COBRA: A Framework for Cost-Based Rewriting of Database Applications

K. Venkatesh Emani
Dept. of Computer Science and Engineering
IIT Bombay
venkateshek@cse.iitb.ac.in

S. Sudarshan
Dept. of Computer Science and Engineering
IIT Bombay
sudarsha@cse.iitb.ac.in

***Abstract*—Database applications are typically written using a mixture of imperative languages and declarative frameworks for data processing. Data processing logic gets distributed across the declarative and imperative parts of a program. Often, there is more than one way to implement the same program, whose efficiency may depend on a number of parameters. In this paper, we propose a framework that automatically generates all equivalent alternatives to a given program using a given set of program transformations, and chooses the least cost alternative. We use the concept of program regions as an algebraic abstraction of a program and extend the Volcano/Cascades framework for optimization of algebraic expressions, to optimize programs. We illustrate the use of our framework for optimizing database applications. We show through experimental results, that our framework has wide applicability in real-world applications and provides significant performance benefits.**

***Index Terms*—query optimization; database applications; program regions; cost-based optimization**

I. INTRODUCTION

Database applications are typically written using a mixture of imperative languages such as Java for business logic, and declarative frameworks for data processing. Examples of such frameworks include SQL (JDBC) with Java, object-relational mappers (ORMs), large-scale data processing frameworks such as Apache Spark, and Python data science libraries (example: pandas), among others. These frameworks provide high-level operators/library functions for expressing common data processing operations and contain efficient implementations of these functions.

However, in many applications, data processing operations are often (partially) implemented in imperative code. The reasons for this include modularity, limited framework expertise of the developer, need for custom operations that cannot be expressed in the declarative framework, etc. Consequently, data processing is distributed across the imperative and declarative parts of the application. Often, there is more than one way to implement the same program, and the best approach may be chosen depending on a number of parameters.

This raises an interesting question for an optimizing compiler for data processing applications. Given an application program, is it possible to generate semantically equivalent alternatives to the program using program transformations, and choose the program with the least cost depending on the context? In this paper, we propose the COBRA[1] framework to achieve this, as illustrated in Fig. 1.

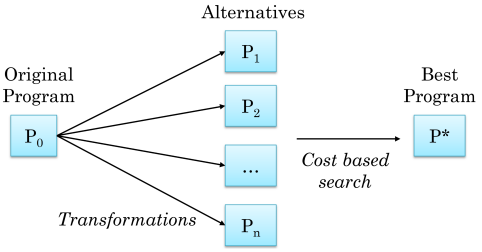


Fig. 1: COBRA Illustration

There has been work on rewriting data processing programs for improved performance using program transformations [1], [2], [3], [4], [5]. However, existing techniques fail to consider all possible alternatives for cost-based rewriting. They either apply the proposed transformations in a specific order [2] or carefully craft the transformation rules so that the rule set is confluent and terminating [4]. This is not a viable solution for all rule sets, especially as the number/complexity of rules increases. A brute force solution is to keep applying all possible transformations as long as any one of them is applicable; however, this may cause the transformation process to never terminate, in case of cyclic transformation rules. For example, in their work on translating imperative code to map-reduce, Radoi et al. [5] state that their transformation rules are neither confluent nor terminating, and use a heuristic driven by a cost function to guide the search for possible rewrites. However, such an approach, in general, has the disadvantage of missing out on useful rewrites that are not considered by the heuristic.

A similar problem has been solved for the purpose of query optimization in databases. Graefe et al. proposed the Volcano/Cascades framework [6], [7], which uses an AND-OR DAG representation (details in Section III) to enumerate all alternative rewrites for a given SQL query (relational algebra expression) generated using transformation rules, and to choose the best query (plan) by searching through the space of possible rewrites. Although designed for query optimiza-

[1]An acronym formed from COst Based Rewriting of (database) Applications.

tion, the Volcano/Cascades framework can be used with any algebra.

Such a framework can be used with transformations on expressions in imperative programs, as described in [8]. Examples of such transformations amy include algebraic simplifications and many peephole optimizations such as constant folding, strength reduction, etc. However, transformations proposed for optimizing data processing applications typically involve rewriting conditional statements, loops, functions, or even the entire program. Such transformations involving larger program units are not amenable to direct integration into an algebraic framework like Volcano/Cascades.

In this paper, we identify that *program regions* [9], which we used for transformations in our previous work [4], provide a natural abstraction for dividing an imperative program into parts, which can then be optimized individually and collectively using an extension of the Volcano/Cascades framework. Program regions are structured fragments of a program such as straight line code, if-else, loops, functions, etc. (details in Section III). Our framework, COBRA, represents a program as an AND-OR DAG using program regions. Program transformations add alternatives to this AND-OR DAG. COBRA can be used for cost-based transformations in any program with well-defined program regions. However, in this paper, we restrict our attention to the use of COBRA for optimizing database applications.

Our contributions in this paper are as follows:

- We describe the AND-OR DAG representation of an imperative program with regions and discuss how the alternatives generated using program transformations are represented using the AND-OR DAG (Section IV).
- We illustrate the use of our framework for optimizing database applications. To this end, we discuss an intermediate representation for transformations in database applications (Section V) building on our earlier work [4].
- We present a cost model (Section VI) to estimate the cost of database application programs, with a focus on the cost of query execution statements and loops over query results.
- We built the COBRA optimizer by incorporating our techniques into a system that implements the Volcano/Cascades framework. We present an experimental evaluation (Section VIII) of COBRA on real-world application programs, to show the applicability of our techniques and their impact on application performance.

We present a motivating example in Section II and discuss the necessary background in Section III. We discuss related work in Section VII and conclude in Section IX.

II. MOTIVATING EXAMPLE

The COBRA framework can be used for optimizing programs using a variety of data access methods such as JDBC, web services, object-relational mappers (ORM) etc. In this section we discuss an example program that uses the Hibernate ORM [10], to motivate the need for COBRA.

```
@Entity @Table(name="orders")
class Order{
  @Column(name="o_id");
  int o_id;
  @ManyToOne(targetEntity = Customer.class)
  @JoinColumn(name="customer_sk")
  Customer customer;
  ...
}
```

Fig. 2: Hibernate object-relation mapping specification

Object-relational mapping frameworks enable access to the database using the same language as the application [1] without writing explicit SQL queries. The framework automatically generates relevant queries from object accesses and translates query results into objects, based on a specified mapping between database tables and application classes.

For example consider Fig. 2, which shows a schema definition in the Hibernate ORM. The class *Order* is mapped to the database table *orders*. When *Order* objects are retrieved, the framework implicitly creates a query on *orders* and populates the attributes of *Order*. The relationship from table *orders* to table *customers* (mapped by class *Customer*) is expressed as an attribute of *Order*.

Objects (rows) retrieved from the database are cached upon first access using their id (primary key). Thereafter, these objects can be accessed inside the application without having to query the database again. Hibernate supports lazy loading, i.e., fetching an attribute of an object only when the attribute is accessed; this facilitates fetching information from a related table (such as *customer* in *Order*) only when needed. Most ORMs also allow users to express complex queries using SQL or object-based query languages. ORMs are widely used in OLTP applications [1], and their use in reporting applications is not uncommon [11]. Inefficiencies due to the usage of ORM frameworks are also well known [12], and have been addressed by earlier optimization techniques [1], [4] (refer related work, Section VII).

Fig. 3a shows a sample program that uses the Hibernate ORM and processes a list of orders along with customer related information. The program uses an ORM API (*loadAll*) to fetch all *Orders* objects and then processes each order inside a loop. However, for each order, the framework generates a separate query to fetch the related *customer* information, which resides in another table. This causes a lot of network round trips, leading to poor performance. This issue is known as the *N+1 select problem* in ORMs [12].

