Extracting Equivalent SQL from Imperative Code in Database Applications

K. Venkatesh Emani
Subhro Bhattacharya†
Karthik Ramachandra*
S. Sudarshan

Indian Institute of Technology, Bombay
{venkateshek, sudarsha}@cse.iitb.ac.in, {karthik.s.ramachandra, subhro.bhattacharya}@gmail.com

ABSTRACT

Optimizing the performance of database applications is an area of practical importance, and has received significant attention in recent years. In this paper we present an approach to this problem which is based on extracting a concise algebraic representation of (parts of) an application, which may include imperative code as well as SQL queries. The algebraic representation can then be translated into SQL to improve application performance, by reducing the volume of data transferred, as well as reducing latency by minimizing the number of network round trips. Our techniques can be used for performing optimizations of database applications that techniques proposed earlier cannot perform. The algebraic representations can also be used for other purposes such as extracting equivalent queries for keyword search on form results. Our experiments indicate that the techniques we present are widely applicable to real world database applications, in terms of successfully extracting algebraic representations of application behavior, as well as in terms of providing performance benefits when used for optimization.

1. INTRODUCTION

Database applications are written using a mix of declarative SQL queries and imperative code written in languages such as Java. Techniques that optimize across the declarative and imperative parts of a database application are referred to as holistic optimization techniques. Such holistic techniques, which exploit program analysis and rewriting in conjunction with query rewriting, can perform optimizations that are beyond the scope of a database query optimizer or an optimizing compiler for the imperative language.

In this paper, we present a novel holistic optimization technique which derives algebraic representations, or expressions, for program variables in database applications. The algebraic representation (D-IR) captures the effect of multiple program statements on a variable as a single expression. D-IR is then translated into a functional representation (F-IR) based on relational algebra and `fold`. Various transformation rules are presented to optimize F-IR, which is then translated into SQL. Our techniques have multiple applications, listed below.

Optimization of database applications: Our techniques allow many operations performed in imperative code in database backed applications to be translated to SQL queries making use of selections, joins, projections, and aggregate operations. Our techniques can detect when conditional execution, nested loops, and collection of results into an aggregate variable can be translated into SQL, reducing data transfer and even the number of queries executed.

There has been recent work by Cheung et al. [4] to transform parts of application logic into SQL queries, with a focus on database applications using Hibernate [13]. They rely on program synthesis technology, which is very resource intensive. Our techniques, on the other hand, rely on static program analysis, which is cheaper. For all programs that our techniques could successfully optimize, our techniques extracted equivalent SQL in much less time than [4], as shown by our experiments. Radoi et al. [17] proposed techniques for translating imperative code into MapReduce programs. While we use a functional representation which is similar to theirs, our goal is to infer SQL queries, so the techniques we propose are different. A detailed comparison with [4, 17], and other techniques for finding equivalent SQL queries, is given in Section 6.

Enhancing applicability of existing techniques: When the results of one query do not directly feed as parameters to another query, batching [11] is unable to combine these two queries into a single query. Techniques in this paper resolve assignments to intermediate variables and allow query parameters to be expressed in terms of program inputs or results of other queries. This enhances the applicability of batching as well as hybrid approaches proposed in [18], by combining related queries.

*Current affiliation: Microsoft Gray Systems Lab
†Current affiliation: Citrix Systems

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

ISBN 978-1-4503-3531-7/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2882903.2882926

Keyword Search: Keyword search systems such as [6] accept a manually extracted set of queries for each form, along with mappings of form parameters to query parameters as input. Our algebraic representations can be used to automate extraction of equivalent SQL queries for keyword search.

Contributions: The key novel contributions of this paper are as follows:

1. We present (in Section 3) a DAG based intermediate representation (D-IR) for application code, that expresses the value of a variable at a point in the program as an expression in terms of values available at an earlier point in the program. D-IR represents straight line and conditionally evaluated code algebraically, while loops have a non algebraic representation (only cursor loops are considered). We present techniques for deriving D-IR representations for program variables. Our techniques are based on program regions (refer Section 3.1), and can be applied to complex programs that include function calls.

2. We show (in Section 4) how to translate D-IR for cursor loops into a functional representation which uses `fold` along with relational algebra (we call it fold intermediate representation, or F-IR). F-IR is a convenient declarative representation for imperative code.

3. We present (in Section 5) transformation rules for F-IR, which help us in moving computation out of cursor loops and generating equivalent SQL queries. We then describe (in Section 5.2) how to rewrite the source program to use equivalent SQL. Our techniques are able to extract equivalent SQL partially for some variables that are amenable to algebraic analysis, while leaving other parts of code intact. Our techniques can translate many instances of nested loops where the inner loop computes aggregation for each value of the outer loop, into a `GROUP BY` query; earlier techniques are unable to perform such translations.

4. Our techniques have been implemented in the DBridge [2] system, to analyze and optimize Java programs that use JDBC or Hibernate. (The techniques themselves are not specific to any language or API.)

5. We present (in Section 7) an experimental evaluation of the proposed techniques on real world applications, which show the applicability of our techniques and their impact on application performance.

Section 2 presents an overview of our approach. We discuss related work in Section 6, and conclude the paper in Section 8.

2. OVERVIEW

An overview of our system is given in Figure 1. Given the source program (fragment), we first construct our

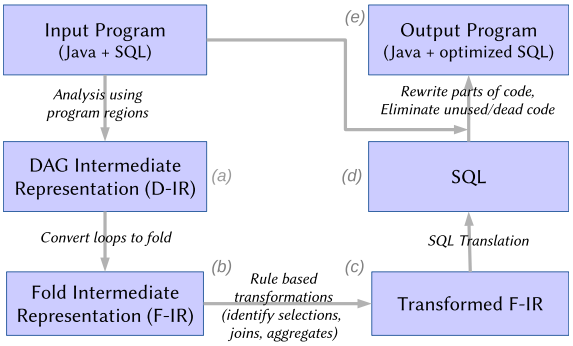


Figure 1: System Overview

DAG-based intermediate representation (which we call D-IR) for program variables. D-IR serves two purposes: (i) it resolves intermediate variable assignments, so the value of each variable at any program point is expressed in terms of values of variables at the beginning of the program, and (ii) it provides a semantic representation of the program.

For a variable whose computation we wish to optimize, its D-IR is translated into a functional representation using `fold` and relational algebra (F-IR), provided the required preconditions are met. The motivations behind translation of D-IR into F-IR are twofold: (i) F-IR provides a convenient declarative representation of the imperative program which is easy to translate into SQL. (ii) Transformations on F-IR are easy to describe and reason about, as F-IR uses higher order functions, which have well established properties. This makes it easy to prove correctness of transformation rules.

Rule based transformations are applied on the F-IR to push computation into the relational algebra query where possible; when no more transformations can be done, the rewritten F-IR representation is translated into SQL. The original program is then rewritten to derive the value of that particular variable, using the extracted equivalent SQL. Parts of the original program which are now rendered redundant/unused are removed. Translation of imperative code into SQL can greatly reduce the number of queries executed, and the amount of data transferred from the database, as compared to the original program.

Consider the code fragment shown in Figure 2, which is extracted from an open source gaming tournament software [15][1]. Variable types have not been displayed in this code, for ease of presentation (we will stick to this practice throughout the paper). It is part of a ranking page generator which tries to find the highest score across all tables in a round of the game Mahjong, where there are four players per table. The original code also finds the player who has the highest score along with

[1]Some changes have been made from the actual code for ease of presentation: (i) queries are made explicit and (ii) schema of the class `Board` has been simplified. Also, we use an abstract syntax for queries, which uses the pseudo function `executeQuery` that takes an SQL/HQL query, executes it and returns the result set/list of objects. Our implementation uses the actual source code.

```
findMaxScore(){
  boards = executeQuery("from Board as b
      where b.rnd_id = 1");
  scoreMax = 0;
  for(t : boards) {
    p1 = t.getP1();
    p2 = t.getP2();
    p3 = t.getP3();
    p4 = t.getP4();

    score = Math.max(p1, p2);
    score = Math.max(score, p3);
    score = Math.max(score, p4);
    if(score > scoreMax)
      scoreMax = score;
  }
  return scoreMax;
}
```

Figure 2: Code for highest score calculation

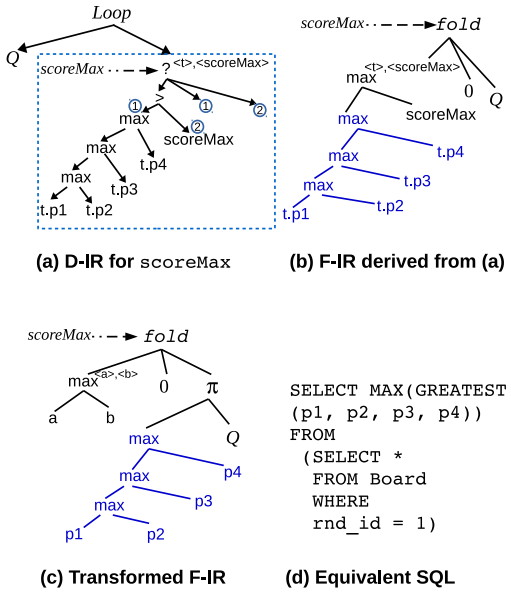


Figure 3: Walk-through of equivalent SQL derivation. Q denotes the query $\sigma_{rnd\_id=1}(Board)$

the score itself, for each round. Appendix B discusses how to generate equivalent SQL for such cases.

The optimized SQL for the `scoreMax` is shown in Figure 3(d)[2]. Parts (a), (b) and (c) show the various intermediate stages from the source code to optimized SQL, each of which will be described in detail in future sections. In parts (b) and (c), `max` is a function which returns the greatest of two elements.

