Niijima is a sound and automated compiler technique designed to perform global optimizations across the language boundaries and can be readily used in production settings. While the main ideas were implemented for *Scope*, the proposed IR can also be used to perform a variety of dataflow optimizations.

Acknowledgements

We thank the anonymous reviewers for their thorough and insightful comments. We are especially grateful to our shepherd Alexandra (Sasha) Fedorova for her feedback. Harry Xu acknowledges support from MSR's academic visiting program as well as National Science Foundation grants CNS-1613023, CNS-1703598, and CNS-1763172, and Office of Naval Research grants N00014-16-1-2913 and N00014-18-1-2037.

References

[1] Apache arrow: A cross-language development platform for in-memory data. https://arrow.apache.org, 2019.

[2] Hadoop: Open-source implementation of MapReduce. http://hadoop.apache.org.

[3] Borkar, V. R., Carey, M. J., Grover, R., Onose, N., and Vernica, R. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE* (2011), pp. 1151–1162.

[4] Bu, Y., Borkar, V., Xu, G., and Carey, M. J. A bloat-aware design for big data applications. In *ISMM* (2013), pp. 119–130.

[5] Chaiken, R., Jenkins, B., Larson, P.-A., Ramsey, B., Shakib, D., Weaver, S., and Zhou, J. SCOPE: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow. 1*, 2 (2008), 1265–1276.

[6] Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R. R., Bradshaw, R., and Weizenbaum, N. FlumeJava: easy, efficient data-parallel pipelines. In *PLDI* (2010), pp. 363–375.

[7] Cheney, J., Lindley, S., and Wadler, P. A practical theory of language-integrated query. In *ICFP* (2013), pp. 403–416.

[8] Cheung, A., Madden, S., Arden, O., and Myers, A. C. Automatic partitioning of database applications. *Proc. VLDB Endow. 5*, 11 (2012), 1471–1482.

[9] Cheung, A., Madden, S., and Solar-Lezama, A. Sloth: Being lazy is a virtue (when issuing database queries). In *SIGMOD* (2014), pp. 931–942.

[10] Cheung, A., Solar-Lezama, A., and Madden, S. Optimizing database-backed applications with query synthesis. In *PLDI* (2013), pp. 3–14.

[11] Condie, T., Conway, N., Alvaro, P., Hellerstein, J. M., Elmeleegy, K., and Sears, R. MapReduce online. In *NSDI* (2010), pp. 21–21.

[12] DataBricks. Spark tungsten. https://databricks.com/glossary/tungsten, 2015.

[13] Dean, J., and Ghemawat, S. MapReduce: Simplified data processing on large clusters. In *OSDI* (2004), pp. 137–150.

[14] Fang, L., Nguyen, K., Xu, G., Demsky, B., and Lu, S. Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs. In *SOSP* (2015), pp. 394–409.

[15] Garbervetsky, D., Pavlinovic, Z., Barnett, M., Musuvathi, M., Mytkowicz, T., and Zoppi, E. Static analysis for optimizing big data queries. In *FSE* (2017), pp. 932–937.

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

[17] Guo, Z., Fan, X., Chen, R., Zhang, J., Zhou, H., McDirmid, S., Liu, C., Lin, W., Zhou, J., and Zhou, L. Spotting code optimizations in data-parallel pipelines through PeriSCOPE. In *OSDI* (2012), pp. 121–133.

[18] Herodotou, H., Lim, H., Luo, G., Borisov, N., Dong, L., Cetin, F. B., and Babu, S. Starfish: A self-tuning system for big data analytics. In *CIDR* (2011), pp. 261–272.

[19] Hueske, F., Peters, M., Sax, M. J., Rheinländer, A., Bergmann, R., Krettek, A., and Tzoumas, K. Opening the black boxes in data flow optimization. *Proc. VLDB Endow. 5*, 11 (2012), 1256–1267.

[20] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys* (2007), pp. 59–72.

[21] Jarke, M., and Koch, J. Query optimization in database systems. *ACM Comput. Surv. 16*, 2 (1984), 111–152.

