Optimizing Data Access in Database Applications using Static Analysis

Thesis

Submitted in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

by

K. Venkatesh Emani
Roll No: 134058001

Advsior

Prof. S. Sudarshan

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai
2019

To the memory of my grandfather, Emani Venkateswarlu.

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

(Signature)

(Name of the student)

(Roll No.)

Date:

Abstract

Database applications are typically written using a mixture of imperative languages and embedded queries (expressed using SQL/other frameworks) for data access. Traditionally, the imperative and declarative parts of these applications have been optimized separately. Techniques for optimization that span across the declarative and imperative parts in database applications are called *holistic* optimization techniques.

In this thesis, we present novel holistic optimization techniques for optimizing data access in applications that access data using database abstractions. In such applications, data processing logic often gets distributed across the declarative and imperative parts of a program. Consequently, data access from the application is often inefficient due to iterative queries, transfer of unused data, over-specification of queries, and other reasons. We propose techniques based on static program analysis and program transformations to automatically rewrite database applications for efficient data access. When more than one transformations are applicable on a particular program, our techniques can systematically generate all equivalent alternatives using the given set of program transformations, represent these alternatives efficiently, and choose the best rewrite based on a cost model. We also improve upon existing techniques for optimizing the evaluation of user defined functions in databases, which, similar to database applications, contain a mix of imperative code and declarative SQL queries. We demonstrate that the static analysis techniques we develop can be applied to other optimizations as well as test data generation for queries in database applications. Our experiments on real world and benchmark applications show that our techniques have wide applicability and provide significant performance benefits.

Contents

List of Figures

Chapter 1

Introduction

Enterprise applications typically rely on a persistence backend to store user and application data [98]. Relational databases, along with SQL, provide a simple interface for data definition and manipulation, and are the standard choice of storage for a lot of such applications. Database applications written using imperative languages like Java, C#, etc. access the database through interfaces such as JDBC and object-relational mappers [21].

Traditionally, query execution calls to the database within the application program have been treated as black boxes [85]. This also resulted in the optimization of such applications independently on two fronts: (i) source code optimizations by the language compiler and (ii) query optimization at the database. However, optimizing individual components does not ensure the global optimum. Although the individual components are optimal, how they interact with each other can lead to a large number of correctness, security and performance issues which can go unnoticed during development time. Thus, we need a *holistic* view of the database and application as a single unit to address these issues. Techniques for optimization of database applications that span the application program and database have been referred to as *holistic optimization techniques* [69].

1.1 Problem Overview and Motivation

The interaction between a typical application with embedded SQL queries and the database is shown in Figure 1.1. The application submits a query, which is sent to the database. The database optimizes and executes the query, and the results are returned to the application in the form of a `ResultSet`. The query results are then used to perform other operations in the application. This view of database applications presumes a clear separation of concerns between the application program and the database:

- The database performs computations related to data processing including filtering, joins, aggregation, and others.
- The application performs business related computations such as displaying information, taking user input, page navigation, and others, leveraging the database for accessing relevant data.

A relational database stores data in the form of tables and columns, whereas an application program typically uses data in the form of objects. This is called an *object-relational*

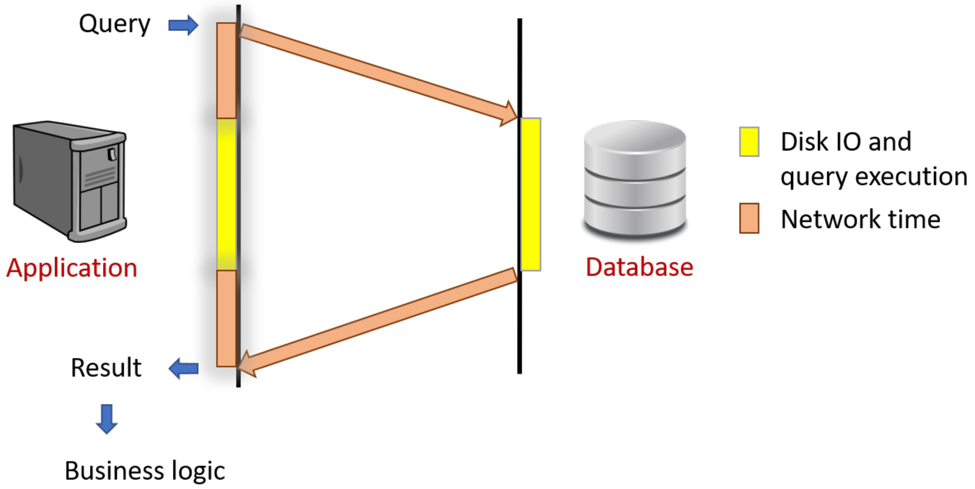


Figure 1.1: Interaction between the application and the database in a typical JDBC application (image source: [83])

impedance mismatch. Object-relational mapping (ORM) frameworks address this impedance mismatch by allowing developers to access data stored in the database using object accesses, without writing explicit SQL queries. The interaction between applications using such abstractions and the database is shown in Figure 1.2. Developers using these abstractions access data simply using object accesses; the framework takes care of generating the required SQL queries, and translating the results back into objects for use in the application. Examples of popular ORM frameworks include Hibernate [60] for Java, Entity Framework [12] for .NET, Django ORM [11] for Python, Active Record [13] for Ruby on Rails, and others.

An alternative approach to addressing the impedance mismatch is to make database relations as first class citizens inside a programming language. The language provides a model of the database, and custom operations that can be used to construct queries. The language compiler takes care of translating the models and operations into their SQL counterparts. Examples of such languages/libraries include SAP ABAP [14], Microsoft LINQ [15], etc. In essence, the frameworks, languages and libraries discussed above provide an abstraction of the relational

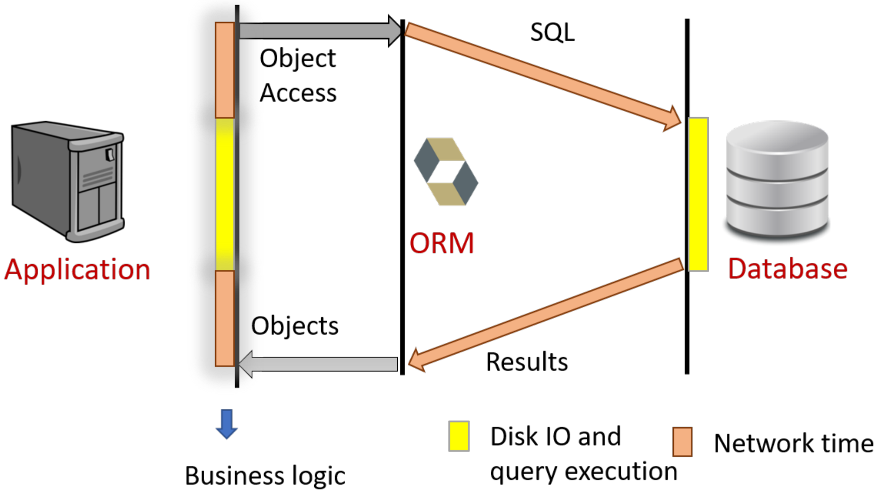


Figure 1.2: Interaction between the application and the database in a typical ORM application

```
List getUnfinishedProjects() {
 List unfinishedP = new ArrayList<Project>();
 List projects = loadAll(Project.class);
 //loadAll() internally uses SQL to fetch all rows of the table Project
 for (Project project :  projects){
  if (!(project.getIsFinished()))
   unfinishedP.add(project);
 }
 return unfinishedP;
}
```

(a) Program using Hibernate ORM

```
List getUnfinishedProjects() {
 List unfinishedP = executeQuery(
     "SELECT *
     FROM Project
     WHERE isFinished <> 1");
 return unfinishedP;
}
```

(b) Program from Figure 1.3a rewritten to use SQL

Figure 1.3: Distribution of data processing logic in ORM applications

model of the database. The use of database abstraction frameworks in database applications has been increasing. Chen et al. [28] note that in a recent survey, 67.5% of Java developers use the Hibernate ORM framework to access the database.

In applications using ORM frameworks (or other abstractions), data processing gets distributed across the database and the application program. Developers of ORM applications tend to express complex queries using simple queries (generated by ORM) coupled with imperative code. For example, consider Figures 1.3 and 1.4. Figures 1.3a and 1.4a both use constructs provided by the database abstraction framework to read rows of a table into an intermediate collection and filter using an `if` inside a loop. Further, Figure 1.4a performs aggregation after filtering. These frameworks enable modularity and code reuse, and help developers without

```
SELECT * INTO TABLE gt_doc FROM ekko.
LOOP AT gt_doc INTO lwa_vbfa
 IF lwa_vbfa-matnr EQ 5.
  lwa_qty = lwa_qty + lwa_vbfa-rfmng.
 ENDIF.
ENDLOOP.
```

(a) An ABAP program

```
SELECT SUM(rfmng)
FROM gt_doc
WHERE matnr = 5
```

(b) SQL query for Figure 1.4a

Figure 1.4: Distribution of data processing logic in programs using database abstractions

SQL expertise to develop applications that need to access a relational database, hence they are popular.

However, the abstraction provided by ORMs comes at the cost of performance issues due to multiple network round trips, transfer of unused data, and other inefficiencies [118, 28]. For example, in Figure 1.3a, only a fraction of the rows fetched are used, and in Figure 1.4a the rows fetched are used to compute a single aggregate value (`lwa_qty`). This problem can be alleviated by using an SQL query; for example, the program in Figure 1.3a can be rewritten using an SQL query as shown in Figure 1.3b, to fetch only the required rows. There is a need for extending holistic optimization techniques beyond optimization of applications that use JDBC/SQL, to fix such inefficiencies in applications that use database abstractions.

In this thesis, we develop techniques to optimize database programs where the data processing logic is distributed across the application program and the database executing SQL. There has been prior work in this area [33, 114, 84] (only the most relevant work is mentioned here, refer Chapter 2 for a detailed survey of prior work). However, either the applicability of these approaches [114] is limited, or the approach [33] is resource intensive, and the approaches [33, 114, 84] may not perform the best rewrite due to the limitations of using heuristics. In our work, we develop practical solutions to this problem that provide have wide applicability, significant performance benefits, and choose the best rewrite.

Similar to database applications, user defined functions (UDFs) inside databases contain a mix of imperative constructs and SQL queries. Techniques developed for optimizing database applications can be used to optimize UDFs as well. Optimization of UDFs has been explored in the past [99]; however the approach in [99] is intrusive to the database optimizer, hence unviable for integration into commercial database systems. In our work, we improve on the work of [99] and propose a light-weight approach to UDF optimization that is amenable for easy integration into the optimizer.

1.2 Summary of Contributions

The key contributions of this thesis are summarized below.

1. We propose techniques based on static program analysis to identify relational operations such as selections, projections, joins, aggregations, etc. performed in imperative code and automatically rewrite the program to use SQL queries for these operations. The SQL queries can significantly improve program performance by reducing (a) the number of queries (fewer network round trips), (b) amount of data transferred from the database to the application, and (c) overall program execution time.

   We propose techniques based on program regions [73] (i.e., structured fragments of a program such as straight line code, if-else, loops, etc.), which provide a hierarchical division of the program into smaller parts that combine to form larger parts, enabling us to handle complex control flow. We propose a bottom up recursive algorithm on program regions to construct an algebraic representation (named *fold intermediate representation*, or F-IR) for representing loops in imperative code that iterate over query results. We formalize a set of preconditions expressed in terms of inter-statement read/write dependencies that determine whether or not a program can be translated to SQL. Multiple simple transformations, the correctness of each of which is easy to prove, operate in tandem to simplify F-IR expressions to eliminate loops. The transformed expression is amenable for translation into SQL, using which the program is rewritten.

2. We present the COBRA framework for cost-based rewriting of database applications. Our framework addresses gaps in existing approaches, which are based on heuristics, which in turn can lead to sub-optimal rewrites. Given a program containing database accesses, and a set of transformations on the program, our framework systematically explores the space of possible rewrites and chooses the best rewrite.

   Our techniques use the notion of program regions [74] to represent a program algebraically, and extend the Volcano/Cascades [52, 54] framework for cost-based transformations of algebraic expressions, to rewrite programs. Our techniques bring the benefits of Volcano/Cascades such as handling cycles in transformations, representing a large number of alternative rewrites efficiently, etc. to program transformations. The cost based choice is driven by a proposed cost model that estimates the cost of a program containing imperative statements and queries; our cost model is able to consult the database optimizer for estimates related to costs and cardinalities of queries.

3. We develop the Froid framework for optimization of imperative code in a relational database, by inlining user defined function (UDF) invocations in queries. Such inlining provides up to order of magnitude performance improvements by enabling set oriented execution of UDFs, which are otherwise evaluated iteratively once per each row in the calling query.

   To this end, we propose techniques for constructing algebraic representations of scalar T-SQL user defined functions in the Microsoft SQL Server database, building on our work from translating imperative code to SQL. We evaluate alternative strategies for UDF inlining at different stages of query optimization and present our learnings. Our techniques are also able to provide some compiler optimizations such as dead code elimination and dynamic slicing to UDFs in relational databases.

4. The static analysis techniques that we developed for optimizing database applications and UDFs can also be used for extracting useful information about queries embedded in imperative code. We have extended our techniques to perform other optimizations as well as test data generation for embedded queries in database applications, as outlined below:

   - Developers of database applications often use ORDER BY queries, which are expensive. We propose techniques based on data flow analysis to identify whether the order of query results is not used in the program, and develop transformations to rewrite queries and collections to remove the unnecessary ordering, for improved performance.
   - We propose techniques to generate test data for imperative programs containing SQL queries. The XData [24] system developed at IIT Bombay generates test data for automatic grading of SQL queries. Our techniques use static analysis to extract queries and relevant conditions from database applications containing embedded SQL queries, and use XData to generate test data for queries, as well as unit tests for functions containing queries, to assess the correctness of these queries.

1.3 Organization of the Thesis

The thesis is organized as follows. In Chapter 2, we survey existing approaches for holistic optimization of database applications. Techniques for translating imperative code to SQL are detailed in Chapter 3. We discuss the COBRA framework for cost-based transformations in

Chapter 4. Chapter 5 elaborates on the Froid framework for optimizing UDFs. In Chapter 6, we discuss other applications for static analysis of database applications. We conclude the thesis in Chapter 7 with a few directions for future work.

Chapter 2

Literature Survey

In his 1979 paper describing the Theseus database programming language [97], Shopiro noted that the majority of uses of a database are by specialized application programs rather than by interactive queries. One of the design goals of Theseus was to facilitate research in automatic program optimization. Hardikar et al. [58] proposed techniques for automatically generating COBOL data processing reports and test data based on user specifications. As early as 1992, Lieuwen et al. [67] have argued about the need for extending compilers to include database-style optimizations. Although databases have evolved since [97, 58, 67], queries embedded in database applications are widely used in a large variety of applications, thus global (*holistic*) program optimization assumes particular significance, apart from optimization of individual queries.

Over the last decade, there have been many interesting developments in optimizing database applications. Techniques from database query optimization, program analysis and program synthesis have been adopted to solve various problems in the area of holistic optimization. We now survey early approaches and then discuss the major developments in holistic optimization of database applications.

2.1 Early Approaches

Dasgupta et al. [36] presented a static analysis framework for analyzing database applications that use ADO.NET to automatically identify various security, correctness and performance problems in the database application. This framework provides a common infrastructure to incorporate database awareness into the existing compilers. Since the framework is very generic, many analysis and optimization tasks can be incorporated into it.

Manjhi et al. [69] proposed the MERGING transformation to identify patterns in the code where the application first issues a query to get multiple values and then for each value, issues another database query using a loop. They then replace this with a single join query. Although loops containing query execution statements are promising targets for optimization in database applications [85], techniques in [69] do not handle conditional branches like if-else, which limits the applicability of their techniques. Further, loops in database applications often iterate over program collections, so the precondition that the looped values should be results from a previous query is quite restrictive.

```
int sum = 0;
stmt = con.prepareStatement("SELECT COUNT(partkey) FROM part WHERE p
category=?");
while(!categoryList.isEmpty()) {
 category = categoryList.removeFirst();
 stmt.setInt(1, category);
 ResultSet rs = stmt.executeQuery();
 int count = rs.getInt(count);
 sum += count;
}
```

Figure 2.1: Code with opportunities for loop fission transformation (source: [85])

2.2 Recent Approaches

We discuss recent developments in this area, classified according to the type of techniques used.

2.2.1 Batching of Query Results

Techniques for batching of query results aim to reduce the number of query invocations by combining multiple queries into a single call to the database. Consider the program shown in Figure 2.1, which counts the number of parts in a category including all of its sub-categories in the hierarchy. For this, it issues queries synchronously and iteratively, thus incurring a large latency due to multiple network round trips.

Rewriting Procedures for Batched Bindings

Batching transformations proposed by Guravannavar et al. [56] replace iterative invocations of a query using a single invocation of its *batched* query using *loop fission transformations*. The transformations proposed in [56] are part of the DBridge system [26], which is a program transformation system for programs containing embedded queries. We briefly describe the DBridge system before discussing techniques for batching from [56].

DBridge

The DBridge system [26, 43] developed at IIT Bombay uses techniques from static program analysis and program transformations to rewrite programs containing embedded SQL queries, for optimized data access. DBridge focuses on rewriting Java programs that use the JDBC or Hibernate [60] frameworks for accessing a relational database. Since DBridge performs transformations on bytecode, any program that can be compiled into bytecode can also be optimized using the DBridge system. The techniques involved are generic, however, and can be used for rewriting programs that use other languages/data access APIs.

The architecture of DBridge is shown in Figure 2.2. DBridge is a source (.java/.class) to source (.java/.class) rewrite system. It leverages the Soot [102] Java optimization framework, which provides structures such as the control flow graph for implementing data flow analyses and a convenient intermediate representation for transformations called Jimple. Using these structures, DBridge performs multiple transformations on the Jimple code. The modified Jimple code is finally decompiled into the target program.

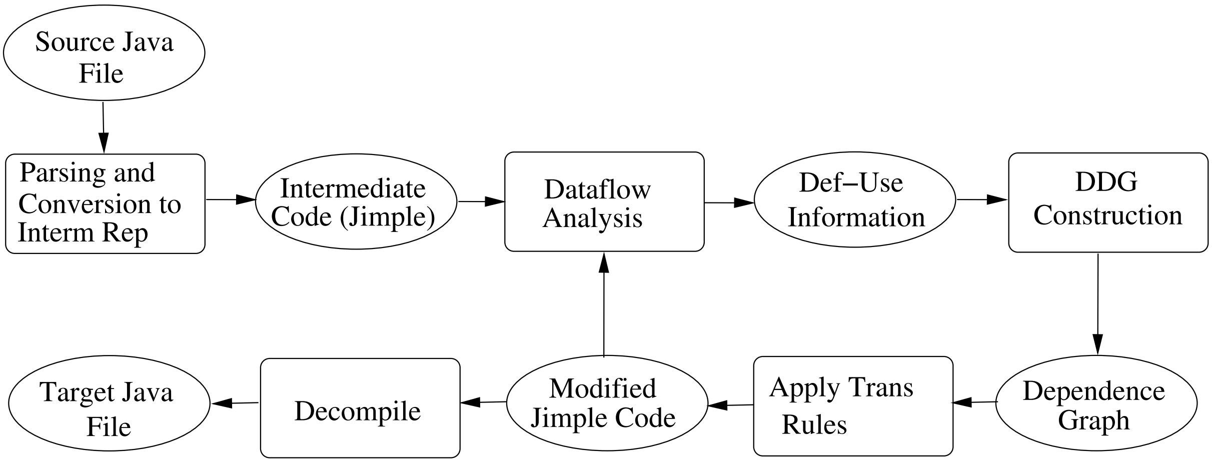
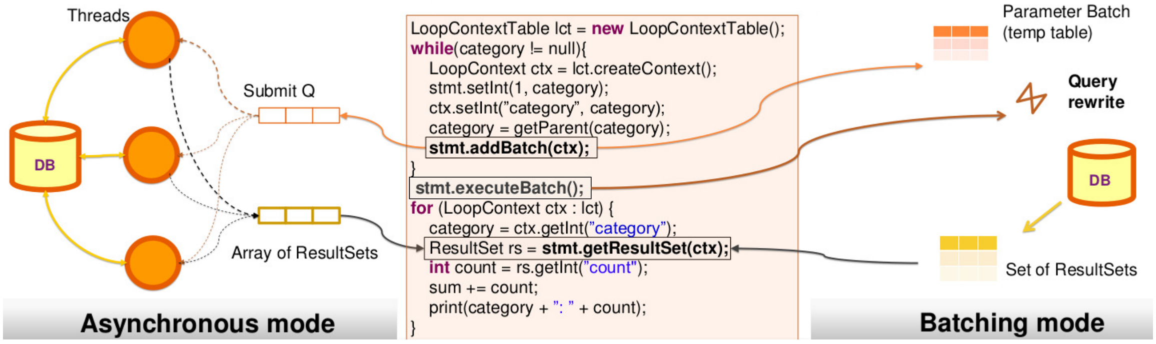


Figure 2.2: DBridge Architecture (image source: [26])



```
LoopContextTable lct = new LoopContextTable();
while(category != null){
    LoopContext ctx = lct.createContext();
    stmt.setInt(1, category);
    ctx.setInt("category", category);
    category = getParent(category);
    stmt.addBatch(ctx);
}
stmt.executeBatch();
for (LoopContext ctx : lct) {
    category = ctx.getInt("category");
    ResultSet rs = stmt.getResultSet(ctx);
    int count = rs.getInt("count");
    sum += count;
    print(category + ": " + count);
}
```

Figure 2.3: Transformed program for Figure 2.1 after loop fission [56]. Figure reproduced from [85].