The discussion in this paper targets loops that iterate over a collection, which we call *cursor loops*. If the iterated collection can be inferred (directly or indirectly) as equivalent to the result of a database query, we use the query to represent the collection. Otherwise, it is possible to create a temporary table at the database with the contents of the collection, and use a query (Q) on the temporary table to represent the collection. For simplicity, in this paper, we focus on the former case. For the latter case, we assume that Q is available. We omit details.

Furthermore, our discussion focuses on aggregates/collections built inside cursor loops. In addition to building aggregates/collections, another common use of cursor loops is to print values as they are computed in the loop. In such cases, we preprocess the program to replace output statements with appends to a (global) string (which can be treated as an ordered collection), and print its contents at the end of the program. The preprocessed program is then optimized using our techniques. We defer details to Appendix B.

Query execution calls are usually enclosed within exception handling code. Our implementation conservatively considers code that lies within a `try-catch` block, so that exception handling behavior is not altered due to optimizations. We also assume that loops do not contain unconditional exit statements like `break`, although certain cases of loop exit can be handling by some more engineering effort. Our experiments show that despite these restrictions, techniques presented in this paper have wide applicability. Inferring equivalent SQL across multiple `try-catch` blocks is work in progress.

[2]We illustrate using the `GREATEST` function of PostgreSQL. Translation into other dialects is possible using similar functions, or using `CASE..WHEN` construct.

3. DAG BASED IR

The goal of the first intermediate representation we use is to represent the values of program variables as algebraic expressions. We use a DAG representation that allows sharing of common sub-expressions between multiple expressions. We refer to our DAG-based intermediate representation as D-IR. DAG representation for basic blocks has been used in various code optimization techniques in traditional compilers [1]. We extend it to construct DAG representations for other program regions (Section 3.3).

In this section, we first present a background on program regions. We then discuss our D-IR representation and describe an algorithm for D-IR construction.

3.1 Background

A *Control Flow Graph* (CFG) is a directed graph in which nodes correspond to basic blocks in the program and edges correspond to control flow [1]. There are two specially designated nodes: the *Start* node, through which control enters into the graph, and the *End* node, through which all control flow leaves. CFGs are usually built on intermediate representations such as Java bytecode. Our techniques apply to any CFG; our implementation uses CFGs built on a representation called Jimple, provided by the Soot optimization framework [25].

Regions represent structured fragments of programs such as basic blocks, if-else blocks, loops, functions etc. A region in a flow graph is a set of nodes that includes a *header* that dominates all other nodes in the region,

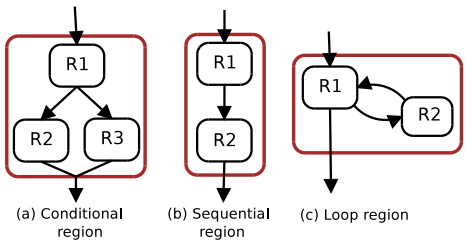


Figure 4: Types of regions

and has a single entry and exit. Regions are constructed from the CFG using rules described in [12]. Alternatively, it is possible to use an abstract syntax tree to identify program regions.

In our work, we handle four types of regions: basic block, sequential region, conditional region, and loop region (see Figure 4). In Figure 4, *R1*, *R2* and *R3* represent constituent regions of a parent region.

- **Basic Block Region**: A basic block represents a maximal group of consecutive statements that are executed together with sequential control flow between them. By definition, basic blocks cannot contain conditional constructs (if-else), loops or jumps of any sort. Regions R1, R2, R3 and R4 in Figure 5(a) represent basic blocks.
- **Sequential Region**: Sequential regions are regions composed of two sub-regions with sequential control flow between them (Figure 4(b)). In Figure 5(a), R6 is a sequential region.
- **Conditional Region**: A conditional region is comprised of three sub-regions (Figure 4(a)). The first sub-region (*R1*) contains the condition. The second sub-region (*R2*) is executed if the condition evaluates to true, otherwise the third sub-region (*R3*) is executed. We refer to *R1* as the "condition region", *R2* as the "true region" and *R3* as the "false region". In Figure 5(a), $R5$ is a conditional region composed of the condition region $R2$, true region $R3$ and false region $R4$.
- **Loop Region**: Loop regions are composed of two regions (loop header and loop body) with a cycle, as shown in Figure 4(c). Control flow starts at the loop header which contains the looping condition. If the condition evaluates to true, the loop body is executed, and control returns to the loop header to re-evaluate the condition. This is repeated until the condition becomes false, and then the loop exits.

By definition, regions compose other regions. We note that the program as a whole is also a region.

3.2 D-IR

D-IR is an intermediate representation for imperative code which may also contain database queries. It has two components: equivalent expression DAG (*ee-DAG*) and variable-expression map (*ve-Map*). Each region in

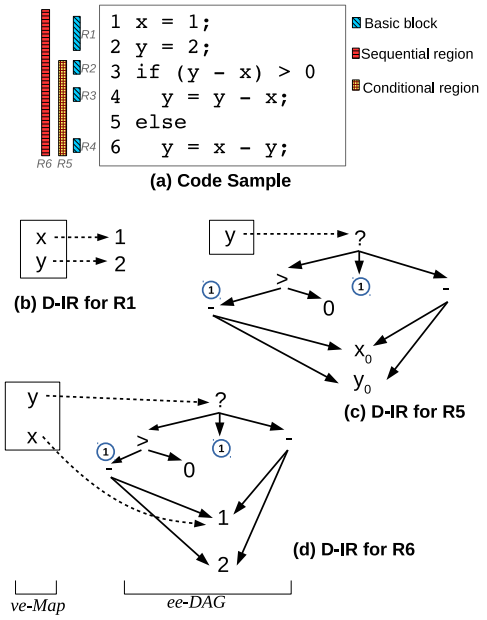


Figure 5: D-IR construction for a simple code fragment

the program has an ee-DAG and its associated ve-Map. We describe these data structures below.

3.2.1 *ee-DAG*

We define an equivalent expression DAG (ee-DAG) as a directed acyclic graph in which each node represents an *expression*. An expression (i) is a constant, a variable or a query attribute (base case) (ii) consists of an operator and its operands; each operand can in turn be an expression. The operator is connected to its operands through directed edges.

Consider the code sample shown in Figure 5(a). The ee-DAG for the program is shown in Figure 5(d). Circled numerals denote pointers to another part of the ee-DAG (to avoid clutter).

Parameterized queries in the source program can be treated as parameterized expressions in the multiset relational algebra. All relational algebraic operators (project (π), project (σ), join ($\bowtie$) etc.) are available in ee-DAG. Note that in this paper, π denotes projection without duplicate elimination. We also include extended relational algebraic operators for aggregation (γ), sorting (τ) and eliminating duplicates (δ) [27][3].

Relational operators do not guarantee that the order of input tuples is preserved in the output, unless explicitly sorted using τ. However, in this paper, we assume that for the operator π (projection without duplicate elimination), the input ordering is preserved in the output.

All arithmetic operators ($+$, $-$, $*$, / etc.), and logical operators ($<$, $>$, $==$, AND, OR etc.), are available

[3]Usage: ${}_G\gamma_{Agg(E)}(R)$ groups R on G and performs aggregations in Agg (G could be empty), $\tau_L(R)$ sorts R on L, and $\delta(R)$ eliminates all duplicate rows from R.

in the ee-DAG. In addition, we introduce the following operators to enable representation of imperative code constructs:

- Conditional evaluation ("?"): Its semantics are similar to ternary operator or if-else in imperative languages like Java/C.

- Cursor loop (*Loop*): The *Loop* operator represents computations inside cursor loops. It accepts two operands.. The first operand is the relation or query result over which the loop iterates. The second operand represents the loop body. Unlike other operators, the *Loop* operator does not represent a single value. In this sense, the *Loop* operator is not algebraic. The loop body is represented by enclosing it inside a dotted rectangle (refer Figure 3(a)). Expressions inside this rectangle correspond to a single iteration of the loop.

Since our implementation can take imperative programs with rich language features as input, equivalent ee-DAG operators are necessary to model semantics of source program statements. We chose Java programs to demonstrate the working of our tool, so equivalent ee-DAG operators were created for the following Java constructs:

- String operations, addition/deletion of elements into/from collections (Array, List and Set), getter and setter functions for object attributes

- Important library functions – for example, in Figure 2, our system understands that `Math.max` is a function which returns the maximum of two numbers. This is modeled in D-IR using the `max` operator. Adding support for more library functions is not hard, and our ee-DAG can easily evolve as needed.

In our ee-DAG, an operator is represented as a node, and its operands are represented as children of the operator node. For example, in Figure 5(c), the condition `y - x > 0` is represented by the node rooted at $>$, and the operation `y - x` is represented by the node rooted at $-$. Similarly, in Figure 3(c), the ee-DAG rooted at π represents the query $\pi_{\texttt{max(max(max(}p1,p2\texttt{),}p3\texttt{),}p4\texttt{)}}(Q)$.

3.2.2 ve-Map

The ve-Map is a key-value data structure where a key is the label of a program variable (v) and its value is a pointer to a node e in the ee-DAG. The expression e when evaluated gives the value of variable v, in terms of values available at the beginning of the region. We refer to this ee-DAG expression as the equivalent ee-DAG expression (or simply the *equivalent expression*) of the program variable. In the illustrations in this paper, we denote ve-Map values (pointers) with dotted arrows. Note that these dotted arrows are not part of the ee-DAG edges. The ve-Map for the program in Figure 5(a) is shown in Figure 5(d). In this paper, we skip showing entries in the ve-Map for variables in which we are not interested (to make diagrams more readable).

3.3 Algorithm for D-IR Construction

D-IR construction works on top of the region hierarchy. Construction of regions was discussed in Section 3.1. We now outline a bottom up recursive algorithm for D-IR construction for a region. Appendix D describes the algorithm in full detail.

