Consolidation of Queries with User-Defined Functions

Marcelo Sousa
University of Oxford
marcelo.sousa@cs.ox.ac.uk

Isil Dillig
UT Austin
isil@cs.utexas.edu

Dimitrios Vytiniotis
Microsoft Research
dimitris@microsoft.com

Thomas Dillig
UT Austin
tdillig@cs.utexas.edu

Christos Gkantsidis
Microsoft Research
chrisgk@microsoft.com

Abstract

Motivated by streaming and data analytics scenarios where many queries operate on the same data and perform similar computations, we propose *program consolidation* for merging multiple user-defined functions (UDFs) that operate on the same input. Program consolidation exploits common computations between UDFs to generate an equivalent optimized function whose execution cost is often much smaller (and never greater) than the sum of the costs of executing each function individually. We present a sound consolidation calculus and an effective algorithm for consolidating multiple UDFs. Our approach is purely static and uses symbolic SMT-based techniques to identify shared or redundant computations. We have implemented the proposed technique on top of the Naiad data processing system. Our experiments show that our algorithm dramatically improves overall job completion time when executing user-defined filters that operate on the same data and perform similar computations.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Program Analysis; H.2.4 [*Database Management*]: Query Processing

General Terms Consolidation, user-defined functions, query optimization

1. Introduction

Streaming and analytics with big data is becoming ubiquitous. Today, private and public data centers process tens of thousands of jobs per day [6, 25]. In this paper, we focus on the common scenario where many data center jobs manipulate the *same data* and perform *similar* computations, such as in the following two examples:

1. In a stream computing platform that mediates between a client application and a stream provider, many queries may be coming from a small number of popular applications, configured with different parameters. For instance, many queries issued by a popular price monitoring application may filter airlines that fly between two cities and whose cost is lower than a certain amount. Here, cities and cost are the query parameters.
2. In data analytics scenarios, similar scripts may be running on behalf of different teams, often sharing parts of computations or running slightly modified versions of the same code customized to individual needs. In fact, business analytics companies do try to exploit the similarities between different computations running on the same data [28].

While *multiple query optimization* (MQO) [30] research in the database comnunity has extensively studied the optimization of multiple similar queries in a shared execution environment, most MQO research has focused on optimizing queries written in some relational calculus. However, today's programming model has shifted to query languages that support *user-defined functions* (UDFs) written in a conventional imperative programming language, such as C#, Python, or Java [24, 26, 32]. For example, Microsoft's LINQ extension for C# adds native data querying capabilities to the language and can be used as an interface for writing distributed data applications with UDFs [24, 34].

When a machine in a data center is assigned the execution of UDF code from several applications that operate on the same data,[1] a promising further optimization is to *reuse computations* of these UDFs. When many UDFs perform similar computations, we can potentially infer their result based on previous computations to substantially improve overall job completion time.

Motivated by this scenario, we identify a new form of optimization called *program consolidation* for merging multiple UDFs that operate on the same data. Program consolidation exploits shared or redundant computations between these UDFs and produces a merged function whose execution cost is often much smaller (and never larger) than the sum of the costs of executing each function individually. Similar to compiler optimizations, our technique is purely static and does not incur run-time overhead. However, unlike traditional compiler optimizations which focus on a single program, our techniques reuse computations *across* different programs by performing transformations to expose interesting dependencies and by using symbolic logic to identify reusable computations.

This paper makes the following key contributions:

- We propose *program consolidation* as a multi-program optimization technique and present a new *consolidation calculus* for merging different user-defined functions (Section 4).

[1] A sensible choice for a system that tries to optimize co-location of jobs that use the same input data.

PLDI '14, June 9–11, 2014, Edinburgh, United Kingdom.
Copyright © 2014 ACM 978-1-4503-2784-8/14/06. . . $15.00.
http://dx.doi.org/10.1145/2594291.2594305

We prove that the transformations enabled by our calculus are sound and do not increase overall execution cost (although they may increase latency).

- We present a *consolidation algorithm* that specifies how to apply the calculus rules in order to improve overall completion time of n jobs running on the same machine (Section 5). Our algorithm uses symbolic techniques based on SMT solvers to identify useful computation sharing opportunities.
- To test the effectiveness of program consolidation under realistic parallel execution of tasks, we implement a new C#-based LINQ operator on top of the Naiad data processing engine [24]. Our new operator allows for consolidated execution of multiple UDFs. We experiment with mixes of similar realistic UDFs operating on real-world data sets, and report substantial performance improvements with sub-second consolidation time for hundreds of UDFs (Section 6).

Our evaluation gives strong evidence that our technique can result in better resource utilization for parallel query execution. However, the ideas in this paper are broader: our calculus and consolidation algorithm are applicable in any scenario where multiple programs operate over the same data and where the worker threads saturate the parallelism of the host machine.

2. Program consolidation in action

In this section, we give examples that illustrate how program consolidation can be applied for sharing common computations and eliminating redundant operations in UDF-based queries.

Example 1. *Consider the following UDF* `f1` *which filters for flights that are operated by either United or Southwest airlines:*

```
1. bool f1(FlightInfo fi) {
2.   Airline c = fi.airline;
3.   string name = c.name.toLower();
4.   if (name == "united") return true;
5.   return (name == "southwest"); }
```

Now, consider the following UDF `f2` *which returns true iff the flight costs less than $200 and is operated by United airlines:*

```
1. bool f2(FlightInfo fi) {
2.  if(fi.price >= 200) return false;
3.  return (fi.airline.name.toLower() == "united"); }
```

If both functions operate over the same input, some of the `f1` computation can be reused in `f2`. Specifically, if `f1` returns false, then `f2` will also return false. Furthermore, if `f1` returns at line 4, then the `f2` computation at line 3 is redundant, since we know the comparison must evaluate to true. Our technique takes advantage of these dependencies between `f1` and `f2` and generates the following consolidated program returning a pair of booleans:[2]

```
1. pair<bool,bool> f(FlightInfo fi) {
2.   Airline c = fi.airline;
3.   string name = c.name.toLower();
4.   if(name == "united") {
5.      if(fi.price >= 200) return (true, false);
6.      else return (true, true); }
7.   else return (c == "southwest", false); }
```

Here, the consolidated program `f` reuses computation in two ways: First, we retrieve the name of the airline and convert it to lower case letters exactly once, whereas this computation is duplicated in both `f1` and `f2`. Second, `f` checks if the flight is operated by United airlines only once, whereas this test is duplicated in both `f1` and `f2`. Furthermore, note that the sequential execution of `f1` and `f2` performs four comparisons in the worst case, whereas the consolidated program performs at most two comparisons.

[2] To minimize increases in latency, both our implementation and technical formalization broadcasts the result of a UDF as soon as it is computed.

Program Π_i	$:=$	$\lambda\alpha_1, \ldots \alpha_k.\, S$
Statement S	$:=$	skip $\mid x_{ij} :=$ IE $\mid S_1; S_2 \mid S_1 \oplus^{\text{BE}} S_2$
		$\mid$ notify^i b $\mid$ while BE do S
Int Expr IE	$:=$	int $\mid \alpha_j \mid x_i \mid f(\text{IE}_1, ..., \text{IE}_k)$
		$\mid \text{IE}_1 \oslash \text{IE}_2$ ($\oslash \in \{+, -, *\}$)
Bool Expr BE	$:=$	$b \mid \text{IE}_1 \odot \text{IE}_2$ ($\odot \in \{<, =, \leq\}$)
		$\mid \neg\text{BE} \mid \text{BE}_1 \curlywedge \text{BE}_2$ ($\curlywedge \in \{\wedge, \vee\}$)
Bool Const b	$:=$	$\top \mid \bot$

Figure 1. Imperative language used for formal development

Example 2. *Consider the following UDF* `g1` *which filters cities based on their minimum temperature:*

```
1.  bool g1(WeatherInfo wi)  {
2.    int min = getTempOfMonth(1); int i = 2;
3.    while(i <= 12) {
4.      t = wi.getTempOfMonth(i);
5.      if(t < min) min = t;
6.      i++; }
7.   return min > 15; }
```

Similarly, `g2` *filters cities based on their maximum temperature:*

```
1.  bool g2(WeatherInfo wi) {
2.    int j = 1; int max = getTempOfMonth(j);
3.    while(j < 12) {
4.      j++;  cur = wi.getTempOfMonth(j);
5.      if(t > max) max = t; }
6.    return max < 10; }
```

Our technique can generate the following consolidated program:

```
1.  pair<bool, bool> g(WeatherInfo wi) {
2.   int min = getTempOfMonth(1);
3.   int i = 2; int j = 1;
4.   int max = min;
5.   while (i<=12) {
6.     t = wi.getTempOfMonth(i);
7.     j = i; cur = t; i++;
8.     if(t < min) min = t;
9.     else if (cur > max) max = t;
10. }
11.  if(min > 15) return (true, false);
12.  else return (false, max < 10); }
```

Here the consolidated program reuses computation in many ways: First, we have fused the two loops from `g1` and `g2` into one common loop in `g`. Second, in the loop body, we have reused the result of the call to `getTempOfMonth`. Third, when `t<min` evaluates to true at line 8, we do not perform the test `cur>max`. Fourth, when the test `min > 15` evaluates to true at line 11, we do not perform the redundant test `max < 10`.