The main transformations in DBridge are[1] batching transformations [84] and prefetching [87] transformations; we discuss batching in this section, and prefetching in Section 2.2.2.

Batching using Loop Fission

Loop fission transformations focus on identifying query execution statements within a loop and rewriting the queries along with the loop after splitting it, in order to minimize latency due to iterative queries. The loop split and transformation happens systematically using a series of program transformation rules. Each of these rules identifies a program pattern and, provided certain preconditions are satisfied, replaces it with an equivalent program. The loop in the program from Figure 2.1 can be rewritten using loop fission as shown in Figure 2.3. The techniques batching [56] and asynchronous query submission [84] both employ loop fission.

In Figure 2.3 (Batching mode), the first loop adds all the queries to a batch, which is then executed using the `stmt.executeBatch()` statement to fetch a single result for all invocations of the query across all iterations of the first loop. In Figure 2.3, the batched query would be as follows:

```
SELECT pb.category, le.c1
FROM pbatch pb OUTER APPLY
```

[1] The contributions of this thesis have been incorporated into the DBridge system, so the discussion on DBridge will be limited to approaches prior to the work described in this thesis.

```
        (SELECT COUNT(partkey) AS c1)
        FROM part WHERE category = pb.category) le
```

where pb is a temporary table in which parameter bindings are materialized.

The second loop uses the results of the batched query, and the results for each iteration are identified by using a *loop context* that was created when creating the batch.

Sloth

The Sloth [32] system aims to reduce latency in database applications by collecting batches of queries and executing them together. Unlike the batching approach of Guravannavar et al. [56], Sloth performs batching at program runtime. Sloth determines the set of queries to be batched by using *lazy evaluation*, as follows. Query invocations simply register a query for execution and the query is added to the next batch of queries to be sent to the database. Computations involving the results of a query are deferred until the results are critical (such as for displaying on the console), at which point the collected batch of queries is sent to the database for execution.

Batching queries using this technique reduces the number of network roundtrips and also allows the database to optimize and execute multiple queries together, using techniques for multi query optimization [90].

2.2.2 Prefetching Query Results

Prefetching is another technique that helps reduce the latency due to remote database calls by inserting asynchronous requests for query results, even before program execution has reached the query call location. Ramachandra and Chavan et al. [87, 84, 27] propose two techniques for prefetching query results in database applications.

Asynchronous Query Submission

Asynchronous query submission aims to minimize the latency due to iterative queries by overlapping query execution and program execution. Asynchronous query submission also uses loop fission, and the user facing API is the same as in batching transformations (refer Figure 2.3 (Asynchronous mode)).

However, in asynchronous submission, when the query is added to a batch using the statement `stmt.addBatch(ctx)`, the query is asynchronously submitted to the database using a separate thread. The first loop is thus responsible for initiating the query execution for all parameters, and the `addBatch` call returns with a handle. As the results of each query are available, they are cached at the client. In the second loop, when the query results are actually used, the results are retrieved from the cache using the handle obtained earlier. In case some query results are still unavailable in the cache at the time of use in the second loop, the application then falls back to a synchronous block until the results are available.

Hybrid Approaches to Batching

Since the API for batching and asynchronous query submission is the same, a hybrid approach of asynchronous batching [84] has also been proposed. In this approach, a certain number of parameter bindings (which may be determined dynamically at runtime) are batched, and the batched query is executed asynchronously. This approach has the advantage of making the first result available sooner than in batching, while reducing the number of asynchronous query invocations. Another important contribution of [84] is an algorithm for reordering statements within a loop to eliminate loop carried flow dependencies; this increases the applicability of these techniques.

2.2.3 Prefetching Query Results at the Earliest Program Point

Ramachandra et al. [87] propose a technique for prefetching results of queries ahead of their actual execution in the program. The technique is called *query anticipability analysis*, and is based on the data flow framework *anticipable expressions analysis* [65]. Query anticipability analysis determines the earliest program point at which a prefetch instruction for a query can be inserted, without any prefetches being wasted. Intuitively, this is the earliest point in the program where:

- The values of all parameters that are used in a query are available.
- It is guaranteed from the program's control flow that the actual query statement is executed whenever the prefetch statement is executed.

For a formal description of the techniques, refer [87]. Note that although asynchronous query submission also achieves prefetching for queries within loops, techniques in [87] generalize it to prefetch results for queries at any point in the program, including interprocedural prefetching, i.e., inserting prefetch requests across procedure calls.

2.2.4 Pushing Computation to the Database

Cheung et al. [31, 33] propose two techniques for optimizing database applications by pushing computation to the database. They are described below.

Query By Synthesis

Query By Synthesis (QBS) [33] is a tool that extracts SQL from fragments of imperative code that use query results. Cheung et al. [33] note that the widespread usage of object-relational mapping (ORM) libraries in applications to interact with databases, "often leads to poor performance as modularity concerns encourage developers to implement relational operations in application code. Such an implementation does not allow the application to take advantage of the optimized implementations that databases provide". It would be useful to identify such relational logic embedded in imperative code and transform it into SQL queries. The authors present a system (QBS) based on query synthesis that achieves this goal. For example, the program in Figure 2.4 can be rewritten Figure 2.5 by extracting an SQL query.

QBS uses program synthesis technology for extracting queries. The authors define a *theory of ordered relations*, which is similar to relational algebra but uses ordered lists rather than multisets. Reasoning with ordered lists is important because ORM frameworks use an ordered list interface for persistent data. QBS works in two steps:

1. Analyze the input code and come up with templates for the annotations (annotations are algebraic representations of the variables in the code) in the theory of ordered relations. For example, a nested loop could be modeled as a join, an if condition could be modeled as a selection condition, etc.

```
List<User> getRoleUser() {
 List<User> listUsers = new ArrayList<User>();
 List<User> users = this.userDao.getUsers();
 List<Role> roles = this.roleDao.getRoles();
 for (User u:  users) {
  for (Role r:  roles) {
   if (u.roleId().equals(r.roleId())){
    User userok = u;
    listUsers.add(userok);
 }}}
 return listUsers;
}
```

Figure 2.4: Imperative program implementing join (source: [33])

```
List<User> getRoleUser() {
 List<User> listUsers = db.executeQuery(
    "SELECT u
    FROM users u, roles r
    WHERE u.roleId = r.roleId
    ORDER BY u.roleId, r.roleId");
 return listUsers;
}
```

Figure 2.5: Program from Figure 2.4 rewritten using SQL query (source: [33])

2. Formulate equations for post conditions and loop invariants using these templates, in the predicate logic derived from the theory of ordered relations. The authors use the Sketch [101] constraint based synthesis system to solve these equations, to obtain the solution for the annotations. The resulting expressions for variables are then translated to SQL for program rewriting.

Pyxis

Pyxis is a tool that partitions a database application program so that a part of the program can be pushed to run at the server. Data transfer between the database and the application program leads to multiple network round trips. Pyxis [31] aims to fix this problem by identifying parts of the program that can be run at the database itself, so as to minimize the number data transfers and control flow transfers between the application and the database.

For example, the program fragment in Figure 2.6 is partitioned as shown in Figure 2.7 where `APP` denotes code that runs at the application, and `DB` denotes code that runs at the database. The functions `sendAPP`, `sendDB`, and `sendNative` facilitate data exchange between the application to the database (marked using the comment *//exchange* in Figure 2.7).

In Pyxis, the data and control flow dependencies between statements in the original program are encoded in the form of a graph with statements being nodes and dependencies being edges. The edges have a weight that denotes the cost incurred if the two statements are on different partitions. The partitioning decision is modeled as an integer programming problem, with the goal to minimize the cost incurred due to partitioning. For further details, refer [31]. Although Pyxis can automatically identify the partitions, the database server may need to be equipped with additional software to be able to run the partitioned application code, leading to additional

```
class Order {
 int id;
 double[] realCosts;
 double totalCost;
 Order(int id) {
  this.id = id;
 }
 void placeOrder(int cid, double dct) {
  totalCost = 0;
  computeTotalCost(dct);
  updateAccount(cid, totalCost);
 }
 void computeTotalCost(double dct) {
  int i = 0;
  double[] costs = getCosts();
  realCosts = new double[costs.length];
  for (itemCost :  costs) {
   double realCost;
   realCost = itemCost * dct;
   totalCost += realCost;
   realCosts[i++] = realCost;
   insertNewLineItem(id, realCost);
  }
 }
}
```

Figure 2.6: A program accessing the database (source: [31])

maintenance concerns.

2.2.5 Optimizing Transactions in Database Applications

Yan et al. [116] proposed techniques that leverage application semantics to improve the performance of transactions in OLTP applications. The idea behind their techniques is that some queries in a transaction require locking of the same tuples as other concurrently executing transactions (determined by profiling), so such queries should be issued at the end of the transaction. To this end, they propose techniques to reorder statements within a transaction to minimize lock contention, while preserving the application semantics.

2.2.6 Optimizing User Defined Functions in Databases

Decorrelation of nested sub-queries has been studied well in databases [66, 37, 93, 49, 42]. In particular, the *Apply* operator introduced by Galindo-Legaria et al. [42] explicitly models sub-query execution as the evaluation of a parameterized expression for each row in the outer query. Transformation rules replace the *Apply* operator with standard relational operations, thus achieving sub-query decorrelation.

Simhadri et al. [99] propose extensions to the *Apply* operator and transformation rules to decorrelate user defined function (UDF) invocations within a query. For example, the query

```
class Order {
 :APP: int id;
 :APP: double[] realCosts;
 :DB: double totalCost;
 Order(int id) {
  :APP: this.id = id;
  :APP: sendAPP(this); //exchange
 }
 void placeOrder(int cid, double dct) {
  :APP: totalCost = 0;
  :APP: sendDB(this); //exchange
  :APP: computeTotalCost(dct);
  :APP: updateAccount(cid, totalCost);
 }
 void computeTotalCost(double dct) {
  int i; double[] costs;
  :APP: costs = getCosts();
  :APP: realCosts = new double[costs.length];
  :APP: sendAPP(this); //exchange
  :APP: sendNative(realCosts, costs); //exchange
  :APP: i = 0;
  for (:DB: itemCost :  costs) {
   :DB: double realCost;
   :DB: realCost = itemCost * dct;
   :DB: totalCost += realCost;
   :DB: sendDB(this); //exchange
   :DB: realCosts[i++] = realCost;
   :DB: sendNative(realCosts); //exchange
   :DB: insertNewLineItem(id, realCost);
  }
 }
}
```

Figure 2.7: Partition of program from Figure 2.6 (source: [31])

with UDF invocation in Figure 2.8 is rewritten using the techniques in [99] to Figure 2.9. The rewritten query allows the database engine to execute the code inside the UDF in a set-oriented manner instead of iterative invocation as in the original query.

The extensions model imperative constructs used in UDF invocations, such as formal to actual parameter mappings, assignment statements, and branching (if-else) statements. Although the techniques proposed in [99] are widely applicable and provide significant performance benefits, they require intrusive changes to the database query optimizer due to the introduction of new operators and transformation rules, which may be undesirable in commercial databases.

2.2.7 Optimizations for ORM Applications

A few approaches have been developed for optimization of applications that use object-relational mapping (ORM) frameworks. The Query By Synthesis system [33] has been discussed earlier in Section 2.2.4. Here, we discuss other approaches. The goal of these approaches is to use

```
create function service_level(int ckey) returns char(10) as
begin
 float totalbusiness; string level;
 SELECT SUM(totalprice) INTO :totalbusiness
 FROM orders WHERE custkey=:ckey;
 if (totalbusiness > 1000000)
  level = Platinum;
 else if (totalbusiness > 500000)
  level = Gold;
 else
  level = Regular;
 return level;
end
```

Query: `SELECT custkey, service level(custkey) FROM customer;`

Figure 2.8: Query with a scalar UDF (source: [99])

program analysis to identify performance anti-patterns in ORM applications, which can help developers to write more performant ORM code.

ORM Application Maintenance and Caching

The work by Chen et al. [28] focuses on issues pertaining to the maintenance and performance of applications that use object-relational mapping [21] (ORM) frameworks. Their work contains a mix of empirical studies performed on enterprise and open source applications, and a set of tools to aid development of efficient ORM code. Their contributions are summarized below.

- Chen et al. identify that ORMs cannot completely abstract away data access in terms of objects and ORM code is scattered across the application. Changes to ORM code are frequent and the most common case of changes to ORM code is due to security and performance reasons. However, traditional programming language compilers are unaware of ORM syntax and semantics, hence these issues go unnoticed.

- They develop a framework to automatically detect ORM performance anti-patterns. The framework is able to rank the anti-patterns based on their potential impact on application performance.

- They build a tool called CacheOptimizer that can suggest the best cache configuration for database mapped classes in the application. CacheOptimizer uses web application

```
SELECT c.custkey, CASE e.totalbusiness > 1000000: Platinum
CASE e.totalbusiness > 500000: Gold
DEFAULT: Regular
FROM customer c LEFT OUTER JOIN e ON c.custkey=e.custkey;
```

where e stands for the query:

```
SELECT custkey, SUM(totalprice) AS totalbusiness
FROM orders GROUP BY custkey;
```

Figure 2.9: Decorrelated form of query from Figure 2.8 (source: [99])

Approach	Benefits	Static (Program Rewriting)	Runtime
Batching	Reduce #queries, disk IO sharing	Loop fission [56]	Lazy evaluation [32]
Prefetching	Overlap program and query execution	Query anticipability analysis [87]	Adaptive batching [84]
Pushing computation to DB	Reduced data transfer	Query by synthesis [33], Automatic program partitioning [31]	
Statement reordering	Dependency removal	Loop fission [27], Optimizing transactions [116]	
ORM Application Analysis	Aid programmers	ORM Maintenance and caching [28], Analysis of performance bugs [117]	

Figure 2.10: Summary of Prior Work

logs to infer the workload characteristics and data access patterns and automatically adds appropriate cache configurations to the application.

Analysis of Performance Bugs in ORM Applications

The Hyperloop project [10] focuses on detecting and solving performance problems in web applications that use ORM frameworks. Yang et al. [118] surveyed 12 open source Ruby on Rails ORM applications and identify various causes for performance inefficiencies in ORM applications including retrieval of unused fields, missing indexes at the database, unnecessary computation involving query results, and others. Subsequently, they developed a tool [119] for automatically fixing some of these inefficiencies.

2.3 Summary

In this chapter, we have surveyed various techniques for optimizing database applications, with a focus on major developments in this area over the past decade. The techniques are summarized in Figure 2.10, categorized according to the approach and whether the techniques used are static/runtime techniques. We observe that holistic optimization of database applications has received a lot of attention and techniques from multiple disciplines have been adopted to solve a number of interesting problems. Research in recent years indicates the increasing use of ORM frameworks, and motivates the need for a database aware compiler for optimizing ORM applications in addition to applications that use explicit SQL queries.

Chapter 3

Translating Imperative Code to SQL

Database applications perform poorly due to inefficient data access caused by iterative queries and transfer of unused data. These inefficiencies are particularly prevalent in applications that use object-relational mapping (ORM) frameworks. In this chapter we present an approach to this problem, 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. The contents of this chapter have been published in [44].

3.1 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 chapter, 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. [33] to transform parts of application logic into SQL queries, with a focus on database applications using Hibernate [60]. 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 [33], as shown by our experiments. Radoi et al. [82] 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 [33, 82], and other techniques for finding equivalent SQL queries, is given in Section 3.6.

Enhancing applicability of existing techniques

When the results of one query do not directly feed as parameters to another query, batching [56] is unable to combine these two queries into a single query. Techniques in this chapter 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 [56, 84] by combining related queries.

Keyword Search

Keyword search systems such as [41] 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 chapter are as follows:

1. We present (in Section 3.3) 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 3.3.1), and can be applied to complex programs that include function calls.

2. We show (in Section 3.4) 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.

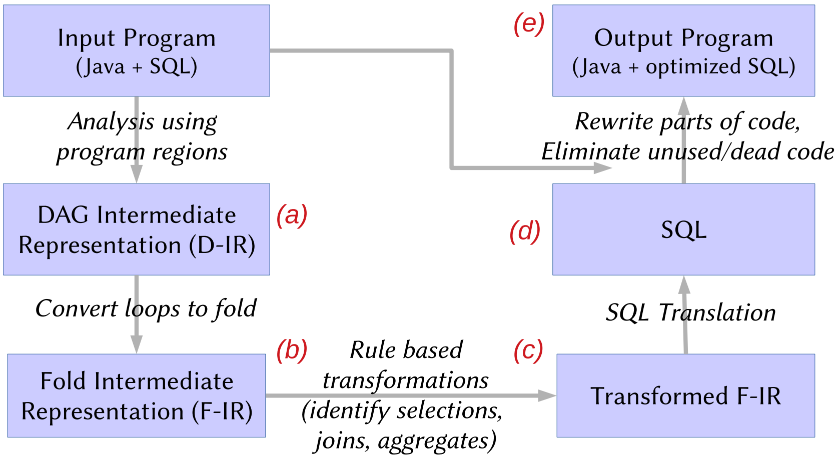

Input Program
(Java + SQL)
Analysis using program regions
DAG Intermediate Representation (D-IR)
(a)
Convert loops to fold
Fold Intermediate Representation (F-IR)
(b)
Rule based transformations (identify selections, joins, aggregates)
(c)
Transformed F-IR
SQL Translation
(d)
SQL
Rewrite parts of code, Eliminate unused/dead code
(e)
Output Program
(Java + optimized SQL)

Figure 3.1: System Overview

3. We present (in Section 3.5) 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 3.5.2) 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 inner loop computes aggregation for each value of outer loop, into a `GROUP BY` query, which earlier techniques cannot do.

4. Our techniques have been implemented in the DBridge [26] tool for static analysis and program transformations, 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 3.7) 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 3.2 presents an overview of our approach. We discuss related work in Section 3.6, and summarize the chapter in Section 3.8.

3.2 Overview

An overview of our system is given in Figure 3.1. Given the source program (fragment), we first construct our 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.

```
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;
}
```

Figure 3.2: Code for highest score calculation

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 3.2, which is extracted from an open source gaming tournament software [71][1]. Variable types have not been displayed in this code, for ease of presentation (we will stick to this practice throughout the chapter). 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 the score itself, for each round. Section 3.5.4 discusses how to generate equivalent SQL for such cases.

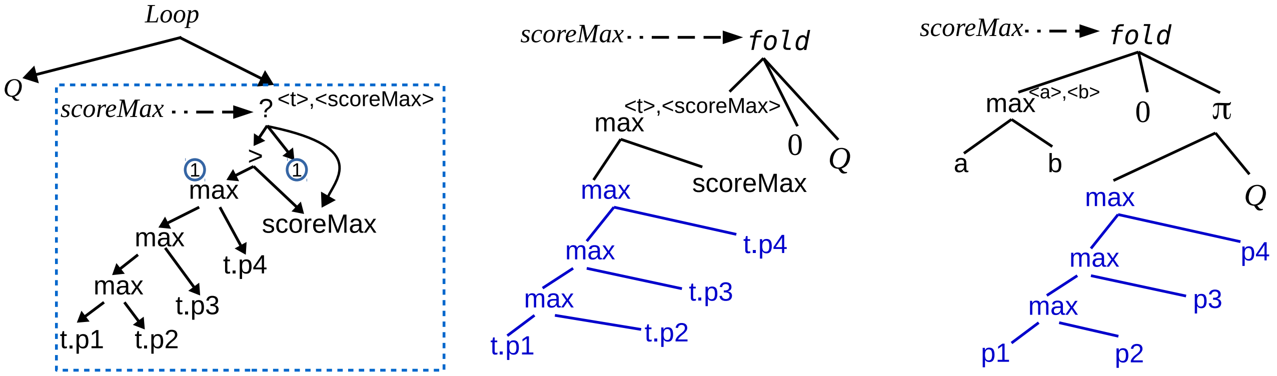
The optimized SQL for the `scoreMax` is shown in Figure 3.3(d)[2]. Parts (a), (b) and (c) show the various intermediate stages from the source code to optimized SQL, each of which will be described in detail in future sections. In parts (b) and (c), *max* is a function which returns the greatest of two elements.