1. Construct D-IR (ee-DAG and ve-Map) for each constituent region (sub-region). All leaves in the ee-DAG which are variables are marked as region inputs.

2. Merge D-IRs of sub-regions appropriately (depending on type of parent region) to obtain D-IR for the parent region. The aim of merging is to replace region inputs with their ee-DAG expressions, which are expressed terms of inputs to a preceding region.

In the illustrations in this paper, region inputs are denoted by tagging them with a subscript 0, for example, $\texttt{x}_0$ and $\texttt{y}_0$ in Figure 5(c).

The smallest sub-region in a program is a single statement. Thus, D-IR construction for a program starts by constructing D-IR for simple statements, which are merged to get D-IR for basic blocks, which are merged to get D-IR for other composite regions. This process halts when the variable values are expressed in terms of inputs to the outermost region of interest.

We consider each simple source program statement as consisting of an expression (comprising of an operator and its operands) whose value is assigned to an optional target variable. For example, the statement `sum = 5 + 10` consists of an addition operation involving the operator + and its operands `5` and `10`, with `sum` as the target variable. A source language expression is represented in D-IR using an equivalent ee-DAG operator and equivalent expressions for operands as its children. Assignment is captured by adding an entry in ve-Map for the target variable (or updating, if an entry already exists), with its value as the ee-DAG expression.

We now outline the steps for D-IR construction for each type of region.

For a sequential region R comprising of regions R_1 and R_2 such that R_2 follows R_1, the ee-DAG for R is obtained by replacing each leaf variable (region input) in the ee-DAG of R_2, with the ee-DAG of the variable from R_1. For a conditional region R comprising of a condition c, true region R_1 and false region R_2, the ee-DAG for each variable in R is obtained by creating a conditional evaluation node ("?") with its three children as c, ee-DAG of variable from R_1 and ee-DAG of variable from R_2. For a loop region, the ee-DAG is obtained by creating a "*Loop*" parent node with its two children as the looping query and the loop body. Two ve-Maps are merged by creating a union of entries from both ve-Maps. In case of duplicate keys, entries from the following region are retained.

Consider the code sample shown in Figure 5(a). Figures 5(b), (c) and (d) show the step by step construction of D-IR for the program. In Figure 5(c), the variables `x` and `y` are leaves, so they are marked as region inputs

(by tagging them with a subscript 0). These are resolved to constants in Figure 5(d), while merging with the preceding basic block *R1*. Note that in the final D-IR (Figure 5(d)), all intermediate variable references have been resolved to inputs at the beginning of the program. In order to efficiently check the existence of a node in the ee-DAG, a composite id – comprising of id's of its operator and operands – is assigned to each node, and a hash table is used for searching.

User defined functions/procedures are also handled by our techniques. D-IR is separately constructed for a user defined function/procedure. It is then merged with the preceding region at the caller location, by considering them to form a sequential region, taking into account actual to formal parameter mapping. We defer details to Appendix D.

4. F-IR REPRESENTATION

As we have described in Section 3.3, D-IR gives an algebraic representation for computations in sequential and conditional regions. However, the *Loop* operator used to represent cursor loops in D-IR is not algebraic. We now describe our second intermediate representation, which we call the *fold intermediate representation* (F-IR). F-IR combines D-IR and `fold` function to enable algebraic/functional representation of cursor loops. F-IR abstracts away details about imperative logic while still retaining the result ordering, unlike relational algebra. This allows easy translation from cursor loops to F-IR. Our transformation rules then operate on F-IR (Section 5). There has been earlier work on using `fold` to represent loops [17].

We use the following textual notation to represent a dag/tree. In general, $\triangle$ represents an ee-DAG, and e represents an expression tree. We write $op[c_1, c_2, \ldots]$ to describe an expression with root node as op, and $c_1, c_2, \ldots$ as its children. Angular brackets $\langle\rangle$ denote a parameter. Thus, $op[c, \langle p\rangle]$ denotes that the second child of op is a parameter p. Similarly, we use $e^{\langle p_1\rangle,\langle p_2\rangle,\ldots}$ to denote an expression, where values of some operands in the expression are given by parameters p_1, p_2 etc. A parameterized expression (say, $\mathtt{f}^{\langle p_1\rangle,\langle p_2\rangle}$) can be treated as a function taking parameters (p_1 and p_2). These parameters are substituted when the expression is evaluated with actual values. Parentheses are used to denote invocation of a function or a parameterized expression with actual values, for example, $\mathtt{f}(v_1, v_2)$ denotes a call to function `f` with parameter values v_1 and v_2, and $e(v)$ denotes evaluating the parameterized expression $e^{\langle p\rangle}$ by substituting p with v.

4.1 Fold

The `fold` function is a higher order function i.e., a function which can take other functions as parameters and can return functions as return values. `fold` takes 3 arguments: a folding function `f`, an identity element (*id*) and a recursive data structure on which `fold` is to be applied. In the context of lists, the `fold` function processes each element in an input list and combines the results into a single return value. The list elements can be processed from first to last, or last to first, depending on whether it is a left fold (`foldl`), or right fold (`foldr`). In this paper, by `fold`, we mean `foldl`. `foldl` is more suitable for representing computations on lists in imperative programs, as the elements are usually processed from first to last. `fold` on lists is defined recursively, as follows:

$$\mathtt{fold}\ [\ \mathtt{f},\ id,\ [\ ]\ ] = id$$
$$\mathtt{fold}\ [\ \mathtt{f},\ id,\ [a_1]\ ] = \mathtt{f}(id,\ a_1)$$
$$\mathtt{fold}\ [\ \mathtt{f},\ id,\ [a_1, .., a_{n+1}]\ ] = \mathtt{f}(\ \mathtt{fold}\ [\ \mathtt{f},\ id,\ [a_1, .., a_n]\ ],\ a_{n+1})$$

Note that [and] are used both to denote lists, and to enclose arguments to the higher order function `fold`. (In the latter case, we could have used parentheses in place of [ ], or omitted them altogether, as is customary in the functional programming community, but we use [ ] to improve readability of our transformation rules.) For example, $\mathtt{fold}\ [\ +,\ 0,\ [1, 2, 3, 4, 5]\ ]$ evaluates to

$$((((0 + 1) + 2) + 3) + 4) + 5 = 15.$$

The definition of `fold` can be easily extended to operate on results of queries, which are ordered/unordered collections, instead of lists.

We extend ee-DAG to allow the use of a `fold` operator. We refer to D-IR extended with `fold` operator as F-IR. The `fold` operator takes three arguments, corresponding to each argument of a `fold` function. Note that the first argument can be a parameterized expression, which is treated as a function. Input queries are expressed using extended relational algebra. For example, the ee-DAG rooted at `fold` in Figure 3(c) is written as

$$\mathtt{fold}\ [\ \mathtt{max},\ 0,\ Q'\ ]$$

where `max` is the binary maximum function, and Q' denotes $\pi_{\mathtt{max(max(max}(p1,p2),p3),p4)}(\sigma_{rnd\_id=1}(Board))$.

4.2 Converting Loops to Fold

In this section, we describe how to use `fold` to precisely represent cursor loops, in F-IR. Equivalent SQL cannot be extracted for collections other than those constructed from query results (directly or indirectly), so we focus on cursor loops. Our system currently supports the following aggregations inside cursor loops: set/multiset insertion (`insert`), appending to list (`append`) or scalar aggregation (`min`, `max`, `sum`).

Before we present the algorithm for D-IR to F-IR translation (Figure 6), we describe a few terms commonly used in program analysis. A *loop carried flow dependency* is said to exist between two statements S_1 and S_2 in a loop, if S_2 follows S_1 in the control flow, and S_2 writes to a location which is read by S_1 in a future iteration. An *external dependency* is said to exist between S_1 and S_2 if both the statements access the same external location (file, database etc.), and at least one of S_1 or S_2 writes to the external location. For the purpose of dependence analysis, we conservatively treat the entire database/file as a single location. This is required since writes to a relation may trigger updates on another relation. Also, reading/writing an element in a collection is treated as accessing the entire collection. A *data dependence graph*(DDG) [16] of a program is a directed multi-graph in which program statements are nodes, and the edges represent data dependencies between the statements. Directions and labels on edges

procedure `toFIR`(R):

R: A program region.
Let $R.\triangle$ be the ee-DAG for R, and $R.M$ be its ve-Map.
begin
foreach sub-region C of R
`toFIR(C)`
if R is not a (cursor) loop region, return
else `loopToFold`(R)
end

procedure `loopToFold`(R):

R: A cursor loop region
begin
foreach variable v that is updated in R {
Let S = `slice`(R, l, v) represent a slice for v over R, where l is the program point at the end of R
Let S_{acc} be the subset of statements from S updating v.
Let D_S be the part of the data dependence graph for R that corresponds to statements from S.
if all the preconditions below are satisfied in D_S:
(P1) there should be a cycle of dependencies containing S_{acc} and a loop carried flow dependence (*lcfd*) edge (E).
(P2) there should be no other *lcfd* edge apart from E and the *lcfd* edge due to update of the loop cursor variable.
(P3) there should be no external dependencies.
{
Extract the expression tree $e = Loop[Q, e_{acc}]$ from the ee-DAG given by $R.M$.`lookup`(v).
Let v_0 be the initial value of v at beginning of the loop.
$foldExpr$ = `fold`$[e'^{\langle v \rangle, \langle t \rangle}_{acc}, v_0, Q]$
where, e'_{acc} is obtained from e_{acc} by replacing each reference to attributes of Q, with reference to corresponding attributes of t (t is a new tuple variable).
Add *foldEpr* to $R.\triangle$
Update the entry for v in $R.M$ to point to *foldEpr*
Replace pointers to v in $R.\triangle$ with pointers to *foldEpr*
Insert statement(s_{fold}) "v = *foldExpr*" at the end of R
`updateDDG(R)` /*reconstruct the DDG for R by ignoring unused/dead code (see description)*/
}
}
end

Figure 6: Algorithm for Conversion to F-IR

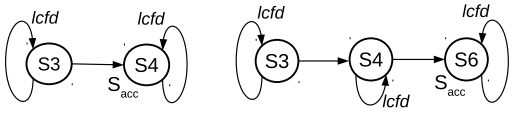
identify the direction and type of dependence, respectively. A program slice S = `slice`(P, n, v) is defined [26] as the subset (S) of all statements and control predicates of the program P that directly or indirectly affect the value of a variable v at the program point n. For example, in Figure 7(a), let the program point at the end of line 7 be l_7. Let P_{37} represent the loop, which contains lines 3 to 7. Then, `slice`$(P_{37}, l_7,$ `agg`$) = \{S3,S4\}$. Similarly, `slice`$(P_{37}, l_7,$ `dummyVal`$) = \{S3,S4,S6\}$. The algorithm is given in Figure 6.