To avoid this problem, a join query is usually suggested to fetch the required data, while restricting the number of queries to one. This is shown in program P_1 in Fig. 3b[2]. P_1 follows the general rule of thumb where data processing is pushed into the database as much as possible, thus allowing the database to

[2]We use a pseudo-function *executeQuery* that takes a query, executes it and returns the results as a collection of objects. Also, variable types have not been displayed for ease of presentation. Our implementation uses the actual source code.

```
1  processOrders(result) {
2    result = {}; //empty collection
3    for(o : loadAll(Order.class)){
4        cust = o.customer; // requires a separate query
5        val = myFunc(o.o_id, cust.c_birth_year, ...);
6        result.add(val);
7    }
8  }
```

(a) P_0: Program using Hibernate ORM

```
1  processOrders(result) {
2    result = {};
3    joinRes = executeQuery("select * from orders o join
       customer c on o.o_customer_sk = c.c_customer_sk");
4    for(r : joinRes){
5        val = myFunc(r.o_id, r.c_birth_year, ...);
6        result.add(val);
7    }
8  }
```

(b) P_1: P_0 rewritten to use Hibernate SQL query API

```
1  processOrders(result) {
2    result = {};
3    customers = loadAll(Customer.class);
4    Utils.cacheByColumn(customers,'c_customer_sk');
                                  // refer footnote 3
5    for(o : loadAll(Orders.class);){
6        cust = Utils.lookupCache(o.o_customer_sk);
7        val = myFunc(o.o_id, cust.c_birth_year, ...);
8        result.add(val);
9    }
10 }
```

(c) P_2: P_0 rewritten to use prefetching

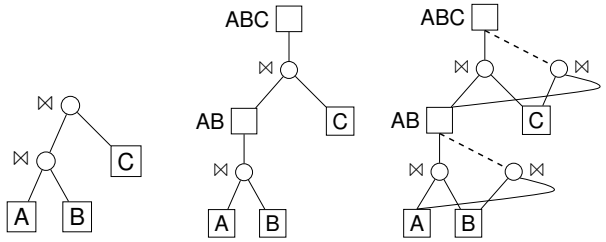
Fig. 3: Alternative implementations of the same program

use clever execution plans to minimize query execution time.

The join query shown in P_1 may lead to duplication of *customer* rows in the join result (as each customer typically places multiple orders). For small data sizes or a few rows when the *orders* fetched are filtered using a selection, this duplication may not have a significant impact. However, for higher cardinalities, the join result may be large and transferring the results over a slow remote network from the database to the application may incur significant latency. In such cases, an equivalent program P_2 shown in Fig. 3c[3] may be faster, provided the tables *orders* and *customers* fit in the application server memory. This is because P_2 fetches individual tables and performs a join at the application, thus avoiding transfer of a large amount of data over the network.

Existing approaches for rewriting ORM applications with SQL, such as [4], [1] apply transformations with the sole aim of pushing data processing to the database; thus, they transform P_0 to P_1. Other transformations, such as prefetching

[3]The pseudo-function *cacheByColumn* caches a query result collection based on the value of a given column as key and *lookupCache* fetches a value from the cache using a given key. The cache may be in the form a simple hashmap or use caching frameworks such as Memcache or EhCache, which are used by many applications for client-side query result caching. ORM frameworks such as Hibernate provide caching implicitly.



(a) Initial Query (b) DAG representation of query (c) Expanded DAG after applying commutativity

Fig. 4: Representing alternative query rewrites using the AND-OR DAG

query results [13] may be used to transform P_0 to P_2. However, neither P_1 nor P_2 is the best choice in all situations. Using COBRA, all alternatives such as P_1, P_2, and others can be generated using program transformations proposed earlier [1], [4], [13], [3], and the best program can be chosen in a cost-based manner.

III. BACKGROUND

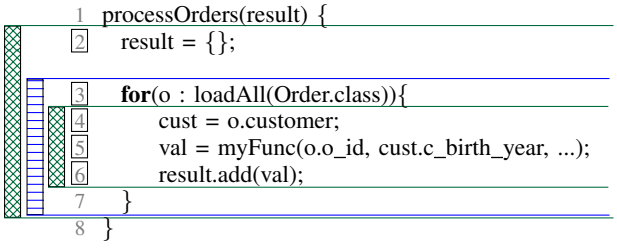
In this section, we give a background of (a) the AND-OR DAG representation for cost-based query optimization in the Volcano/Cascades framework, and (b) program regions.

A. Volcano/Cascades AND-OR DAG

Our discussion of AND-OR DAGs is based on [14]. An AND-OR DAG is a directed acyclic graph where each node in the graph is classified as one of two types: an AND node, or an OR node. The children of an OR-node can only be AND-nodes, and vice versa. In the case of queries (relational algebra expressions), AND nodes represent operators, and OR nodes represent relations. For example, consider the join query $(A \bowtie B) \bowtie C$, which is shown as a tree in Fig. 4a. The AND-OR DAG representation for this query is shown in Fig. 4b.

The Volcano framework for optimization of algebraic expressions is based on equivalence rules. This framework allows the optimizer implementor to specify transformation rules that state the equivalence of two algebraic expressions; examples of such rules include join commutativity ($A \bowtie B \leftrightarrow B \bowtie A$) and join associativity ($(A \bowtie B) \bowtie C \leftrightarrow A \bowtie (B \bowtie C)$), in the case of query optimization. Transformation rules are applied on an expression; while new expressions are added, the old ones are retained in the AND-OR DAG.

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. For the query $(A \bowtie B) \bowtie C$, the AND-OR DAG after applying commutativity is shown in Fig. 4c. The alternatives added are shown using a dotted line connecting the OR node to the root operator of the new expression. Thus, we obtain the following alternatives for the root OR node: $(A \bowtie B) \bowtie C$, $(B \bowtie A) \bowtie C$, $C \bowtie (A \bowtie B)$, and $C \bowtie (B \bowtie A)$. Note that repeated application of commutativity may lead back to the original expression, thus causing cycles in the transformation



Regions naming convention: $P_i.T_{m-n}$ denotes a region of type T in program P_i that starts at line m and ends at line n.

☐ Basic block (B) – $P_0.B_2$, $P_0.B_3$, $P_0.B_4$, $P_0.B_5$, $P_0.B_6$
☒ Sequential region (S) – $P_0.S_{4-6}$, $P_0.S_{2-7}$
▤ Loop region (L) – $P_0.L_{3-7}$

Fig. 5: Program regions for program P_0 from Fig. 3a

process. The Volcano/Cascades framework has efficient techniques for identifying duplicates, so the transformation process will terminate even in the presence of such cycles.

Each operator in the DAG may be implemented using one of a few alternatives. For example, a join operator may be implemented using a hash join, indexed nested loops join, or a merge join. This adds further alternatives to the AND-OR DAG (not shown in Fig. 4). The cost of any node in the AND-OR DAG is calculated using the cost of child nodes, as shown in the table below.

Node type	Cost formula
OR node	Minimum of cost of each child (base case: single relation)
AND node	Cost of operator + Sum of costs of children

The plan corresponding to the least cost at the root node of the AND-OR DAG is the optimized plan.

In the case of query optimization, the cost assigned to a particular node depends on factors such as the number of rows in the relation, the type of the operator and its implementation, presence of indexes etc. We skip further details of costing for query optimization and refer the reader to [6], [7].

B. Program regions

A region is any structured fragment in a program with a single entry and single exit [15]. Examples of regions include a single statement (*basic block region*), if-else (*conditional region*), loop (*loop region*), etc. A sequence of two or more regions is called a *sequential region*[4]. Regions can contain other regions, so they present a hierarchical view of the program. The contained region is called a *sub-region* and the containing region is called the *parent region*. The outermost region represents the entire program.