The discussion in this chapter 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 chapter, 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

[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. “?”s represent place holders for query parameters to be substituted in the corresponding order. Our implementation uses the actual source code.

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



Q denotes the query: $\sigma_{rnd\_id = 1}$(Board)

(a) D-IR for `scoreMax` **(b) F-IR derived from (a)** **(c) Transformed F-IR**

```
SELECT MAX(GREATEST (p1, p2, p3, p4)) FROM Board WHERE rnd_id = 1
```

(d) Equivalent SQL

Figure 3.3: Walk-through of equivalent SQL derivation

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 Section 3.5.4.

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 chapter have wide applicability. Inferring equivalent SQL across multiple `try-catch` blocks is part of future work.

3.3 DAG Based IR

The goal of the first intermediate representation we use is to represent the values of program variables as algebraic expressions. We chose a DAG representation to allow 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 [17]. We extend it to construct DAG representations for other program regions (Section 3.3.3).

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.

3.3.1 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 [17]. There are two specially designated

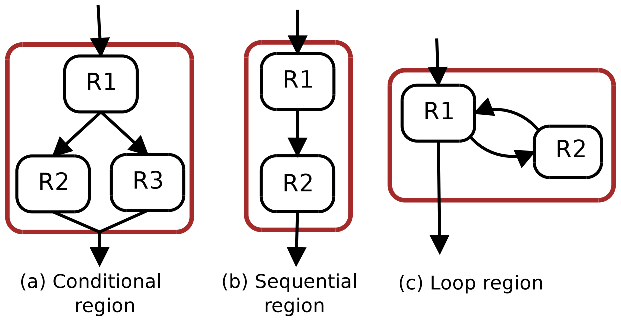


Figure 3.4: Types of regions

nodes: the *Start* node, through which control enters into the graph, and the *End* node, through which all control flow leaves. Additionally, for each node *n*, $Entry(n)$ and $Exit(n)$ represent the program points just before the execution of the first statement, and just after the execution of the last statement of *n*. Directed edges represent control flow; sets *pred(n)* and *succ(n)* denote the predecessors and successors of a node *n* respectively. The predecessor and successor relationships are as defined below:

- *succ(n):* The successors set of node *n* in the CFG G is the set of all nodes n_2 such that there exists an edge from *n* to n_2 in G. i.e.
  $succ(n) = \{n_2 | n_2 \in G \text{ and } n \rightarrow n_2\}$

- *pred(n):* The predecessors set of node *n* in the CFG G is the set of all nodes n_2 such that there exists an edge from n_2 to *n* in G. i.e.
  $pred(n) = \{n_2 | n_2 \in G \text{ and } n_2 \rightarrow n\}$

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 [102].

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, and has a single entry and exit. Regions are constructed from the CFG using rules described in [59]. 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 3.4). In Figure 3.4, *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 3.5(a) represent basic blocks.

- **Sequential Region**: Sequential regions are regions composed of two sub-regions with sequential control flow between them (Figure 3.4(b)). In Figure 3.5(a), R6 is a sequential region.

- **Conditional Region**: A conditional region is comprised of three sub-regions (Figure 3.4(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 3.5(a), *R*5 is a conditional region composed of the condition region *R*2, true region *R*3 and false region *R*4.

- **Loop Region**: Loop regions are composed of two regions (loop header and loop body) with a cycle, as shown in Figure 3.4(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.

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

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 3.5(a). The ee-DAG for the program is shown in Figure 3.5(d). Circled numerals denote pointers to another part of the ee-DAG (to avoid clutter).

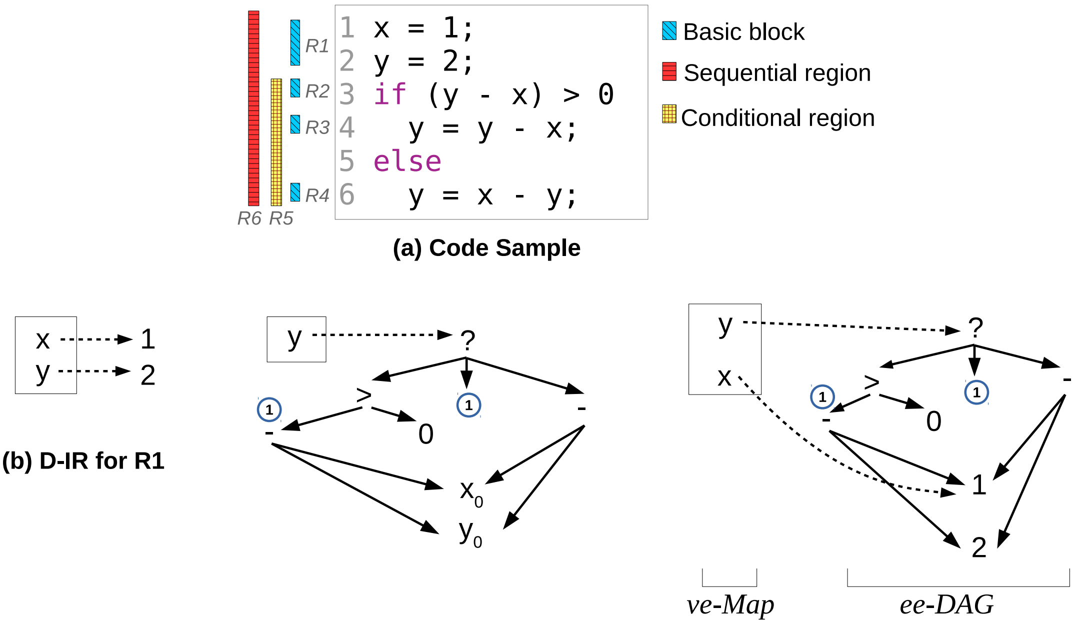
Parameterized queries in the source program can be treated as parameterized expressions in the multiset relational algebra. All relational algebraic operators (project (π), project (σ), join ($\bowtie$) etc.) are available in ee-DAG. Note that in this chapter, π denotes projection without duplicate elimination. We also include extended relational algebraic operators for aggregation (γ), sorting (τ) and eliminating duplicates (δ) [112][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 chapter, 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 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

[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.



```
x = 1;
y = 2;
if (y - x) > 0
  y = y - x;
else
  y = x - y;
```

(a) Code Sample

(b) D-IR for R1

(c) D-IR for R5

(d) D-IR for R6

Figure 3.5: D-IR construction for a simple code fragment

algebraic. The loop body is represented by enclosing it inside a dotted rectangle (refer Figure 3.3(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 3.2, 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 3.5(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 3.3(c), the ee-DAG rooted at π represents the query $\pi_{max(max(max(p1,p2),p3),p4)}(Q)$.

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 chapter, 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 3.5(a) is shown in Figure 3.5(d). In this chapter, we skip showing entries in the ve-Map for variables in which we are not interested (to make diagrams more readable).

3.3.3 Algorithm for D-IR Construction

D-IR construction works on top of the region hierarchy. Construction of regions was discussed in Section 3.3.1. We now outline a bottom up recursive algorithm for D-IR construction for a region. Appendix A.2 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 chapter, region inputs are denoted by tagging them with a subscript 0, for example, $\mathtt{x}_0$ and $\mathtt{y}_0$ in Figure 3.5(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 3.5(a). Figures 3.5(b), (c) and (d) show the step by step construction of D-IR for the program. In Figure 3.5(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 3.5(d), while merging with the preceding basic block *R1*. Note that in the final D-IR (Figure 3.5(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 A.2.

3.4 F-IR Representation

As we have described in Section 3.3.3, 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 3.5). There has been earlier work on using *fold* to represent loops [82].

We use the following textual notation to represent a dag/tree. In general, Δ 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 $e^{\langle p_1 \rangle, \langle p_2 \rangle, \cdots}$ 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, $f^{\langle p_1 \rangle, \langle p_2 \rangle}$) 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, $f(v_1, v_2)$ 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.

3.4.1 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. 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 [47] that allow *fold* to represent computations in loops on ordered collections that cannot be represented by *reduce*.

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 chapter, 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:

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

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, *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 3.3(c) is written as

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

where *max* is the binary maximum function, and Q' denotes

$$\pi_{\,max(max(max(p1,p2),p3),p4)}(\sigma_{rnd\_id=1}(Board)) \qquad .$$

3.4.2 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, we describe a few terms commonly used in program analysis.

- A *loop carried flow dependency* (*lcfd*) 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) [75] 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 identify the direction and type of dependence, respectively.

- A *program slice* $S = slice(P, n, v)$ is defined [105] 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 3.6(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, $slice(P_{37}, l_7, dummyVal) = \{S3,S4,S6\}$.

- *Dead code* [38] 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.

The algorithm for constructing F-IR for a region is given in Algorithm 1. Given a program region R, the algorithm recursively constructs F-IR for each sub-region, before constructing F-IR for the parent region. The algorithm first constructs D-IR for a region and then translates the D-IR expression for each variable into F-IR, provided the preconditions are satisfied. If the

Algorithm 1 Construct F-IR for a Region

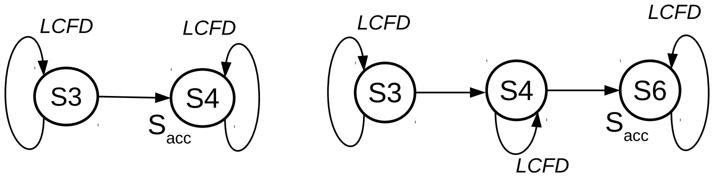
Input: A program region (R)
Output: F-IR for R

```
procedure ConstructFir($R$)
    $\Delta \leftarrow$ D-IR for $R$                ▷ Obtained using the algorithm from Section 3.3.3
    $M \leftarrow$ ve-Map for $R$

    for each sub-region $C$ of $R$ do
        ConstructFir($C$)                              ▷ Recursive call
    end for

    if $R$ is not a cursor loop region then
        return
    else                         ▷ Contains a loop represented using the D-IR Loop operator
        LoopToFold($R$, $\Delta$, $M$)                  ▷ Updates $\Delta$ and $M$
    end if
end procedure

procedure LoopToFold($R$, $\Delta$, $M$)
    $V \leftarrow$ all variables updated in $R$

    for each variable $v \in V$ do
        $l \leftarrow$ program point at the end of $R$
        $S \leftarrow slice(R, l, v)$
        $D_S \leftarrow$ part of the data dependence graph for $R$ that contains only statements from $S$
        $S_{acc} \leftarrow$ all statements from $S$ that write to $v$
                          ▷ See description of algorithm for details of $l$, slice, $D_S$, and $S_{acc}$

        if CheckPreconditions($D_S$, $S_{acc}$) then
            $Loop[Q, expr] \leftarrow M$.lookup($v$)
            $v_0 \leftarrow$ initial value of $v$ at the beginning of the loop
            $expr' \leftarrow$ ParameterizeQueryRefs($expr$)   ▷ See description of algorithm for details
            $foldExpr \leftarrow fold\ [expr', v_0, Q]$

            $\Delta$.add($foldExpr$)
            $M$.put($v$, $foldExpr$)
            Replace pointers to $v$ in $\Delta$ with pointers to $foldExpr$

            Insert statement $s_{fold}$: $v = foldExpr$ at the end of $R$
            UpdateDDG($R$)                          ▷ See description of algorithm for details
        end if
    end for
end procedure

function CheckPreconditions($G$, $S$)        ▷ $G$: data dependence graph, $S$: set of statements
    $P_1$: There should be a cycle of dependencies containing $S$ and an lcfd edge $E$.
    $P_2$: There should be no other lcfd edge apart from $E$ and an lcfd edge due to the update of the
          loop cursor variable.
    $P_3$: There should be no external dependencies.
    if $P_1$ and $P_2$ and $P_3$ hold then
        return true
    else
        return false
    end if
end function
```

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

Q: SELECT x, y, z from R

(a) Code Sample



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

Figure 3.6: Demonstration of preconditions for translation into F-IR

preconditions fail for a variable, the algorithm simply proceeds to attempt F-IR translation for other variables.

The function PARAMETERIZEQUERYREFS (line 23) takes as input an expression *expr*, which is a child of a *Loop* operator and may contain references to the query Q over which the loop iterates. It then replaces each reference to attributes of Q in *expr* with reference to a corresponding attribute of a tuple parameter variable t, where t has the same schema as the result of Q. The statement "$v = foldExpr$" (labeled s_{fold}) is a stub. The goal is to translate *foldExpr* into SQL. However, for the purpose of dependence analysis, we treat *foldExpr* as an algebraic expression.

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 3.5.1). However, dependences due to S_{dead} may cause preconditions to fail in an enclosing region. The procedure UPDATEDDG (line 29) reconstructs the DDG for R by incorporating newly generated/dead code to the region statements in each iteration of the loop inside LOOPTOFOLD. For example, statements in S_{dead} are ignored, and s_{fold} is included to update the DDG. The updated DDG may then be used in the next iteration of 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.1.

Consider the code sample shown in Figure 3.6(a). Figure 3.6(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 3.6(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 Section 3.5.4, we describe how some cases of dependent aggregations can be handled.

Note that *min*, *max* aggregations are typically represented in a loop using the following structure:
if expr OP v then v = expr, 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.

3.5 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.

3.5.1 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,
$fold[\ append,\ [\,],\ Q\ ] = Q$ (Rule T1.1)
$fold[\ insert,\ \{\},\ Q\ ] = \delta(Q)$ (Rule T1.2)

Rule T2 (Predicate push):
If $f^{\langle v \rangle, \langle t \rangle} = ?[\ pred(t),\ g\ ]$, then
$fold\ [f,\ id,\ \pi_L(\tau_Z(Q))\ ]$
$\equiv fold\ [\ g,\ id,\ \pi_L(\tau_Z(\sigma_{pred}(Q)))\ ]$

where $pred(t)$ is a predicate expression parameterized only on attributes of tuple t, and τ is the relational sort operator (Section 3.3.2). 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 $f^{\langle v \rangle, \langle t \rangle} = g(v,\ h(t.A))$, then
$fold\ [f,\ id,\ \pi_A(Q)\ ]$
$\equiv fold\ [\ g,\ id, \pi_{h(A)}(Q)]$

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 $f^{\langle v\rangle,\langle t\rangle} = \mathit{fold}\ [\ \mathit{append},\ v, \pi_L(\tau_{Z_2}(\sigma_{pred(t)}(Q_2)))\ ]$,
then $\mathit{fold}\ [f,\ [\,],\ \tau_{Z_1}(Q_1)\ ]$
$\equiv \pi_L(\tau_{Z_1,Q_1.K,Z_2}(Q_1 \bowtie_{pred} Q_2))$

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. [33], 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 $f^{\langle v\rangle,\langle t\rangle} = \mathit{fold}\ [\ \mathit{insert},\ v,\ \pi_L(\tau_{Z_2}(\sigma_{pred(t)}(Q_2)))\ ]$, then
$\mathit{fold}\ [f,\ \{\},\ \tau_{Z_1}(Q_1)\ ] \equiv \delta(\pi_L(Q_1 \bowtie_{pred} Q_2))$

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):

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

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

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 [49]. 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 Section 3.5.4, 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. Section 3.5.4 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 Section 3.5.4. Similar to database query optimizer rules, more transformations can be added to exploit other opportunities for inferring relational operations performed in imperative code.

3.5.2 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 3.5.1) 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. Thus, our techniques are able to partially extract SQL for as many variables as possible. This is an improvement over existing techniques [33], which attempt to translate an entire program fragment into SQL without considering slices for specific variables. Consequently, although our techniques currently do not support SQL translation of update operations, we are able to extract SQL partially for read-only queries that may be interspersed with update queries, as long as the updates do not introduce any critical dependencies for SQL translation.

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.

3.5.3 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 3.6(a). Our techniques will extract a separate query for the aggregated

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

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. We discuss our approach to a cost based rewriting in Chapter 4. This approach uses a top down search algorithm using an AND-OR DAG, based on the Volcano/Cascades query optimizer [52, 54].

3.5.4 Extensions

In this section, we present extensions to the F-IR representation and some more transformations that we have used in our implementation.

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 3.4). To address this common case, we add two new operators to F-IR:

- *tuple*: The *tuple* operator simply represents a tuple of expressions. It has $n > 1$ children, each of which is an expression in D-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 chapter, we represent the index *i* along with the *project* operator. For example, *project0* projects the first expression from its child *tuple*.

We now relax the precondition from Section 3.4 for converting loops to fold, 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 tuple (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[5]. 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 ORDER BY and LIMIT, or using a construct like SQLs RANK if the SQL dialect supports it. We omit details.

[5] Most databases today allow users to define user defined aggregates that can return a tuple.

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*.

Rule T6
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.

Outer Apply

Before we present Rule T7, we describe the *outer apply* construct. The *outer apply* construct [49], 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 in Q_1, *OApply* evaluates $Q_2(t)$ with appropriate values substituted for the parameters, and returns a union of all the results. If the result of $Q_2(t)$ is empty for a row, the row is returned with NULL values substituted for the fields obtained from $Q_2(t)$.

The `outer apply` syntax that we use in this chapter is of SQL Server. It 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 3.7.

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

where 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 set insertion, in place of *append*.

Exists/Not exists

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. A common example is checking for existence of a tuple in a table.

If the initial value of v is *false*, and value assigned to v in the loop is *true*, then we translate the loop to a *fold* with the folding function as logical OR. If the initial value and assigned value are reversed, the folding function is a logical AND. 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.

```
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 3.7: Cursor loop with nested scalar queries

```
((((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 3.8: Optimized query for data access in Figure 3.7

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 3.7. 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 chapter (Rule T7), a single SQL query can be extracted to fetch the required data for this code sample. The query is shown in Figure 3.8. As all the queries inside the cursor loop of Q1 in Figure 3.7 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 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 [29]. 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.

3.5.5 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 Section 3.5.4. 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 we shall discuss in Chapter 6, can be used to improve the applicability of our rules in such cases.

3.6 Related Work

We now briefly discuss and contrast our work with some of the related work in the area of holistic optimization of database applications.

Wiederman et al. [114] propose a source-to-source transformation technique that transforms an object oriented program with transparent persistence into an equivalent one with explicit queries. There has also been recent work on inferring SQL queries from procedural code using program synthesis by Cheung et al. [33]. Their approach generates possible equivalent SQL queries and uses the Sketch framework [101] 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 converting to SQL. The intermediate representation we use (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. [120] 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. [82] 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 such as projections, filters, joins and aggregates, while they focus on enabling parallelism and extracting map and reduce operations. Another point to note is that 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. [61] propose a syntax (JQS) through which certain complex SQL queries can be expressed using normal (imperative) Java constructs. Similarly, Giorgidze et al. [51] present a Haskell library which allows developers to express database queries using Haskell constructs. Such constructs are then translated into SQL for execution at the database. However, an important difference of our techniques from [61] and [51] is that our techniques automatically infer which parts of imperative code can be pushed into the database. In contrast, [61] and [51] require developers to provide this information, in a syntax that uses source language constructs. Some of the techniques of Cheney et al. [29], for translating XQuery to SQL, could be useful for handling print statements as discussed in Section 3.5.4. However, their goals and techniques are otherwise very different from ours.

Shi et al. [95] 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. [99] proposed techniques to algebrize 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. [56] proposed program analysis and transformation methods to exploit set oriented query execution to improve performance of iterative execution of parameterized queries. Ramachandra et al. [87] proposed a technique to prefetch query results across function calls. As discussed in Section 3.1, our techniques can be used in conjunction with the techniques of [56, 84, 87], to further enhance application performance.

3.7 Experimental Evaluation

Our implementation is in Java. We used the Soot framework [102] 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 [115] – an orchestration software, Matoso [71] – 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 [92] – a bidding system modeled after *ebay.com*, and RuBBoS [91] – 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 chapter, and QBS to denote techniques by Cheung et al. [33].

3.7.1 Applicability

The techniques presented in this chapter 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 other approaches with similar goals [33])

Cheung et al. [33] 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 3.1. The values in these columns denote the time taken for equivalent SQL extraction in cases where the system succeeded. The numbers for QBS have been taken from [33]. "–" 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 [33] and those presented in this chapter do not handle database updates. However, while [33] 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 (refer Section 3.4.2 for details). Similar to [33], 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.