Given a program region R, the algorithm attempts to translate all variables updated in R into F-IR. Updates to variables in a loop are translated into `fold` using the `loopToFold` procedure. The statement "v = *foldExpr*" (labeled s_{fold}) is a stub. The goal is to translate *foldExpr*

```
1 rs = executeQuery(Q);
2 agg = 0;
3 while(rs.next()){ //S3
4   agg += rs.getInt("x"); //S4
5   prettyPrint(rs.getString("y"),
                rs.getString("z")); //S5
6   dummyVal += agg; //S6
7 }
```

Q: SELECT x, y, z from R

(a) Code Sample



(b) Slice for `agg` **(c) Slice for** `dummyVal`

Figure 7: Demonstration of preconditions for translation into F-IR

into SQL. However, for the purpose of dependence analysis, we treat *foldExpr* as an algebraic expression.

Dead code [5] refers to code whose results are not used in any other computation. It may be transitive, i.e., identifying a part of the code as dead may reveal more dead code. Let S_{dead} denote the set of all statements which are rendered dead, due to insertion of s_{fold}. The actual decision of whether to use equivalent SQL (and remove S_{dead}) or not, happens after F-IR transformations on *foldExpr* (Section 5.1). However, dependences due to S_{dead} may cause preconditions to fail in an enclosing region. The procedure `updateDDG` reconstructs the DDG by ignoring statements in S_{dead} and including s_{fold}. Note that if the preconditions fail for a variable, the algorithm proceeds to attempt F-IR translation for other variables in the loop.

Theorem 1: Given a cursor loop region R, the value of a variable v after termination of the loop is equivalent to the result of *foldExpr* for v obtained by `loopToFold`(R), when executed on the same input.
A proof sketch for this theorem is given in Appendix A.

Consider the code sample shown in Figure 7(a). Figure 7(b) shows that the loop body slice for `agg` at the end of the loop satisfies the preconditions for translation into F-IR. The F-IR representation for `agg` is given as: `fold` [`f`, 0, Q] where `f`$^{\langle v \rangle, \langle t \rangle} = +$ [v, $t.x$].

The slice for `dummyVal`, as shown in Figure 7(c) violates P2, due to the presence of an additional *lcfd* edge from *S4* to itself. Thus, the ee-DAG for `dummyVal` cannot be translated into F-IR. We note that although our preconditions disallow an F-IR representation for `dummyVal`, in general, it is possible to represent `dummyVal` as a `fold`, where the folding function aggregates a pair of values (`agg` and `dummyVal`). However, SQL translation of `dummyVal` is not possible (without using a custom aggregation function), as it is dependent on `agg`. In Appendix B, we describe how some cases of dependent aggregations can be handled.

Note that the structure "`if` (*expr OP v*) `then` v = *expr*" is often used to implement `min` and `max` aggrega-

tions in a loop, where OP is one of $<$, $>$, $<=$, $>=$. This structure is translated into $v = OP_1(v, expr)$ where OP_1 is `max` for $>$, $>=$, and `min` for $<$, $<=$. If the program uses v OP $expr$, then it can easily be translated to the form $expr$ OP v before applying the above translation. Translation into F-IR is done after applying the above translation.

5. F-IR TRANSFORMATIONS

In this section, we present a number of transformations on F-IR representation. Our transformations are expressed as equivalence rules. Each rule has an input F-IR which can be replaced by an equivalent output F-IR by applying the rule. The aim of our transformations is to obtain an optimized F-IR from the given F-IR. By optimized F-IR, we mean an F-IR which when translated into SQL, reduces the number of queries and/or data transferred as compared to the original F-IR.

5.1 Transformation Rules

We now present transformation rules. Many of these rules specify a pattern for the relational algebra input to `fold`. The actual input may not directly match this pattern. However, standard relational algebra transformations can be used to bring the input query to the required structure to apply transformations.

Rule T1 (Simplification): If `append` denotes the list append operator, `insert` denotes the set insertion operator, and δ denotes the duplicate elimination operator,

$\texttt{fold}[\ \texttt{append},\ [\ ],\ Q\ ] = Q$ (Rule T1.1)

$\texttt{fold}[\ \texttt{insert},\ \{\},\ Q\ ] = \delta(Q)$ (Rule T1.2)

Rule T2 (Predicate push):

If $\texttt{f}^{\langle v\rangle,\langle t\rangle} = ?[\ pred(t),\ \texttt{g}\ ]$, then

$\texttt{fold}\ [\ \texttt{f},\ id,\ \pi_L(\tau_Z(Q))\ ]$

$\equiv \texttt{fold}\ [\ \texttt{g},\ id,\ \pi_L(\tau_Z(\sigma_{pred}(Q)))\ ]$

where $pred(t)$ is a predicate expression parameterized only on attributes of tuple t, and τ is the relational sort operator (Section 3.2.1). Z can be empty, which signifies absence of ordering on Q.

The selection predicate $pred$ is obtained from $pred(t)$ by replacing references to attributes of t with corresponding attributes of Q. We will use the terms $pred(t)$ and $pred$ in similar contexts in other transformation rules, without describing them again.

Rule T3 (Push scalar functions into the query):

If $\texttt{f}^{\langle v\rangle,\langle t\rangle} = \texttt{g}(v,\ \texttt{h}(t.A))$, then

$\texttt{fold}\ [\ \texttt{f},\ id,\ \pi_A(Q)\ ]$

$\equiv \texttt{fold}\ [\ \texttt{g},\ id, \pi_{\texttt{h}(A)}(Q)]$

This rule can easily be extended to the case when `h` operates on more than one attribute of tuple t.

Rule T4 (Join identification)

(Rule T4.1 – list append):

If $\texttt{f}^{\langle v\rangle,\langle t\rangle} = \texttt{fold}\ [\ \texttt{append},\ v, \pi_L(\tau_{Z_2}(\sigma_{pred(t)}(Q_2)))\ ]$, then

$\texttt{fold}\ [\ \texttt{f},\ [\ ],\ \tau_{Z_1}(Q_1)\ ]$

$\equiv \pi_L(\tau_{Z_1, Q_1.K, Z_2}(Q_1 \bowtie_{pred} Q_2))$

provided Q_1 has a unique key K, where `append` is the list append operator. Z_2 can be empty.

This rule is the same as the join identification rule used by Cheung et al. [4], but the output should be sorted on $(Z_1, Q_1.K, Z_2)$, and not just (Z_1, Z_2).
For insertion into a set, the result ordering does not matter, but duplicates should be eliminated. Thus, the rule can be given as follows.

(Rule T4.2 – set insertion):

If $\texttt{f}^{\langle v\rangle,\langle t\rangle} = \texttt{fold}\ [\ \texttt{insert},\ v,\ \pi_L(\tau_{Z_2}(\sigma_{pred(t)}(Q_2)))\ ]$, then

$\texttt{fold}\ [\ \texttt{f},\ \{\},\ \tau_{Z_1}(Q_1)\ ] \equiv \delta(\pi_L(Q_1 \bowtie_{pred} Q_2))$

Z_1 and Z_2 can be empty. In the case of multiset `insert`, duplicate elimination is not required, so the RHS would simply be $\pi_L(Q_1 \bowtie_{pred} Q_2)$ (Rule T4.3).

Rule T5 (Aggregations)

(Rule T5.1 – entire relation):

$\texttt{fold}[\ op,\ id,\ \pi_A(Q)\ ] \equiv \gamma_{op\_agg(A)}(Q)$

where op_agg is the relational aggregation operator corresponding to the binary operator op, id is the identity element for op. For op=+, op_agg = `sum`; for op = `max`, op_agg = `max`; for op = `min`, op_agg = `min`. Note that we overloaded `max` to represent both binary and aggregation operators.

(Rule T5.2 – group by):

If $\texttt{f}^{\langle v\rangle,\langle t\rangle} = \texttt{append}[v, (t.B, \gamma_{op\_agg(A)}(\sigma_{pred(t)}(Q_2)))\ ]$, then $\texttt{fold}\ [\texttt{f},\ [\ ],\ Q_1\ ] \equiv$

$\pi_{Q_1.B,\ op\_agg(Q_2.A)}(_{Q_1.*}\gamma_{op\_agg(Q_2.A)}\ (Q_1 \rtimes_{pred} Q_2))$

provided the order of Q_1 is not deterministic, and Q_1 has a key. This rule can be extended easily to handle `insert` in the place of `append`.

Note that the above transformation works with standard SQL semantics for aggregates involving NULL values. The general case that Q_1 may not have a key, and may be ordered can be handled using extensions to techniques for decorrelation of aggregate queries [7]. We omit details for lack of space. In rules T5.1 and T5.2, the initial variable value (second argument) passed to `fold` was the identity element for the folding function. In Appendix B, we discuss transformation rule T6 that enables us to handle the case when the above assumption does not hold.