3. Formal setup

In this section, we present a simple imperative programming language that we use for formalizing program consolidation. To justify our design choices, we emphasize the application domain of our work, which is queries with UDFs. Official guidelines for writing such "well-behaved" user-defined code require that *"the expressions used for filtering should not have side effects and must be deterministic. Also, the expressions should not contain any logic that depends on a set number of executions, because the filtering operations might be executed any number of times."* [2]. Some systems further require these properties for correctness [34].

With these assumptions in mind, we focus on a language that includes a local store and deterministic functions without side-effects. Hence, programs in our language do have local state, but different programs cannot have shared mutable state.

The language syntax is given in Figure 1 and is quite conventional. Each program Π_i accepts arguments $\alpha_1, \dots \alpha_k$ and consists of a statement S. Statements include skip, assignments to local variables (denoted by x_{ij}), sequencing, conditionals (written $S_1 \oplus^{\mathrm{BE}} S_2$), while loops, and notifications. A conditional statement $S_1 \oplus^{\mathrm{e}} S_2$ executes S_1 if e evaluates to true and S_2 otherwise. A notification statement is $\mathrm{notify}^i\ b$ where b is a boolean constant and i is the unique identifier associated with Π_i. We say that program Π_i *broadcasts* value b if it executes $\mathrm{notify}^i\ b$. Our language contains notifications rather than the more conventional return statement because it allows us to model the early broadcasting of a result from a consolidated program, thereby minimizing increases in latency. Conveniently, the Naiad framework, on top of which we have implemented our analysis, implements precisely this notification primitive as a method call.

Expressions in our language are either integer or boolean expressions denoted IE and BE respectively. Integer expressions include constants, arguments α_j, local variables x_{ij}, integer operations $\oslash$ $(+, -, *)$ and function calls. We do not include function definitions, so we assume all functions called by Π_i are externally provided by a library. Observe that local variables x_{ij} in this language are labeled by the program identifier i; hence, any pair of programs contain a disjoint set of local variables.

Figure 2 gives a big-step, cost-annotated operational semantics of this language using judgments of the form:

$$E, e \Downarrow_k c \quad \text{and} \quad E, S \Downarrow_k E', N$$

where e is an expression, c is a constant, S is a statement, and E is an environment mapping variables to constants. The symbol N represents the *notification environment*, which maps program identifiers to boolean constants. The purpose of the notification environment N is to collect all broadcasted values so that we can assert the correctness of our program transformations by showing that they preserve N. The judgment $E, e \Downarrow_k c$ asserts that, under environment E, expression e evaluates to value c with cost k. Similarly, the judgment $E, S \Downarrow_k E', N$ says that, under environment E, the execution of statement S has cost k and generates a new environment E' and a notification environment N.

The cost semantics is given using an abstract function *cost* which assigns a cost to a particular type of operation. For the operational semantics of function calls, the *eval* operation computes the return value and cost of the function call. Specifically,

$$\mathrm{eval}(f(c_1, \dots, c_k)) = (c, m)$$

means that evaluating f with arguments $c_1, \dots, c_k$ has cost m and returns value c. While our cost semantics does not model low-level system aspects such as cache contention and instruction pipeline, it is a reasonable high-level proxy for quantifying execution cost.

3.1 Definition of Program Consolidation

Having presented our formal setup, we now give a formal statement of the problem we address. Given k programs $\Pi_1, \dots, \Pi_k$ that perform computations on the same input, the goal of program consolidation is to generate a single program Π such that:

1. Π has the same behavior as a sequential execution of $\Pi_1, \dots, \Pi_k$
2. For any input, the cost of executing Π is no more than the sum of the costs of executing $\Pi_1, \dots, \Pi_k$ individually

We use the notation $\Pi_1 \otimes \Pi_2$ to denote the consolidation of Π_1 and Π_2. The consolidated program $\Pi_1 \otimes \Pi_2$ contains local variables from both Π_1 and Π_2 and notifications from both programs. A consolidated program can execute multiple notifications in a given run, but the notifications must be associated with different program identifiers. Based on the operational semantics, we define the soundness of program consolidation as follows:

$$\frac{\mathrm{cost}(\mathrm{int}) = k}{E, c \Downarrow_k c}\ (c\ \mathrm{int}) \qquad \frac{E(v) = c \quad \mathrm{cost}(\mathrm{var}) = k}{E, v \Downarrow_k c}$$

$$\frac{\begin{array}{c} E, e_i \Downarrow_{n_i} c_i \\ \mathrm{eval}(f(c_1, \dots c_k)) = (c, m) \\ w = \sum_{i=1}^{k} n_i + m \end{array}}{E, f(e_1, \dots, e_k) \Downarrow_w c} \qquad \frac{\begin{array}{c} E, e_1 \Downarrow_{k_1} c_1 \\ E, e_2 \Downarrow_{k_2} c_2 \\ \mathrm{cost}(\oslash) = n \end{array}}{E, e_1 \oslash e_2 \Downarrow_{k_1+k_2+n} c_1 \oslash c_2}$$

$$\frac{b \in \{\top, \bot\} \quad \mathrm{cost}(\mathrm{bool}) = k}{E, b \Downarrow_k b} \qquad \frac{E, e \Downarrow_k b \quad \mathrm{cost}(\neg) = m}{E, \neg e \Downarrow_{k+m} \neg b}$$

$$\frac{\begin{array}{c} E, e_1 \Downarrow_{k_1} b_1 \quad E, e_2 \Downarrow_{k_2} b_2 \\ \mathrm{cost}(\mathbin{\wedge\!\!\wedge}) = m \end{array}}{E, e_1 \mathbin{\wedge\!\!\wedge} e_2 \Downarrow_{k_1+k_2+m} b_1 \mathbin{\wedge\!\!\wedge} b_2} \qquad \frac{\begin{array}{c} E, e_1 \Downarrow_{k_1} c_1 \quad E, e_2 \Downarrow_{k_2} c_2 \\ \mathrm{cost}(\odot) = m \end{array}}{E, e_1 \odot e_2 \Downarrow_{k_1+k_2+m} c_1 \odot c_2}$$

$$\frac{}{E, \mathrm{skip} \Downarrow_0 E, \emptyset} \qquad \frac{E, e \Downarrow_k c \quad \mathrm{cost}(\mathrm{assign}) = m}{E, x := e \Downarrow_{k+m} E[c/x], \emptyset}$$

$$\frac{E, S_1 \Downarrow_{k_1} E_1, N_1 \quad E_1, S_2 \Downarrow_{k_2} E_2, N_2}{E, S_1; S_2 \Downarrow_{k_1+k_2} E_2, N_1 \uplus N_2} \qquad \frac{\mathrm{cost}(\mathrm{notify}) = k}{E, \mathrm{notify}^i\ b \Downarrow_k E, [i \mapsto b]}$$

$$\frac{\begin{array}{c} E, e \Downarrow_k \top \\ E, S_1 \Downarrow_m E_1, N_1 \\ \mathrm{cost}(\mathrm{branch}) = n \end{array}}{E, S_1 \oplus^e S_2 \Downarrow_{k+m+n} E_1, N_1} \qquad \frac{\begin{array}{c} E, e \Downarrow_k \bot \\ E, S_2 \Downarrow_m E_2, N_2 \\ \mathrm{cost}(\mathrm{branch}) = n \end{array}}{E, S_1 \oplus^e S_2 \Downarrow_{k+m+n} E_2, N_2}$$

$$\frac{\begin{array}{c} E, e \Downarrow_k \top \quad E, S \Downarrow_m E_1, N_1 \\ E_1, \mathrm{while}\ e\ \mathrm{do}\ S \Downarrow_n E_2, N_2 \\ \mathrm{cost}(\mathrm{branch}) = o \end{array}}{E, \mathrm{while}\ e\ \mathrm{do}\ S \Downarrow_{k+m+n+o} E_2, N_1 \uplus N_2} \qquad \frac{\begin{array}{c} E, e \Downarrow_k \bot \\ \mathrm{cost}(\mathrm{branch}) = m \end{array}}{E, \mathrm{while}\ e\ \mathrm{do}\ S \Downarrow_{k+m} E, \emptyset}$$

Figure 2. Operational semantics

Definition 1. (Soundness of Consolidation) *Given two programs $\Pi_1 = \lambda\vec{\alpha}.S_1$ and $\Pi_2 = \lambda\vec{\alpha}.S_2$, $\Pi_1 \otimes \Pi_2 = \lambda\vec{\alpha}.S$ is a sound consolidation of Π_1 and Π_2 if, for any input vector $\vec{c}$, whenever*

$$[\vec{\alpha} \mapsto \vec{c}], S_1 \Downarrow_{k_1} E_1, N_1 \quad \text{and} \quad [\vec{\alpha} \mapsto \vec{c}], S_2 \Downarrow_{k_2} E_2, N_2$$

then: $[\vec{\alpha} \mapsto \vec{c}], \Pi_1 \otimes \Pi_2 \Downarrow_{\leq k_1+k_2} E_1 \cup E_2, N_1 \uplus N_2$

In other words, consolidated program $\Pi_1 \otimes \Pi_2$ has the same observable behavior as $\Pi_1; \Pi_2$, but executing $\Pi_1 \otimes \Pi_2$ is potentially cheaper than sequentially executing Π_1 and Π_2.