Experiment 2 (Comparison with [56, 84, 87])

Our techniques can perform optimizations, which existing holistic optimization techniques like batching [56], prefetching of queries [87], and hybrid techniques [84] 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 3.1, 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 chapter 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 [84] can combine batching and prefetching.

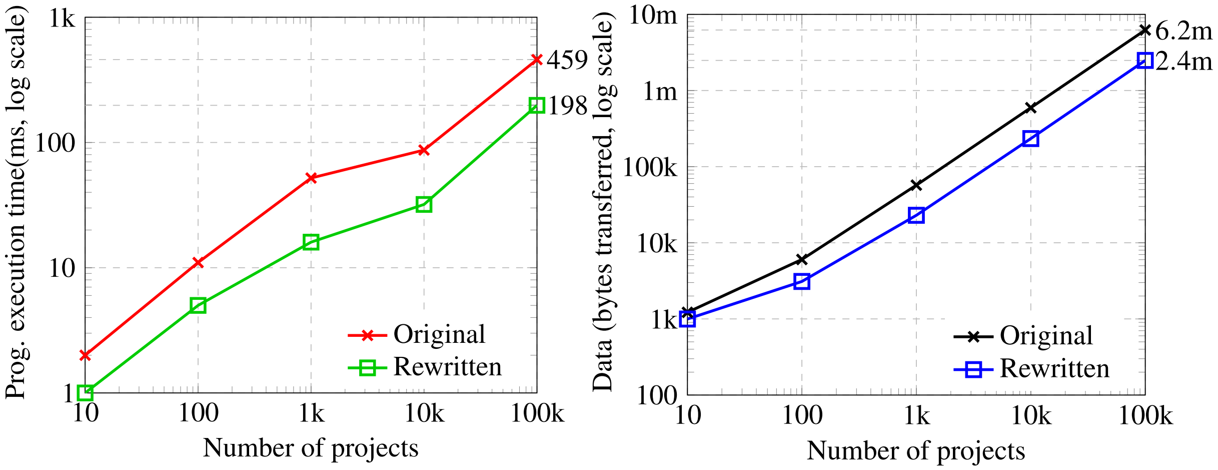
Sl.	File (Line No.)	QBS	EqSQL
1	ActivityService (401)	–	< 1
2	ActivityService (328)	–	< 1
3	Guidance Service (140)	–	< 1
4	Guidance Service (154)	–	< 1
5	ProjectService (266)	–	–
6	ProjectService (297)	19	< 1
7	ProjectService (338)	–	–
8	ProjectService (394)	21	< 2
9	ProjectService (410)	39	< 1
10	ProjectService (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 3.1: Comparison of time taken (s) by QBS (128GB RAM, 32 cores) and EqSQL (8GB RAM, 8 cores) for SQL extraction

Experiment 3 (Extraction of equivalent SQL for keyword search systems)

As mentioned in Section 3.1, 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 [41]. 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



(a) Time

(b) Data

Figure 3.9: Selection

AcadPortal. The benchmarks RuBiS and RuBBoS are simple servlet based applications with SQL queries. Our implementation was able to derive all the queries and dependences for these applications. The *JobPortal* application is much more complex. 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.

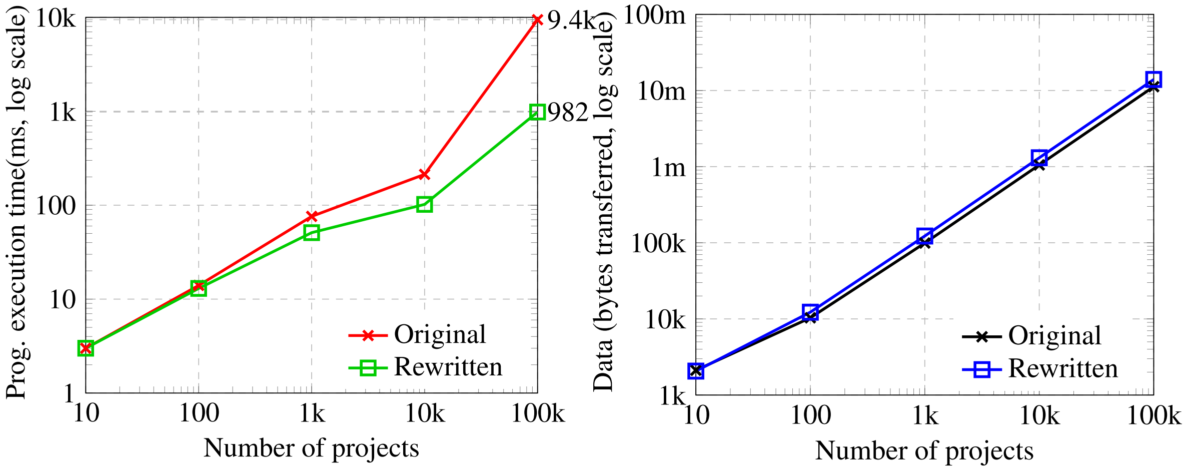
Approaches for batching and prefetching are not suitable for this purpose. QBS can be used, but we are unable to give a comparison as we do not have access to their source code.

3.7.2 Performance Impact

In this section, we first compare our tool with QBS [33] 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 3.1, for the code samples that we could successfully optimize, our techniques extract equivalent SQL in much less time than those of [33], 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.



(a) Time

(b) Data

Figure 3.10: Join

Experiment 5 (Selection)

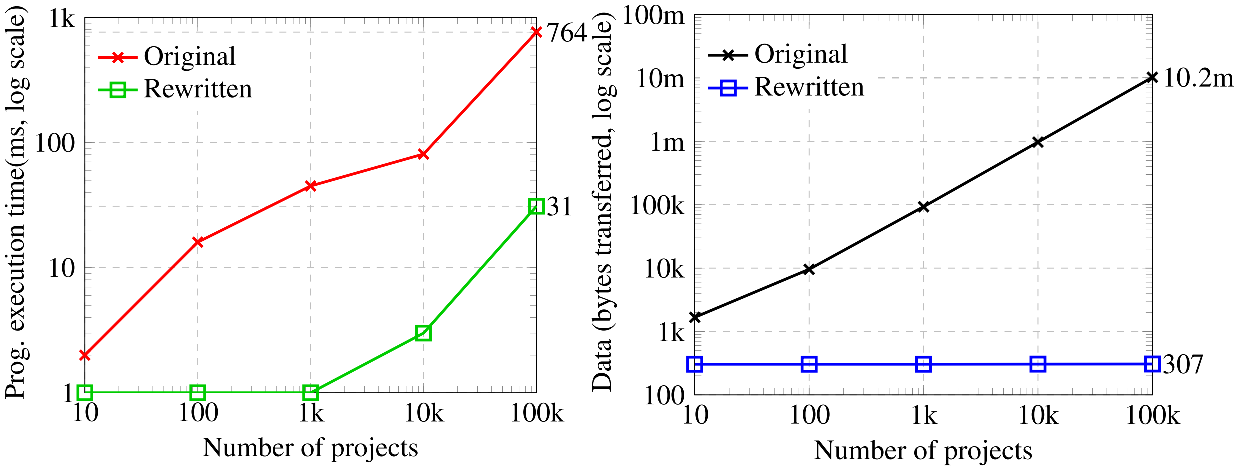
We use code based on sample #6 from Table 3.1 which computes the list of unfinished projects, where all tuples are fetched, and filtered inside Java code. Our tool optimizes it to fetch only the required tuples by pushing the predicate into the query. The results, shown in Figure 3.9, indicate that the 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.

Experiment 6 (Join)

We consider code based on sample #30 from Table 3.1 (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 3.10. 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 (11.2m in the original program vs 13.9m in the rewritten program) 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 3.2 which is based on a ranking page generator from Matoso. The results are shown in Figure 3.11. 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.



(a) Time

(b) Data

Figure 3.11: Aggregation

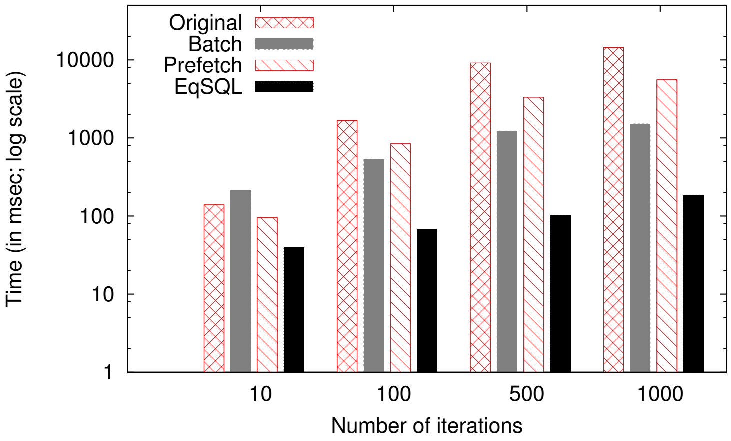


Figure 3.12: Comparison With Existing Techniques

Experiment 8 (Comparison with batching [56] and prefetching [87])

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 3.7 of Section 3.5.4. The results are shown in Figure 3.12. In the figure, *Batch* refers to optimizations using techniques described in [56], *Prefetch* refers to techniques in [87]. Though existing techniques do lead to improved performance, they are limited in their applicability, as discussed in Section 3.5.4. EqSQL enhances performance by upto two orders of magnitude compared to the original program, and upto one order of magnitude compared to other optimizations.

3.8 Summary

In this chapter, 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 3.5.5, future work includes extracting equivalent SQL for database update operations performed in imperative code.

Chapter 4

Cobra: A Framework for Cost-based Rewriting of Database Applications

Database applications are typically written using a mixture of imperative languages and declarative frameworks for data processing. Application 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 chapter, we propose a framework that automatically generates all equivalent alternatives of 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. The contents of this chapter have been published in [45].

4.1 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 of the program using program transformations, and choose the program with the least cost

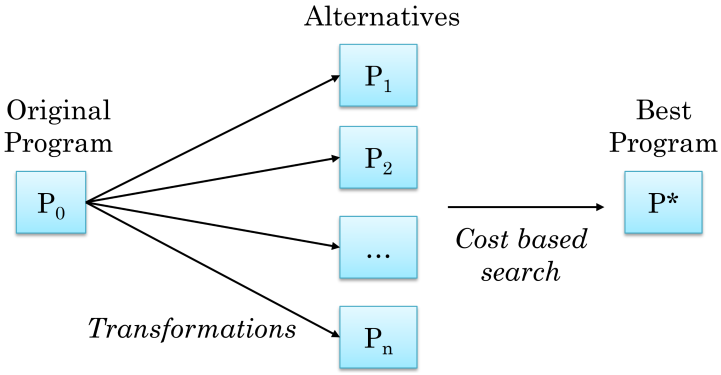


Figure 4.1: COBRA Illustration

depending on the context? In this chapter we propose the COBRA[1] framework to achieve this, as illustrated in Figure 4.1.

There has been work on rewriting data processing programs for improved performance using program transformations [33, 27, 84, 82]. However, existing techniques fail to consider all possible alternatives for cost based rewriting. They either apply the proposed transformations in a specific order [27], or carefully craft the transformation rules so that the rule set is confluent and terminating (such as the transformation rules we proposed in Chapter 3). 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 [82], Radoi et al. 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 [54, 52], which uses an AND-OR DAG representation (details in Section 4.3) 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 optimization, the Volcano/Cascades framework can be used with any algebra.

Such a framework can be used with program transformations based on expressions, as described in [104]. Examples of such transformations include 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 for direct integration into an algebraic framework like Volcano/Cascades.

In this chapter, we identify that *program regions* [74], which we used for transformations in Chapter 3, 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 4.3). 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 chapter, we restrict our attention to the use of

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

```
@Entity @Table(name=‘‘orders’’)
class Order{
 @Column(name=‘‘o_id");
 int o_id;

 @ManyToOne(targetEntity = Customer.class)
 @JoinColumn(name=‘‘customer_sk")
 Customer customer;

 ...
}
```

Figure 4.2: Hibernate object-relation mapping specification

COBRA for optimizing database applications.

Our contributions in this chapter 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 4.4).
- We illustrate the use of our framework for optimizing database applications using program transformations from Chapter 3 and other transformations from earlier work [87].
- We present a cost model (Section 4.6) to estimate the cost of database application programs, with a focus on 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 4.8) of COBRA on a real world application, to show the applicability of our techniques and their impact on application performance.

We present a motivating example in Section 4.2, and discuss the necessary background in Section 4.3. We discuss related work in Section 4.7, and summarize the chapter in Section 4.9.

4.2 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 [60], to motivate the need for COBRA.

Object relational mapping frameworks such as Hibernate enable access to the database using the same language as the application [33] 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 Figure 4.2, 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

```
processOrders(result) {
 result = {}; //empty collection
 for(o :  loadAll(Order.class)){
   cust = o.customer; // requires a separate query
   val = myFunc(o.o_id, cust.c_birth_year, ...);
   result.add(val);
 }
}
```

(a) P_0: Program using Hibernate ORM

```
processOrders(result) {
 result = {};
 joinRes = executeQuery(‘‘select * from orders o join
  customer c on o.o_customer_sk = c.c_customer_sk’’);
 for(r :  joinRes){
   val = myFunc(r.o_id, r.c_birth_year, ...);
   result.add(val);
 }
}
```

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

```
processOrders(result) {
 result = {};
 customers = loadAll(Customer.class);
 Utils.cacheByColumn(customers,‘c_customer_sk’);
                                   // refer footnote 3
 for(o :  loadAll(Orders.class);){
   cust = Utils.lookupCache(o.o_customer_sk);
   val = myFunc(o.o_id, cust.c_birth_year, ...);
   result.add(val);
 }
}
```

(c) P_2: P_0 rewritten to use prefetching

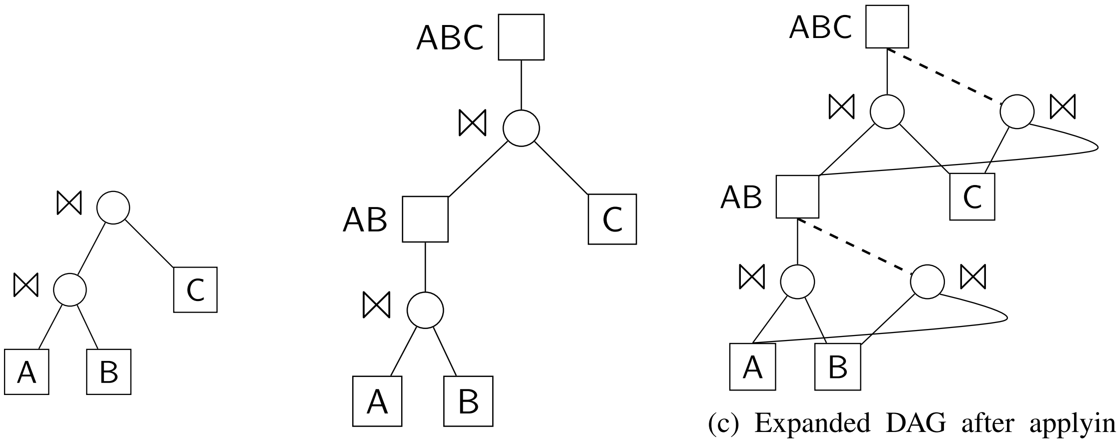
Figure 4.3: Alternative implementations of the same program

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 [33], and their use in reporting applications is not uncommon [62]. Inefficiencies due to the usage of ORM frameworks are also well known [28], and have been addressed by earlier optimization techniques such as techniques in [33] and our techniques from Chapter 3 (refer related work, Section 4.7).

Figure 4.3a shows a sample program using the Hibernate ORM that 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 [28].

To avoid this problem, a join query is usually suggested to fetch the required data, while



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

Figure 4.4: Representing alternative query rewrites using the AND-OR DAG

restricting the number of queries to one. This is shown in program P_1 in Figure 4.3b[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 use clever execution plans to minimize query execution time.

The join query shown in P_1 may lead to duplication of the customers 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 Figure 4.3c[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.

Current approaches for rewriting ORM applications with SQL, such as our techniques from Chapter 3 and those proposed in [33], 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 query results [87] 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, and the best program can be chosen in a cost-based manner.

4.3 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.

[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.

[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. 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.

4.3.1 Volcano/Cascades AND-OR DAG

Our discussion of AND-OR DAGs is based on [90]. 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 relational algebra expressions (queries), 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 Figure 4.4a. The AND-OR DAG representation for this query is shown in Figure 4.4b.

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 Figure 4.4c. 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 the commutativity transformation is cyclic. The Volcano/Cascades framework has efficient techniques for identifying duplicates, so the transformation process will terminate even in the presence of cyclic transformations.

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 Figure 4.4). The cost of any node in the AND-OR DAG is calculated using 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

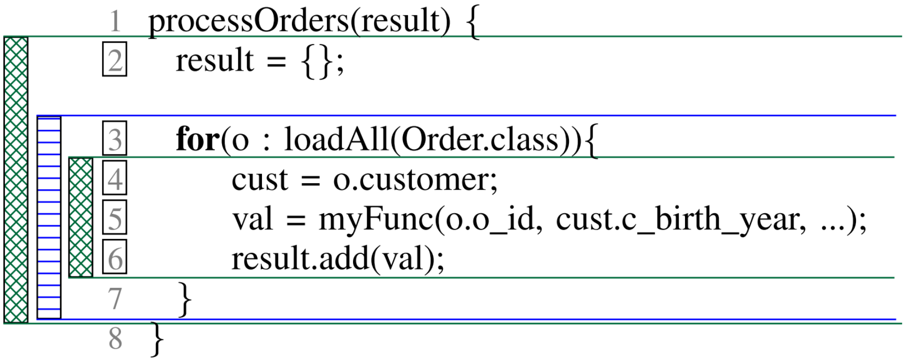
The plan corresponding to the least cost at the root node of the AND-OR DAG is the optimized plan.

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 [54, 52].

4.3.2 Program regions

We have discussed program regions in Chapter 3. Here, we present a brief recap. A region is any structured fragment in a program with a single entry and single exit [57]. 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

[4] Some approaches consider a basic block region as a sequence of statements. In this chapter, we consider each statement as a basic block, and treat a sequence of statements as a sequential region consisting of basic blocks. In



```
1  processOrders(result) {
2    result = {};

3    for(o : loadAll(Order.class)){
4        cust = o.customer;
5        val = myFunc(o.o_id, cust.c_birth_year, ...);
6        result.add(val);
7    }
8  }
```

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) – $P_0.L_{3-7}$

Figure 4.5: Program regions for program P_0 from Figure 4.3a

is called a *sub-region* and the containing region is called the *parent region*. The outermost region represents the entire program.

For example consider Figure 4.5, which replicates the program P_0 from Figure 4.3a with program regions shown alongside the code (note the naming convention for regions). The outermost region in Figure 4.5 is a sequential region $P_0.S_{2-7}$, which consists of 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 (breakup of $P_0.S_{4-6}$ into its basic blocks is not shown).

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.

4.4 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. Transformations on an expression generate alternative expressions to compute the same result. 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 straight forward. 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.

our implementation, we use an intermediate representation of bytecode [102], where each statement is represented using a three-address code [17].

4.4.1 Region as a State Transition

An imperative program can be considered as a specification for transition from one state to another. For example, the function *processOrders* from program P_0 (Figure 4.3a) specifies the following transition: *by the end* of *processOrders*, variable *result* contains the join of *orders* and *customers* with *myFunc* applied on each tuple. Alternative implementations of the program (such as P_1 and P_2 from Figure 4.3) 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 cache.

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

$$R : X_0 \rightarrow X_1 \tag{4.1}$$

where R is a region, X_0 is the state at the beginning of R and X_1 is the state at the end of R. We call X_0 the *input state*, and X_1 the *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 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 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 program.