Rule 5.2 is used to translate a common implementation of group by in imperative code, where there are two nested cursor loops, the outer loop defines the groups, and the inner loop performs aggregation of rows in the group, and appends the aggregated result to a result list. Appendix B describes how to handle another common case where, in addition to the aggregated value, a tuple of values from the row containing the aggregated value is returned (for example, one may want the name of a student who scored the highest marks in a test, along with his/her marks).

We present some more transformation rules which are used in our implementation, in Appendix B. Similar to database query optimizer rules, more transformations

can be added to exploit other opportunities for inferring relational operations performed in imperative code.

5.2 Generating and Using Equivalent SQL

After a program has been translated into F-IR, we use a top down traversal of its regions to rewrite the program to use equivalent SQL, by processing the parent region first, and then its sub-regions, as follows.

For a region R, we consider each statement (s_{fold}) "$v = foldExpr$" that is directly inside R (inserted during F-IR translation), and apply transformations (Section 5.1) on *foldExpr*. Let *transExpr* denote the resultant F-IR obtained after all transformations have been applied. If *transExpr* does not contain any `fold`s (i.e., it is a relational algebra expression), and all functions in *transExpr* have equivalent SQL functions, then translation of *transExpr* into SQL is straight forward. In some cases, if the folding function does not have an equivalent SQL aggregate function, it is possible to use a custom aggregation function (either as a user defined function inside the database, or as a stored procedure defined in the application source language, if the database allows it). In other cases, SQL translation fails.

If an SQL query (Q) could be obtained from *transExpr*, we replace the stub s_{fold} with the statement (s_{sql}) "$v =$ `executeQuery`(Q)"[4]. Parts of region R which are now rendered dead due to s_{sql} are removed by dead code elimination. If SQL translation for *transExpr* fails, then the assignment "$v = foldExpr$" is removed. The original code for v remains intact.

Replacing the original source with SQL generated from transformed F-IR is most often a good idea. However, from an optimization view point, the decision to replace should be taken in a cost based manner, in general, as discussed in the next section.

5.3 Application of Transformation Rules

In our implementation, we apply transformation rules in the left to right direction. Our transformation rules match the LHS for a syntactic pattern, and replace it with the RHS. We assume that translation into SQL is always beneficial.

In case multiple transformation rules are applicable for a given program fragment, we choose any one of the applicable rules and proceed. In the current set of rules (T1 through T7), a transformation from LHS to RHS does not destroy any syntactic patterns in the LHS, which are amenable for transformation by other rules. So, the order of application of the competing rules does not matter. Thus, the rule set is confluent. It can be verified that our current set of rules always push computation from the folding function into the query, and not in the other direction. Thus, infinite derivations are not possible, and neither are cyclic derivations. Thus, our current rule set always terminates. However, addition of new transformation rules may result in cycles.

Translation into SQL may not be beneficial for all programs. For example, consider the code sample from Figure 7(a). Our techniques will extract a separate query for the aggregated variable `agg`, but the entire data still has to be fetched to print other information with rich formatting. In this case, the cost of an additional query will outweigh the benefit of pushing aggregation into the database.

For this particular case, a simple heuristic can be used to decide whether or not to do the transformation: transform only if equivalent SQL could be extracted for all variables inside the loop that use query results. However, in general, a cost based exploration of the space of possible rewrites of the program is necessary, to choose the best possible rewrite. This is an important area of future work. We sketch an approach to a cost based choice of rewriting which we plan to implement, in Appendix C. This approach uses a top down search algorithm using an AND-OR DAG, based on the Volcano/Cascades query optimizer [9, 10].

5.4 Limitations

Our techniques focus on optimization of programs that iterate over a query result, performing actions that can be translated into SQL. Our system cannot handle cases where there are language constructs that cannot be represented in F-IR, like custom comparators, type based selection, retrieving the i'th element in a list etc. Expanding F-IR to address some of these cases is an area of future work. We note, however, that other parts of the program may still be amenable to optimization.

There are complex F-IR expressions that cannot be translated into SQL. One such example where the order of print statements needs to be preserved is discussed in Appendix B. Often, data structures in imperative programs are over-specified (for example, using a list in place of a set). Respecting such over-specification sometimes makes our transformation rules inapplicable. Techniques for "weakening" the data structures (for example, using a set instead of a list), which could make our rules applicable, are part of future work.

6. RELATED WORK

Wiederman et al. [28] propose a source-to-source program transformation technique that analyzes programs using Hibernate to identify conditions under which retrieved data is used, and rewrites the program to use explicit queries where the conditions are included in the query. However, they do not extract queries for other relational operations such as joins and aggregations expressed in imperative code.

There has also been recent work on inferring SQL queries from procedural code using program synthesis by Cheung et al. [4]. Their approach generates possible equivalent SQL queries and uses the Sketch framework [24] to check for equivalence. This approach is quite powerful, but, as evidenced by their results, can be quite expensive. While Cheung et al. identify continuous code fragments which can be replaced by an SQL query, our techniques can also transform intermittent fragments of code into SQL, thus enhancing their applicability. Cheung et al. developed a Theory of Ordered Relations (TOR) as an intermediate representation to express loop invariants and post conditions before con-

[4] `executeQuery` is a short form notation described earlier in footnote 1

verting to SQL. Our intermediate representation (F-IR), on the other hand, does not need a new algebra, and makes use of `fold` and existing operators from extended relational algebra.

Zhang et al. [30] propose techniques to infer queries, using the output table and database schema information by treating the source code (query) that generated the result as a black box. However, with this approach, guarantees for correctness of the query cannot be given for all inputs, since test inputs may not be exhaustive.

Recently, Radoi et al. [17] have proposed an approach for automatic translation of sequential array-based code into a parallel MapReduce framework. They use a functional intermediate representation (IR) and present rewrite rules that enable parallelism and translate the IR into Scala MapReduce code. Although their IR is similar to our F-IR based representation, their goals are quite different from ours. Our transformation rules are designed with the aim of inferring relational operations from the IR, while they focus on enabling parallelism and extracting map and reduce operations. Also, the work of Radoi et al. is suited purely for batch processing programs, while our work, in addition, considers application code where data access is interspersed with presentation (UI) logic, such as Web and mobile applications.

Iu et al. [14] propose a syntax (JQS) through which certain complex SQL queries can be expressed using normal (imperative) Java constructs. Similarly, Giorgidze et al. [8] present a Haskell library which allows developers to express database queries using Haskell constructs. However, an important difference of our techniques from [14] and [8] is that our techniques automatically infer which parts of imperative code can be pushed into the database. In contrast, [14] and [8] require developers to provide this information, in a syntax that uses source language constructs. Some of the techniques of Cheney et al. [3], for translating XQuery to SQL, could be useful for handling print statements as discussed in Appendix B. However, their goals and techniques are otherwise very different from ours.

Shi et al. [22] propose the UniAD system to unify execution of imperative code and queries at a single execution engine. They target only ad-hoc data processing tasks with small data sets, and use a custom database engine, hence they cannot leverage the query optimization capabilities of popular database systems.

Simhadri et al. [23] proposed techniques to algebraize imperative constructs in user defined functions (UDFs). Their aim was to extract a single relational algebra expression for the entire UDF body. Our techniques are applicable over a much richer set of imperative constructs including objects and collections.

Guravannavar et al. [11] proposed program analysis and transformation methods to exploit set oriented query execution to improve performance of iterative execution of parameterized queries. Ramachandra et al. [19] proposed a technique to prefetch query results across function calls. As discussed in Section 1, our techniques can be used in conjunction with the techniques of [11, 18, 19], to further enhance application performance.

7. EXPERIMENTAL EVALUATION

Our implementation is in Java. We used the Soot framework [25] for program analysis, and we incorporated a region based analysis framework in Soot. Our framework builds a hierarchical region tree over the CFG, and provides the infrastructure for traversing through regions, as well as merging the results of our analysis across regions.

For evaluation, we used our tool on code samples adapted from four real world applications namely, Wilos [29] – an orchestration software, Matoso [15] – a ranking software for Mahjong tournaments, *AcadPortal* and *JobPortal* – two real world applications in production use at IIT Bombay; and two benchmark applications namely, RuBiS [21] – a bidding system modeled after *ebay.com*, and RuBBoS [20] – a bulletin board like *slashdot.org*. Our experiments were run on a machine with 8GB RAM with Intel Core i7-3770, 3.40GHz CPU running Ubuntu Linux, with MySQL 5.5 database server. The client was on the same machine.

We use *EqSQL* to refer to techniques in this paper, and QBS to denote techniques by Cheung et al. [4].

7.1 Applicability

The techniques presented in this paper can be used with any language and data access API. We have implemented our techniques for database backed applications written in Java. Our implementation supports applications using Hibernate for database access.

Experiment 1 (Comparison with Cheung et al. [4]): Cheung et al. [4] reported the applicability of their techniques for code samples extracted from Wilos, an open source application that uses Hibernate. We tested our implementation on the same code samples. The results are shown in Table 1. The values in these columns denote time taken for equivalent SQL extraction in cases where the system succeeded. The numbers for QBS have been taken from [4]. "–" denotes that a particular code sample could not be optimized due to limitations in the techniques. ✓denotes that the code fragment can be handled by the techniques we propose, although they are not handled by our current implementation.

While QBS could automatically extract equivalent SQL in 21/33 cases, our system succeeded in 17/33 cases, although there are 7 further cases which can be handled by our techniques, but are not handled by our current implementation; we are working on extending our implementation to handle such cases. In 6 of the cases where our current implementation is able to extract equivalent SQL, QBS fails.