For example, consider Fig. 5, which replicates the program P_0 from Fig. 3a with program regions shown alongside the code (note the naming convention for regions). The outermost region in Fig. 5 is a sequential region $P_0.S_{2-7}$, which consists of a basic block $P_0.B_2$ followed by a loop region $P_0.L_{3-7}$. The loop region, in turn, is composed of a basic block $P_0.B_3$ and a sequential region $P_0.S_{4-6}$, and so on.

Regions can be built from the control flow graph (CFG) using rules described in [9]. We use this approach in our implementation. Alternatively, it is possible to use an abstract syntax tree of code written in a structured programming language to identify program regions. Exceptions may violate the normal control flow in a region. Currently, our techniques do not preserve exception behavior in the program; handling this is part of future work.

IV. AND-OR DAG Representation of Programs

The Volcano/Cascades framework is well suited for optimizing algebraic expressions, which combine a set of input values using operators to produce an output value. The availability of sub-expressions (parts) of an expression is key to Volcano/Cascades, as alternatives for an expression are generated by combining alternatives for sub-expressions (OR nodes) using operators (AND nodes). However, adapting an algebraic framework such as Volcano/Cascades for optimizing imperative programs is not straightforward. Apart from computing expressions, imperative programs can modify the program stack/heap and contain operations that have side effects (such as writing to a console). Further, real-world programs contain complex control and data flow (due to branching, loops, exceptions etc.).

In this section, we argue that program regions provide a natural abstraction for parts of an imperative program. We then discuss the representation of program alternatives using an AND-OR DAG that we call the Region DAG.

A. Region as a State Transition

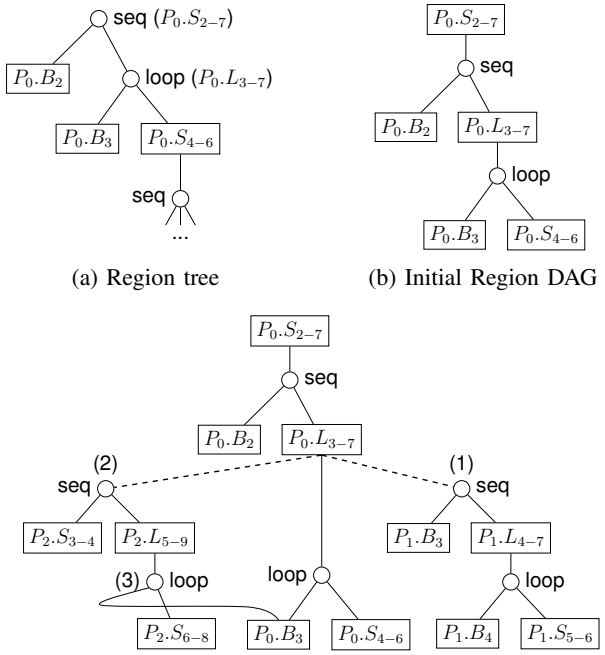
An imperative program can be considered as a specification for a transition from one state to another. For example, the function *processOrders* from program P_0 (Fig. 3a) specifies the following transition: *by the end* of *processOrders*, the join of *orders* and *customers* is computed, *myFunc* is applied on each tuple in the join result and stored in the collection *result*. Alternative implementations of the program (such as P_1 and P_2 from Fig. 3) are alternative ways to perform the same transition.

The same argument can be extended to regions. Consider the loop body from program P_0 (lines 4 to 6), which is a sequential region. The transition specified by this region is: by the *end of the region*, the contents of the collection *result* at the *beginning of the region* are appended with another element obtained by processing the current tuple. The loop body from program P_2 (lines 6 to 8) performs the same computation, however, instead of fetching customer information using a separate query as in P_0, P_2 fetches it from the cache.

We now formally define a program region as a transition, as follows.

$$R : X_0 \rightarrow X_1 \quad (1)$$

[4]Some approaches consider a basic block region as a sequence of statements. In this paper, we consider each statement as a basic block and treat a sequence of statements as a sequential region consisting of basic blocks. In our implementation, we use an intermediate representation of bytecode [16], where each statement is represented using a three-address code [17].



(a) Region tree (b) Initial Region DAG

(c) Expanded Region DAG

Fig. 6: Representing alternative programs using the Region DAG

where R is a region, X_0 is a state at the beginning of R and X_1 is a state at the end of R. We call X_0 an *input state*, and X_1 an *output state*. Since the entire program is also a region, the same definition extends to a program as well.

Our framework is agnostic to the definition of a *state*. For example, in our discussion above, we used the values of program variables (such as *result*) to represent a state. If an application writes to the console, the contents of the console could be included in the definition of the state. In general, other definitions may be considered depending on the program transformations used.

For a single statement (basic block), the transition from the input state X_0 to the output state X_1 involves only the states X_0 and X_1. For regions that may contain other regions, the transition may involve multiple intermediate states: $(X_0 \rightarrow X_{a1} \rightarrow \ldots \rightarrow X_{an} \rightarrow X_1)$ where $X_{a1} \ldots X_{an}$ are the results of transitions in sub-regions. The output state of one sub-region feeds as the input state to another sub-region according to the control flow in the program.

Our definition of a program region as a transition allows regions to be identified as parts of a program performing local computations that together combine to form the entire program, similar to sub-expressions in an algebraic expression. In this paper, we use the term "computation in a region R" to refer to the transition from an input state to an output state specified by a region R.

B. Region AND-OR DAG

Region AND-OR DAG, or simply Region DAG, is an AND-OR DAG that can represent the various alternative, but equivalent programs. Given a program with regions, the program and its alternatives can be represented using the Region DAG as follows.

Step 1 – Region tree: Firstly, we identify regions in the program, as described in Section III-B. The hierarchy of regions in a program can be represented as a tree, which we call the *region tree*. The region tree for the regions in Fig. 5 is shown in Fig. 6a.

The leaves of a region tree are basic block regions. Intermediate nodes are operators that denote the type of control flow between sub-regions for combining them to form the parent region. A sequential region is formed using the *seq* operator, a conditional region is formed using the *cond* operator, a loop region using the *loop* operator, and so on. Child nodes are ordered left to right according to the starting line of the corresponding region in the program. In Fig. 6a, we mention the label of the parent region in parentheses along with the operator. The region tree in COBRA is analogous to the query expression tree in Volcano/Cascades (Fig. 4a).

Step 2 – Initial Region DAG: The next step is to translate the region tree into an AND-OR DAG, which we call the *initial Region DAG*. The initial Region DAG for the region tree from Fig. 6a is shown in Fig. 6b. Operator nodes in the region tree are represented as AND nodes, and leaf nodes and intermediate results are represented using OR nodes. The initial Region DAG is analogous to the DAG representation of a query in Volcano/Cascades (Fig. 4b).

An OR node in the Region DAG represents all alternative ways to perform the computation in a particular region. An AND node represents operators to combine sub-regions into the parent region. The initial Region DAG contains a single alternative for each region, which is the original program. For example, Fig. 6b represents the following alternative for the region $P_0.S_{2-7}$: perform the computation in the basic block $P_0.B_2$ and then the loop $P_0.L_{3-7}$, sequentially. Similarly, the loop region has a single alternative. Other alternatives may be generated by program transformations.

Step 3 – Program transformations: Program transformations rewrite a program/region to perform the same computation in different ways. In our work, we assume that we are provided with transformations that preserve the equivalence of the original and rewritten programs on any valid input state. COBRA then represents these alternative programs efficiently using Region DAG for cost-based rewriting. Our framework does not infer equivalence of programs or of transformations. It is up to the transformation writer to verify the correctness of transformations. In this paper, we use the transformations from [4], [13], with some extensions. We discuss them in Section V.