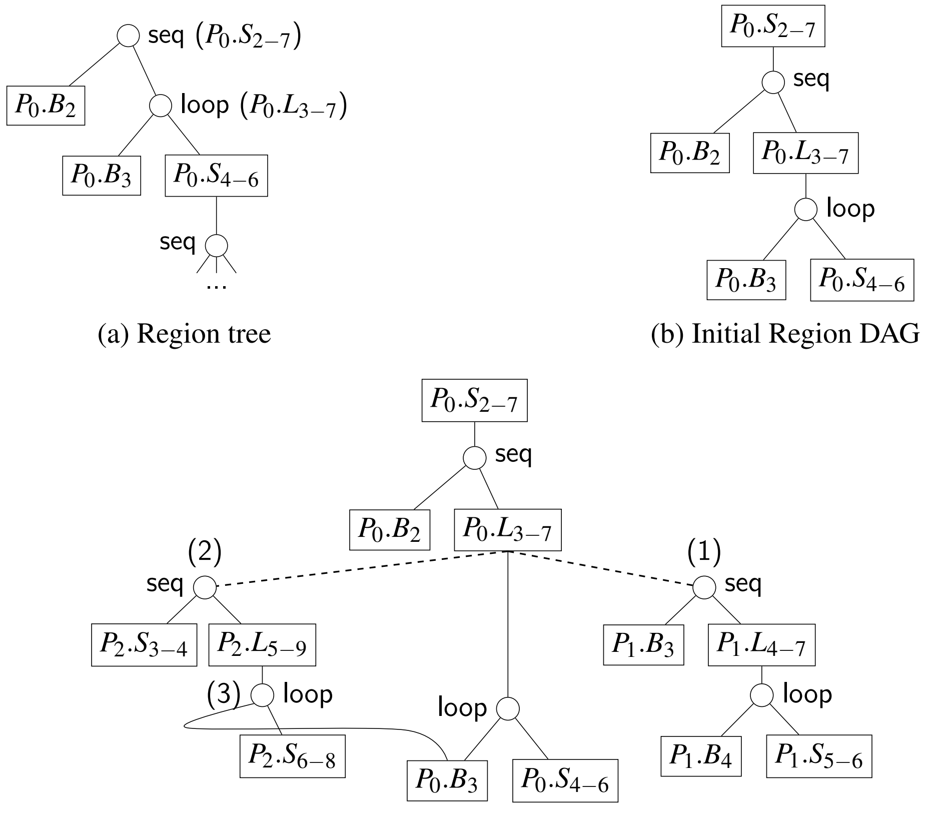
Our definition of a program region as a transition allows regions to be identified as parts of a program performing local computations that together combine to form the entire program, similar to sub-expressions in an algebraic expression. In this chapter, we use the term "computation in a region R"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

4.4.2 Region AND-OR DAG

Region AND-OR DAG, or simply Region DAG, is an AND-OR DAG that can represent 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 4.3.2. 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 Figure 4.5 is shown in Figure 4.6a.



(a) Region tree

(b) Initial Region DAG

(c) Expanded Region DAG

Figure 4.6: Representing alternative programs using the Region DAG

The leaves of a region tree are basic block regions. Intermediate nodes are operators that specify how results of sub-regions should be combined 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 Figure 4.6a, 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 (Figure 4.4a).

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 Figure 4.6a is shown in Figure 4.6b. 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 (Figure 4.4b).

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, Figure 4.6b 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 chapter, we use the transformations from Chapter 3 and [87], with some extensions. We discuss them in Section 4.5.

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 node for a 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 Figure 4.4c).

For example, program transformations such as SQL translation (refer Chapter 3) and prefetching [87] identify iterative query invocation inside a loop region in P_0, and rewrite the loop as shown in P_1 and P_2 respectively (refer Figure 4.3). They are represented in the Region DAG as shown in Figure 4.6c. Figure 4.6c 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 ($P_0.B_3$) 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 [104]. 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 present further comparison of our work with Peggy in Section 4.7.

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 4.5. 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 4.3.2) hinder the applicability of COBRA. For example, in a *try-catch* block, control may enter the *catch* block from any statement in the *try* block, so it does not conform to the region patterns that we identify. We refer to such fragments with complex control flow as *unstructured regions*. Another example of an unstructured region is an if-else with a complex predicate (combination of two or more predicates using AND (&&) or OR (||)), which is broken down into simpler predicates by the compiler

```
mySum(){
 sum = 0;
 cSum = new Map(); //creates a new empty map
 for(t :  executeQuery(‘‘select month, sale_amt
               from sales order by month’’)){
  sum = sum + t.sale_amt;
  cSum.put(month, sum);
 }
 print(sum);
 print(cSum);
}
```

Figure 4.7: Program M_0: Aggregations inside a loop

thereby resulting in complex control flow.

Alternatively, these unstructured regions may still be identified using a syntactic representation of the program such as an abstract syntax tree (AST). Unstructured regions may have structured regions within them. For example, a *try* block may contain an *if-else* statement. In such cases, the unstructured region can be encapsulated into a black box, and alternatives can be represented for other parts of the program nested within, and outside the unstructured region. We omit details.

4.5 Transformations using IR

In this section, we discuss the representation of alternative implementations of a program obtained using F-IR transformations from Chapter 3, and prefetching transformations from earlier work [87].

4.5.1 F-IR Recap

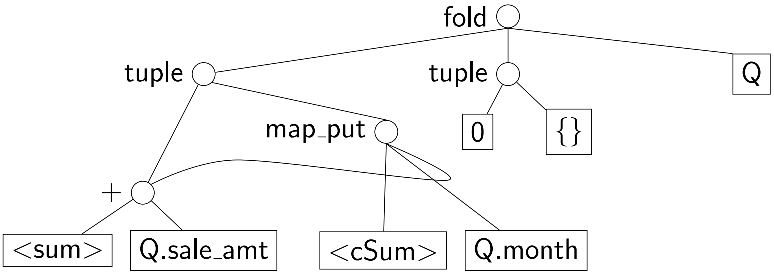
In Chapter 3, we proposed a DAG based intermediate representation named F-IR (*fold intermediate representation*) for imperative code that may also contain database queries. F-IR is based on program regions, and has been used to express program transformations for rewriting database applications by pushing relational operations such as selections, projections, joins, and aggregations that are implemented in imperative code to the database using SQL. The list of F-IR transformations used in this chapter are summarized in Figure 4.10[5].

Consider the program shown in Figure 4.7, which computes two aggregates – sum and cumulative sum (`cSum`) – using a loop over query results. The F-IR representation for the loop from Figure 4.7 is shown in Figure 4.8.

4.5.2 Integration into Region DAG

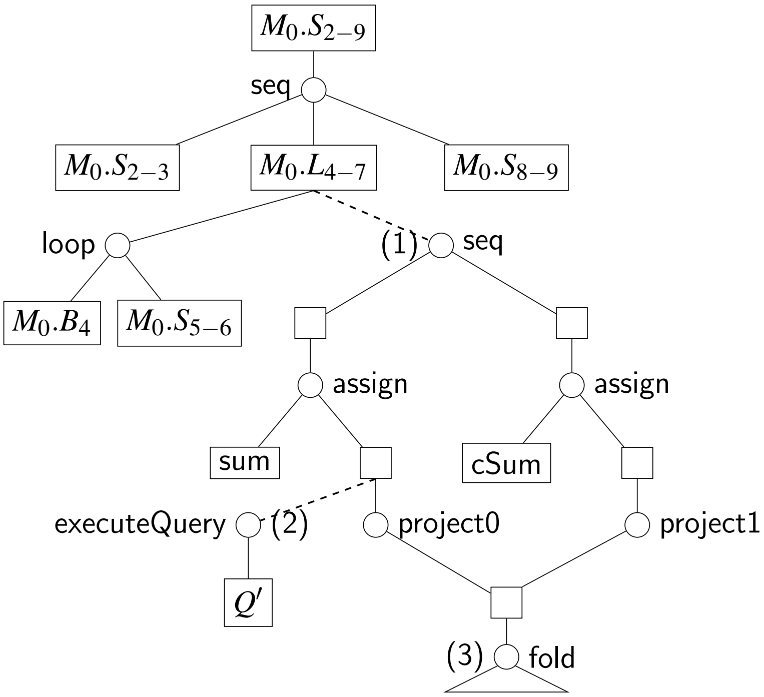
As we mentioned earlier in Section 4.5, F-IR is based on regions, and F-IR expressions represent values of program variables at the end of a region in terms of values available at the beginning

[5] Note that prefetching transformation from earlier work [87] has also been expressed as an F-IR transformations.



Q: select month, sale_amt from sales order by month

Figure 4.8: F-IR representation for the loop in Figure 4.7



Q': select sum(sale_amt) from sales
The *fold* expression (node 3) is as shown in Figure 4.8

Figure 4.9: Region DAG for Figure 4.7 after transforming to F-IR

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 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 (refer Chapter 3) 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.

Figure 4.9 shows the Region DAG for program M_0 from Figure 4.7. 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 Figure 4.8 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.

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

Figure 4.10: F-IR Transformation Rules (T1 to T5 are from Chapter 3)

4.5.3 Transformations

Transformations on F-IR expressions add further alternatives to the Region DAG. In Chapter 3, we proposed F-IR transformations with the aim of translating imperative code into SQL. These transformations are summarized in Figure 4.10 (T1 to T5)[6]. (There are other transformation rules in Chapter 3, 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 chapter, we propose transformations N1 and N2 (Figure 4.10) based on earlier work on prefetching [87]. Rule N1 transforms iterative lookup queries inside a loop into a prefetch[7] 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*).

We use Rule T5 to extract an SQL query for *sum*. 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. Using the cost model described in Section 4.6, 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 Chapter 3 for details on generating imperative code from F-IR.

[6] 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 Chapter 3.

[7] 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.

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)

Figure 4.11: Cost parameters

4.6 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 4.3.1).

In our work we focus on optimizing programs for data access. Figure 4.11 describes the parameters we consider for cost estimation. We use a parameter *amortization factor* (AF_Q) that estimates the number of invocations of a query Q, to allocate the prefetching cost across each invocation.

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 [90] (in our case, caching). 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 Figure 4.10), or (b) an entire relation is fetched without any filters/grouping. AF may be tuned individually for various queries depending on the particular application's workload.

We note however, that using prefetching, the first access to the query may have significantly higher latency compared to the original program, as typically a large number of rows are prefetched using a single query. This can be mitigated by prefetching asynchronously, and dynamically deciding to prefetch only after a certain number of accesses to minimize the overhead of prefetching. This is similar to the classical ski-rental problem [63] and has been applied earlier in the context of join optimizations in parallel data management systems [25]. Extending COBRA to adapt heuristics from [90] to efficiently handle alternatives generated due to caching is part of future work, and dynamic approaches for prefetching are part of future work.

Currently, we calculate cost only in terms of the time taken to execute the program. Our cost model can be extended to include other parameters such as CPU cost, memory usage etc., if needed. Using the parameters from the table above, the cost of 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:

$$C_Q = C_{NRT} + C_Q^F + max(N_Q * S_{row(Q)}/BW, C_Q^L - C_Q^F)$$

Prefetch

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

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

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 cost of each child.

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

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.

4.7 Related Work

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

Program transformations for database applications

In earlier work [26], techniques for optimizing database applications using static program analysis have been proposed as part of the DBridge system. Various program transformations such as batching, asynchronous query submission and prefetching [87, 84] have been incorporated in DBridge. DBridge also contains transformations for rewriting Hibernate applications using SQL for improved performance (Chapter 3); the QBS system [33] 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, the focus of this chapter 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 independent of DBridge with any set of program transformations.

There has been work on automatically rewriting programs with embedded queries for evolving schemas, using program transformations that are derived from schema modifications [96]. The transformations we considered in our work instead focus on rewriting queries for a fixed schema, by pushing computation from imperative code into SQL. However, COBRA can be used for cost based rewriting of applications using transformations from [96].

Enumeration and application of transformations

The Peggy compiler optimization framework [104] facilitates the application of transformations (compiler optimizations) in any order. It uses a data structure called PEG that operates 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 4.4). 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 chapter, is desired.

Pushing computation to the database

The Pyxis [31] 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 [15] allow developers to express relational database queries using the same language as the application, and later translate these queries into SQL [15, 55]. Our techniques focus on automatically identifying parts of imperative code that can be pushed into SQL, whereas [55] require developers to completely specify these queries, albeit in a syntax that uses source language constructs.

4.8 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 [90], which is based on Volcano/Cascades. COBRA leverages the region based analysis framework and program transformations from the DBridge system (refer [87] and Chapter 3) for optimizing database applications. DBridge internally uses the Soot framework [102] 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 of the program.

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 Figure 4.3), 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 4.5.2) 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 [3] 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 [2]. We used the following conditions: *slow remote network* (bandwidth: 500kbps, latency: 250ms (taken from [19])) 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 number of loop iterations (size of query result set). The cost of executing any other instruction apart from a query execution statement in the imperative program (C_z from Section 4.6) 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 get an estimate of query execution times, based on past executions of the queries. The cost metrics we used were provided to our system as a cost catalog file.

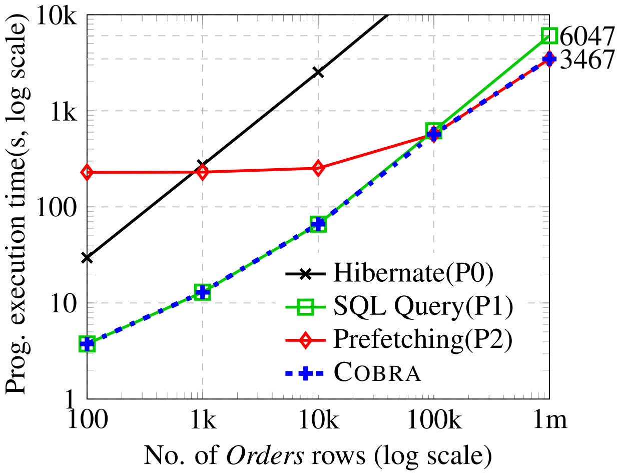


Figure 4.12: Performance of alternative implementations of Figure 4.3a – Slow remote network, varying Orders

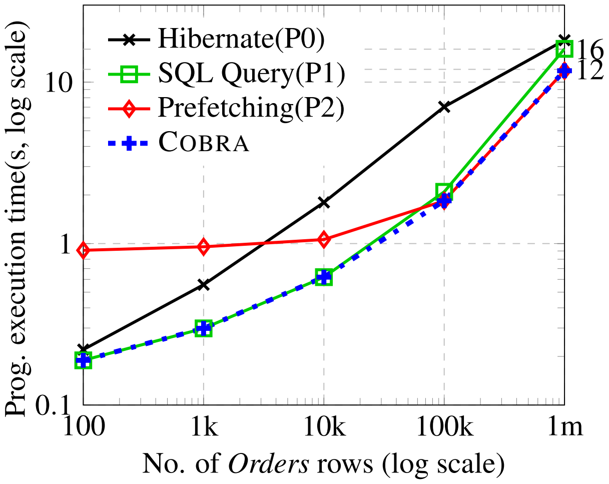


Figure 4.13: Performance of alternative implementations of Figure 4.3a – Fast local network, varying Orders

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. Figure 4.12 shows the actual running times of these programs, and the choice suggested by COBRA. At 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 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.

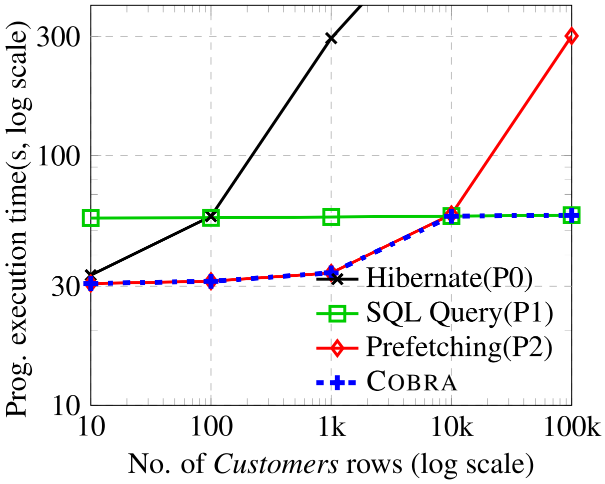


Figure 4.14: Performance of alternative implementations of Figure 4.3a – Slow remote network, varying Customers

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 Figure 4.13. Although P_2 performs better than P_1 at high cardinality of *Order* in both Figure 4.13 and Figure 4.12, 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 Figure 4.14 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 Figures 4.13 and 4.12, 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 [115], which uses the Hibernate ORM 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. Figure 4.15 lists for each category, the cost based choice of transformations and the number of cases identified. Details of each code fragment are listed

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 translation 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 whole table once and filter from 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

Figure 4.15: Cases for cost based based optimization in real world application (pattern id, description, number of cases)

in Figure 4.17.

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 Figure 4.16.

The x-axis in Figure 4.16 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

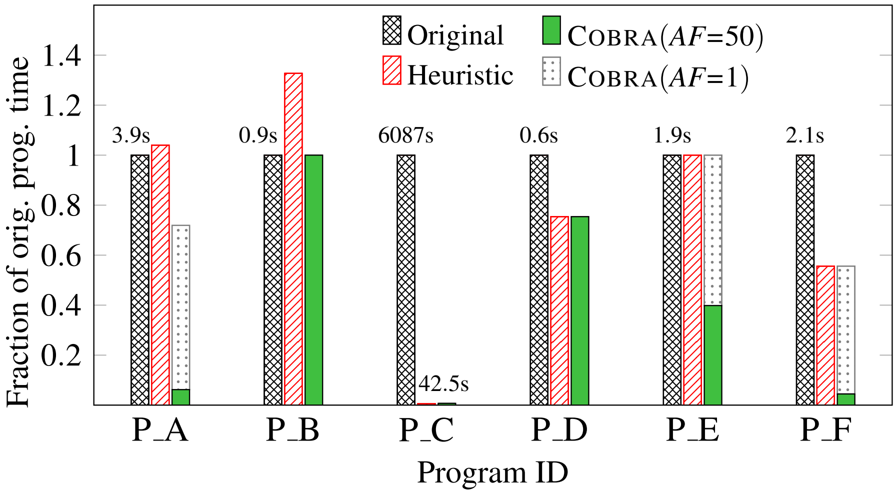


Figure 4.16: Performance benefits due to COBRA

Sl.No.	Pattern ID	File Name (Line Number)
1	A	ProjectService (1139)
2		TaskDescriptorService (198)
3		ConcreteWorkBreakdownElementService (144)
4	B	IterationService (139)
5		PhaseService (185)
6	C	ConcreteRoleAffectationService (60)
7		ConcreteTaskDescriptorService (312)
8		ConcreteTaskDescriptorService (1276)
9		ConcreteTaskDescriptorService (1302)
10		ConcreteWorkBreakdownElementService (63)
11		ConcreteWorkProductDescriptorService (445)
12		ParticipantService (129)
13		RoleService (15)
14		ActivityService (407)
15	D	IterationService (293)
16		PhaseService (307)
17		ActivityService (229)
18		RoleDescriptorService (276)
19		TaskDescriptorService (140)
20		TaskDescriptorService (142)
21		WorkProductDescriptorService (310)
22	E	ProjectService (346)
23		ProjectService (567)
24		ProjectService (647)
25		ProjectService(704)
26		ProcessService (1212)
27		ProcessService (1253)
28		ProcessService (1593)
29		ProcessService (1631)
30		ProcessService (1740)
31	F	ProcessService (406)
32		ProcessService (921)

Figure 4.17: Code fragments for cost based rewriting

original program, *Heuristic* – program rewritten using the heuristic: push as much computation as possible into SQL query (using transformations from Chapter 3), then prefetch the query results at the earliest program point, COBRA$_{(AF=50)}$ – program rewritten using COBRA with AF=50, and 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 (listed in Figure 4.10).

The results from Figure 4.16 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 Figure 4.15 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.

We now compare the plans (program implementations) chosen by the heuristic optimizer and COBRA. Remember that the heuristic optimizer pushes as much computation as possible into SQL. For programs that could entirely be translated into SQL (programs *C* and *D* in our workload), COBRA chose full SQL translation - same as the heuristic optimizer. For other programs where only a part of the program could be translated to SQL, COBRA differs from the heuristic optimizer.

For instance, in program *A* (nested loops with intermittent updates), the heuristic optimizer chose to translate the inner loop (which performs a filter) to SQL, whereas the outer loop could not be translated due to presence of updates. COBRA instead, chose to prefetch the inner loop query without the filter, thus eliminating iterative queries. Program *B* contained two aggregations inside a loop on a query result - a scalar count, and a collection that accessed all the rows in the query result. While the heuristic optimizer translated the count computation into an additional SQL aggregate query, COBRA chose the original program with a single query. Programs *E* and *F* originally each contained SQL queries with a where clause, where the predicate differed. While this was deemed optimal by the heuristic optimizer, COBRA rewrote the queries without the where clause (similar to program A) to leverage multiple accesses to the same relation and employed prefetching.

COBRA *Optimization Time*

The time taken for program optimization using COBRA is usually not a concern, as the program is optimized once for a particular environment and run multiple times. However, we note that in our experiments, the time taken for optimization was very small (<1s) for all programs.

Threats to validity

Our evaluation uses programs that use the Hibernate ORM as part of the Spring framework [103]. 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.