4. Consolidation Calculus

The main idea behind our consolidation calculus is that it explores the space of all possible consolidations of Π_1 and Π_2 that can be obtained by applying (semantic) expression simplification and (semantic) dead code elimination to a program that represents a possible interleaving of the statements in Π_1 and Π_2. In other words, any program derivable using our calculus can be viewed as an optimized version of an interleaved execution of Π_1 and Π_2. Specifically, there are three core ideas underlying our calculus:

1. Cross-simplification: Given a high-cost expression e in Π_i that is equivalent to another lower-cost expression e' over Π_j variables, our consolidation calculus allows rewriting e to e'. In its simplest form, this type of consolidation allows (static) memoization across programs. For example, if Π_1 contains expression $f(\alpha)$ where f performs an expensive computation and Π_2

$$\text{(Int)}\ \frac{\Psi \models e = e' \quad \text{cost}(e') \leq \text{cost}(e)}{\Psi \vdash_{\rm i} e : e'}$$

$$\text{(Bool 1)}\ \frac{\Psi \models e}{\Psi \vdash_{\rm b} e : \top} \qquad \frac{\Psi \models \neg e}{\Psi \vdash_{\rm b} e : \bot}\ \text{(Bool 2)}$$

$$\text{(Bool 3)}\ \frac{\begin{array}{cc}\Psi \not\models e_1 \odot e_2 & \Psi \not\models \neg(e_1 \odot e_2) \\ \Psi \vdash_{\rm i} e_1 : e_1' & \Psi \vdash_{\rm i} e_2 : e_2'\end{array}}{\Psi \vdash_{\rm b} e_1 \odot e_2 : e_1' \odot e_2'}$$

$$\text{(Bool 4)}\ \frac{\Psi \vdash_{\rm b} e_1 : e_1' \qquad \Psi \vdash_{\rm b} e_2 : e_2'}{\Psi \vdash_{\rm b} e_1 \mathbin{\wedge\!\!\!\vee} e_2 : \text{fold}(e_1' \mathbin{\wedge\!\!\!\vee} e_2')}$$

$$\text{(Bool 5)}\ \frac{\Psi \vdash e : e'}{\Psi \vdash_{\rm b} \neg e : \text{fold}(\neg e')}$$

Figure 3. Rules for cross-simplifying expressions

variable y stores the result of $f(x)$ where $x = \alpha$, then the consolidated program can replace the expensive call to f with y.

2. Cross-embedding: Our calculus allows embedding Π_i statements within the true and false branches of Π_j conditionals, which often exposes many cross-simplification opportunities. For example, if a Π_1 predicate P_1 implies a Π_2 predicate P_2, by embedding P_2 inside the true and false branches of P_1, we can save the computation of P_2 as a result of cross-simplifying it to *true* in the then branch of Π_1. Hence, cross-embedding allows dead code elimination across multiple programs.
3. Interleaving: Since Π_1 and Π_2 do not share state, any interleaving of Π_1 and Π_2 is sound, but some interleavings expose more optimization opportunities than others. For instance, suppose Π_1 contains statement $(L_1 \oplus^{e_1} R_1)$ and Π_2 contains $(L_2 \oplus^{e_2} R_2)$. If e_2 is true most of the time and it can be shown that e_2 implies e_1, evaluating e_2 before e_1 is beneficial since we can save the computation of e_1 in most executions (when combined with cross-embedding and cross-simplification).

Figure 5 presents our calculus as inference rules of the form:

$$\Psi \vdash S_1 \otimes S_2 : S$$

where Ψ is a logical formula which we refer to as a *context*, S_1 and S_2 are statements from two different programs Π_1 and Π_2, and S is the consolidation of S_1 and S_2. Context Ψ represents the strongest post-condition of the code that comes before S_1 and S_2.[3] In the remainder of the paper, we assume that Ψ is a first-order formula in the combined theory of integer arithmetic and uninterpreted functions: Specifically, arithmetic expressions in the source language correspond to integer arithmetic terms in the constraint language, and function call expressions are represented using uninterpreted function terms in the constraint language.

The rules in Figure 5 use two auxiliary judgments:

$$\Psi \vdash_i e : e' \quad \text{and} \quad \Psi \vdash_b e : e'$$

which correspond to the cross-simplification of integer and boolean expressions respectively. The meaning of both of these judgments is that, under context Ψ, expression e is provably equivalent to expression e' where the cost of e' is smaller than that of e. The rules for both of these judgments are presented in Figure 3. The first rule (Int) in Figure 3 describes the simplification of integer expressions. According to this rule, if Ψ entails that e and e' are equal (i.e., $\Psi \Rightarrow e = e'$ is logically valid) and the cost of e' is less than or equal to e according to our cost semantics, then e can be simplified to e'. The rules labeled (Bool 1) and (Bool 2) state that e can be simplified to boolean constant true (resp. false) under context Ψ if Ψ entails e (resp. the negation of e). According to (Bool 3), an arithmetic constraint of the form $e_1 \odot e_2$ can be simplified to $e_1' \odot e_2'$ under Ψ if e_1 and e_2 are equivalent to lower-cost expressions e_1' and e_2' according to Ψ. Rules Bool 4 and 5 allow simplifying predicates that contain boolean connectives. These rules first simplify their operands under context Ψ and re-combine the simplified operands using an operation called fold which performs constant folding (e.g., $\text{fold}(e_1 \wedge \top) = e_1$, $\text{fold}(\bot \wedge e_2) = \bot$, etc.).

Example 3. *Consider context* $\Psi : \alpha_1 > 0 \wedge x = f(\alpha_2) \wedge y = \alpha_1$. *According to Figure 3, we have:*

$$\Psi \vdash_{\rm b} (y \geq 0 \wedge f(\alpha_2) \neq 0) : \ x \neq 0$$

since $\Psi \models y \geq 0$ *and* $\Psi \models f(\alpha_2) = x$.

We now turn to Figure 5 which presents our core consolidation calculus. In this figure, we use the letters S, C, R, P etc. to range over statements. The first rule, called (Com), states that consolidation is commutative. That is, if we can obtain a program S by consolidating S_2 with S_1, then S is also a valid consolidation of S_1 with S_2. Effectively, the (Com) rule allows our calculus to explore different cross-optimization opportunities that arise from different interleavings of programs Π_1 and Π_2.

The second rule (Skip 1) is a base case in our consolidation calculus and allows eliminating skip statements. In contrast, the rules labeled Skip 2 and Skip 3 allow introducing skip statements. While these skip introduction rules may not seem useful at first glance, they greatly simplify the presentation of our calculus.

The fifth rule (Assign) applies when one of the programs starts with an assignment $x := e$. In this case, we first simplify e by using the $\vdash_{\rm i}$ judgment defined earlier. If we find a lower cost expression e' that is equivalent to e under Ψ, we replace the original assignment with $x := e'$. Now, if C is the remainder of the first program fragment and if P is the second program, the consolidated program still needs to execute C and P after the assignment. Hence, the second premise in the Assign rule performs the consolidation of C and P under the new context $\text{sp}(\Psi, x := e)$ which denotes the strongest postcondition of Ψ with respect to $x := e$.
Remark: While the Assign rule only applies when the first program starts with an assignment, we can use the Com rule followed by the Assign rule when the second program starts with an assignment.

Example 4. *Figure 4 shows how the Assign, Com, and Skip rules in the calculus can be applied to consolidate the statements* $x := f(\alpha) + 1$ *and* $y := f(\alpha) - 1$ *under context* true. *In this example, the consolidated program is* $x := f(\alpha) + 1; y := x - 2$.

The next rule labeled Step allows "stepping over" a statement in the first program without consolidating it with statements from the second program. Given two statements $S; C$ and P, the Step rule first executes S followed by the consolidation of C and P. In the premise of this rule, $C \otimes P$ is computed under context $\text{sp}(\Psi, S)$ because S executes before $C \otimes P$.[4] Observe that an application of the Step rule can potentially miss useful cross-simplification opportunities since we do not simplify S under context Ψ. Hence, as we will see in Section 5, it often only makes sense to use the Step rule when none of the other rules apply or when simplifying S under Ψ is unlikely to yield performance improvements.

[3] Observe that, since Π_1 and Π_2 do not share state, the postcondition of any interleaving of Π_1 and Π_2 code is the same.

[4] Since notify statements are non-standard, we define $\text{sp}(\Psi, \text{notify}^i b) = \Psi$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x = f(\alpha)+1 \vdash_{\rm i} f(\alpha)-1 : x-2} \quad \dfrac{}{x = f(\alpha)+1 \ \wedge\ y = f(\alpha)-1 \vdash \text{skip} \otimes \text{skip} : \text{skip}}}{x = f(\alpha)+1 \vdash (y := f(\alpha)-1; \text{skip}) \otimes \text{skip} : y := x-2}\text{(Assign)}}{x = f(\alpha)+1 \vdash (y := f(\alpha)-1) \otimes \text{skip} : y := x-2}\text{(Skip 2)}}{x = f(\alpha)+1 \vdash \text{skip} \otimes (y := f(\alpha)-1) : y := x-2}\text{(Com)}}{\text{true} \vdash (x := f(\alpha)+1; \text{skip}) \otimes (y := f(\alpha)-1) : x := f(\alpha)+1; y := x-2}\text{(Assign)}}{\text{true} \vdash (x := f(\alpha)+1) \otimes (y := f(\alpha)-1) : x := f(\alpha)+1; y := x-2}\text{(Skip2)}$$