In a Region DAG, the rewritten program/region is represented as an alternative under the OR node for that particular region. This may create new nodes in the Region DAG. If a sub-region in the rewritten program already exists in the Region DAG, it is reused (leveraging techniques

in Volcano/Cascades for detecting duplicates and merging nodes). We call the Region DAG after adding alternatives from program transformations as the *expanded Region DAG*, analogous to the expanded query DAG in Volcano/Cascades (refer Fig. 4c).

For example, program transformations such as SQL translation [4] and prefetching [13] identify iterative query invocation inside the loop region in P_0, and rewrite the loop as shown in P_1 and P_2 respectively (refer Fig. 3). They are represented in the Region DAG as shown in Fig. 6c, which shows three alternatives to perform the computation in the loop region $P_0.L_{3-7}$. The newly added alternatives (nodes labeled 1 and 2) are both sequential regions containing a loop region within and achieve the same result as the original loop region. The loop operator from P_2 (node labeled 3) shares a basic block subregion ($P_0.B_3$) with the loop region from P_0. The loop headers $P_2.B_5$ and $P_0.B_3$ are the same region and the latter already exists in the Region DAG, so it is reused. In summary, there are three alternatives for the root node $P_0.S_{2-7}$, corresponding to the programs P_0, P_1, and P_2. Note that the AND-OR DAG structure allows the node $P_0.B_2$ to be represented only once, although it is part of all three programs corresponding to alternatives for $P_0.L_{3-7}$.

Representing alternative programs in a Region DAG is not dependent on an intermediate representation or the program transformations used. Given a program/region and its rewritten version, COBRA can represent both the original and transformed programs using the Region DAG. This is a key improvement of our representation over Peggy [8]. Peggy aims to represent multiple optimized versions of a program, for the purpose of eliminating the need for ordering compiler optimizations. Representation of programs in Peggy is tied to a specific intermediate representation (IR), which may be provided by the user. Program transformations must be expressed in this IR. COBRA, on the other hand, does not necessitate the use of an IR, and the transformation process can be unknown to the framework. We compare our work with Peggy further in Section VII.

Nevertheless, COBRA supports representing programs using an IR and expressing transformations on the IR. We discuss one such IR for database applications next, in Section V. In fact, since the original program is represented intact in the Region DAG, it is possible to use multiple IRs simultaneously, each of which may target a specific set of transformations.

Program regions are essential to representing alternatives using the Region DAG. Limitations in the construction of program regions (discussed in Section III-B) hinder the applicability of COBRA. Although it is possible to use COBRA to optimize some parts of a program without well-formed regions, this is currently not handled by our framework. We refer the reader to [18] for details.

V. TRANSFORMATIONS USING IR

In our earlier work [4], we proposed a DAG-based intermediate representation named F-IR (*fold intermediate representation*) for imperative code that may also contain database queries. F-IR has been used to express program transformations for rewriting database applications by pushing relational operations such as selections, projections, joins, and aggregations that are implemented in imperative code to the database using SQL. In this section, we first present a recap of F-IR from our previous work [4]. Later, we describe extensions to F-IR to overcome some of the limitations from [4]. We then discuss the integration of F-IR into COBRA.

```
1  mySum(){
2    sum = 0;
3    cSum = new Map(); //creates a new empty map
4    for(t : executeQuery("select month, sale_amt
                           from sales order by month")){
5      sum = sum + t.sale_amt;
6      cSum.put(month, sum);
7    }
8    print(sum);
9    print(cSum);
10 }
```

Fig. 7: Program M_0: Aggregations inside a loop

A. F-IR Recap

F-IR is based on regions. Variables in a region are represented using expressions only in terms of constants and values available at the beginning of the region; any intermediate assignments are resolved. F-IR contains operators for representing imperative language operations, as well as relational algebraic operators for representing database queries. Specifically, F-IR uses the *fold* operator (borrowed from functional programming) to algebraically represent loops over collections/query results (which are called *cursor loops*).

For example, consider the program shown in Fig. 7, which computes two aggregates, sum and cumulative sum, using a loop over query results. The value of the variable *sum* over the loop region is represented using *fold* as follows:

$$fold(<sum> + Q.sale\_amt, 0, Q)$$

The first argument to *fold* is the aggregation function. Angular brackets $<$ and $>$ denote that the value of *sum* in the aggregation function is parametric and is updated in each iteration. The second argument is the initial value of the aggregate (*sum*) before the loop, in this case, 0 (this feeds as the value of $<sum>$ in the first iteration). The third argument is the query Q over which the loop iterates, in this case: *select month, sale_amt from sales order by month*. We use the notation $Q.x$ to refer to column x of a tuple in Q. Transformations in [4] identify the 'fold with plus' pattern and infer an SQL query for the variable *sum*, as follows:

sum = executeQuery("**select sum**(sale_amt) **from** sales");

The function *fold* is similar to *reduce* in the map-reduce terminology and the two functions are referred to synonymously in some contexts. However, there are important differences [19] that allow *fold* to represent loops on ordered collections that cannot be represented by *reduce*. For a formal

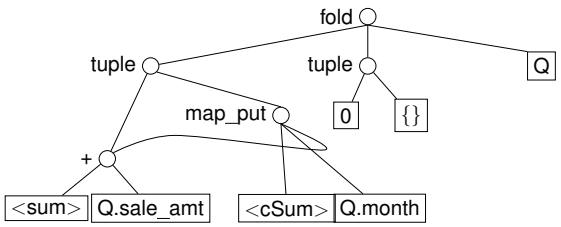


Fig. 8: F-IR representation for the loop in Fig. 7
Q: select month, sale_amt from sales order by month

discussion on *fold*, refer [4].

Not all loops can be represented algebraically. We identified in [4] the set of preconditions (specified as constraints on inter statement data dependencies) to be satisfied by a cursor loop to represent it using F-IR. However, the preconditions in [4] are restrictive as they allow only a single aggregation in a loop to be represented using *fold*. We now discuss this limitation and its impact in the context of cost-based transformations. We extend F-IR with new operators to overcome the limitation.

B. Extensions to F-IR

Consider again the program shown in Fig. 7. The variable *cSum* cannot be represented in F-IR using techniques from [4] due to dependent aggregations: i.e., multiple aggregations in a loop, where one aggregate value is dependent on another. In Fig. 7, the variable *cSum* is dependent on *sum*.

In our previous work [4], the result of *fold* operator is a single scalar/collection value. When multiple aggregations are present in a loop, we considered separately the part (slice) of the loop computing each aggregation and translated it to SQL separately, as our goal in [4] was to translate parts of a program to SQL where possible. For dependent aggregations (such as *cSum* in Fig. 7), extracting such a slice is not possible. Thus, the loop cannot be represented as a *fold* expression using techniques from [4].

An intermediate representation of dependent aggregations in loops is necessary for a cost-based decision of transformations. For example, in Fig. 7, techniques from [4] would extract an SQL query for *sum* (as explained in Section V-A) and leave the computation of *cSum* inside the loop intact. Such a rewrite would result in the following program:

```
for(t : executeQuery("select ... from sales order by month")){
  sum = sum + t.sale_amt;
  cSum.put(month, sum);
}
sum = executeQuery("select sum(sale_amt) from sales);
```

However, this transformation degrades program performance, as a new query execution statement (shown in italics) is added to the program resulting in an extra network round trip. Thus, it is necessary in this case for the entire loop to be represented in F-IR for a cost-based decision.

In this paper, we address this limitation by extending the *fold* operator in F-IR to return a tuple of expressions. To facilitate this, we introduce two new operators, namely *tuple* and *project*.