4.9 Summary

In this chapter, 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 optimization of imperative programs, and extend the Volcano/Cascades framework for optimizing algebraic expressions, to optimize programs with regions. Our experiments show that techniques in this chapter are widely applicable in real world applications with embedded data access, and provide significant performance improvements.

Our cost based search in this chapter assumes that the cost of a node in the AND-OR DAG is determined locally, i.e., using the costs of the operator and/or its children. However, the cost of a particular node may sometimes depend on other nodes in the AND-OR DAG apart from its children. For example, due to side effects such as caching, multiple nodes can access results from the cache without incurring further execution costs for the same expression after it is first computed. In our current implementation, we cache all query results (which are typically small in ORM applications) in memory and reuse them, thus incurring the cost of a query only once. In general, this is similar to the problem of multi-query optimization with materialization. Greedy heuristics for multi-query optimization proposed in [90] can be adapted to efficiently handle alternatives generated due to caching of multiple queries with large results. Implementing this in COBRA is an area of future work.

Chapter 5

Froid: Optimization of Imperative Programs in a Relational Database

For decades, RDBMSs have supported declarative SQL as well as imperative functions and procedures as ways for users to express data processing tasks. While the evaluation of declarative SQL has received a lot of attention resulting in highly sophisticated techniques, the evaluation of imperative programs has remained naïve and highly inefficient. Imperative programs offer several benefits over SQL and hence are often preferred and widely used. But unfortunately, their abysmal performance discourages, and even prohibits their use in many situations. We address this important problem that has hitherto received little attention.

We present Froid, an extensible framework for optimizing imperative programs in relational databases. Froid's novel approach automatically transforms entire User Defined Functions (UDFs) into relational algebraic expressions, and embeds them into the calling SQL query. This form is now amenable to cost-based optimization and results in efficient, set-oriented, parallel plans as opposed to inefficient, iterative, serial execution of UDFs. Froid's approach additionally brings the benefits of many compiler optimizations to UDFs with no additional implementation effort. We describe the design of Froid and present our experimental evaluation that demonstrates performance improvements of up to multiple orders of magnitude on real workloads. The contents of this chapter have been published as a collaborative work[1] in [86].

5.1 Introduction

SQL is arguably one of the key reasons for the popularity of relational databases today. SQL's declarative way of expressing intent has on one hand provided high-level abstractions for data processing, while on the other hand, has enabled the growth of sophisticated query evaluation techniques and highly efficient ways to process data.

Despite the expressive power of declarative SQL, almost all RDBMSs support procedural extensions that allow users to write programs in various languages (such as Transact-SQL, C#, Java and R) using imperative constructs such as variable assignments, conditional branching, and loops. These extensions are quite widely used. For instance, we note that there are of the

[1] This is joint work done during an internship with Microsoft Gray Systems Lab, Madison. My contribution was to build an end-to-end prototype for inlining multi-statement scalar UDFs using the idea of program regions, and perform an experimental evaluation on real customer workloads.

order of tens of millions of Transact-SQL (T-SQL) UDFs in use today in the Microsoft Azure SQL Database service, with billions of daily invocations.

UDFs and procedures offer many advantages over standard SQL. (a) They are an elegant way to achieve modularity and code reuse across SQL queries, (b) some computations (such as complex business rules and ML algorithms) are easier to express in imperative form, (c) they allow users to express intent using a mix of simple SQL and imperative code, as opposed to complex SQL queries, thereby improving readability and maintainability.These benefits are not limited to RDBMSs, as evidenced by the fact that BigData systems (Hive, Spark, etc.) support UDFs as well.

Unfortunately, the above benefits come at a huge performance penalty, due to the fact that UDFs are evaluated in a highly inefficient manner. It is a known fact amongst practitioners that UDFs are "evil" when it comes to performance considerations [109, 88]. In fact, users are advised by experts to avoid UDFs for performance reasons. The internet is replete with articles and discussions that call out the performance overheads of UDFs [108, 110, 111, 78, 79]. This is true for all popular RDBMSs, commercial and open source.

UDFs encourage good programming practices and provide a powerful abstraction, and hence are very attractive to users. But the poor performance of UDFs due to naïve execution strategies discourages their use. The root cause of poor performance of UDFs can be attributed to what is known as the 'impedance mismatch' between two distinct programming paradigms at play – the declarative paradigm of SQL, and the imperative paradigm of procedural code. Reconciling this mismatch is crucial in order to address this problem, and forms the crux of this chapter.

We present Froid, an extensible optimization framework for imperative code in relational databases. The goal of Froid is to enable developers to use the abstractions of UDFs and procedures without compromising on performance. Froid achieves this goal using a novel technique to automatically convert imperative programs into equivalent relational algebraic forms whenever possible. Froid models blocks of imperative code as relational expressions, and systematically combines them into a single expression using the *Apply* [49] operator, thereby enabling the query optimizer to choose efficient set-oriented, parallel query plans.

Further, we demonstrate how Froid's relational algebraic transformations can be used to arrive at the same result as that of applying compiler optimizations (such as dead code elimination, program slicing and constant folding) to imperative code. Although Froid's current focus is T-SQL UDFs, the underlying technique is language-agnostic, and therefore extending it to other imperative languages is quite straightforward, as we show in this chapter.

There have been some recent works that aim to convert fragments of database application code into SQL in order to improve performance, such as the techniques from Chapter3 and those from [33]. However, to the best of our knowledge, Froid is the first framework that can optimize imperative programs in a relational database by transforming them into relational expressions. While Froid is built into Microsoft SQL Server, its underlying techniques can be integrated into any RDBMS.

We make the following contributions in this chapter.

1. We describe the unique challenges in optimization of imperative code executing in relational databases, and analyze the reasons for their abysmal performance.

2. We describe the novel techniques underlying Froid, an extensible framework to optimize UDFs in Microsoft SQL Server. We show how Froid integrates with the query processing lifecycle and leverages existing sub-query optimization techniques to transform inefficient, iterative, serial UDF execution strategies into highly efficient, set-oriented, parallel plans.

3. We show how several compiler optimizations such as dead code elimination, dynamic slicing, constant propagation and folding can be expressed as relational algebraic transformations and simplifications that arrive at the same end result. Thereby, Froid brings these additional benefits to UDFs with no extra effort.

4. We discuss the design and implementation of Froid, and present an experimental evaluation on several real world customer workloads, showing significant benefits in both performance and resource utilization.

The rest of the chapter is organized as follows. Section 5.2 gives the background. Sections 5.3, 5.4, 5.5 and 5.6 describe Froid and its techniques. Design details are discussed in Section 5.7 followed by an evaluation in Section 5.8. We discuss related work in Section 5.12 and summarize the chapter in Section 5.13.

5.2 Background

In this section, we provide some background regarding the way imperative code is currently evaluated in Microsoft SQL Server and analyze the reasons for their poor performance. SQL Server primarily supports imperative code in two forms: UDFs and Stored Procedures (SPs). UDFs cannot modify the database state whereas SPs can. UDFs and SPs can be implemented in either T-SQL or Common Language Runtime (CLR). T-SQL expands on the SQL standard to include imperative constructs, various utility functions, etc. CLR integration allows UDFs and SPs to be written in any .NET framework language such as C# [34]. UDFs can be further classified into two types. Functions that return a single value are referred to as scalar UDFs, and those that return a set of rows are referred to as Table Valued Functions (TVFs). SQL Server also supports inline TVFs, which are single-statement TVFs analogous to parameterized views [48]. In this chapter we focus primarily on *Scalar T-SQL UDFs*. Extensions to support other imperative languages are discussed in Section 5.7.3.

5.2.1 Scalar UDF Example

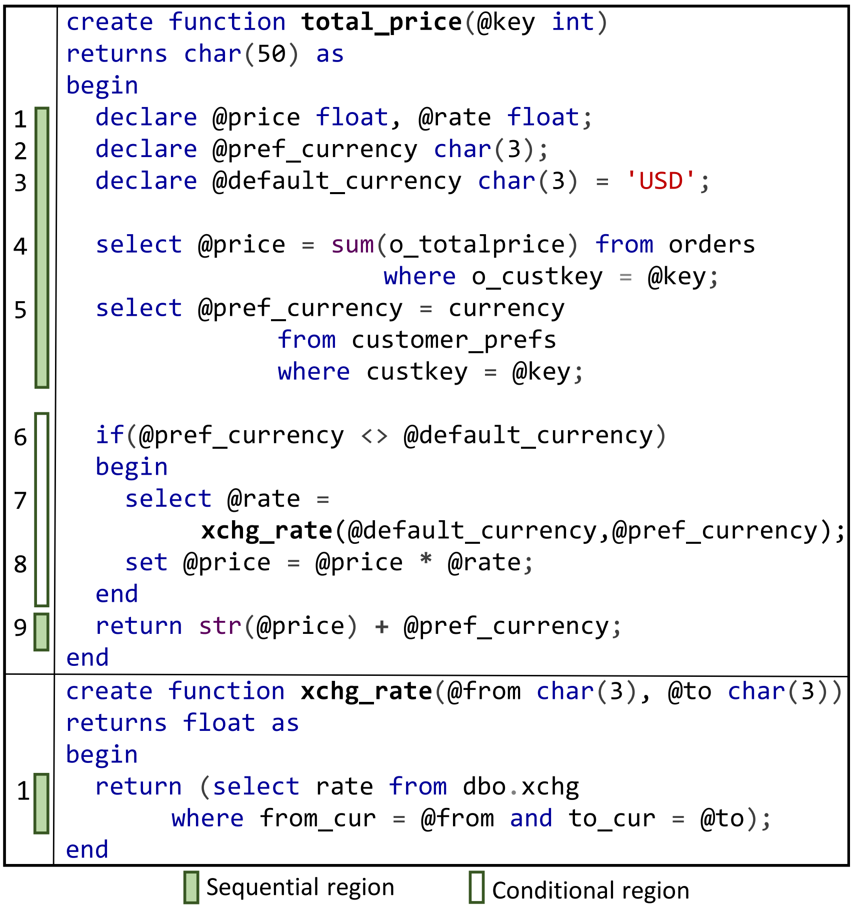
In SQL Server, UDFs are created using the CREATE FUNCTION statement [48] as shown in Figure 5.1. The function *total_price* accepts a customer key, and returns the total price of all the orders made by that customer. It computes the price in the preferred currency of the customer by looking up the currency code from the *customer_prefs* table and performs currency conversion if necessary. It calls another UDF *xchg_rate*, that retrieves the exchange rate between the two currencies. Finally it converts the price to a string, appends the currency code and returns it. Consider a simple query that invokes this UDF.

select *c_name,* ***dbo.total_price****(c_custkey)*
from *customer*;

For each customer, the above query displays the name, and the total price of all orders made by that customer. We will use this simple query and the UDFs in Figure 5.1 as an example to illustrate our techniques in this chapter.

5.2.2 UDF Evaluation in SQL Server

We now describe the life cycle of an SQL query that includes a UDF. At the outset we note that this is a simplified description with a focus on how UDFs are evaluated currently. We refer the



```
   create function total_price(@key int)
   returns char(50) as
   begin
1    declare @price float, @rate float;
2    declare @pref_currency char(3);
3    declare @default_currency char(3) = 'USD';

4    select @price = sum(o_totalprice) from orders
                       where o_custkey = @key;
5    select @pref_currency = currency
                  from customer_prefs
                  where custkey = @key;

6    if(@pref_currency <> @default_currency)
     begin
7      select @rate =
            xchg_rate(@default_currency,@pref_currency);
8      set @price = @price * @rate;
     end
9    return str(@price) + @pref_currency;
   end
```

```
   create function xchg_rate(@from char(3), @to char(3))
   returns float as
   begin
1    return (select rate from dbo.xchg
          where from_cur = @from and to_cur = @to);
   end
```

🟩 Sequential region □ Conditional region

Figure 5.1: Example T-SQL User defined functions

reader to [39, 18, 49] for details.

Parsing, Binding and Normalization: The query first goes through syntactic validation, and is parsed into a tree representation. This tree undergoes binding, which includes validating referenced objects and loading metadata. Type derivation, view substitution and optimizations such as constant folding are also performed. Then, the tree is normalized, wherein most common forms of subqueries are turned into some join variant. A scalar UDF that appears in a query is parsed and bound as a UDF operator. The parameters and return type are validated, and metadata is loaded. The UDF definition is not analyzed at this stage.

Cost-based Optimization: Once the query is parsed and normalized, the query optimizer performs cost-based optimization based on cardinality and cost estimates. Execution alternatives are generated using transformation rules, and the plan with the cheapest estimated cost is selected for execution. SQL Server's cost-based optimizer follows the design of the Volcano optimizer [53]. SQL Server reuses query plans for queries and UDFs by caching chosen plans. A cache entry for a UDF can be thought of as an array of plans, one for each statement in the UDF.

Execution: The execution engine is responsible for executing the chosen plan efficiently. Relational query execution invokes a scalar evaluation sub-system for predicates and scalar computations, including scalar UDFs [42]. The plan for the simple query in Section 5.2.1 is shown in Figure 5.2. For every tuple that is emitted by the Table Scan operator, the execution engine calls into the scalar evaluation sub-system to evaluate the scalar UDF *total_price*.

At this point, the execution context switches to the UDF. Now, the UDF can be thought of as a batch of statements submitted to the engine If the UDF contains SQL queries (e.g. lines 4 and

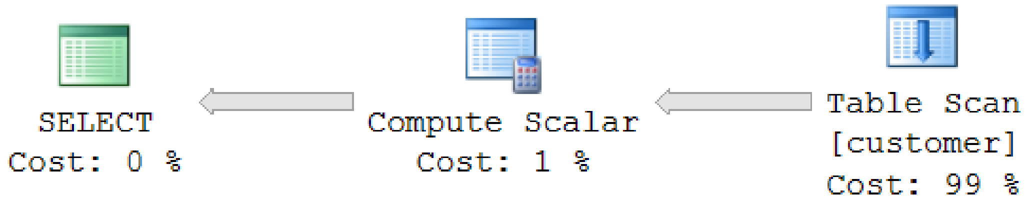


Figure 5.2: Query plan for the query in Section 5.2.1

5 of Figure 5.1), the scalar subsystem makes a recursive call back to the relational execution engine. Once the current invocation of the UDF completes, the context switches back to the calling query, and the UDF is invoked for the next tuple – this process repeats. During the first invocation of the UDF, each statement goes through compilation, and the plan for the UDF is cached. During subsequent invocations, the cached plan for the UDF is used.

5.2.3 Drawbacks in UDF Evaluation

We now enumerate the main causes for poor performance of UDFs. While we describe the reasons in the context of UDFs in SQL Server, they are mostly true for other RDBMSs as well, though the finer details may vary.

Iterative invocation: UDFs are invoked in an iterative manner, once per qualifying tuple. This incurs additional costs of repeated context switching due to function invocation, and mutual recursion between the scalar evaluation sub-system and relational execution. Especially, UDFs that execute SQL queries in their body (which is common in real workloads) are severely affected.

These iterative plans can be highly inefficient, since queries within the function body are executed multiple times, once for each invocation. This can be thought of as a nested loops join along with expensive context switches and overheads. As a consequence, the number of invocations of a UDF in a query has a huge impact on its performance. The query optimizer is rendered helpless here, since it does not look inside UDF definitions.

Lack of costing: Query optimizers treat UDFs as inexpensive black-box operations. During optimization, only relational operators are costed, while scalar operators are not. Prior to the introduction of scalar UDFs, other scalar operators were generally cheap and did not require costing. A small CPU cost added for a scalar operation was enough. This inadvertent simplification is a crucial cause of bad plan choices in cases where scalar operations are arbitrarily expensive, which is often true for scalar UDFs.

Interpreted execution: As described in Section 5.2.2, UDFs are evaluated as a batch of statements that are executed sequentially. In other words, UDFs are interpreted statement-by-statement.

Note that each statement itself is compiled, and the compiled plan is cached. Although this caching strategy saves some time as it avoids recompilations, each statement executes in isolation. No cross-statement optimizations are carried out, unlike in compiled languages. Techniques such as dead code elimination, constant propagation, folding, etc. have the potential to improve performance of imperative programs significantly. Naïve evaluation without exploiting such techniques is bound to impact performance.

Limitation on parallelism: Currently, SQL Server does not use intra-query parallelism in queries that invoke UDFs. Methods can be designed to mitigate this limitation, but they introduce additional challenges, such as picking the right degree of parallelism for each invocation

```
create function toyUDF
returns float as
begin
 declare @a float, @b float, @c float, @d float;
 select @a = sum(amt) from orders;
 set @b = @a + 100;
 set @c = @a * 1.1;
 set @d = c - b;
 return @d;
end
```

Figure 5.3: Simple UDF that reads a variable multiple times

of the UDF.

For instance, consider a UDF that invokes other SQL queries, such as the one in Figure 5.1. Each such query may itself use parallelism, and therefore, the optimizer has no way of knowing how to share threads across them, unless it looks into the UDF and decides the degree of parallelism for each query within (which could potentially change from one invocation to another). With nested and recursive UDFs, this issue becomes even more difficult to manage.

5.2.4 Prior Approaches

Before we discuss the Froid framework, we present the shortcomings of other approaches that we have considered, to motivate Froid's approach. Simhadri et al. [100] describe a technique to decorrelate queries in UDFs using extensions to the *Apply* operator [49, 42]. Froid's approach partly borrows its intuition from this work, but there are some key differences. First, Froid does not require any new operators or operator extensions unlike the approach of [100]. Second, their transformation rules are designed to be a part of a cost based optimizer. Froid, in contrast is designed as a precursor to query optimization. Third, they do not address vital issues such as handling multiple return statements and avoiding redundant computation of predicate expressions, which are found to be quite common in real workloads. In Froid, we improve on the work of [100] to address these problems.

In Chapter 3, we presented techniques for extracting algebraic representations for imperative code containing embedded queries. We prototyped these techniques to extract algebraic representations (F-IR) for UDFs, with the goal of inlining. However, we found that for some UDFs, the F-IR generated was huge and complex due to duplication of expressions. For example, consider the UDF `toyUDF` in Figure 5.3, which assigns the result of a query to a variable `a` and reads `a` multiple times.

Algebrizing `toyUDF` using techniques from Chapter 3 results in the following expression (shown as SQL):

```
((select sum(amt) from orders) * 1.1) -
((select sum(amt) from orders) + 100)
```

Note that the expression for `a` is repeated twice in the expression for the UDF resulting in repeated execution of the corresponding query, whereas in the original UDF, the query for `a` is evaluated only once. Our experience showed that with even moderately large UDFs, such duplication can make the resulting expression for the UDF undesirably large, rendering the query optimizer to fall back to sub-optimal evaluation strategies. In Froid, we use the concept of regions similar to techniques from Chapter 3, and construct algebraic expressions to minimize the size of input to the optimizer.

5.3 The Froid Framework

As mentioned earlier, Froid is an extensible, language-agnostic optimization framework for imperative programs in RDBMSs. The novel techniques behind Froid are able to overcome all the limitations described above. We now describe the intuition and high level overview of Froid. Then, with the help of an example, we walk through the process of optimizing UDFs in Sections 5.4 and 5.5.

5.3.1 Intuition

Queries that invoke UDFs, such as the one in Section 5.2.1 can be thought of as queries with complex sub-queries. In nested sub-queries, the inner query is just another SQL query (with or without correlation). UDFs on the other hand, use a mix of imperative language constructs and SQL, and hence are more complex. A key observation that we make here is that iterative execution of UDFs is similar to correlated evaluation of nested sub-queries.

Optimization of sub-queries has received a lot of attention in the database literature and industry (see Section 5.12 for details). In fact, many of the popular RDBMSs are able to transform correlated sub-queries into joins, thereby enabling the choice of set-oriented plans instead of iterative evaluation of sub-queries.

Given these observations, the intuition behind Froid can be succinctly stated as follows. *If the entire body of an imperative UDF can be expressed as a single relational expression R, then any query that invokes this UDF can be transformed into a query with R as a nested sub-query in place of the UDF.* We term this semantics-preserving transformation as *unnesting* or *inlining* of the UDF into the calling query.

Once we perform this transformation, we can leverage existing sub-query optimization techniques to get better plans for queries with UDFs. This transformation forms the crux of Froid. Note that although we use the term *inlining* to denote this transformation, it is fundamentally different compared to inlining in imperative programming languages.