Figure 4. Example illustrating Assign, Com, and Skip rules in the calculus

$$\text{(Com)} \qquad \dfrac{\Psi \vdash S_2 \otimes S_1 : S}{\Psi \vdash S_1 \otimes S_2 : S}$$

$$\text{(Skip 1)} \qquad \dfrac{}{\Psi \vdash \text{skip} \otimes S : S}$$

$$\text{(Skip 2)} \qquad \dfrac{\Psi \vdash S_1; \text{skip} \otimes S_2 : S}{\Psi \vdash S_1 \otimes S_2 : S}$$

$$\text{(Skip 3)} \qquad \dfrac{\Psi \vdash \text{skip}; S_1 \otimes S_2 : S}{\Psi \vdash S_1 \otimes S_2 : S}$$

$$\text{(Assign)} \qquad \dfrac{\begin{array}{c}\Psi \vdash_{\rm i} e : e' \\ \text{sp}(\Psi, x := e) \vdash C \otimes P : S'\end{array}}{\Psi \vdash (x := e; C) \otimes P : \ (x := e'; S')}$$

$$\text{(Step)} \qquad \dfrac{\text{sp}(\Psi, S) \vdash C \otimes P : R}{\Psi \vdash S; C \otimes P : S; R}$$

$$\text{(Seq)} \qquad \dfrac{\begin{array}{c}\Psi \vdash S_1 \otimes S_2 : R_1 \\ \text{sp}(\Psi, S_1; S_2) \vdash C_1 \otimes C_2 : R_2\end{array}}{\Psi \vdash (S_1; C_1) \otimes (S_2; C_2) : R_1; R_2}$$

$$\text{(If 1)} \qquad \dfrac{\begin{array}{c}\Psi \vdash_{\rm b} e : \top \\ \Psi \vdash (L; C) \otimes P : P'\end{array}}{\Psi \vdash (L \oplus^e R; C) \otimes P : P'}$$

$$\text{(If 2)} \qquad \dfrac{\begin{array}{c}\Psi \vdash_{\rm b} e : \bot \\ \Psi \vdash (R; C) \otimes P : P'\end{array}}{\Psi \vdash (L \oplus^e R; C) \otimes P : P'}$$

$$\text{(If 3)} \qquad \dfrac{\begin{array}{c}\Psi \vdash_{\rm b} e : e' \quad (e' \text{ not } \top \text{ or } \bot) \\ \Psi \wedge e \vdash (L; C) \otimes P : S_1 \\ \Psi \wedge \neg e \vdash (R; C) \otimes P : S_2\end{array}}{\Psi \vdash (L \oplus^e R; C) \otimes P : S_1 \oplus^{e'} S_2}$$

$$\text{(Loop)} \qquad \dfrac{\Psi \vdash ((S;\ \text{while}\ e\ \text{do}\ S; C) \oplus^e C) \otimes P : P'}{\Psi \vdash (\text{while}\ e\ \text{do}\ S; C) \otimes P : P'}$$

Figure 5. Core Consolidation Calculus

The Seq rule in Figure 5 applies to a pair of programs of the form $S_1; C_1$ and $S_2; C_2$. According to this rule, the consolidated program is the sequential composition of $S_1 \otimes S_2$ and $C_1 \otimes C_2$. Similar to the Assign and Step rules, we compute $C_1 \otimes C_2$ under context $\text{sp}(\Psi, S_1; S_2)$ because $C_1 \otimes C_2$ is executed after S_1 and S_2.

The next three rules describe the consolidation of programs starting with a conditional $L \oplus^e R$. In all three rules, we first simplify e with respect to context Ψ. In the If 1 rule, if e is equivalent to true under Ψ, we know that the false branch will not execute. Hence, the consolidated program is $(L; C) \otimes P$ where C is the remainder of the first program and P is the second program. The If 2 rule describes the symmetric case where e simplifies to false under Ψ. In this case, since the true branch is effectively dead code, the consolidated program is obtained as $(R; C) \otimes P$.

The If 3 rule describes the case where the conditional test e simplifies to some e' which is neither true nor false. In this case, we *embed* the second program P into the true and false branches of the first program. The consolidated program is $S_1 \oplus^{e'} S_2$ where S_1 and S_2 are the programs obtained by embedding P into the true and false branches respectively. Specifically, we obtain S_1 by consolidating $L; C$ with P, but the consolidation happens under context $\Psi \wedge e$ since we know that e must be true in the then branch. Similarly, S_2 is obtained by consolidating $R; C$ with P under context $\Psi \wedge \neg e$ since we know $\neg e$ holds in the else branch.

Example 5. *Figure 6 illustrates how the If and Step rules in the calculus are used in consolidating the conditional statements* $\text{notify}^1 \top \oplus^{x > \alpha} \text{notify}^1 \bot$ *and* $\text{notify}^2 \top \oplus^{x \leq \alpha} \text{notify}^2 \bot$ *under context* true. *In this example, the consolidated program is:*

$$(\text{notify}^1 \top; \text{notify}^2 \bot) \oplus^{x > \alpha} (\text{notify}^1 \bot; \text{notify}^2 \top)$$

Observe that the resulting program performs only one test while sequential execution of the programs would perform two tests.

Remark: While the If 3 rule exposes cross-simplification opportunities between two programs, an application of this rule can cause a blow-up in the size of the consolidated program because we duplicate C and P in the then and else branches of the conditional. However, using our calculus, one can derive alternative rules to If 3 that have different simplification vs. code size trade-offs. For instance, the following rule, which we refer to as If 4, can be derived from our calculus using the Com, Skip 2, Seq, and If 3 rules:

$$\dfrac{\begin{array}{c}\Psi \vdash e : e' \\ \Psi \wedge e \vdash L \otimes P : S_1 \\ \Psi \wedge \neg e \vdash R \otimes P : S_2\end{array}}{\Psi \vdash (L \oplus^e R; C) \otimes P : S_1 \oplus^e S_2; C} \ \text{(If 4)}$$

Observe that this derived rule does not duplicate C in the then and else branches and therefore results in a smaller consolidated program. Similarly, the following rule, which we refer to as If 5, is also derivable using Com, Skip 3, Seq, and If 3 rules:

$$\dfrac{\begin{array}{cc}\Psi \wedge e \vdash L \otimes \text{skip} : L' & \Psi \wedge \neg e \vdash R \otimes \text{skip} : R' \\ \Psi \vdash e : e' & \Psi \vdash C \otimes P : Q\end{array}}{\Psi \vdash (L \oplus^e R; C) \otimes P : L' \oplus^{e'} R'; Q} \ \text{(If 5)}$$

This derived rule (If 5) allows us to cross-simplify the conditional without embedding P in the true and false branches. An application of If 5 results in a smaller consolidated program than If 3 but may miss useful cross-simplification opportunities.

The last rule in Figure 5 (Loop) describes the case when the first program starts with a while loop. This rule simply expands the loop to the equivalent, equal-cost statement:

$$S' = \ (S; \text{while}(e)\ \text{do}\ S) \oplus^e \text{skip}$$

and then applies consolidation between S' and the second program P. Observe that expanding the while loop as described above may be beneficial, for example, when there is shared computation be-

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x > \alpha \vdash_b x \leq \alpha : \bot} \quad \dfrac{\dfrac{x > \alpha \vdash \text{skip} \otimes \text{not}^2\bot : \text{not}^2\bot}{x > \alpha \vdash \text{not}^1\top \otimes \text{not}^2\bot : \text{not}^1\top; \text{not}^2\bot}}{x > \alpha \vdash \text{not}^2\bot \otimes \text{not}^1\top : \text{not}^1\top; \text{not}^2\bot}}{x > \alpha \vdash (\text{not}^2\top \oplus^{x\leq\alpha} \text{not}^2\bot) \otimes \text{not}^1\top : \text{not}^1\top; \text{not}^2\bot}}{x > \alpha \vdash \text{not}^1\top \otimes (\text{not}^2\top \oplus^{x\leq\alpha} \text{not}^2\bot) : \text{not}^1\top; \text{not}^2\bot} \quad \dfrac{\dfrac{\neg(x > \alpha) \vdash_b x \leq \alpha : \top \quad \dfrac{\dfrac{\neg(x > \alpha) \vdash \text{skip} \otimes \text{not}^2\top : \text{not}^2\top}{\neg(x > \alpha) \vdash \text{not}^1\bot \otimes \text{not}^2\top : \text{not}^1\bot; \text{not}^2\top}\,(4)}{\neg(x > \alpha) \vdash \text{not}^2\top \otimes \text{not}^1\bot : \text{not}^1\bot; \text{not}^2\top}\,(2)}{\neg(x > \alpha) \vdash (\text{not}^2\top \oplus^{x\leq\alpha} \text{not}^2\bot) \otimes \text{not}^1\bot : \text{not}^1\bot; \text{not}^2\top}\,(3)}{\neg(x > \alpha) \vdash \text{not}^1\bot \otimes (\text{not}^2\top \oplus^{x\leq\alpha} \text{not}^2\bot) : \text{not}^1\bot; \text{not}^2\top}\,(2)}{\text{true} \vdash (\text{not}^1\top \oplus^{x>\alpha} \text{not}^1\bot) \otimes (\text{not}^2\top \oplus^{x\leq\alpha} \text{not}^2\bot) : (\text{not}^1\top; \text{not}^2\bot) \oplus^{x>\alpha} (\text{not}^1\bot; \text{not}^2\top)}\,(1)$$

Figure 6. Example illustrating if rules in the calculus. Due to space constraints, we abbreviate notify as not and omit uses of the Skip 2 rule. Labels (1), (2), (3), (4) correspond to the application of If 3, Com, If 1, and Step rules respectively.

tween e and P or S and P. However, if we keep blindly applying the Loop rule, our consolidation procedure is unlikely to terminate. For example, if both programs are while loops with statically unknown symbolic bounds, the only way to ensure the termination of our consolidation procedure would be to apply the Step rule after some finite number of steps, effectively executing the two loops sequentially. Unfortunately, this strategy does not take advantage of possible consolidation opportunities when there is shared computation in the bodies of the two loops.