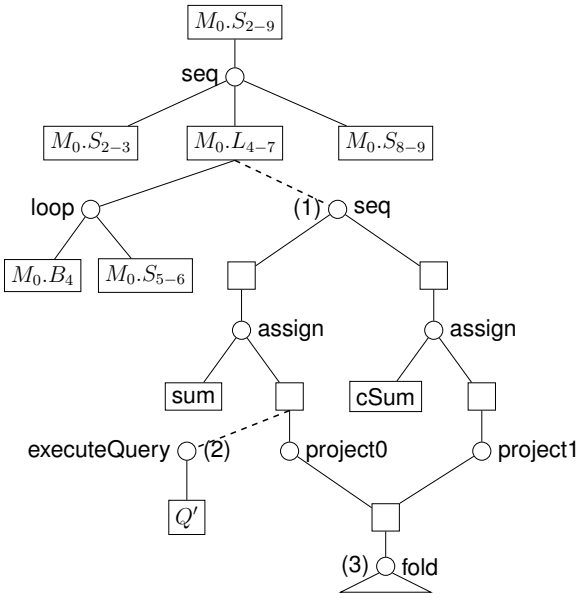


Fig. 9: Region DAG for Fig. 7 after transforming to F-IR (Q': *select sum(sale_amt) from sales*. The *fold* expression (node 3) is as shown in Fig. 8.)

tuple: The *tuple* operator simply represents a tuple of expressions. It has $n>1$ children, each of which is an expression in F-IR. The expressions may have common sub-expressions, which are shared. The output of a *tuple* operator is the n-tuple of outputs of each of its children.

project: Intuitively, the *project* operator performs the reverse operation of *tuple*. It takes as input a *tuple* expression and an index i, and projects the i'th individual expression from *tuple*. In this paper, we represent the index i along with the *project* operator. For example, *project0* projects the first expression from its child *tuple*.

Coupled with *fold*, the operators *tuple* and *project* allow algebraic representation of cursor loops that may have aggregations dependent on one another by removing precondition P2 from [4]. The F-IR construction algorithm with modified preconditions is formally discussed in [18]. Fig. 8 shows the F-IR representation for the loop from Fig. 7 using *fold*. The aggregation function is a tuple of expressions; one for each aggregated variable (*sum* and *cSum*). Similarly, the initial value passed to *fold* is a tuple that combines the initial values (0 and the empty map respectively) for the two aggregates. Q denotes the query from Fig. 7. The result of *fold* is a *tuple*. Subsequently, this F-IR expression is added to the Region DAG for cost-based transformations. We discuss this next.

C. Integration into Region DAG

As we mentioned earlier in Section V, F-IR is based on regions, and F-IR expressions represent values of program variables at the end of a region in terms of values available at the beginning of a region. Thus, an F-IR expression also specifies a transition from an input state to an output state in a region, where the input and output states consist of values

Rule	Definition	Description
T1	$fold(insert,\ \{\},\ Q) = Q$	Fold removal (*insert*: set insertion function)
T2	$fold(?(pred,\ g),\ id,\ Q) \equiv fold(g,\ id, \sigma_{pred}(Q))$	Predicate push into query (*pred*: predicate; *g*: some function; ?: conditional execution (if) operator)
T3	$fold(g(v,\ h(Q.A)),\ id,\ Q) \equiv fold(g,\ id, \pi_{h(A)}(Q))$	Push scalar functions into query (*g*,*h*: functions; *A*: column in Q)
T4	$fold(fold(insert,\ id,\ \sigma_{pred}(Q_2)),\ \{\},\ Q_1) \equiv Q_1 \bowtie_{pred} Q_2$	Join identification (*pred*: a predicate; *insert*: set insertion function)
T5	$fold(op,\ id,\ \pi_A(Q)) \equiv \gamma_{op\_agg(A)}(Q)$	Aggregation (*op*: a binary operation like +, scalar *max*; *op_agg*: corresponding relational aggregation operation like *sum*, *max*)
N1	$fold(f(v, executeQuery(\sigma_{R.A=Q.B}(R))), id, Q) \equiv seq(prefetch(R, A), fold(f(v, lookup(Q.B)), id, Q))$	Prefetching (*prefetch*: fetch query result and cache by column locally. *cacheByColumn, lookup*: Refer footnote 3).
N2	$fold(g,\ id, \sigma_{pred}(Q)) \equiv fold(?(pred, g), id, Q)$	Reverse of T2

Fig. 10: F-IR Transformation Rules (T1 to T5 are from [4])

of all program variables that are live at the beginning and at the end of the region, respectively.

We model the construction of an F-IR expression for a region as a program transformation that takes a region as input and gives the equivalent F-IR expression as output. If the preconditions for F-IR representation (Section V-B) are satisfied, the F-IR expression is constructed and added as an alternative to the corresponding region. If the preconditions fail, no F-IR expressions are added, but other program transformations can still be applied on the Region DAG.

Fig. 9 shows the Region DAG for program M_0 from Fig. 7. The program consists of a sequential region ($M_0.S_{2-9}$) containing a loop region within ($M_0.L_{3-6}$). The F-IR expression from Fig. 8 is used to add an alternative (node 1) to the loop region. Using the *fold* expression for the loop, we first extract the individual variable values using *project*, assign them to the appropriate variables, combine the assignments using a *seq* operator, and add the alternative to the OR node corresponding to the loop.

D. Transformations

Transformations on F-IR expressions add further alternatives to the Region DAG. In our earlier work [4], we proposed F-IR transformations with the aim of translating imperative code into SQL. These transformations are summarized in Fig. 10 (T1 to T5)[5]. (There are other transformation rules in [4], all of which are included in our implementation.) Prefetching is widely used in enterprise settings to mitigate the cost of multiple invocations of the same query. To enable prefetching, in this paper, we propose new transformations N1 and N2 (Fig. 10). Rule N1 transforms iterative lookup queries inside a loop into a prefetch[6] followed by local cache lookups. Rule N2 transforms a selection query into a query without selection followed by a local filter. Note that rule N1 uses a combination of F-IR operators as well as operators for combining regions (such as *seq*, *loop* and *cond*).

[5] γ is the relational aggregation operator. Here, we present abridged versions of the rules, for the sake of brevity. For complete details of these transformations including ordering, duplicates, and variations of each rule, refer [4].

[6] In our current implementation, N1 prefetches an entire relation and all subsequent lookups are performed locally. This can be extended to prefetch queries that result only in a part of the relation.

We use Rule T5 to extract an SQL query for *sum*. As shown in Fig. 9, this is added as an alternative (node 2) to the OR corresponding to the expression for *sum*. Similarly, alternative expressions for *cSum* are added after applying other transformations (not shown in Fig. 9). Using the cost model described in Section VI, COBRA can identify that the alternative with node 2 incurs an extra query execution cost, in addition to the loop computation represented by *fold*. After the least cost program is found, the F-IR representation is translated into imperative code. We refer the reader to [4] for details on generating imperative code from F-IR.

Limitations of F-IR: As discussed earlier (Section V-A), not all loops can be represented in F-IR. The focus of F-IR is to represent set-oriented operations on collections/query results using cursor loops in imperative programs. Further, F-IR currently represents only selection (read) queries, so updates are not part of F-IR. Expanding F-IR to support updates is part of future work. Note that selection queries interleaved with update queries can still be represented using F-IR, leaving the updates intact. We refer the reader to [4], [18] for more details.

VI. COST MODEL

In this section, we discuss how to estimate the cost of a program represented using the Region DAG, and how to find the best alternative from many possible alternatives. We will restrict our attention to cost estimation for individual nodes in the Region DAG; the idea for cost-based search in the Region DAG is similar to that in the Volcano/Cascades AND-OR DAG (refer Section III-A). In our work, we focus on optimizing programs for data access. Fig. 11 describes the parameters we consider for cost estimation. The parameter *amortization factor* (AF_Q) estimates the number of invocations of a query Q, to allocate the prefetching cost across each invocation.