Techniques in [4] and those presented in this paper do not handle database updates. However, while [4] entirely rejects code fragments involving database updates, our tool partially optimizes such code fragments by keeping update statements intact, and extracting equivalent SQL for other variables in the code fragment, provided the update statements do not introduce a dependency between other statements. Similar to [4], our techniques fail for code samples 5 and 7 that contain polymorphic type comparison and selection using custom comparator, which are not handled in EqSQL.

Sl.	File (Line No.)	QBS	EqSQL
1	ActivityService (401)	–	< 1
2	ActivityService (328)	–	< 1
3	Guidance Service (140)	–	< 1
4	Guidance Service (154)	–	< 1
5	ProjectService (266)	–	–
6	ProjectService (297)	19	< 1
7	ProjectService (338)	–	–
8	ProjectService (394)	21	< 2
9	ProjectService (410)	39	< 1
10	ProjectService (248)	150	< 1
11	AffectedtoDao (13)	72	< 2
12	ConcreteActivityDao (139)	–	–
13	ConcreteActivityService (133)	–	✓
14	ConcreteRoleAffectationService (55)	310	✓
15	ConcreteRoleDescriptorService (181)	290	–
16	ConcreteWorkBreakdownElementService(55)	–	–
17	ConcreteWorkProductDescriptorService(236)	284	–
18	IterationService (103)	–	< 1
19	LoginService (103)	125	< 2
20	LoginService (83)	164	< 2
21	ParticipantBean (1079)	31	< 2
22	ParticipantBean (681)	121	–
23	ParticipantService (146)	281	✓
24	ParticipantService (119	301	< 2
25	ParticipantService (266)	260	–
26	PhaseService (98)	–	< 2
27	ProcessBean (248)	82	< 2
28	ProcessManagerBean (243)	50	< 2
29	RoleDao (15)	–	–
30	RoleService (15)	150	✓
31	WilosUserBean (717)	23	✓
32	WorkProductsExpTableBean (990)	52	✓
33	WorkProductsExpTableBean (974)	50	✓

Table 1: Comparison of time taken (s) by QBS (128GB RAM, 32 cores) and EqSQL (8GB RAM, 8 cores) for SQL extraction

Experiment 2 (Comparison with [11, 18, 19]): Our techniques can perform optimizations, which existing holistic optimization techniques like batching [11], prefetching of queries [19], and hybrid techniques [18] cannot perform.

Batching is applicable only when there is parameterized iterative query invocation from a loop. If the loop iterates over a query result, batching is able to extract a join query. In addition to the above case, EqSQL can identify more optimization opportunities for pushing selections, projections and aggregations into the database. We examined all code samples from Wilos listed in Table 1, and identified that batching is applicable in 7/33 cases, whereas EqSQL is applicable in 24/33 cases. In 4 cases where both batching and EqSQL are applicable, EqSQL will perform better or same as batching. This is because, in addition to extracting a join query, EqSQL also pushes selections and projections into the database, unlike batching. However, while techniques in this paper are applicable only on cursor loops, batching can handle while loops also, using loop split transformations. It is possible to extend our techniques, to be used in conjunction with loop split transformations.

Prefetching is possible in all cases we examined. However, prefetching by itself does not push any computation from imperative code into SQL, so data transfer is not reduced, although a hybrid technique described in [18] can combine batching and prefetching.

Experiment 3 (Extraction of equivalent SQL for keyword search systems): As mentioned in Section 1, keyword search systems for form interfaces require an SQL query, which would retrieve exactly the data printed by the form interface. The form can contain imperative code along with SQL queries. This was done manually in [6]. In this set of experiments, our goal was to evaluate whether our techniques can automatically extract equivalent SQL queries from servlets. One difference from the earlier cases is that in keyword search systems, ordering of data is not relevant.

We have analyzed the source code of three applications. The fraction of servlets where all queries were extracted by our tool was 17/17 for RuBiS, 16/16 for RuBBoS and 58/79 for *AcadPortal*. The cases where we were not able to derive queries were mainly due to limitations in our implementation such as the presence of operations which are not yet supported.

We have compared the output of our tool with manually extracted queries on the *AcadPortal* application and found that in about 20% of the cases, the manually extracted query was less precise than that extracted automatically by our tool, as the manual queries fetched more data than what is printed by the form interface.

Approaches for batching and prefetching are not suitable for this purpose. QBS can be used, but we are unable to give a comparison as we do not have access to their source code.

7.2 Performance Impact

In this section, we first compare our tool with QBS [4] based on time taken for optimization. We do not have the queries generated by QBS, so we could not directly compare the queries generated by our tool and QBS. However, we manually verified that for each of these cases, (i) queries generated by our system are correct, (ii) whenever a code fragment could entirely be translated into SQL, our system succeeded in doing so.

Experiment 4 (Comparison of optimization time with QBS): As shown in Table 1, for the code samples that we could successfully optimize, our techniques extract equivalent SQL in much less time than those of [4], even when run on a less powerful machine. The significant difference in time is because QBS relies on synthesis technology, which is resource intensive, while our system uses static program analysis, which is much cheaper.

The next three experiments present the impact of our transformations on applications using Hibernate, in terms of execution time and network data transfer.

Experiment 5 (Selection): We use code based on sample #6 from Table 1 which computes the list of unfinished projects, where all tuples are fetched, and filtered inside Java code. Our tool optimizes it to fetch only the required tuples by pushing the predicate into the query. The results, shown in Figure 8, indicate that the transformed code not only runs faster, but also transfers less data compared to the original code. We used 20% selectivity for the query in this experiment. The performance gain achieved is larger/smaller as the selectivity of the query is less/more.

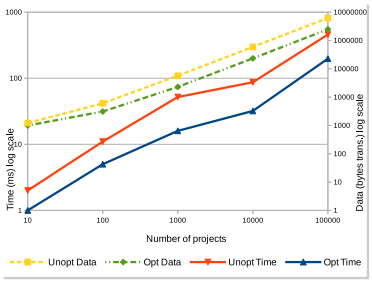


Figure 8: Selection

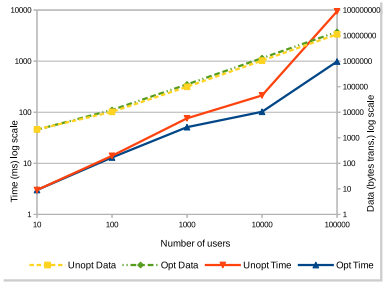


Figure 9: Join

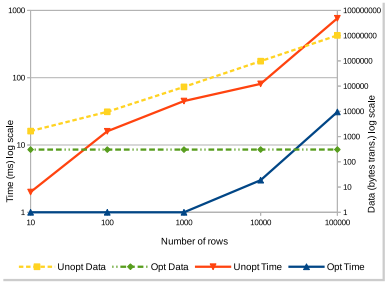


Figure 10: Aggregation

Performance impact of our transformations

Experiment 6 (Join): We consider code based on sample #30 from Table 1 (slightly simplified to be handled by our current implementation). This code computes a join of two tables `WilosUser` and `Role` (ratio of sizes 40:1), and projects the `WilosUser` entity, along with the role name from `Role`. The original code fetches all rows of both tables, and combines them using nested loops in the application, based on a condition. It is rewritten using our transformations, into a join query. The results are shown in Figure 9. The transformed code performs faster than the original code, as the database engine is allowed to choose the best join plan. However, the amount of data transferred is marginally more in the transformed code, because attributes of `Role` get replicated for each row of `WilosUser`.

Experiment 7 (Aggregation): We consider the code sample from Figure 2 which is based on a ranking page generator from Matoso. The results are shown in Figure 10. The data transferred for the optimized query is constant, as only the single result value is transferred in all cases. In contrast, data transfer for the original query increases linearly with increase in table size.

Experiment 8 (Comparison with batching [11] and prefetching [19]): In this experiment, we discuss the improvements due to equivalent SQL extraction over batching and prefetching. We extracted a code sample from the *JobPortal* application where there is opportunity for optimization by all three techniques, namely prefetching, batching and equivalent SQL extraction. This code fetches all relevant applicants for a job based on a search criteria. It then iterates over the results of the above query, and (conditionally) executes multiple scalar queries to fetch relevant information about that particular applicant. The pseudocode for this sample is shown in Figure 12 of Appendix B. The results are shown in Figure 11. In the figure, *Batch* refers to optimizations using techniques described in [11], *Prefetch* refers to techniques in [19]. Though existing techniques do lead to improved performance, they are limited in their applicability, as discussed in Appendix B. EqSQL enhances performance by upto two orders of magnitude compared to the original program, and upto one order of magnitude compared to other optimizations.

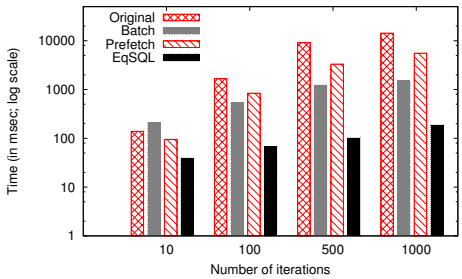


Figure 11: Comparison With Existing Techniques

8. CONCLUSION AND FUTURE WORK

In this paper, we have described novel techniques based on program regions, to translate imperative code to SQL. We presented algorithms to translate the source program into an algebraic/functional intermediate representation (F-IR) that uses `fold` and extended relational algebra to represent cursor loops. Transformation rules on F-IR identify relational operations performed in imperative code, and translate them into equivalent SQL. Our experiments show that techniques in this paper are widely applicable and useful in real world applications, and provide performance improvements that existing approaches cannot provide, on many programs.

Apart from addressing the limitations mentioned in Section 5.4, future work includes extracting equivalent SQL for database update operations performed in imperative code, and deciding in a cost-based manner, whether to rewrite parts of the code into SQL, as sketched in Appendix C.