Hence, to allow the fusion of loop bodies from different programs, we enrich our consolidation calculus with additional rules given in Figure 7. The high-level idea underlying loop consolidation is the following: Given two loops L_1, L_2 of the form:

$$\text{while } (e_1) \text{ do } S_1 \quad \text{and} \quad \text{while } (e_2) \text{ do } S_2$$

the following statement S is semantically equivalent to $L_1; L_2$:

```
1 : while($e_1 \wedge e_2$) do $S_1 \otimes S_2$;
2 : while($e_1$) do $S_1$;
3 : while($e_2$) do $S_2$;
```

Furthermore, S exposes computation sharing opportunities between L_1 and L_2 since we consolidate their bodies at line 1.

While the above discussion explains the key idea underlying loop consolidation, the rules in Figure 7 are more involved because we need to ensure that consolidation can never degrade performance. Unfortunately, the strategy outlined above does not have this guarantee. For example, suppose that the loop bodies do not share any computation (so $S_1 \otimes S_2$ has the same cost as $S_1; S_2$) and that L_1 and L_2 execute the same number of times. In this case, S performs two redundant tests (namely the continuation conditions of the loops at lines 2 and 3) compared to $L_1; L_2$. Thus, to ensure the soundness of our calculus, we must prove that the above transformation never performs worse than $L_1; L_2$.

For this purpose, the loop rules in Figure 7 make use of a loop invariant Ψ_1 for the loop:

$$L_0 \;=\; \text{while}(e_1 \wedge e_2) \text{ do } S_1; S_2$$

In the Loop 2 rule, we check whether Ψ_1 implies that e_1 and e_2 must both be false when L_0 terminates (i.e., under condition $\neg(e_1 \wedge e_2)$). This is equivalent to checking $\Psi_1 \models e_1 \Leftrightarrow e_2$ and corresponds to the second premise of the Loop 2 rule. If this is the case, we have proven that L_1 and L_2 always execute the same number of times; hence L_0 and $L_1; L_2$ are both semantically equivalent to the lower-cost loop:

$$\text{while}(e_1) \text{ do } S_1 \otimes S_2$$

Thus, the consolidated program is while(e_1) do S; R where S is the consolidation of the original loop bodies S_1 and S_2 under context $\Psi_1 \wedge e_1$ and R is the consolidation of the remainders C_1 and C_2 under context $\Psi_1 \wedge \neg e_1$.

The Loop 3 rule is very similar to the previous rule, but instead checks whether one of the loops *must* execute more times than the other loop. Specifically, given the loop invariant Ψ_1 of

$$L_0 = \text{while}(e_1 \wedge e_2) \text{ do } S_1; S_2$$

we now check whether Ψ_1 implies that e_1 must be true when L_0 terminates. This implies that $\Psi_1 \models e_2 \Rightarrow e_1$ and corresponds to the second premise of the Loop 3 rule. If this is the case, when L_0 terminates, we know that e_1 is still true but e_2 is false. Hence, L_0 is semantically equivalent to the lower cost loop:

$$\text{while } e_2 \text{ do } S_1 \otimes S_2$$

Now, of course, after L_0 terminates, we still need to execute the remaining parts of the two programs. Since e_1 is true when L_0 terminates, the remaining part of the first program is:

$$R_1 = \; S_1; \text{while } e_1 \text{ do } S_1; C_1$$

and the remaining part of the second program is just C_2. Hence, the final consolidated program is while(e_2) do $S; R$ where S is the consolidated body $S_1 \otimes S_2$ under context $\Psi_1 \wedge e_2$, and R is the consolidation of R_1 and C_2 under context $\Psi_1 \wedge \neg e_2$.

Example 6. *Consider consolidating the following program P_1:*

```
$i := \alpha;\ x := 0;$
while($i > 0$) do {$i := i - 1;\ t_1 := f(i);\ x := x + t_1$}
```

with the following program P_2:

```
$j := \alpha - 1;\ y := \alpha;$
while($j \geq 0$) do {$t_2 := f(j);\ y := y + t_2;\ j := j - 1$}
```

First, we construct the following loop L and find its loop invariant:

```
while($i > 0 \wedge j \geq 0$) do{
   $i := i - 1;\ t_1 := f(i);\ x := x + t_1;$
   $t_2 := f(j);\ y := y + t_2;\ j := j - 1;$ }
```

where the precondition Ψ is $i = \alpha \wedge x = 0 \wedge j = \alpha - 1 \wedge y = \alpha$. In this case, $j = i - 1$ is a loop invariant of L. Observe that

$$j = i - 1 \wedge \neg(i > 0 \wedge j \geq 0) \models \neg(i > 0) \wedge \neg(j \geq 0)$$

Hence, we have established that the loops in P_1 and P_2 must execute the same number of times. Now, we construct the loop

$$\text{while}(e_1) \text{ do } S_1 \otimes S_2$$

where S_1 and S_2 are the loop bodies in P_1 and P_2 and where the consolidation of S_1 and S_2 happens under context $j = i - 1 \wedge i > 0$. Hence, we can obtain the following consolidation of P_1 and P_2:

```
$i := \alpha;\ x := 0;\ j := \alpha - 1;\ y := 0;$
while($i > 0$) do {
   $i := i - 1;\ t_1 := f(i);\ x := x + t_1;$
   $t_2 := t_1;\ y := y + t_2;\ j := i;$ }
```

Observe that the consolidated program improves upon sequential execution of P_1 and P_2 in that (i) we have eliminated the test $j \geq 0$ in all iterations, (ii) the (potentially expensive) call to $f(j)$ in P_2 is replaced by t_1, and (iii) we have replaced the decrement operation on j with a variable assignment.

$$\text{(Loop 2)} \quad \frac{\begin{array}{c}\Psi_1 := \text{LoopInv}(\text{while } (e_1 \wedge e_2) \text{ do } S_1; S_2, \Psi) \\ \Psi_1 \wedge \neg(e_1 \wedge e_2) \models \neg e_1 \wedge \neg e_2 \\ \Psi_1 \wedge e_1 \vdash S_1 \otimes S_2 : S \qquad \Psi_1 \wedge \neg e_1 \vdash C_1 \otimes C_2 : R\end{array}}{\Psi \vdash (\text{while } e_1 \text{ do } S_1; C_1) \otimes (\text{while } e_2 \text{ do } S_2; C_2) : \text{while } e_1 \text{ do } S; R}$$

$$\text{(Loop 3)} \quad \frac{\begin{array}{c}\Psi_1 := \text{LoopInv}(\text{while } (e_1 \wedge e_2) \text{ do } S_1; S_2, \Psi) \\ \Psi_1 \wedge \neg(e_1 \wedge e_2) \models e_1 \\ \Psi_1 \wedge e_2 \vdash S_1 \otimes S_2 : S \qquad \Psi_1 \wedge \neg e_2 \vdash (S_1; \text{while } e_1 \text{ do } S_1; C_1) \otimes C_2 : R\end{array}}{\Psi \vdash (\text{while } e_1 \text{ do } S_1; C_1) \otimes (\text{while } e_2 \text{ do } S_2; C_2) : \text{while } e_2 \text{ do } S; R}$$

Figure 7. Loop Consolidation

4.1 Soundness of the Calculus

To state a soundness theorem for our calculus, we first define *agreement* between environment E and context Ψ:

Definition 2. (Agreement) *We say that environment E and context Ψ agree with each other, written $E \sim \Psi$, if:*

$$\bigwedge_{f,\vec{c}} (\text{eval}(f(\vec{c})) = r) \models \Psi[E]$$

In other words, E and Ψ agree with each other if Ψ is valid under the variable assignment specified by E and under the function lookup operation specified by the eval function used in the operational semantics. Using this notion of agreement, our soundness theorem can be stated as follows:

Theorem 1. (Soundness) *Suppose $E_1 \cup E_2 \sim \Psi$ and*

$$\Psi \vdash S_1 \otimes S_2 : S$$

If $E_1, S_1 \Downarrow_{n_1} E_1', N_1$ and $E_2, S_2 \Downarrow_{n_2} E_2', N_2$, then:

$$E_1 \cup E_2, S \Downarrow_{\leq(n_1+n_2)} E_1' \cup E_2', N_1 \uplus N_2$$

5. Consolidation Algorithm

While the calculus presented in Section 4 lays out the key principles of program consolidation, it is intentionally not algorithmic and can be used to design a variety of consolidation algorithms. In this section, we present an algorithm that describes a simple but effective strategy for applying the calculus rules from Section 4 in order to improve overall job completion time, which is an important metric to optimize in data processing applications [25]. The key idea underlying our algorithm is that we apply the commutativity rule very sparingly but take advantage of all cross-simplification and most cross-embedding opportunities. Because of the judicious use of commutativity, our algorithm can be both efficient and effective without considering all interleavings between two programs.

The high-level idea underlying our algorithm is the following: Given a pair of statements S_1 and S_2, if S_1 does not start with a control statement (i.e., conditional or loop), we first simplify S_1 with respect to Ψ and then *consume* it by incorporating S_1 into the context. If S_1 starts with a conditional whose test predicate is relevant in S_2, then we embed S_2 into the *then* and *else* branches of the conditional in order to exploit cross-simplification opportunities. When, eventually, all of S_1 is consumed (and our algorithm ensures that it will be), we apply the commutativity rule which allows cross-simplifying S_2 with respect to S_1, which is now incorporated in context Ψ. When we encounter loops from both programs, we try to consolidate the loop bodies, but if we cannot do this in a way that guarantees a performance improvement, we simply sequentially execute the loops. Finally, if S_1 starts with a loop, but S_2 does not, we again apply commutativity, since incorporating S_2 into the context Ψ allows us to cross-simplify the body of S_1 with respect to S_2.