We use a resource consumption model for cost estimation, where network, CPU, and query execution costs are expressed in units of time. Using the parameters from Fig. 11, the cost of the various nodes in the AND-OR DAG is estimated as follows.

Query execution: The cost of execution of a query Q is defined as follows:

$$C_Q = C_{NRT} + C^F_Q + max(N_Q * S_{row(Q)}/BW,\ C^L_Q - C^F_Q)$$

Term	Definition
C_{NRT}	Network round trip time between the client (where the program is running) and the database.
C_Q^F	Time taken by the database since receiving the query to send out the first row in the result.
C_Q^L	Time taken by the database since receiving the query to send out the last row in the result.
N_Q	Cardinality of the result set for Q, i.e., the number of rows in the result after executing Q.
$S_{row(Q)}$	Size in bytes of a single row in the result set for Q.
BW	Network bandwidth (bytes/sec)
AF_Q	Amortization factor – estimated number of invocations of Q.
C_Y	Cost of a program operator node in the Region DAG
C_Z	Cost of executing one imperative program statement (other than query execution statement)

Fig. 11: Cost parameters

Prefetch: The cost of prefetching a relation using a query Q is defined as follows:

$$C_{prefetch(Q)} = C_Q / AF_Q$$

Currently in our framework, we decide to prefetch a query if (a) it is explicitly marked for prefetching as the result of a transformation (such as N1 from Fig. 10), or (b) an entire relation is fetched without any filters/grouping. (AF_Q) may be tuned individually for various queries depending on the particular application's workload.

In general, determining whether or not a relation should be prefetched is non-trivial, as this may affect the cost of other nodes included in a plan. This problem is similar to the multi-query optimization problem, which aims to calculate the best cost and plan for a query considering materialization [14] (in our case, caching). Extending COBRA to adapt heuristics from [14] to efficiently handle alternatives generated due to caching, and dynamic approaches for prefetching (outlined in the extended report [18]) are part of future work.

Basic block node: A basic block node in the Region DAG contains imperative code. The cost of the basic block is the sum of the cost of each statement (C_Z) in the basic block. C_Z can be tuned according to the particular application.

Region operator node: Region operator nodes are rooted at the operators *seq*, *cond*, or *loop*. Their cost is calculated as follows:

C_{seq} = sum of the cost of each child.

$C_{cond} = p * C_{true} + (1-p) * C_{false} + C_p$

where p is the probability that the condition evaluates to true, C_p is the cost of evaluating the condition, and C_{true} and C_{false} are the costs of the sub-regions corresponding to p evaluating to true and false respectively. If the condition is in terms of a query result attribute, our framework estimates the value of p using database statistics. Otherwise, a value of 0.5 is used.

C_{loop}: If the loop is over the results of a query Q, then it may be represented using a *fold* expression, whose cost is calculated as follows:

$$C_{fold} = N_Q * C_f + C_{Db(Q)}$$

where C_f is the cost of the fold aggregation function.

If the number of iterations is known (loop is over the results of a query, or over a collection) but the loop cannot be represented using *fold*, then the cost is calculated as $K * C_{body}$, where C_{body} is the cost of the loop body, and K is the number of loop iterations. If the number of iterations cannot be known (such as in a generic while loop), we use an approximation for the number of loop iterations, which can be tuned according to the application.

Other F-IR operators: We assign a static cost C_Y for evaluating any other F-IR operator. C_Y can be tuned according to the particular application.

VII. RELATED WORK

In this section, we survey related work on various fronts.

Program transformations for database applications: In our earlier work, we have developed the DBridge system [20], [21] for optimizing database applications using static program analysis techniques. Various program transformations such as batching, asynchronous query submission and prefetching [13], [3] have been incorporated in DBridge. DBridge also contains transformations for rewriting Hibernate applications using SQL for improved performance [4]; the QBS system [1] also addresses the same problem. However, existing approaches assume that such transformations are always beneficial. In contrast, our framework allows a cost-based choice of whether or not to perform a transformation and to choose the least cost alternative from more than one possible rewrites.

Note that unlike earlier techniques in DBridge [13], [3], [4], the focus of this paper is not on the program transformations themselves; rather we focus on representing various alternatives produced by one or more transformations of imperative code and choosing the least cost alternative. Our implementation of COBRA uses DBridge as a sub-system for generating alternative programs by applying these transformations. In general, COBRA can be used independently of DBridge with any set of program transformations.

Enumeration and application of transformations: The Peggy compiler optimization framework [8] facilitates the application of transformations (compiler optimizations) in any order. It uses a data structure called PEG that is similar to the Volcano/Cascades AND-OR DAG. However, there are significant differences from our framework.

Peggy is aimed at compiler optimizations and works on expressions. Our framework is aimed at transformations on larger program units such as regions or even an entire program in addition to transformations on expressions, and can support multiple IRs unlike Peggy (as discussed in Section IV). COBRA also improves upon Peggy in terms of program cost estimation. The cost model in Peggy is primitive, especially as the cost of a loop is calculated as a function of its nesting level and a predetermined constant number of iterations. Such a cost model is inadequate for database applications as query execution statements and loops over query results take the bulk

of program execution time. A more sophisticated cost model that can use the database and network statistics, such as the one described in this paper, is desired.

Pushing computation to the database: The Pyxis [22] system automatically partitions database applications so that a part of the application code runs on a co-located JVM at the database server, and another part at the client. In contrast to Pyxis, COBRA generates complete and equivalent programs using program transformations on the original program and does not require any special software at the database server.

LINQ to SQL: A number of language-integrated querying frameworks similar to LINQ [23] allow developers to express relational database queries using the same language as the application, and later translate these queries into SQL [23], [24]. Our techniques focus on automatically identifying parts of imperative code that can be pushed into SQL, whereas [24] require developers to completely specify these queries, albeit in a syntax that uses source language constructs.

VIII. EXPERIMENTAL EVALUATION

In this section, we present an evaluation of the COBRA framework for cost-based rewriting of database applications. We implemented COBRA by extending the PyroJ optimizer [14], which is based on Volcano/Cascades. COBRA leverages the region based analysis framework and program transformations from the DBridge [21] system for optimizing database applications. DBridge internally uses the Soot framework [16] for static analysis.

For our experiments, we used two machines: a server that runs the database (16GB RAM with Intel Core i7-3770, 3.40GHz CPU running MySQL 5.7 on Windows 10), and a client that runs the application programs (8GB RAM with Intel Core i5-6300 2.4GHz CPU running Windows 10, around 4GB RAM was available to the application program). The numbers reported in the experiments are averaged over five runs. Our experiments aim to evaluate the following: (a) applicability of COBRA and our cost model and (b) performance benefits due to cost-based rewriting. Our experiments use real-world and synthetic code samples that use the Hibernate ORM.

In Experiments 1, 2, and 3, we evaluate the performance of program P_0 and its alternatives P_1 and P_2 (which were shown in Fig. 3), along with the choice suggested by COBRA. We implemented P_0 using the Hibernate ORM, and used transformation rule N1 and a variation of transformation rule T5 (refer Section V-C) to generate P_2 and P_1 respectively, from P_0. The size of each row in *Order* and *Customer* has been chosen according to the TPC-DS [25] benchmark specification.

We ran the programs under varying network conditions and cardinalities of the tables *Order* and *Customer*. We connected the client and server directly with an ethernet cable and simulated variations in the network using a network simulator [26]. We used the following conditions: *slow remote network* (bandwidth: 500kbps, latency: 250ms (taken from [27])) and *fast local network* (bandwidth: 6gbps, round trip time: 0.5ms).