[22] Lu, L., Shi, X., Zhou, Y., Zhang, X., Jin, H., Pei, C., He, L., and Geng, Y. Lifetime-based memory management for distributed data processing systems. *Proc. VLDB Endow. 9*, 12 (2016), 936–947.

[23] Maas, M., Harris, T., Asanović, K., and Kubiatowicz, J. Trash Day: Coordinating garbage collection in distributed systems. In *HotOS* (2015).

[24] Maas, M., Harris, T., Asanović, K., and Kubiatowicz, J. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. In *ASPLOS* (2016), pp. 457–471.

[25] Navasca, C., Cai, C., , Nguyen, K., Demsky, B., Lu, S., Kim, M., and Xu, G. H. Gerenuk: Thin computation over big native data using speculative transformation. In *SOSP* (2019).

[26] Nguyen, K., Fang, L., Navasca, C., Xu, G., Demsky, B., and Lu, S. Skyway: Connecting managed heaps in distributed big data systems. In *ASPLOS* (2018), pp. 56–69.

[27] Nguyen, K., Fang, L., Xu, G., and Demsky, B. Speculative region-based memory management for big data systems. In *PLOS* (2015), pp. 27–32.

[28] Nguyen, K., Fang, L., Xu, G., Demsky, B., Lu, S., Alamian, S., and Mutlu, O. Yak: A high-performance big-data-friendly garbage collector. In *OSDI* (2016), pp. 349–365.

[29] Nguyen, K., Wang, K., Bu, Y., Fang, L., Hu, J., and Xu, G. Facade: A compiler and runtime for (almost) object-bounded big data applications. In *ASPLOS* (2015), pp. 675–690.

[30] Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD* (2008), pp. 1099–1110.

[31] Palkar, S., Thomas, J., Narayanan, D., Thaker, P., Negi, P., Palamuttam, R., Shanbhag, A., olger Pirk, Schwarzkopf, M., Amarasinghe, S., Madden, S., and Zaharia, M. Evaluating end-to-end optimization for data analytics applications in Weld, booktitle = VLDB, year = 2018,.

[32] Palkar, S., Thomas, J., Shanbhag, A., Narayanan, D., Pirk, H., Schwarzkopf, M., Amarasinghe, S., and Zaharia, M. Weld: A common runtime for high performance data analytics. In *CIDR* (2017).

[33] Pike, R., Dorward, S., Griesemer, R., and Quinlan, S. Interpreting the data: Parallel analysis with Sawzall. *Sci. Program. 13*, 4 (2005), 277–298.

[34] Proebsting, T. A., and Watterson, S. A. Filter fusion. In *POPL* (1996), pp. 119–130.

[35] Rheinländer, A., Beckmann, M., Kunkel, A., Heise, A., Stoltmann, T., and Leser, U. Versatile optimization of udf-heavy data flows with sofa. In *SIGMOD* (2014), pp. 685–688.

[36] Schlaipfer, M., Rajan, K., Lal, A., and Samak, M. Optimizing big-data queries using program synthesis. In *SOSP* (2017), pp. 631–646.

[37] Scully, Z., and Chlipala, A. A program optimization for automatic database result caching. In *POPL* (2017), pp. 271–284.

[38] Sellis, T. K. Multiple-query optimization. *ACM Trans. Database Syst. 13*, 1 (1988), 23–52.

[39] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., and Murthy, R. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow. 2*, 2 (2009), 1626–1629.

[40] Yang, H.-c., Dasdan, A., Hsiao, R.-L., and Parker, D. S. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD* (2007), pp. 1029–1040.

[41] Yu, Y., Gunda, P. K., and Isard, M. Distributed aggregation for data-parallel computing: Interfaces and implementations. In *SOSP* (2009), pp. 247–260.

[42] Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, U., Gunda, P. K., and Currey, J. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI* (2008), pp. 1–14.

[43] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI* (2012), pp. 2–2.

[44] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. Spark: Cluster computing with working sets. HotCloud, p. 10.

[45] Zhou, J., Larson, P.-Å., and Chaiken, R. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *ICDE* (2010), pp. 1060–1071.

[46] Zhou, Y., Cheng, H., and Yu, J. X. Graph clustering based on structural/attribute similarities. *Proc. VLDB Endow. 2*, 1 (2009), 718–729.