The consolidation algorithm is given in Figure 5 using Haskell-style pseudo-code. The procedure $\mathbf{\Omega}$ is used for consolidating two

```
1.  $\mathbf{\Omega}$ :: Program × Program → Program
2.  $\mathbf{\Omega}(\lambda\vec{\alpha}.S, \lambda\vec{\alpha}.R) = \lambda\vec{\alpha}.\mathbf{\Omega'}(\top, \mathbf{hd}(S); \mathbf{tl}(S), \mathbf{hd}(R); \mathbf{tl}(R))$

3.  $\mathbf{\Omega'}$ :: $\Psi$ × Statement × Statement → Statement
4.  $\mathbf{\Omega'}(\Psi, \text{skip}; \text{skip}, \text{skip}; \text{skip}) = \text{skip}$
5.  $\mathbf{\Omega'}(\Psi, \text{skip}; \text{skip}, R) = \mathbf{\Omega'}(\Psi, R, \text{skip}; \text{skip})$
6.  $\mathbf{\Omega'}(\Psi, \text{skip}; C_1, R; C_2) = \mathbf{\Omega'}(\Psi, \mathbf{hd}(C_1); \mathbf{tl}(C_1), R, C_2)$
7.  $\mathbf{\Omega'}(\Psi, x := e; C, R) = \mathbf{ApplyAssign}(\Psi, x := e; C, R)$
8.  $\mathbf{\Omega'}(\Psi, \text{notify}^i\ b; C, R) = \mathbf{ApplyStep}(\Psi, \text{notify}^i\ b; C, R)$
9.  $\mathbf{\Omega'}(\Psi, S_1 \oplus^e S_2; C, R) =$
10.   if(valid($\Psi \Rightarrow e$))
11.   then ApplyIf1($\Psi, S_1 \oplus^e S_2; C, R$)
12.   else if(valid($\Psi \Rightarrow \neg e$))
13.         then ApplyIf2($\Psi, S_1 \oplus^e S_2; C, R$)
14.         else if(related(($e, R$))
15.               then if(related($C, R$))
16.                       then  ApplyIf3($\Psi, S_1 \oplus^e S_2; C, R$)
17.                       else  ApplyIf4($\Psi, S_1 \oplus^e S_2; C, R$)
18.               else  ApplyIf5($\Psi, S_1 \oplus^e S_2; C, R$)
19. $\mathbf{\Omega'}(\Psi, S; C_1, R; C_2) =$
20.   case ($S, R$) of
21.     (while $e_1$ do $S_1$, while $e_2$ do $R_1$) →
22.       let $\Psi_1$ = LoopInv(while ($e_1 \wedge e_2$) do $S_1; R_1, \Psi$)
23.       in if(valid($\Psi_1 \Rightarrow e_1 \Leftrightarrow e_2$))
24.           then ApplyLoop2($\Psi, S; C_1, R; C_2$)
25.           else if(valid(($\Psi_1 \wedge \neg(e_1 \wedge e_2)) \Rightarrow e_1$))
26.                 then ApplyLoop3($\Psi, S; C_1, R; C_2$)
27.                 else if(valid(($\Psi_1 \wedge \neg(e_1 \wedge e_2)) \Rightarrow e_2$))
28.                       then ApplyLoop3($\Psi, R; C_2, S; C_1$)
29.                       else if($C_1$ = skip ∧ $C_2$ = skip)
30.                             then ApplyStep($\Psi, S, C_1, R, C_2$)
31.                             else ApplySeq($\Psi, S, C_1, R, C_2$)
32.     _ → $\mathbf{\Omega'}(\Psi, R; C_2, S; C_1)$
```

Figure 8. Consolidation Algorithm

programs, while the main function $\mathbf{\Omega'}$ consolidates two statements. Both $\mathbf{\Omega}$ and $\mathbf{\Omega'}$make use of **hd** and **tl** functions. Given a statement S, **hd** returns the first (non-sequence) statement in S and **tl** returns the remainder. For example, **hd**(x := 1; y := x) = x := 1, and **tl**(x := 1; y := x) = y := x. When S is not a sequence, **tl**(S) yields skip. Hence, the use of the term **tl**(S) in the algorithm may correspond to an application of the Skip 2 rule from the calculus.

In the statement consolidation algorithm $\mathbf{\Omega'}$, we maintain the invariant that both input statements are sequences. Hence, line 4 in Figure 5 corresponds to the base case for our consolidation algorithm which yields skip when both statements are of the form skip; skip. Line 5 in the algorithm describes the case where the first input statement is completely consumed (i.e., reduced to skip; skip), but the other statement R is not. In this case, we commute the two statements so that R can be simplified with respect to context Ψ as mentioned earlier. On the other hand, if the first statement starts with skip but has a non-skip remainder, we simply consume the skip and keep consolidating in the same order.

Next, at line 7, we check whether the first statement starts with an assignment, and, if so, we apply the Assign rule from Figure 5 using the ApplyAssign function. Observe that a use of the consolidation operator $\otimes$ in the premises of the rules corresponds to a recursive invocation of $\mathbf{\Omega}'$. Specifically, any judgment of the form $\Psi \vdash S \otimes P : Q$ in the rules corresponds to a call $\mathbf{\Omega}'(\Psi, S', P')$ with return value Q where $S' = S$ (resp. $P' = P$) if S (resp. P) is a sequence and S' is S; skip (resp. P; skip) otherwise. Observe that in the recursive invocation of $\mathbf{\Omega}'$ in the ApplyAssign function, the assignment $x := e$ is consumed and incorporated into context Ψ by computing its strongest postcondition. Similarly, the call to ApplyStep in line 8 of the algorithm consumes the notification statement by applying the Step rule from Figure 5.

Lines 9-18 in Figure 5 handle the case where the first statement starts with a conditional. If context Ψ logically implies the test predicate e or its negation, we apply the If 1 or If 2 rules as appropriate to eliminate redundant computation. On the other hand, if Ψ does not entail e or its negation, we heuristically decide whether to apply the If 3 rule or the derived If 4 or If 5 rules since all of these choices offer different cross-simplification vs. code size tradeoffs. For this purpose, we assume a boolean function called **related** which (heuristically) decides whether there are cross-simplification opportunities between two code snippets (for example, by checking for similar predicates or calls to the same function). If we decide that both the test predicate e and the remainder C offer cross-simplification opportunities for R, we then apply the If 3 rule, which takes full advantage of any cross-simplication opportunities. If only e but not C is deemed relevant for simplifying R, we then apply the derived If 4 rule discussed in Section 4. Finally, if we decide that embedding R within the true and false branches is unlikely to yield performance benefits, we simply simplify and consume the if statement by applying the If 5 rule.

Lines 19-31 in the algorithm deal with loops. If both statements start with a loop, we check whether it is possible to consolidate the loop bodies. If we can prove that both loops execute the same number of times (the test in line 22), then we apply the Loop 2 rule to consolidate the loop bodies and simplify the loop continuation condition. Similarly, if we can prove that the first loop executes more times than the second loop, we consolidate the loop bodies by applying the Loop 3 rule from the calculus (lines 24-25). On the other hand, if the second loop must execute more times than the first loop, we again apply the Loop 3 rule but with the arguments swapped. Hence, the call to ApplyLoop3 at line 27 also implicitly uses the commutativity rule. Finally, if we cannot prove any relationship between the iteration counts of the two loops, we simply execute them sequentially by using Step and Seq rules.

The pattern match at line 31 of the algorithm covers the case where the first statement starts with a while loop L, but the second statement does not. In this case, since it may be possible to simplify the body of L with respect to the second statement, we apply commutativity by recursively calling $\mathbf{\Omega}'$ with the arguments swapped. Observe that, if there is some other loop L' in the second statement, we may later be able to consolidate the bodies of L and L' through the chain of recursive calls to $\mathbf{\Omega}'$.

6. Implementation and Evaluation

In this section, we describe our implementation and present experimental results.

6.1 Implementation

We implemented our consolidation algorithm on top of the Microsoft Naiad framework [24] and using the Z3 SMT solver [9]. Naiad is a data processing system which allows user-specified queries to be efficiently mapped over large data sets. In the Naiad system, users specify queries using LINQ syntax [1]. While LINQ syntax is superficially similar to SQL, LINQ queries can contain arbitrary C# code. For instance, consider the following LINQ query:

```
var selected = customers.Where(c =>
        GetDistance(c.zip, 94305) < 10 && c.Age > 18);
```

This query filters customers who live within 10 miles of the zip code 94305 and whose age is over 18. Observe that the UDF specified in the `where` operator contains a call to the `GetDistance` function which computes the distance between a pair of zip codes.

In our implementation, we added functionality to Naiad for consolidating user-defined code in the `where` operator of LINQ queries. For this purpose, we added a new Naiad operator called `whereConsolidated` which consolidates the UDFs in `where` clauses of different queries. Furthermore, we amortize the cost of consolidating multiple UDFs using a parallel divide-and-conquer consolidation algorithm. Therefore, given a set of queries with different `where` clauses, our implementation executes a single query with a `whereConsolidated` clause.