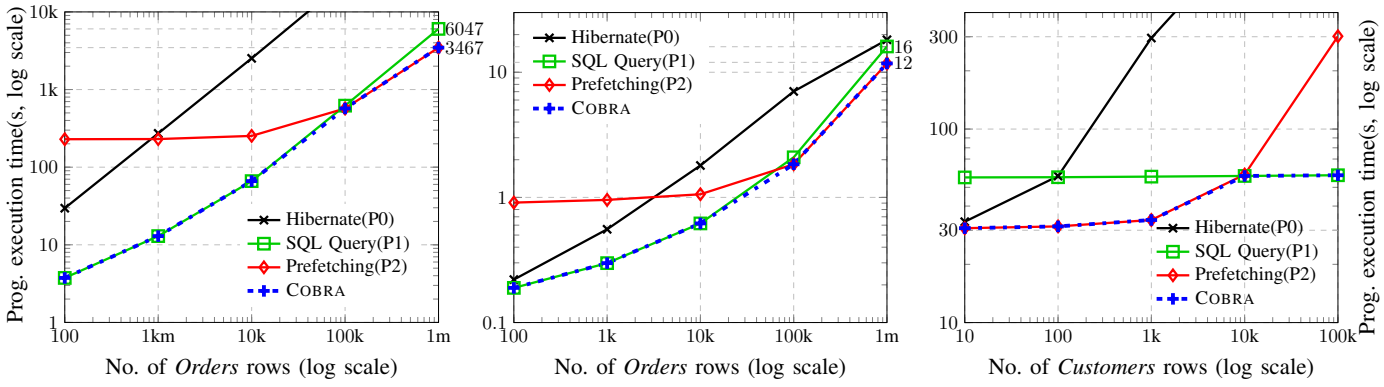
For estimating the cost of generated alternatives using our cost model, we focused on data transfer costs and the number of loop iterations (size of query result set). The cost of executing any other instruction apart from a query execution statement in the program (C_z from Section VI) was set to 30ns, after profiling the applications to estimate the same. We set the amortization factor to 1 (for experiments 1, 2 and 3). We consulted the database query optimizer to estimate query execution times, based on past executions of the queries. The cost metrics were provided to our system as a cost catalog file.

Experiment 1: We first ran the programs using a slow remote network. We fixed the number of rows in *Customer* to 73,000 and varied the number of *Order* rows from 100 to 1 million. Fig. 12a shows the actual running times of these programs, and the choice suggested by COBRA. At a lower number of *Order* rows, COBRA chose the program using SQL query API (P_1), as the other two alternatives incur high latency. Program P_0 suffers from a large number of network round trips due to iterative queries, and P_2 prefetches a relatively large amount of *Customer* data. However, as the number of *Order* rows approaches the number of *Customer* rows, program P_1 causes increasing duplication of *Customer* data in the join result. At this point, COBRA switched to program P_2. The performance of prefetching (P_2) does not vary much for lower cardinalities as the bulk of the time is spent on fetching the larger relation (*Customer*) data. In each case, COBRA correctly identified the least cost alternative.

Experiment 2: We use the same cardinalities as in Experiment 1, but use a fast local network. Again, COBRA estimated P_1 to be the least cost alternative until the number of *Order* rows approaches the number of *Customer* rows, and switched to P_2 after that. This is reflected in the running times of these programs, as shown in Fig. 12b. Although P_2 performs better than P_1 at high cardinality of *Order* in both Fig. 12b and Fig. 12a, the performance difference is much more significant in a slow remote network (3467s vs 6047s) than in a fast local network (12s vs 16s). Note that the performance of SQL query (P_1) and Hibernate (P_0) is comparable at high cardinalities in fast local network. This can be understood as follows. The overhead of a network round trip is very small in a fast local network. Hibernate program internally caches each *Customer* row once fetched, so the latency is minimized after all *Customer* rows have been fetched using individual queries.

Experiment 3: In this experiment, we use a slow remote network, fix the number of *Order* at 10,000 and vary the number of *Customer* rows. As the results from Fig. 12c indicate, the time taken by P_1 is nearly constant (as the size of the join result does not vary with increasing number of *Customer* rows). However, the time taken by P_2 increases with the number of *Customer* rows as P_2 prefetches the entire *Customer* table. This demonstrates that unlike Fig.s 12b and 12a, it is not necessary that P_1 performs better at lower cardinalities, and P_2 performs better at higher cardinalities. COBRA correctly chose the least cost program in each case based on its cost model.

Experiment 4: In this experiment, we used a real-world open source application, Wilos [28], which uses the Hibernate ORM



(a) Slow remote nw - varying Orders (b) Fast local nw - varying Orders (c) Slow remote nw - varying Customers

Fig. 12: Performance of alternative implementations of Fig. 3a in slow remote and fast local networks

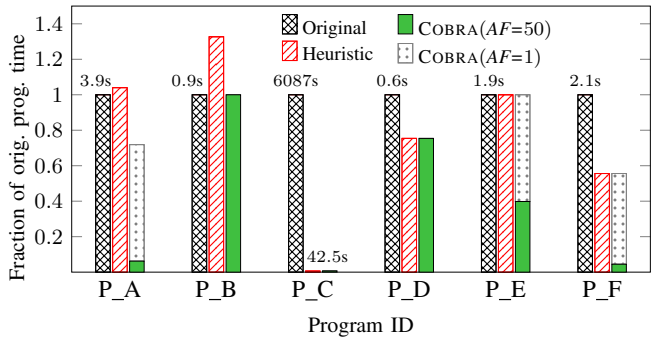


Fig. 14: Performance benefits due to COBRA

Id	Description of cost-based choice	#
A	*Nested loops with intermittent updates*: Inner loop can be translated to SQL for better performance vs overall performance may degrade due to iterative queries	3
B	*Multiple aggregations inside loop*: Faster aggregation/fetch only result by translating to SQL vs multiple queries (NRT) instead of one	2
C	*Nested loops join*: Better join algo. at the database and fetch (large) result of SQL join vs Cache tables at application and join locally	9
D	*Function that is called inside a loop can be rewritten using SQL*: overall performance may degrade due to iterative queries if caller loop cannot be translated	7
E	*Collection filtered differently across different calls of a recursive function*: Multiple point look up queries vs prefetch the whole table once and filter from the cache	9
F	*Different parts of a collection are used across different callee functions*: Multiple select/project queries to fetch only required data vs prefetch all data with one query	2

Fig. 13: Cases for cost-based optimization in a real-world application (pattern id, description, number of cases)

framework. By manual examination of the Wilos source code, we identified 32 code samples where cost-based transformations are applicable. These samples can be broadly classified into six categories. Fig. 13 lists for each category, the cost-based choice of transformations and the number of cases identified. For more details of the code fragments, refer the extended report [18].

We ran COBRA on a representative sample from each category. We used a data generator to generate test data based on the application schema, with the size of the largest relation(s) as 1 million. In particular, the following setup was used: fast local network, many to one mapping ratio 10:1, selectivity of any predicate used 20%. Since we do not know the Wilos application characteristics to estimate the amortization factor, we evaluated COBRA with three different amortization factors (*AF*=1, *AF*=50, and *AF*=∞) in the cost model. The results for *AF*=50 and *AF*=∞ were only marginally different, so for clarity, we only show the results for *AF*=1 and *AF*=50, in Fig. 14.