Acknowledgments

The work of K. Venkatesh Emani is supported by a fellowship from Tata Consultancy Services. The work of Karthik Ramachandra was supported by a fellowship from Microsoft Research, India. We thank Uday Khedker for his valuable guidance, Tarun Jain and Tejas Deshpande for help with implementation, Prasanna Kumar for his inputs, and Kunal Shah for his work which motivated the techniques proposed in this paper.

9. REFERENCES

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2006.

[2] M. Chavan, R. Guravannavar, K. Ramachandra, and S. Sudarshan. Dbridge: A program rewrite tool for set-oriented query execution. In *ICDE*, pages 1284–1287, 2011.

[3] J. Cheney, S. Lindley, and P. Wadler. Query shredding: Efficient relational evaluation of queries over nested multisets. In *SIGMOD*, pages 1027–1038, 2014.

[4] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing database-backed applications with query synthesis. PLDI, pages 3–14, 2013.

[5] S. K. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.*, 22(2):378–415, Mar. 2000.

[6] C. Duda, G. Frey, D. Kossmann, and C. Zhou. Ajaxsearch: crawling, indexing and searching web 2.0 applications. *PVLDB*, 1(2):1440–1443, 2008.

[7] C. A. Galindo-Legaria and M. Joshi. Orthogonal optimization of subqueries and aggregation. In *SIGMOD*, pages 571–581, 2001.

[8] G. Giorgidze, T. Grust, T. Schreiber, and J. Weijers. Haskell boards the ferry. In *Implementation and Application of Functional Languages*, pages 1–18. Springer, 2011.

[9] G. Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[10] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Data Engineering*, pages 209–218. IEEE, 1993.

[11] R. Guravannavar and S. Sudarshan. Rewriting Procedures for Batched Bindings. In *Procs. VLDB*, pages 1107–1123, 2008.

[12] M. S. Hecht and J. D. Ullman. Flow graph reducibility. STOC, pages 238–250, 1972.

[13] Hibernate http://www.hibernate.org.

[14] M.-Y. Iu, E. Cecchet, and W. Zwaenepoel. Jreq: Database queries in imperative languages. In R. Gupta, editor, *Compiler Construction*, volume 6011 of *LNCS*, pages 84–103. Springer, 2010.

[15] MAhjong TOurnament SOftware https://code.google.com/p/matoso/.

[16] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[17] C. Radoi, S. J. Fink, R. Rabbah, and M. Sridharan. Translating imperative code to mapreduce. In *OOPSLA*, pages 909–927. ACM, 2014.

[18] K. Ramachandra, M. Chavan, R. Guravannavar, and S. Sudarshan. Program transformations for asynchronous and batched query submission. *TKDE*, 27(2):531–544, 2015.

[19] K. Ramachandra and S. Sudarshan. Holistic optimization by prefetching query results. In *SIGMOD*, pages 133–144, 2012.

[20] ObjectWeb Consortium. Rice University bulletin board system http://jmob.objectweb.org/rubbos.html.

[21] ObjectWeb Consortium. Rice University bidding system http://rubis.objectweb.org/.

[22] X. Shi, B. Cui, G. Dobbie, and B. C. Ooi. Towards unified ad-hoc data processing. In *SIGMOD*, pages 1263–1274, 2014.

[23] V. Simhadri, K. Ramachandra, A. Chaitanya, R. Guravannavar, and S. Sudarshan. Decorrelation of user defined function invocations in queries. In *ICDE*, pages 532–543, March 2014.

[24] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs, 2006.

[25] Soot: A Java Optimization Framework http://www.sable.mcgill.ca/soot (Oct 2014).

[26] F. Tip. A survey of program slicing techniques. Technical report, 1994.

[27] J. D. Ullman and J. Widom. *A First Course in Database Systems*. Pearson, 2007.

[28] B. Wiedermann, A. Ibrahim, and W. R. Cook. Interprocedural query extraction for transparent persistence. In *OOPSLA*, pages 19–36, 2008.

[29] Wilos Orchestration Software http://www.ohloh.net/p/6390.

[30] M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava. Reverse engineering complex join queries. In *SIGMOD*, pages 809–820, 2013.

APPENDIX

A. PROOF SKETCH FOR LOOP TO FOLD TRANSLATION

We now present a sketch of the proof of correctness for theorem 1. We reuse the terms from Figure 6, without describing them here again.

Theorem 1: Given a cursor loop region R, the value of a variable v after termination of the loop is equivalent to the result of *foldExpr* for v obtained by `loopToFold`(R), when executed on the same input.

Proof Sketch: The proof is given in two parts. Part (a) proves correctness in the case of a single loop, and part (b) proves correctness in the presence of nested loops.

Part (a): Here, we prove that F-IR translation for a single variable in a cursor loop using `fold` is correct. Since `loopToFold` operates on one variable at a time, correctness for multiple variables follows. We use induction on the number of iterations of the loop (i.e., the number of rows in the result set, in order).

The base case is 0 rows (empty result set). For the inductive step, let Q_k denote the top k rows of query Q, v_k denote the value of u after k iterations of the loop, and t_k denote the k'th record of Q. Assume correctness for k iterations. We refer to preconditions P1 and P2 to claim that v_{k+1} depends only on v_k and the current tuple (t_{k+1}). Thus,

$v_{k+1} = e'_{acc}(v_k, t_{k+1})$
$= e'_{acc}(\texttt{fold}[\ e'_{acc}\ v_0\ Q_k\ ], t_{k+1})$
$= \texttt{fold}[\ e'_{acc},\ v_0,\ Q_{k+1}\ ]$ /* *defn of* `fold` */
Hence, proved.

Part (b): The procedure `toFIR` first translates all subregions for a given region into F-IR, before translating the region itself. In the case of a loop, all inner loops, if any, are translated into F-IR before translation is attempted for the loop. Thus, at any point of time, F-IR translation happens only for a single loop, whose correctness was proved in Part (a). Hence, correctness for nested loops follows.

B. MORE TRANSFORMATION RULES

In this section, we present some more transformation rules that we used in our implementation apart from those in Section 5.1, and then discuss some extensions to handle common cases in database applications.

Rule T6 (`fold` with non-id): Consider an F-IR expression `fold`[`f`, x, Q]. In some cases, the initial value passed to `fold` (x) may not be the identity element of the folding function (`f`). This limits the applicability of some of our transformation rules which assume that x must be the identity element for `f`. The following transformation rule allows `fold` to be expressed in terms of the identity element (id) for `f`.

If `f` is associative and $x \neq id$,
then
`fold` [`f`, x, Q] $\equiv$ `f` [x, `fold` [`f`, id, Q]]

Examples of associative functions include +, `max`, `min`, `append` etc.

Before we present Rule T7, we describe the `outer apply` construct. The `outer apply` construct [7], which we denote by Q_1 *OApply* $Q_2(t)$, accepts two arguments: an outer query (Q_1), and an inner query (or expression) which is parameterized on the outer query ($Q_2(t)$). For each row t_1 in Q_1, *OApply* evaluates $Q_2(t_1)$, and for each tuple t_2 in $Q_2(t_1)$, it adds t_1 concatenated with t_2 to the result. However, if $Q_2(t_1)$ is empty, t_1 concatenated with NULLs for attributes from Q_2 is added to the result. We use the `outer apply` syntax of SQL Server in our SQL queries; this is equivalent to the left outer join version of the `lateral` construct in SQL.

We now present Rule T7, which is used to extract a query for a common pattern in database applications, when the data is organized as a star schema. An example is given in Figure 12 in Appendix B.

Rule T7 (Outer Apply):
If $\texttt{f}^{\langle v\rangle,\langle t\rangle} = \texttt{append}[\ v,\ \texttt{g}(\ \pi^s_{L_1}(Q_2(t)), \pi^s_{L_2}(Q_3(t))\ )\ ]$
then `fold`[`f`, [ ], $\tau_{Z_1}(Q_1)$] $\equiv$
$\pi_{\texttt{g}(L_1,L_2)}(\tau_{Z_1}((Q_1\ \ OApply\ Q'_2)$
$OApply\ Q'_3))$

where `append` is the list append operator, π^s represents scalar projection (single row), and $Q_2(t)$, $Q_3(t)$ are parameterized queries. Q'_2 and Q'_3 are obtained from $Q_2(t)$ and $Q_3(t)$ respectively by replacing references to attributes of t with reference to corresponding attributes of Q_1. This rule can also be used for `insert` (set insertion operator), in place of `append`.

Extensions: As we mentioned in Section 5.1, similar to database query optimizers, more transformation rules can easily be added to our system to exploit other opportunities for inferring relational operations performed in imperative code. We now discuss some extensions to our rules presented so far, which can be used to handle some common cases in database applications.

Checking for existence using cursor loops: So far, the focus of our discussion was to extract equivalent SQL from cursor loops that build the value of an aggregate/collection. However, in some cases, a single boolean value is conditionally assigned to a variable (v) inside the cursor loop. For example, a variable v may be initialized to `false`, and the loop may have a statement of the form "if (*pred(t)*) then v=`true`". Such an assignment can be treated as $v = v \vee$ *pred(t)*, which can be handled using our techniques. Similarly, a variable initialized to `true` and set conditionally to `false` can be handled using $v = v \wedge$ *pred(t)*.

A common use of the above constructs is for checking for existence/non-existence of a tuple in a table. Our implementation contains transformation rules to infer `EXISTS` and `NOT EXISTS` queries from the F-IR.

Sometimes, the loop can have an early exit, i.e., it may return/break immediately after a value is assigned once. Currently, we do not handle early exits. However, if the only computation inside the loop is the boolean value assignment, the return/break can potentially be removed, and equivalent SQL can be extracted using our techniques. We omit details.