Of course, merging different queries into a single query with a `whereConsolidated` clause has other benefits beyond merging UDFs: Since this single query reads the data only once, we also improve performance due to IO sharing and better cache utilization compared to sequential query execution which reads the data multiple times. Hence, to have a fair comparison, we implemented another Naiad operator called `whereMany` which takes multiple UDFs and executes them sequentially. Thus, by comparing the execution time of queries that use `whereMany` with those that use `whereConsolidated`, we only measure the direct impact of consolidating UDFs in the `where` clauses of the original queries.

6.2 Experimental Setup

To evaluate the benefits of program consolidation, we considered Naiad queries over five distinct data sets, including Twitter data, weather data, flight information, stock prices, and news. We synthetically generated two datasets for the weather and flight applications and used real-world data for the remaining experiments. For each domain, we attempted to design realistic query families based on templates from real queries as well as existing Naiad benchmarks. For example, one of the query families in the news domain is modeled after the `WordCount` program provided as part of the Naiad tutorial [4], while query families in the weather domain are modeled after industrial use cases for mobile applications such as the Weather app. In addition to considering single query families configured with different parameters, we also considered mixes of multiple query families for each domain.

For each query family, we consolidated 50 different queries consisting of UDFs written in C#, where each different query within the family was drawn from a realistic distribution of query parameters. While each of the original queries are small imperative C# programs with no more than 25 lines of code, the size of the UDFs grow rapidly as we consolidate them using the divide-and-conquer approach described in Section 6.1. In our experimental evaluation, the last iteration of the algorithm typically consolidates a pair of C# programs, each containing a few thousand lines of code. Hence, the UDFs we consolidate become quite large and much more complex as the algorithm progresses. In what follows, we give a more detailed description for each data set and their corresponding query families:

Weather For the *weather* application, we synthetically generated hourly weather data for two years across 500 cities. Each weather data object contains information about the average temperature and the rain fall. The data generated varies the average hourly temperature from -1 to 10 and the rain fall between 0 and 200 milimeters. For this application, we specified five query families that filter cities by:

Q1. Monthly average temperature varying month and temperature;

Q2. Monthly average rain fall varying month and the rain falll;

Q3. Yearly average temperature varying month and temperature;

Q4. Yearly average rain fall varying month and rain fall;

Q5. 50 queries sampled from the above query families according to the following distribution: $\{15, 15, 10, 10\}$.

Flight For the *flight* application, we synthetically generated flight information during the first half of November 2013 for 500 airlines across 10 world cities. We generated 12 daily flights between all cities where $1/4$ of the flights represent domestic flights. For each flight, price is computed by a multiple arithmetic progression dependent on the airline and the identifiers of the origin and destination cities. The query families filter airlines by:

Q1. Direct flight between two cities varying the cities and the price;

Q2. Flight with connections between two cities varying the cities and the price;

Q3. Average price between two cities varying cities and price;

Q4. 50 queries sampled from the above query families according to the following distribution: $\{15, 20, 15\}$.

News For the *news* application, we used the *Reuters-21578* text categorization test collection. [5] The collection is composed of 21 datasets (totalling 19043 English news articles from Reuters) and is widely used for text categorization research. The query families filter articles by:

Q1. Word containment varying the word from a list of specified words;

Q2. Average word length varying the length;

Q3. Maximum word length varying the length;

Q4. 50 queries with UDFs that are boolean combinations of the UDFs in the query families above.

Twitter For the *Twitter* application, we retrieved 11 datasets of real Twitter users from the *IBM Research Many Eyes* database. [6] In total, our database consists of 31152 tweets in English, Spanish and Portuguese. The query families filter tweets by:

Q1. Number of smileys varying the number;

Q2. Sentiment analysis varying the sentiment from a list of common sentiments, e.g. happiness;

Q3. Topic analysis varying the topic, e.g. movies;

Q4. 50 queries with UDFs that are boolean combinations of the UDFs in the query families above.

Stock For the *stock market* application, we retrieved the historical prices of the Nasdaq-100 index from Yahoo Finance. [7] In total, the dataset is composed of 377423 daily stock rows. Each row contains price information at open, close and adjusted close times, as well as the high and low stock prices and the daily volume of transactions. In this application, the query families filter companies by:

Q1. Average volume varying the volume;

Q2. Maximum stock value varying the value;

Q3. Standard deviation varying the deviation;

Q4. 50 queries with UDFs that are boolean combinations of the UDFs in the query families above.

[5] Available at: `http://aka.ms/Hdh6ti`.

[6] Availabe at: `http://aka.ms/W9dgb9`

[7] Available at: `http://finance.yahoo.com/q/hp`.

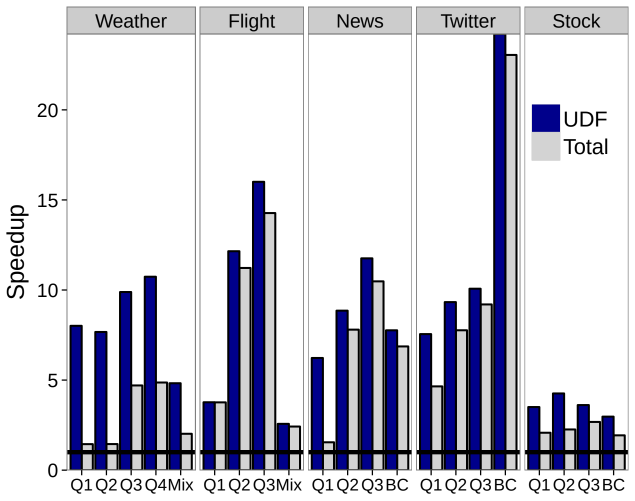

Figure 9. Experimental results. Each pair of bars labeled "Q" correspond to an experiment using query family Q. The label "Mix" denotes a random mix of unrelated query families, and "BC" stands for boolean combinations of different query families.

6.3 Experimental Results

Figure 9 shows the results of our experiments as a bar graph where the bars represent the speed-up obtained when `whereMany` is replaced with `whereConsolidated`. For each experiment, we measured both the total execution time (including consolidation) for each query as well as the time taken by the computation in the UDFs. The light gray bars in Figure 9 represent the speed-up in total query execution time (labeled "Total" in the graph), and the dark blue bars represent speed-up in the UDF execution time. The black horizontal line in Figure 9 corresponds no speed up (i.e., 1x).

As Figure 9 shows, we achieve significant speedups by consolidating UDFs. Specifially, for UDF execution time, speedups range from 2.6x to 24.2x, with an average speedup of 8.4x. For total query execution time (which includes IO, system overhead etc.), the speedup ranges from 1.4x to 23.1x, with an average speedup of 6.0x. Furthermore, consolidation takes very little time with an average of 0.3 seconds to consolidate 50 UDFs, which is only 0.4% of total query execution time. Since our evaluation considers mixes of different query families as well as queries from the same family, our results suggest that UDF consolidation is still useful even when the amount of redundancy is considerably reduced. We believe these results give preliminary but sound evidence that program consolidation can achieve large performance improvements in data processing applications.

In a second experiment, we explore the scalability of our approach with respect to the number of queries for a particular benchmark (in this case, mixes of query families in the News domain). Figure 10 plots the number of queries against running time in seconds; observe that the y-axis is given in log scale. As expected, consolidation time increases with the number of UDFs, but even for 300 queries, consolidation time remains under 1 second. Furthermore, when we use the `whereConsolidated` operator, the total running time remains roughly constant as we increase the number of UDFs, but increases roughly linearly when we use the `whereMany` operator. This experiment shows that the speed-up obtained from program consolidation increases with the number of UDFs when they perform similar computations.

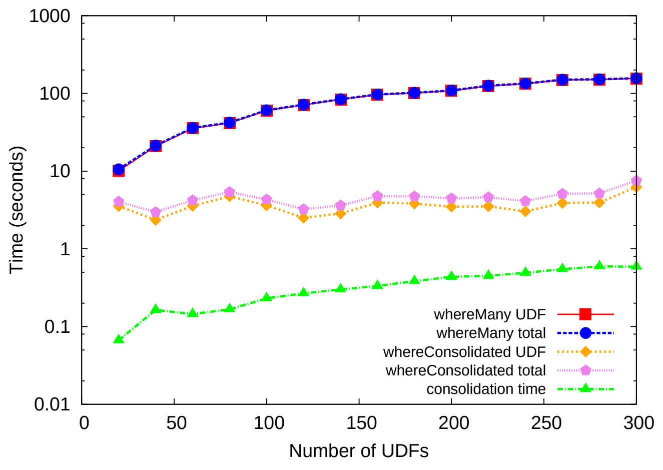

Figure 10. Scalability with the number of UDFs

7. Related Work

There is a large and diverse body of work touching the topic of this paper, from databases to compilers research, that we discuss below.

Multiple query optimization The first optimizations studied for shared query execution are based on common expression analysis [11, 27, 31, 36]. An appealingly simple idea is to treat a set of queries as a single query (extended with query identifiers) and reuse an existing single-query optimizer that can identify common expressions [12]. In this context, "expression" refers to an SQL expression, column, or temporary table. Hence, these optimizations operate on relational algebra and do not deal with user-defined code. An improvement to common expression analysis is multiple-query optimization [30]. Similar to common expression analysis, MQO also operates in the context of a relational algebra, but also takes advantage of implication relations between predicates in order to combine data accesses across different queries. Similar to MQO, our method exploits implication relations, but a key difference is that we focus on the sharing of computation. Furthermore, our approach is applicable to user-defined code written in general-purpose programming languages rather than relational algebra.