The x-axis in Fig. 14 shows the program identified by its pattern ID, and the y-axis shows the fraction of the actual execution time taken by a rewritten program in comparison to that of the original program. We plot the following bars for each program. *Original* – the original program, *Heuristic* – program rewritten using the heuristic from [4] (push as much computation as possible into SQL query, then prefetch the query results at the earliest program point), $\text{COBRA}_{(AF=50)}$ – program rewritten using COBRA with AF=50, and $\text{COBRA}_{(AF=1)}$ – program rewritten using COBRA with AF=1. The actual time in seconds for *Original* is shown above the bar. We use transformation rules proposed by earlier techniques [13], [4] (listed in Fig. 10).

The results from Fig. 14 suggest that performance benefits due to COBRA are significant. In the examples considered for this experiment, programs rewritten using COBRA gave up to 95% improvement over the heuristic optimized program, when the cost was computed using *AF*=50. Even with *AF*=1, COBRA outperforms the original and heuristic optimized programs in some cases like A, as COBRA's calculated iterative query invocations to be costlier and chose the prefetch alternative (refer Fig. 13 pattern A). In cases B, C, and D, COBRA chose the same plan with *AF*=1 as well as *AF*=50, hence the bars are identical. Note that in each case, the program rewritten using COBRA (with *AF*=1 or 50) always performs at least as

well as the original/heuristic optimized program. For lack of space, we skip a discussion on the program plans chosen by COBRA. We refer the reader to the extended report [18] for a detailed description of these plans.

Threats to validity: Our evaluation uses programs that use the Hibernate ORM as part of the Spring framework [29]. Spring automatically takes care of transaction semantics based on annotations that specify which functions are to be executed within a transaction. Each sample that we considered in our evaluation runs under a single transaction (as is typical of a *service* function in Spring), so cache invalidation across transactions is not a problem. Further, Hibernate contains built-in cache management for database mapped objects. In general for other database application programs, optimizing across transactions may not preserve the program semantics and/or affect the amortization factor due to stale caches. Identifying such cases automatically using program analysis is part of future work.

The values of parameters in our cost model have been tuned with respect to the Wilos application, which we used in our evaluation. However, in some cases, there was some deviation of the estimated program execution cost from the actual cost. We observed that this is due to multiple factors including (a) parameters not considered in the cost model (example: Hibernate's cost of constructing mapped objects from the result set), (b) fluctuating values of parameters (example: the utilized bandwidth is a fraction of the maximum bandwidth and varies across different query results), etc. Although our cost model correctly predicted the least cost alternative in all the evaluated samples despite these limitations, a more refined cost model may be desired in general.

IX. CONCLUSION

In this paper, we proposed a framework for generating various alternatives of a program using program transformations and choosing the least cost alternative in a cost-based manner. We identify that program regions provide a natural abstraction for parts of an imperative program, and extend the Volcano/Cascades framework for optimizing algebraic expressions, to optimize programs with regions. Our experiments show that techniques in this paper are widely applicable in real-world applications with embedded data access, and provide significant performance improvements.

Apart from various extensions identified throughout the paper, future work includes expanding the set of program transformations available in COBRA. For instance, program partitions in Pyxis [22] can be modeled as partitions of regions in COBRA, with the location as a physical property and enforcers to transfer data between locations (these concepts already exist in Volcano/Cascades). Although we have focused on data access optimizations for imperative programs in this paper, COBRA could be used for other cost-based program transformations with an appropriate cost model, examples being optimization of stored procedures and automatic parallelization.

ACKNOWLEDGMENTS

The work of K. Venkatesh Emani is supported by a fellowship from Tata Consultancy Services. We thank Uday Khedker, Krithi Ramamritham, and Amitabha Sanyal for their valuable inputs throughout the course of this work.

REFERENCES

[1] A. Cheung, A. Solar-Lezama, and S. Madden, "Optimizing database-backed applications with query synthesis," in *PLDI '13*, 2013, pp. 3–14.
[2] M. Chavan, R. Guravannavar, K. Ramachandra, and S. Sudarshan, "Program transformations for asynchronous query submission," in *ICDE*, 2011, pp. 375–386.
[3] K. Ramachandra, M. Chavan, R. Guravannavar, and S. Sudarshan, "Program transformations for asynchronous and batched query submission," *TKDE '15*, vol. 27, no. 2, pp. 531–544.
[4] K. V. Emani, K. Ramachandra, S. Bhattacharya, and S. Sudarshan, "Extracting Equivalent SQL from Imperative Code in Database Applications," in *SIGMOD '16*.
[5] C. Radoi, S. J. Fink, R. Rabbah, and M. Sridharan, "Translating imperative code to MapReduce," in *OOPSLA*. ACM, 2014, pp. 909–927.
[6] G. Graefe and W. J. McKenna, "The Volcano optimizer generator: Extensibility and efficient search," in *Data Engineering*. IEEE, 1993, pp. 209–218.
[7] G. Graefe, "The Cascades Framework for Query Optimization," *IEEE Data Eng. Bull.*, vol. 18, no. 3, pp. 19–29, 1995.
[8] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: A new approach to optimization," in *POPL*, 2009, pp. 264–276.
[9] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., 1997.
[10] "Hibernate," http://www.hibernate.org (retrieved Oct 1, 2017).
[11] "JasperReports Hibernate Sample (retrieved Oct 1, 2017) http://jasperreports.sourceforge.net/sample.reference/hibernate/."
[12] T.-H. Chen, "Improving the performance of database-centric applications through program analysis," Ph.D. dissertation, Queen's University, 2016.
[13] K. Ramachandra and S. Sudarshan, "Holistic optimization by prefetching query results," in *SIGMOD*, 2012.
[14] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe, "Efficient and Extensible Algorithms for Multi Query Optimization," in *SIGMOD*, 2000.
[15] M. W. Hall, B. R. Murphy, S. P. Amarasinghe, S.-W. Liao, and M. S. Lam, "Interprocedural analysis for parallelization," in *LCPC*, 1995, pp. 61–80.
[16] "Soot: A Java Optimization Framework," https://sable.github.io/soot/ (retrieved Jul 16, 2017).
[17] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2006.
[18] K. V. Emani and S. Sudarshan, "Cobra: A framework for cost based rewriting of database applications," https://arxiv.org/abs/1801.04891.
[19] "Reduce vs foldleft," https://stackoverflow.com/a/25158790/1299738 (retrieved Oct 1, 2017).
[20] M. Chavan, R. Guravannavar, K. Ramachandra, and S. Sudarshan, "DBridge: A program rewrite tool for set-oriented query execution (demo)," in *ICDE*, 2011.
[21] K. V. Emani, T. Deshpande, K. Ramachandra, and S. Sudarshan, "DBridge: Translating Imperative Code to SQL," in *SIGMOD '17*, pp. 1663–1666.
[22] A. Cheung, S. Madden, O. Arden, and A. C. Myers, "Automatic partitioning of database applications," *Proc. VLDB Endow.*, vol. 5, no. 11, pp. 1471–1482, Jul. 2012.
[23] "Language Integrated Query (LINQ)," https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq (retrieved Feb 25, 2018).
[24] T. Grust, J. Rittinger, and T. Schreiber, "Avalanche-safe LINQ compilation," *VLDB*, vol. 3, no. 1-2, pp. 162–172, 2010.
[25] "TPC-DS specification," http://www.tpc.org/ (retrieved Oct 1, 2017).
[26] "Network Emulator Toolkit https://blog.mrpol.nl/2010/01/14/network-emulator-toolkit/ (retrieved Jul 23, 2017)."
[27] "AWS Network Latency Map," https://datapath.io/resources/blog/aws-network-latency-map (retrieved Oct 1, 2018).
[28] "Wilos Orchestration Software," http://www.ohloh.net/p/6390 (retrieved Oct 12, 2017).
[29] "Spring Framework," https://spring.io/ (retrieved Jan 15, 2018).