5.3.2 The APPLY operator

Froid makes use of the *Apply* operator while building a relational expression for UDFs. Specifically, it is used to combine multiple relational expressions into a single expression. The *Apply* operator ($\mathscr{A}$) was originally designed to model correlated execution of sub-queries algebraically in SQL Server [49, 42]. It accepts a relational input R and a parameterized relational expression $E(r)$. For each row $r \in R$, it evaluates $E(r)$ and emits tuples as a join between r and $E(r)$. More formally, it is defined as follows [49]:

$$R \mathscr{A}^{\otimes} E = \bigcup_{r \in R} (\{r\} \otimes E(r))$$

where $\otimes$, known as the join type, is either cross product, left outer-join, left semijoin or left antijoin. SQL Server's query optimizer has a suite of transformation rules for sub-query decorrelation, which remove the *Apply* operator and enable the use of set-oriented relational operations whenever possible. Details with examples can be found in [49, 42, 100].

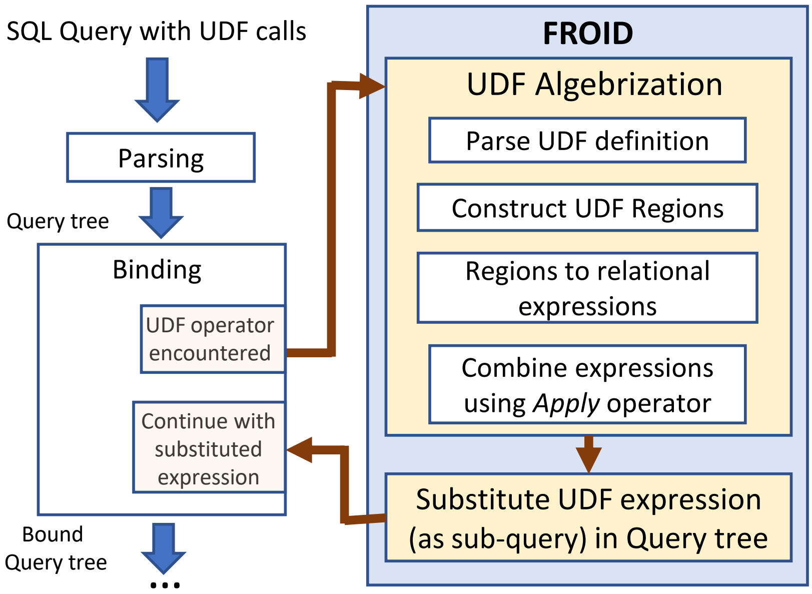

SQL Query with UDF calls
Parsing
Query tree
Binding
UDF operator encountered
Continue with substituted expression
Bound Query tree
...
FROID
UDF Algebrization
Parse UDF definition
Construct UDF Regions
Regions to relational expressions
Combine expressions using *Apply* operator
Substitute UDF expression (as sub-query) in Query tree

Figure 5.4: Overview of the Froid framework

5.3.3 Overview of Approach

For a UDF with a single RETURN statement in its body, such as the function *xchg_rate* in Figure 5.1, the transformation is straightforward. The body of such a UDF is already a single relational expression, and therefore it can be substituted easily into the calling context, like view substitution.

Expressing the body of a multi-statement UDF (such as the function *total_price* in Figure 5.1) as a single relational expression is a non-trivial task. Multi-statement UDFs typically use imperative constructs such as variable declarations, assignments, conditional branching, and loops. Froid models individual imperative constructs as relational expressions and systematically combines them to form one expression.

Figure 5.4 depicts the high-level approach of Froid, consisting of two phases: UDF algebrization followed by substitution. As a part of binding, the query tree is traversed and each node is bound, as described in Section 5.2.2. During binding, if a UDF operator is encountered, the control is transferred to Froid, and UDF algebrization is initiated. UDF algebrization involves parsing the statements of the UDF and constructing an equivalent relational expression for the entire UDF body (described in Section 5.4). This resulting expression is then substituted, or embedded in the query tree of the calling query in place of the UDF operator (described in Section 5.5). This query tree with the substituted UDF expression is bound using the regular binding process. If references to other (nested) UDF operators are encountered, the same process is repeated. This transformation finally results in a bound query tree, which forms the input to normalization and optimization.

5.3.4 Supported UDFs and queries

Froid currently supports the following imperative constructs in scalar UDFs.

- **DECLARE, SET:** Variable declaration and assignments.
- **SELECT:** SQL query with multiple variable assignments.
- **IF/ELSE:** Branching with arbitrary levels of nesting.
- **RETURN:** Single or multiple return statements.
- **UDF:** Nested/recursive function calls.

Imperative Statement (T-SQL)	**Relational expression (T-SQL)**
DECLARE $\{@var\ data\_type\,[= expr]\}[,\ldots n]$;	SELECT $\{expr \mid null$ AS $var\}[,\ldots n]$;
SET $\{@var = expr\}[,\ldots n]$;	SELECT $\{expr$ AS $var\}[,\ldots n]$;
SELECT $\{@var1 = prj\_expr1\}[,\ldots n]$ FROM sql_expr;	$\{$SELECT prj_expr1 AS $var1$ FROM $sql\_expr\}$; $[,\ldots n]$
IF $(pred\_expr)$ $\{t\_stmt;[\ldots n]\}$ ELSE $\{f\_stmt;[,\ldots n]\}$	SELECT CASE WHEN $pred\_expr$ THEN 1 ELSE 0 END AS $pred\_val$; $\{$SELECT CASE WHEN $pred\_val = 1$ THEN t_stmt ELSE $f\_stmt;\}[\ldots n]$
$RETURN\ expr$;	SELECT $expr$ AS $returnVal$;

Table 5.1: Relational algebraic expressions for imperative statements (using standard T-SQL notation from [107])

- **Others:** Relational operations such as EXISTS, ISNULL.

Table 5.1 (column 1) shows the supported constructs more formally. In Table 5.1, $@var$ and $@var1$ denote variable names, *expr* is any valid T-SQL expression including a scalar subquery; prj_expr represents a projected column/expression; *sql_expr* is any SQL query; *pred_expr* is a boolean expression; t_stmt and f_stmt are T-SQL statements [107].

Froid's techniques do not impose any limitations on the size or depths of UDFs and complexity of queries that invoke them. The only precondition for our transformations is that the UDF has to use the supported constructs. However, in practice, there are certain special cases where we partially restrict the application of our transformations; they are discussed in Section 5.7.2.

5.4 UDF Algebrization

We now describe the first phase of Froid in detail. The goal here is to build a single relational expression which is semantically equivalent to the UDF. This involves transforming imperative constructs into equivalent relational expressions and combining them in a way that strictly adheres to the procedural intent of the UDF. UDF algebrization consists of the following three steps.

5.4.1 Construction of Regions

First, each statement in the UDF is parsed and the body of the UDF is divided into a hierarchy of program *regions*. We have discussed program regions in Chapter 3. Here, we present a brief recap. A region is any structured fragment in a program with a single entry and single exit [57]. 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*[2]. Regions by definition contain other regions; the UDF as a whole is also a region.

Function *total_price* of Figure 5.1 is a sequential region R0 (lines 1-9). It is in turn composed of three consecutive sub-regions denoted R1, R2 and R3. R1 is a sequential region (lines 1-5), R2 is a conditional region (lines 6-8), and R3 is a sequential region (line 9) as indicated in Figure 5.1. Regions can be constructed in a single pass over the UDF body.

5.4.2 Relational Expressions for Regions

Once regions are constructed, the next step is to construct a relational expression for each region.

Imperative statements to relational expressions

Froid first constructs relational expressions for individual imperative statements, and then combines them to form a single expression for a region. These constructions make use of the *ConstantScan* and *ComputeScalar* operators in SQL Server [68]. The *ConstantScan* operator introduces one row with no column. A *ComputeScalar*, typically used after a *ConstantScan*, adds computed columns to the row.

[2] Some approaches consider a basic block region as a sequence of statements. In this chapter, we consider each statement as a basic block, and treat a sequence of statements as a sequential region consisting of basic blocks.

Variable declarations and assignments: The T-SQL constructs DECLARE, SET and SELECT fall under this category. These statements are converted into relational equivalents by modeling them as projections of computed columns in relational algebra as shown in Table 5.1 (rows 1, 2, 3). For example, consider line 3 of Figure 5.1:

set @*default_currency* = *'USD'*;

This is represented in relational form as

select *'USD'* **as** *default_currency*.

Observe that program variables are transformed into attributes projected by the relational expression. The RHS of the assignment could be any scalar expression including a scalar valued SQL query (when the SELECT construct is used). In this case, we construct a *ScalarSubQuery* instead of *ComputeScalar*. For example, the assignment statement in line 4 of Figure 5.1 is represented in relational form as

select(**select** *sum(o_totalprice)* **from** *orders*
where *o_custkey* = @*key)* **as** *price* .

Variable declarations without initial assignments are considered as assignments to *null* or the default values of the corresponding data types. Note that the DECLARE and SELECT constructs can assign to one or more variables in a single statement, but Froid handles them as multiple assignment statements. Modeling them as multiple assignment statements might lead to RHS expressions being repeated. However, common sub-expression elimination can remove such duplication in most cases.

Conditional statements: A conditional statement is typically specified using the IF-ELSE T-SQL construct. It consists of a predicate, a *true* block, and a *false* block. This can be algebrized using SQL Server's CASE construct as given in Table 5.1 (row 4). The *switch-case* imperative construct is also internally expressed as the IF-ELSE construct, and behaves similarly. Consider the following example:

```
if(@total > 1000)
    set @val = 'high';
else
    set @val = 'low';
```

The above statement is represented in relational form as

select(**case when** *total* > *1000* **then** *'high'*
else *'low'* **end** *)* **as** *val*.

This approach works for simple cases. For complex and nested conditional blocks, this approach may lead to redundant computations of the predicate thereby violating the procedural intent of the UDF. Re-evaluating a predicate multiple times not only goes against our principle of adherence to intent, but it might also hurt performance if the predicate is expensive to evaluate. Froid addresses this by assigning the value of the predicate evaluation to an implicit boolean variable (shown as *pred_val* in row 4 of Table 5.1). Subsequently, whenever necessary, it uses the CASE expression to check the value of this implicit boolean variable.

Return statements: Return statements denote the end of function execution and provide the value that needs to be returned from the function. Note that a UDF may have multiple return statements, one per code path. Froid models return statements as assignments to an implicit variable called *returnVal* (shown in row 5 of Table 5.1) followed by an unconditional jump to the end of the UDF. This unconditional jump means that no statement should be evaluated once the *returnVal* has been assigned a valid return value (note that *null* could also be a valid return value). Froid implicitly declares the variable *returnVal* at the first occurrence of a return statement. Any subsequent occurrence of a return statement is treated as an assignment to *returnVal*.

Region	Write-sets (Derived table schema)
R1	DT1 (price *float*, rate *float*, default_currency *char(3)*, pref_currency *char(3))*
R2	DT2 (price *float*, rate *float*)
R3	DT3 (returnVal *char(50)*)

Table 5.2: Derived tables for regions in function *total_price*.

Froid models unconditional jumps using the *probe* and *pass-through* functionality of the *Apply* operator [42]. The *probe* is used to denote whether *returnVal* has been assigned, and the *pass-through* predicate ensures that subsequent operations are executed only if it has not yet been assigned.

Although unconditional jumps could be modeled without using *probe* and *pass-through*, there are disadvantages to that approach. First, it increases the size and complexity of the resulting expression. This is because all successor regions of a return statement would need to be wrapped within a *case* expression. Second, the introduction of *case* expressions hinders the applicability of scalar expression folding and simplification. As we shall describe in Section 5.6, Froid brings optimizations such as constant folding and constant propagation to UDFs. The applicability of these optimizations would be restricted by the use of *case* expressions to model unconditional jumps.

Function invocations: Functions may invoke other functions, and may be recursive as well. Froid can unnest such nested function calls to achieve more gains. When a function invocation statement is encountered during UDF algebrization, Froid simply retains the UDF operator as the relational expression for that function. As part of the normal binding process in SQL Server, Froid is again invoked for the nested function, thereby inlining it as well. Some special cases with deeply nested/recursive functions, where we choose not to optimize are discussed in Section 5.7.2.

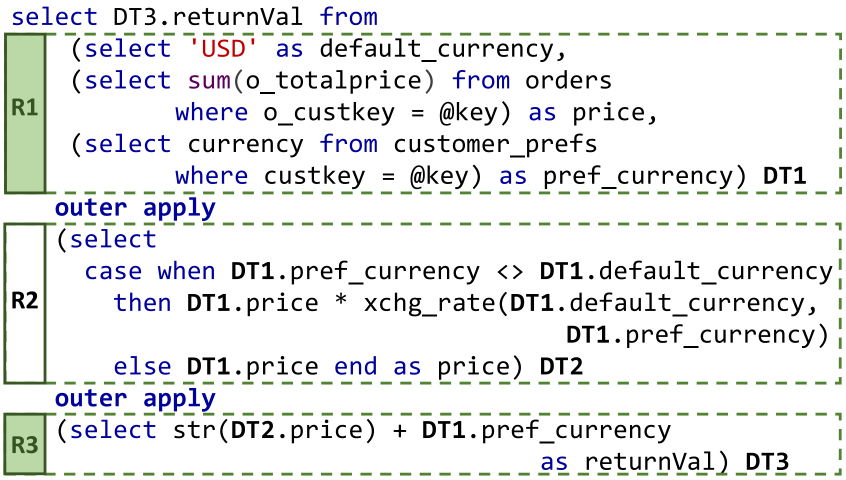
Others: Relational operations such as EXISTS, NOT EXISTS, ISNULL etc. can appear in imperative constructs such as the predicate of an IF-ELSE block. Froid simply uses the corresponding relational operators in these cases. In addition to the above constructs, we have prototyped algebrization of cursor loops. However, from our analysis of many real world workloads, we found that scalar UDFs with loops are quite rare (see Section 5.8). Therefore, we have currently disabled support for loops and may enable it in future.

Derived table representation

We now show how expressions for individual statements are combined into a single expression for a region using derived tables. A *derived table* is a statement-local temporary table created by a subquery. Derived tables can be aliased and referenced just like normal tables. Froid constructs the expression of each region as a derived table as follows.

Every statement in an imperative program has a ‘Read-Set’ and a ‘Write-Set’, representing sets of variables that are read from and written to within that statement respectively. Similarly, every region R can be seen as a compound statement that has a Read-Set and a Write-Set. Informally, the Read-Set of region R is the union of the Read-Sets of all statements within R. The Write-Set of R is the union of the Write-Sets of all statements within R.

A relational expression that captures the semantics of a region R has to expose the Write-Set of R to its subsequent regions. This is because the variables written to in region R would be read/modified in subsequent regions of the UDF. The Write-Set of region R is therefore used to



```
select DT3.returnVal from
    (select 'USD' as default_currency,
    (select sum(o_totalprice) from orders
           where o_custkey = @key) as price,
    (select currency from customer_prefs
           where custkey = @key) as pref_currency) DT1
   outer apply
   (select
     case when DT1.pref_currency <> DT1.default_currency
       then DT1.price * xchg_rate(DT1.default_currency,
                                      DT1.pref_currency)
       else DT1.price end as price) DT2
   outer apply
   (select str(DT2.price) + DT1.pref_currency
                                    as returnVal) DT3
```

Figure 5.5: Relational expression for UDF total_price

define the schema of the relational expression for R. The schema is defined by treating every variable in the Write-Set of R as an attribute. The implicit variable *returnVal* appears in the Write-Set of all regions that have a RETURN statement.

The Write-Sets of all the regions in function *total_price* of Figure 5.1 are given in Table 5.2. Using the schema, along with the relational expressions for each statement, we can construct a relational expression for the entire region R. A single *ConstantScan* followed by *ComputeScalar* operators, one per variable, results in a derived table with a single tuple. This derived table represents the values of all variables written to in R. The derived table aliases for regions R1, R2 and R3 are shown as DT1, DT2, and DT3 in Table 5.2.

5.4.3 Combining expressions using APPLY

Once we have a relational expression per region, we now proceed to create a single expression for the entire function. The relational expression for a region R uses attributes from its prior regions, and exposes its attributes to subsequent regions. Therefore, we need a mechanism to connect variable definitions to their uses and (re-)definitions.

Froid makes use of the relational *Apply* operator to systematically combine region expressions. The derived tables of each region are combined depending upon the type of the parent region. For a region R, we denote the corresponding relational expression as $E(R)$. For the *total_price* function in Figure 5.1, $E(R1) = DT1, E(R2) = DT2, E(R3) = DT3$.

Figure 5.5 shows the relational expression for the entire UDF. The dashed boxes in Figure 5.5 indicate relational expressions for individual regions R1, R2 and R3. Note that Froid's transformations are performed on the relational query tree structure and not at the SQL language layer. Figure 5.5 shows an SQL representation for ease of presentation.

The relational expression for a sequential region such as R0 is constructed using a sequence of *Apply* operators between its consecutive sub-regions i.e.,

$$E(R0) = (E(R1) \ \mathscr{A}^o \ E(R2)) \ \mathscr{A}^o \ E(R3)$$

The SQL form of this equation can be seen in Figure 5.5. The *Apply* operators make the values in DT1 available for use in DT2, the values in DT1 and DT2 available for DT3, and so on. We use the outer join type for these *Apply* operators ($\mathscr{A}^o$). In the presence of multiple return statements, we make use of *Apply* with *probe* (which internally uses left semijoin) and *pass-through* (outer join) [42].

Consider the variable @*pref_currency* as an example. It is first computed in R1, and hence is an attribute of the derived table DT1 (as shown in Figure 5.5). R2 uses this variable, but does not modify it. Therefore @*pref_currency* is not in the schema of DT2. All the uses of @*pref_currency* in R2 now refer to it as *DT1.pref_currency*. R3 also uses @*pref_currency* but does not modify it. The value of @*pref_currency* that R3 uses comes from R1. Therefore R3 also makes use of *DT1.pref_currency* in its computation of *returnVal*.

Observe that the expression in Figure 5.5 has no reference to the intermediate variable @*rate*. As a simplification, we generate expressions for variables only when they are first assigned a value, and we expose only those variables that are live at the end of the region (i.e., used subsequently). The @*rate* variable gets eliminated due to these simplifications. Finally, observe that the only attribute exposed by R0 (the entire function) is the *returnVal* attribute. This expression shown in Figure 5.5, is a relational expression that returns a value equal to the return value of the function *total_price*.

5.4.4 Correctness and Semantics Preservation

We now reason about the correctness of our transformations, and describe how they preserve the procedural semantics of UDFs. As described earlier, Froid first constructs equivalent relational expressions for individual imperative statements (Section 5.4.2). The correctness of these individual transformations directly follows from the semantics of the imperative construct being modeled, and the definition of the relational operations used to model it. The updated values of variables due to assignments are captured using derived tables consisting of a single tuple of values.

Once individual statements (and regions) are modeled as single-tuple relations (Section 5.4.2), performing an *Apply* operation between these relations results in a single-tuple relation, by definition. By defining derived table aliases for these single-tuple relations and using the appropriate aliases, we ensure that all the data dependencies are preserved. The relational *Apply* operator is composable, allowing us to build up more complex expressions using previously built expressions, while maintaining correctness.

In order to strictly adhere to the procedural intent of the UDF, Froid ensures that any computation in the relational equivalent of the UDF occurs only if that computation would have occurred in the procedural version of the UDF. This is achieved by (a) using the *probe* and *pass-through* extensions of the *Apply* operator to ensure that unconditional jumps are respected, (b) avoiding re-evaluation of predicates by assigning their results into implicit variables, and (c) using CASE expressions to model conditional statements.