Computation sharing and runtime optimizations The recent work described in [22] optimizes the shared execution of filters with expensive predicates. Here, filters are restricted to conjunctions of simple predicates and no sharing within arbitrary UDF-based predicates is supported. Hence, the approach presented in [22] would not be able to optimize many of the query families considered in our experimental evaluation. However, an interesting aspect of [22] is that it presents static and adaptive strategies that use information about the cost, selectivity, and participation of predicates to optimally execute a collection of filters. An interesting direction for future work may be to extend and adapt these strategies to program consolidation.

An emerging paradigm for data processing is the use of distributed stream platforms such as S4 [25] and Storm [3]. In S4, users express computations as distributed queries by breaking them up in smaller processing nodes called *bolts*. Other queries can "attach" subcomputations to already installed bolts, allowing the system to share computations. However, it is the responsibility of the *programmer* to identify such sharing opportunities. In contrast, our approach *automatically* identifies sharing opportunities within the user computation, and we can improve the execution of a "bolt" by merging it with another from a different user.

Nectar [14] provides memoization for DryadLINQ [34]. Nectar keeps track of run-time dependencies between input data and UDFs; hence, a new query that reuses some input and UDFs can also reuse pre-computed outputs. In contrast, our method can merge even *non-identical* UDFs that perform similar computations.

Static analyses for user-defined functions Typical RDBMS optimizers either treat UDFs as black-boxes or associate selectiviy and cost metrics with each UDF [7]. Recently, there have been proposals to use static analysis to reveal properties of the UDFs to the optimizer in the context of distributed data processing [17, 35]. These approaches perform static analysis of UDFs to decide whether two successive map operations can be safely reordered in data flow applications. While our approach also uses static analysis, our goal is to share computation rather than minimizing data accesses.

Relation to compiler optimizations and code generation Program consolidation is closely related to traditional compiler optimizations such as common subexpression elimination and constant folding, but we focus on sharing computation across multiple programs rather than within a single program. Of course, standard compiler optimization techniques could, in principle, be used for optimizing multiple UDFs that operate on the same data. Specifically, a naive way to consolidate two programs p_1 and p_2 is to create a single program that executes p_1 and p_2 sequentially and then feed this program to an optimizing compiler.

However, a distinguishing feature of our approach is that we expose shared or redundant computations between different programs by embedding code fragments of one within branches of another. An optimizing compiler would not perform such transformations because of potential code size blowup. In our setting, since different UDFs often have control flow with correlated predicates, this duplication is key for eliminating redundant computation. Another distinguishing feature of our approach is the use of SMT-based implication checks and symbolic execution with strongest post-conditions. While there have been some proposals for using heavier-weight decision procedures and static analyses particularly for automatic parallelization (e.g., [15, 29]), deep static analysis is typically considered too heavy-weight in standard compiler optimizations. However, in our setting, since the UDFs we consolidate tend to be similar, we obtain large benefits with small overhead. Finally, our loop consolidation performs a rich form of loop fusion but again uses logical implication checks and loop invariants for this purpose. We believe some of the ideas underlying our loop consolidation may also be applicable for optimizing compilers, and we plan to explore this in future work.

In the context of functional language optimizations, our work resembles map fusion [8, 13, 20]. However map fusion does not reuse computation between the fused functions. Aditionally, there is a connection to partial evaluation [18]. A system that receives N calls from the same application, $q(prms_i, data)$ $(i \in 1..N)$, can evaluate q on $data$ so that each query has to only evaluate the result on its parameters. This is impractical as the input data set is large. A promising alternative is to partially evaluate calls with the same *parameters*. This however requires query grouping heuristics by common parameters and does not handle situations where computation can be shared even when the query parameters do not match, but induce implications in the bodies of the functions.

Finally, previous work has used compiler optimizations to improve query execution in databases and distributed systems. The Steno system [23] optimizes the implementation of operators that contain UDFs to high-performance imperative code by specialization, unboxing, and elimination of high-level LINQ iterators and virtual method calls. Holistic query evaluation [19, 33] generates high-performance native code from high-level SQL queries, taking into consideration architecture-specific parameters. We view these techniques as complementary to ours; the merged UDF code we generate could be further optimized using these techniques.

Relation to refinement calculi Program consolidation can be viewed as a form of program refinement [5, 10]. Standard refinement calculi have been used to derive programs from specifications and reason about general program correctness [21]. In our context, we can view sequential composition as a specification and use our calculus as a refinement calculus to generate the consolidated program. Unlike standard refinement calculi, our consolidation rules are restricted by a notion of optimality similar to [16]. However, by abstracting optimality as the cost semantics of the language, we soundly incorporate general cost analysis into our calculus for free.

8. Conclusion and Future Work

We presented program consolidation, a new optimization technique motivated by the shared execution of queries with user-defined code, and demonstrated that it dramatically improves job completion times for mixes of similar UDFs operating on real-world data.

In future work, we plan to explore the applicability of program consolidation in high-throughput databases [12] and data-analytics clusters [31]. Since consolidation is quite efficient in practice (in the range of a few hundred milliseconds), we believe integrating program consolidation in such systems is feasible.

Another direction for future work is to explore the applicability of program consolidation in latency-critical applications. In general, when the physical parallelism of a system is saturated, incoming jobs are queued, so without some priority ordering or specialized system design, it is very hard to guarantee latency constraints. In such scenarios, latency-agnostic consolidation may not have a significant impact on the latency characteristics of the system. When latency guarantees or job priorities are present, it may be desirable to run the UDFs in some particular order and still ensure that consolidation does not increase the response time of any individual query. Our current consolidation algorithm does not address this problem, and we plan to investigate consolidation strategies that impose a partial or total query execution order.

Finally, we plan to study extensions to our calculus for programs with exceptions and error handling in order to address fault-tolerance concerns.

References

[1] Linq project, . URL `http://msdn.microsoft.com/en-us/library/vstudio/bb397926.aspx`.

[2] Msdn linq documentation, . URL `http://msdn.microsoft.com/en-us/library/bb669073(v=vs.110).aspx`.

[3] Storm project. URL `http://storm-project.net`.

[4] Wordcount example. URL `https://github.com/MicrosoftResearchSVC/Naiad/blob/release_0.2/Examples/`.

[5] R.-J. Back. A calculus of refinements for program derivations. *Acta Inf.*, 25(6):593–624, 1988.

[6] N. Bruno, S. Agarwal, S. Kandula, B. Shi, M. Wu, and J. Zhou. Recurring job optimization in scope. SIGMOD, pages 805–806, 2012.

[7] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. *ACM Trans. Database Syst.*, 24(2):177–228, 1999.

[8] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. ICFP, pages 315–326, 2007.

[9] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340. Springer, 2008.

[10] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1976.

[11] S. Finkelstein. Common expression analysis in database applications. SIGMOD '82, pages 235–245, 1982.

[12] G. Giannikis, G. Alonso, and D. Kossmann. SharedDB: killing one thousand queries with one stone. *VLDB*, 5(6):526–537, Feb. 2012.

[13] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. FPCA '93, pages 223–232, 1993.

[14] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: automatic management of data and computation in datacenters. OSDI'10, pages 1–8, 2010.

[15] M. Haghighat and C. D. Polychronopoulos. Symbolic analysis: A basis for parallelization, optimization, and scheduling of programs. In *LCPC*, pages 567–585, 1993.

[16] E. C. R. Hehner. Formalization of time and space. *Formal Asp. Comput.*, 10(3):290–306, 1998.

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

[18] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993. ISBN 0-13-020249-5.

[19] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, pages 613–624, 2010.

[20] E. Meijer, M. M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA*, pages 124–144, 1991.

[21] C. Morgan. *Programming from Specifications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990. ISBN 0-13-726225-6.

[22] K. Munagala, U. Srivastava, and J. Widom. Optimization of continuous queries with shared expensive filters. PODS'07, pages 215–224.

[23] D. G. Murray, M. Isard, and Y. Yu. Steno: automatic optimization of declarative queries. PLDI '11, pages 121–131, 2011.

[24] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *SOSP*, pages 439–455, 2013.

[25] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. ICDMW '10, pages 170–177, 2010.

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

[27] J. Park and A. Segev. Using common subexpressions to optimize multiple queries. In *ICDE*, pages 311–319, 1988.

[28] G. Press. Scaling users, not data: Sisense new take on machine learning and crowdsourcing. URL `http://aka.ms/Fzo7sh`.

[29] W. Pugh. The Omega Test: a fast and practical integer programming algorithm for dependence analysis. *CACM*, 8:4–13, 1992.

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

[31] Y. N. Silva, P. Larson, and J. Zhou. Exploiting common subexpressions for cloud query processing. ICDE, pages 1337–1348, 2012.

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

[33] S. D. Viglas. Just-in-time compilation for sql query processing. *Proc. VLDB Endow.*, 6(11):1190–1191, Aug. 2013.

[34] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. Gunda, and J. Currey. Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language. OSDI'08, pages 1–14.

[35] J. Zhang, H. Zhou, R. Chen, X. Fan, Z. Guo, H. Lin, J. Y. Li, W. Lin, J. Zhou, and L. Zhou. Optimizing data shuffling in data-parallel computation by understanding user-defined functions. NSDI'12, pages 22–22.

[36] J. Zhou, P.-A. Larson, J.-C. Freytag, and W. Lehner. Efficient exploitation of similar subexpressions for query processing. SIGMOD '07, pages 533–544, 2007.