Dependent Aggregations: In database applications, especially in reporting contexts, it is a common requirement to return a tuple of values from the row containing the aggregated value, along with the value itself. Specifically, if the aggregating function is `max` or `min`, this is an `argmax`/ `argmin` on a column for all rows in the relation. For example, one may want the name of a student who scored the highest marks in a test, along with his/her marks.

However, our techniques, described thus far, disallow conversion of D-IR to F-IR in the above case, because the tuple of row attributes to be returned depends on the aggregated value (causing a loop carried flow dependence, refer Section 4.2). We relax this precondition as follows.

If a variable v is being aggregated in a cursor loop, and another variable w has a loop carried dependence due to v, then the values of v and w after the loop can be represented in F-IR, using a folding function which returns a pair (v', w'). This allows us to obtain an F-IR from D-IR, which can then be transformed to optimized F-IR, to extract optimized SQL.

In general, this fold can then be translated into SQL using user defined aggregates. (Most databases today allow users to define user defined aggregates that can return a tuple.) However, for the special case of `argmax`, we can obtain an equivalent SQL query using any of several techniques such as sub-query, a combination of

```
ResultSet rs = fetchJobApplicants() //Q1
while(rs.next()) {
  String id = rs.getString("applicantId");
  String applnMode = rs.getString("applnMode");

  fetchAndPrintPersonalDetails(id); //Q2
  fetchAndPrintCommittee1Feedback(id); //Q3
  fetchAndPrintCommittee2Feedback(id); //Q4

  if(applnMode = "online")
    fetchAndPrintEducationalQualifs(id); //Q5
}
```

Figure 12: Cursor loop with nested scalar queries

ORDER BY and LIMIT, or using a construct like SQL's RANK if the SQL dialect supports it. We omit details.

Handling Output Ordering: It is not uncommon to find cases in database applications, that avoid intermediate collections by printing values as they are computed, in loops. In such cases, we preprocess the program to replace output statements with appends to a (global) string (which can be treated as an ordered collection), and print its contents at the end of the program. The preprocessed program is then optimized using our techniques.

When all output statements are present in the same level of loop nesting, this is straight forward. We now discuss optimization of database applications in the case where output statements may be distributed across different nesting levels of multiply nested loops. This approach can also be used for collection variables, when there is an ordering requirement on the contents of a collection.

Consider the sample program shown in Figure 12. This code is extracted from an administrative portal in production use at our organization. It fetches a list of job applicants (Q1), and for each applicant, it (conditionally) fetches and prints further information about the applicant using parameterized scalar queries (Q2, Q3, Q4, and Q5). We note that this is a frequent occurrence when data is organized as a star schema.

Although batching and prefetching techniques are applicable to this program, benefit due to batching is limited because of the overhead of creating four parameter tables, while prefetching is unable to chain queries Q1 and Q5, since parameters from Q1 feed into Q5 through the condition `applnMode == "online"`. However, using techniques described in this paper (Rule T7), a single SQL query can be extracted to fetch the required data for this code sample. The query is shown in Figure 13. As all the queries inside the cursor loop of Q1 in Figure 12 are scalar queries, Rule T7 is applicable. The source program is rewritten to refer to corresponding attributes from the extracted query, instead of attributes from the original queries (Q1 to Q5).

If some queries inside the cursor loop can return multiple rows, then combining them using the apply construct can result in cross products of the results of the

```
((((Q1 outer apply Q2 on Q1.applicantId
= Q2.applicantId)
outer apply Q3 on Q1.applicantId=Q3.applicantId)
outer apply Q4 on Q1.applicantId=Q4.applicantId)
outer apply Q5 on Q1.applicantId=Q5.applicantId
            and Q1.applnMode = 'online')
```

Figure 13: Optimized query for data access in Figure 12

sub-queries. This would be very inefficient, and not preserve ordering. In such cases, it is still possible to retrieve the data with proper ordering using techniques borrowed from [3]. Implementation of these techniques is part of future work. We note that batching and prefetching techniques may be applicable to such programs, even if our techniques are not applicable.

C. COST BASED APPLICATION OF TRANSFORMATIONS

We briefly sketch an approach that we are currently exploring for cost-based application of transformations.

Our approach is based on the Volcano/Cascades framework [10] for optimization of algebraic expressions, which is based on equivalence rules. This framework allows the optimizer implementor to specify rules that state the equivalence of two algebraic expressions; examples of such rules include join commutativity and join associativity.

One of the key ideas underlying the framework is an AND-OR DAG representation of the space of alternative expressions. Each OR-node can have multiple children representing alternative ways of computing the same result, while each AND-node represents the root operator of a tree that computes the result. The children of an OR-node can only be AND-nodes, and vice versa.

The framework allows transformations to be applied on an expression, and retains the old and all new expressions in the AND-OR DAG framework; as a result, the order in which transformations are applied is immaterial. Efficient techniques for detecting duplicate derivations of an expression are also a key part of the framework.

Although designed to deal with algebraic representations, the Volcano/Cascades framework can be used with the region-based representation that we use. Each region can be thought of as logically taking in input variables, and returning values for output variables; there are many ways of computing the same result. Thus, a region is mapped to an OR-node (also called equivalence node) in the AND-OR representation. Each way to compute the results in a region can be modeled as an operation node (AND node) in the Volcano/Cascades framework, with the children of the operation node being the child regions.

Similarly, an expression in the F-IR representation represents a specific way of computing a logical result. The logical result is represented by an OR-node; for each logically equivalent way of computing the result, which

may be initially present or generated by transformation rules, the top-level operation is represented as a child operation node of the OR-node. Implementation of this approach is an area of future work.

D. D-IR CONSTRUCTION

In Section 3.3, we gave an outline of D-IR construction for various types of regions. We now describe the algorithms for D-IR construction in detail.

D.1 Simple Statement

Let s be a simple source language statement, with *op* being the operator of the contained source language expression, and n_1, n_2 etc. being its operands. The ee-DAG for s is a node with the equivalent ee-DAG operator for *op* as the root, and equivalent ee-DAGs for n_1, n_2 etc., as children. A ve-Map is created with a single entry, with key as the target variable, and value as a pointer to the ee-DAG root.

D.2 Basic Block

A basic block is treated as a special case of a sequential region (which we describe next) with each statement being a sub-region. Initially, the first two statements are merged to form a sequential region, and the result is merged with the next statement repeatedly, until the entire block results in a single region.

D.3 Sequential Region

Given two regions *r1* and *r2*, with *eeDag1* and *eeDag2* being their corresponding ee-DAGs, *veMap1* and *veMap2* being their corresponding ve-Maps, such that *r1* and *r2* (in order) form a sequential region r, the ee-DAG and ve-Map for r are obtained using the following algorithm.

- For each leaf in *eeDag2* that is a 0 subscripted variable (i.e., initial value), check and if present, replace it with ee-DAG obtained from a lookup in *veMap1* with the variable as key.
- The ee-DAG for r is the single ee-DAG obtained after step 1. In case *eeDag1* and *eeDag2* are disjoint after step 1, we combine them into a single ee-DAG using the NOP operator.
- Create a new map that is a union of entries from *veMap1* and *veMap2*. In case of duplicate keys, the entry from *veMap2* is retained. This map constitutes the ve-Map for r.

D.4 Conditional Region

Consider three regions *rc*, *rt* and *rf* that form a conditional region r, with *rc* containing the condition c, *rt* being the true region, and *rf* being the false region, *eeDag-t* and *eeDag-f* being the ee-DAGs, and *veMap-t* and *veMap-f* being the ve-Maps for *rt* and *rf* respectively, the ee-DAG and ve-Map for r are obtained using the following algorithm.

- Create a new ee-DAG and ve-Map for r.
- For each non local variable v modified inside r, create a conditional evaluation expression with c as the condition, expression for v in *eeDag-t* as its true operand, and expression for v in *eeDag-f* as its false operand (obtained by looking up in respective ve-Maps). Add this node to the ee-DAG of r.
- If there is no entry for v in one of *veMap1* or *veMap2*, then use its value at the beginning of the region (v_0).
- After creating the conditional evaluation expression, make an entry in the ve-Map of r with v as the key and a pointer to the conditional evaluation expression as its value.

D.5 Loop Region

The ee-DAG for a loop region can be created as follows:

- Create a *Loop* node with the looping query and the loop body as its two children.
- For each variable v that is a key in the ve-Map of the loop body and is also live at the program point immediately after the loop, add an entry *(v, ND)* to the ve-Map of the loop region.

Here, *ND* stands for not yet determined. Uses of v after the loop region will point to *ND* temporarily, until an expression for v over all iterations of the loop is obtained.

D.6 Functions

We classify functions into the following categories.

Library functions: Library functions that have an equivalent ee-DAG operator are represented using that operator. If there is so such operator, then D-IR construction fails for the target variable (v) of that statement, and target variables of statements which read the value of v after this assignment.

User defined functions: For a user defined function, we use the following approach:

1. Create the IR separately for the function. Let e denote the ee-DAG expression for the return value of the function. If an unknown statement is encountered inside the function, fail. Formal parameters are region inputs, so their initial values are denoted by appending a 0 subscript to the variable name.
2. If the above step succeeds, merge the function with its preceding region at the caller location, by considering them to form a sequential region. We update the value of the target variable (which is assigned the return value of the function, if any) in the ve-Map of the caller region to point to e. Formal parameters are mapped to actual parameters and resolved during the merge.

User defined procedures: User defined procedures, in addition to returning a value, can also modify the input parameters such that the change in their values is reflected at the caller location. They can be handled similar to functions, with the following additional step.

3. Remove all entries for local variables in the ve-Map for the procedure. Now, merge the procedure with its preceding region at the caller location, by considering them to form a sequential region (as described in Section D.3).