5.5 Substitution and Optimization

Once we build a single expression for a UDF, the high-level approach to embed this expression into the calling query is similar to view substitution, typically done during binding. Froid replaces the scalar UDF operator in the calling query with the newly constructed relational expression as a scalar sub-query. The parameters of the UDF (if any) form the correlating parameters for the scalar sub-query. At substitution time, references to formal parameters in the function are replaced by actual parameters from the calling query. SQL Server has sophisticated optimization techniques for subqueries [49], which can be then leveraged. In fact, SQL Server

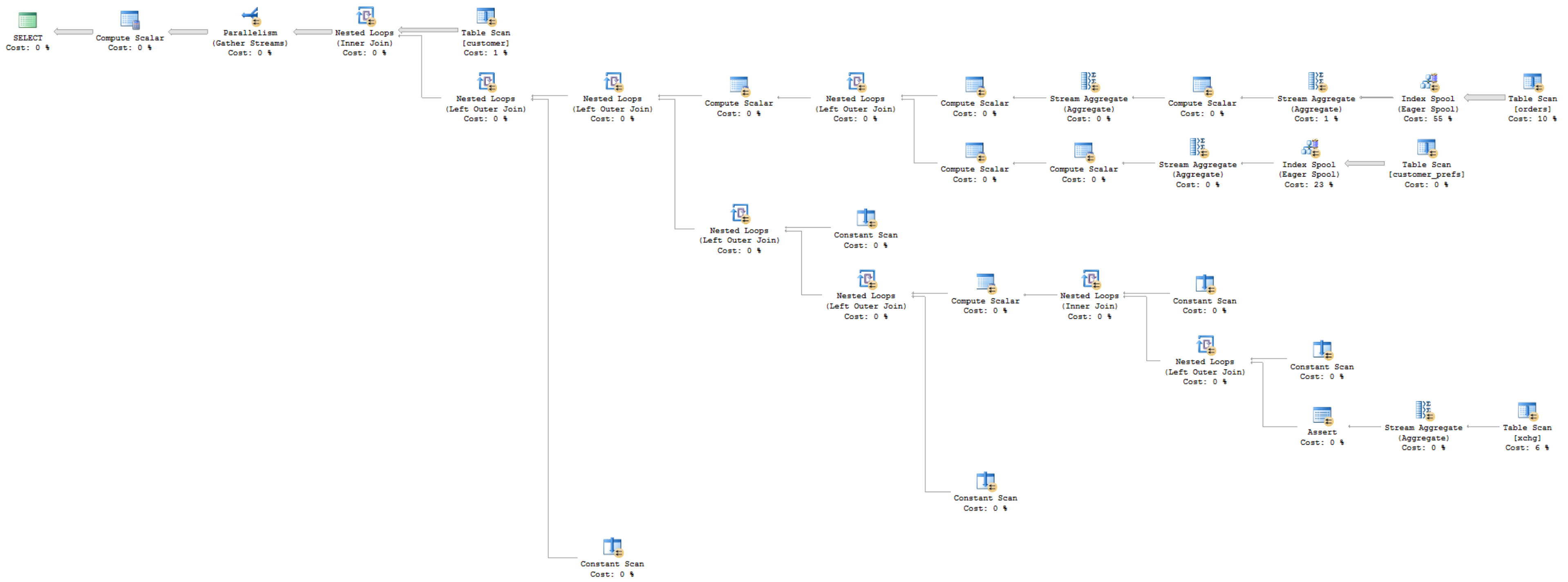


Figure 5.6: Plan for inlined UDF total_price of Figure 5.1

never chooses to do correlated evaluation for scalar valued sub-queries [42]. The plan (with Froid enabled) for the query in Section 5.2.1 is given in Figure 5.6. Although this plan is quite complex compared to the simple plan in Figure 5.2, it is significantly better. From the plan, we observe that the optimizer has (a) inferred the joins between *customer*, *orders*, *customer_prefs* and *xchg* – all of which were implicit, (b) inferred the appropriate *group by* operations and (c) parallelized the entire plan.

Froid overcomes all the four limitations in UDF evaluation enumerated in Section 5.2.3. First, the optimizer now decorrelates the scalar sub-query and chooses set-oriented plans avoiding iterative execution. Second, expensive operations inside the UDF are now visible to the optimizer, and are hence costed. Third, the UDF is no longer interpreted since it is now a single relational expression. Fourth, the limitation on parallelism no longer holds since the entire query including the UDF is now in the same execution context.

In a commercial database with a large user base such as SQL Server, making intrusive changes to the query optimizer can have unexpected repercussions and can be extremely risky. One of the key advantages of Froid's approach is that it requires no changes to the query optimizer. It leverages existing query optimization rules and techniques by transforming the imperative program into a form that the query optimizer already understands.

5.6 Compiler Optimizations

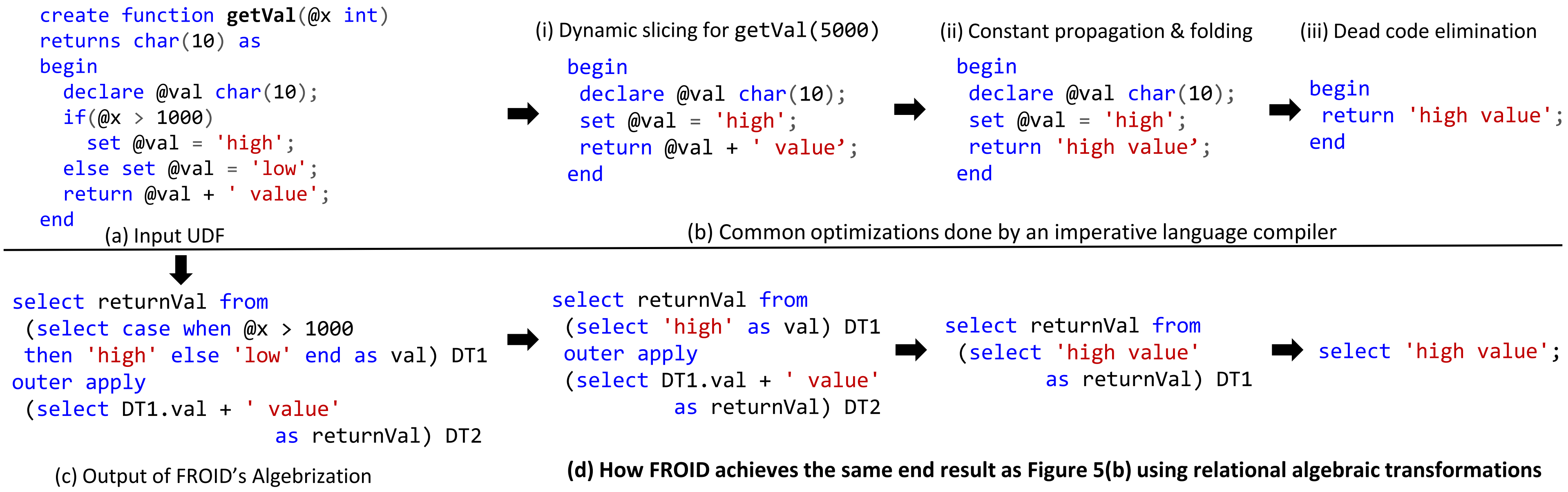
Froid's approach not only overcomes current drawbacks in UDF evaluation, but also adds a bonus: *with no additional implementation effort, it brings to UDFs the benefits of several optimizations done by an imperative language compiler.* In this section, we point out how some common optimization techniques for imperative code can be expressed as relational algebraic transformations and simplifications. Due to this, Froid is able to achieve these additional benefits by leveraging existing sophisticated query optimization techniques present in Microsoft SQL Server.

Using a simple example, Figure 5.7 illustrates the working of Froid's transformations in contrast with compiler optimizations[3]. The function *getVal* (Figure 5.7(a)) sets the value of variable @*val* based on a predicate. Starting with this UDF, a few common optimizations done by an imperative language compiler are shown in Figure 5.7(b) in three steps. Starting from the same input UDF, Figure 5.7(c) shows the output of Froid's algebrization. Then, Figure 5.7(d) shows relational algebraic transformations such as projection-pushdown and apply-removal that Froid uses, to arrive at the same result as the compiler optimizations in Figure 5.7(b).

5.6.1 Dynamic Slicing

Dynamic slicing is a program slicing technique that makes use of information about a particular execution of a program. A dynamic slice for a program contains a subset of program statements that will be visited in a particular execution of the program [64, 75]. For a particular invocation of the UDF in Figure 5.7(a), only one of its conditional branches is taken. For example, the dynamic slice for *getVal(5000)* is given in Figure 5.7(b)(i). As we can observe from Figure 5.7(d), Froid achieves slicing by evaluating the predicate ($@x > 1000$) at compile time and removing the case expression. In such cases where one or more parameters to a UDF are

[3] Note that in Figure 5.7, for ease of presentation, parts (c) and (d) are shown in SQL; these are actually transformations on the relational query tree representation.



```
create function getVal(@x int)
returns char(10) as
begin
  declare @val char(10);
  if(@x > 1000)
    set @val = 'high';
  else set @val = 'low';
  return @val + ' value';
end
```

(a) Input UDF

(i) Dynamic slicing for getVal(5000)

```
begin
 declare @val char(10);
 set @val = 'high';
 return @val + ' value’;
end
```

(ii) Constant propagation & folding

```
begin
 declare @val char(10);
 set @val = 'high';
 return 'high value’;
end
```

(iii) Dead code elimination

```
begin
 return 'high value';
end
```

(b) Common optimizations done by an imperative language compiler

```
select returnVal from
 (select case when @x > 1000
 then 'high' else 'low' end as val) DT1
outer apply
 (select DT1.val + ' value'
                    as returnVal) DT2
```

(c) Output of FROID’s Algebrization

```
select returnVal from
 (select 'high' as val) DT1
 outer apply
 (select DT1.val + ' value'
          as returnVal) DT2
```

```
select returnVal from
 (select 'high value'
        as returnVal) DT1
```

```
select 'high value';
```

(d) How FROID achieves the same end result as Figure 5(b) using relational algebraic transformations

Figure 5.7: Compiler optimizations as relational transformations

compile time constants, Froid simplifies the expression to use the relevant slice of the UDF by using techniques such as projection pushdown and scalar expression simplification.

5.6.2 Constant Folding and Propagation

Constant folding and constant propagation are related optimizations used by modern compilers [17, 64]. Constant folding is the process of recognizing and evaluating constant expressions at compile time. Constant propagation is the process of substituting the values of known constants in expressions at compile time.

SQL Server already performs constant folding within the scope of a single statement. However, since it does not perform cross-statement optimizations, constant propagation is not possible. This leads to re-evaluation of many expressions for every invocation of the UDF. Froid enables both constant propagation and folding for UDFs with no additional effort. Since the entire UDF is now a single relational expression, SQL Server's existing scalar simplification mechanisms simplify the expression. Figure 5.7(d) shows how the expression is simplified by evaluating both the predicate ($@x > 1000$) and then the string concatenation operation ('high' + ' value') at compile time, after propagating the constant 'high'.

5.6.3 Dead Code Elimination

Lines of code that do not affect the result of a program are called *dead code*. Dead code includes code that can never be executed (unreachable code), and code that only affects dead variables (assigned, but never read). As an example, suppose the following line of code was present in function *total_price* (Figure 5.1) between lines 3 and 4:

select *@t=count(*)* **from** *orders* **where** *o_custkey=@key*

The above line of code assigns the result of a query to a variable that is never used, and hence it is dead code. In our experiments, we found many occurrences of dead code. As UDFs evolve and grow more complex, it becomes hard for developers to keep track of unused variables and code. Dead code can also be formed as a consequence of other optimizations. Dead code elimination is a technique to remove such code during compilation [17]. Since UDFs are interpreted, most forms of dead code elimination are not possible.

Now let us consider how Froid handles this. Since the variable *@t* is in the Write-Set of R1, it appears as an attribute of DT1. However, since it is never used, there will be no reference to *DT1.t* in the final expression. Since there is an explicit projection on the *returnVal* attribute, *DT1.t* is like an attribute of a table that is not present in the final projection list of a query. Such attributes are aggressively removed by the optimizer using projection pushdown. Thereby, the entire sub-expression corresponding to the variable *@t* gets pruned out, eliminating it from the final expression.

In summary, we showed how Froid uses relational transformations to arrive at the same end result as that of applying compiler optimizations on imperative code. One might argue that compiler optimizations could be implemented for UDFs without using Froid's approach. However, that would only be a partial solution since it does not address inefficiencies due to iterative UDF invocation and serial plans.

We conclude this section by highlighting two other aspects. First, the semantics of the *Apply* operator allows the query optimizer to move and reuse operations as necessary, while preserving correlation dependencies. This achieves the outcome of *dependency-preserving statement reorderings* and *common sub-expression elimination* [17], often used by optimizing compilers.

Second, due to the way Froid is designed, these techniques are automatically applied across nested function invocations, resulting in increased benefits due to *interprocedural optimization*.

5.7 Design and Implementation

In this section, we discuss key design choices, trade-offs, and implementation details of the Froid framework.

5.7.1 Cost-based Substitution

One of the first questions we faced while designing Froid was to decide whether inlining of UDFs should be a cost-based decision. The answer to this question influences the choice of whether substitution should be performed during Query Optimization (QO) or during binding.

If inlining has to be a cost-based decision, it has to be performed during QO. If not, it can be done during binding. There are trade-offs to both these design alternatives. One of the main advantages to doing this during binding is that it is non-intrusive – the QO and other phases of query processing require no modifications. On the other hand, inlining during query optimization has the advantage of considering the algebrized UDF as an alternative, and making a cost-based decision of whether to substitute or not.

In Froid, we chose to perform inlining during binding due to these reasons: (a) Our experiments on real workloads showed that the inlined version performs better in almost all cases (see Section 5.8), questioning the need for cost-based substitution. (b) It is non-intrusive, requiring no changes to the query optimizer – this is an important consideration for a commercial database system, (c) Certain optimizations such as constant folding are performed during binding. Inlining during QO would require re-triggering these mechanisms explicitly, which is not desirable.

5.7.2 Imposing Constraints

Although Froid improves performance in most cases, there are extreme cases where it might not be a good idea. Algebrization can increase the size and complexity of the resulting query (see Section 5.8.1). From our experiments, we found that transforming a UDF with thousands of lines of code may not always be desirable as it could lead to a query tree with tens of thousands of operators. Additionally, note that the query invoking the UDF might itself be complex as well (see Section 5.8.2). Optimizing such a huge input tree makes the job of the query optimizer very hard. The space of alternatives to consider would increase significantly.

To mitigate this problem, we have implemented a set of algebraic transformations that simplify the query tree reducing its size when possible. However, in some cases, the query tree may remain huge even after simplification. This has an impact on optimization time, and also on the quality of the plan chosen. Therefore, one of the constraints we imposed on Froid is to restrict the size of algebrized query tree. In turn, this restricts the size of UDFs that are algebrized by Froid. Based on our experiments, we found that except for a few extreme cases (see Section 5.8.2), imposing this constraint still resulted in significant performance gains.

Nested and Recursive functions: Froid's transformations can result in deep and complex trees (in the case of deeply nested function calls), or never terminate at all (in the case of recursive UDFs), if it is not managed appropriately. Froid overcomes this problem by controlling the inlining depth based on the size of the algebrized tree. This allows algebrization of deeper nestings of smaller UDFs and shallow nestings of larger UDFs. Note that if there is a deep nesting of large UDFs (or recursive UDFs), algebrizing a few levels might still leave UDFs in the query. This still is highly beneficial in terms of reducing function call overheads and enabling the choice of set-oriented plans, but it does not overcome the limitation on parallelism (Section 5.2.3).

5.7.3 Supporting additional languages

Relational databases allow UDFs and procedures to be written in imperative languages other than procedural SQL, such as C#, Java, R and Python. Although the specific syntax varies across languages, they all provide constructs for common imperative operations such as variable declarations, assignments and conditional branching. Froid is an extensible framework, designed in a way that makes it straightforward to incrementally add support for more languages and imperative constructs.

Froid models each imperative construct as a class that encapsulates the logic for algebrization of that construct. Therefore, adding support for additional languages only requires (a) plugging in a parser for that language and (b) providing a language-specific implementation for each supported construct. The framework itself is agnostic to the language, and hence remains unchanged. As long as the UDF is written using supported constructs, Froid will be able to algebrize them as described in this chapter.

Note that while translating from a different language into SQL, data type semantics need to be taken into account to ensure correctness. Data type semantics vary across languages, and translating to SQL might lead to loss of precision, and sometimes different results.

5.7.4 Implementation Details

We now briefly discuss some special cases and other implementation details.
Security and Permissions Consider a user that does not have *execute* permissions on the UDF, but has *select* permissions on the referenced tables. Such a user will be able to run an inlined query (since it no longer references the UDF), even though it should be disallowed. To mitigate this issue, Froid enlists the UDF for permission checks, even if it was inlined. Conversely, a user may have *execute* permission on the UDF, but no *select* permissions on the referenced tables. In this case, by inlining, that user is unable to run the query even though it should be allowed. Froid handles this similar to the way view permissions are handled.

Plan cache implications: Consider a case where a user with administrative privileges runs a query involving this UDF, and consequently the inlined plan is now cached. Subsequently, if a user without UDF *execute* permissions but with *select* permissions on the underlying tables runs the same query, the cached plan will run successfully, even though it should not. Another implication is related to managing metadata version changes and cache invalidation. Consider the case as described above, where an inlined plan is cached. Now, if the user alters or drops the UDF, the UDF is changed or no longer available. Therefore, any query that referred to this UDF should be removed from the plan cache. Both these issues are solved by enlisting the UDF in schema and permission checks, even if it was algebrized.

Type casting and conversions: SQL Server performs implicit type conversions and casts in many cases when the datatypes of parameters and return expressions are different from the

Workload	W1	W2
Total # of scalar UDFs	178	93
# UDFs optimizeable by Froid	151 (85%)	86 (92.5%)
UDF lines of code (avg,min,max)	(21,6,113)	(26,7,169)

Table 5.3: Applicability of Froid on two customer workloads

declared types. In order to preserve the semantics as before, *Froid* explicitly inserts appropriate type casts for actual parameters and the return value.

Non-deterministic intrinsics: UDFs may invoke certain non-deterministic functions such as GETDATE(). Inlining such UDFs might violate the user's intent since it may invoke the intrinsic function once-per-query instead of once-per-tuple. Therefore, we disable transforming such UDFs.

5.8 Evaluation

We now present some results of our evaluation of Froid on several workloads and configurations. Froid is implemented in SQL Server 2017 in about 1.5k lines of code. For our experiments, SQL Server 2017 with Froid was run on Windows Server 2012(R2). The machine was equipped with Intel Xeon X5650 2.66 Ghz CPU (2 processors, 6 cores each), 96 GB of RAM and SSD-backed storage.

5.8.1 Applicability of Froid

We have analyzed several customer workloads from Azure SQL Database to measure the applicability of Froid with its currently supported constructs. We are primarily interested in databases that make good use of UDFs and hence, we considered the top 100 databases in decreasing order of the number of UDFs present in them. Cumulatively, these 100 databases had 85329 scalar UDFs, out of which Froid was able to handle 51047 (59.8%). The UDFs that could not be transformed contained constructs not supported by Froid. We also found that there are 10526 customer databases with more than 50 UDFs each, where Froid can inline more than 70% of the UDFs. The sizes of these UDFs range from a single line to 1000s of lines of code. These numbers clearly demonstrate the wide applicability of Froid.

In order to give an idea of the kinds of UDFs that are in these proprietary workloads, we have included a set of UDFs in Section 5.9. These UDFs have been modified to preserve anonymity, while retaining program structure. We have randomly chosen two customer workloads (referred to as W1 and W2) for deeper study and performance analysis. The UDFs have been used with no modifications, and there were no workload-specific techniques added to Froid. As summarized in Table 5.3, Froid is able to transform a large fraction of UDFs in these workloads (85% and 92.5%). As described in Section 5.7, UDF algebrization results in larger query trees as input to query optimization. The largest case in W2 resulted in more than 300 imperative statements being transformed into a single expression, having more than 7000 nodes. Note that this is prior to optimizations described in Section 5.6. This illustrates the complexity of UDFs handled by Froid.

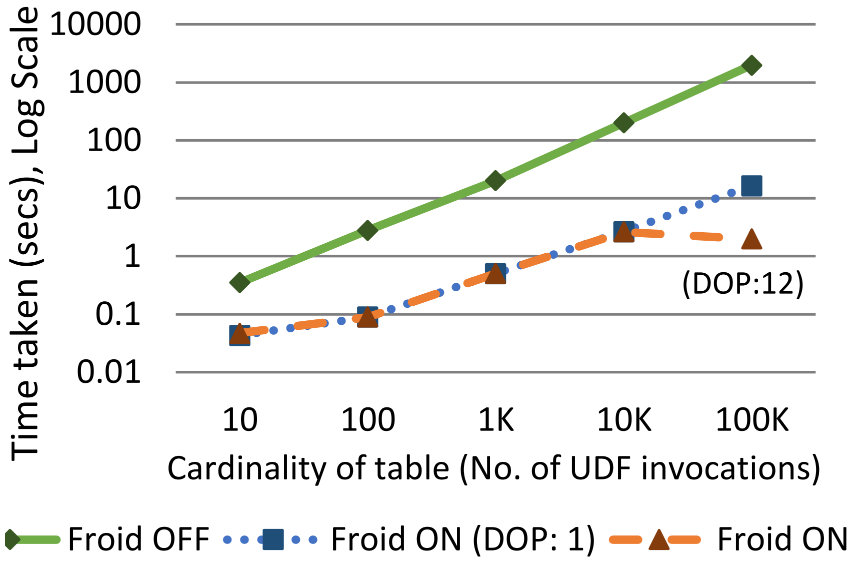


Figure 5.8: Varying the number of UDF invocations

5.8.2 Performance improvements

We now present a performance evaluation of Froid on workloads W1 and W2. Since our primary focus is to measure the performance of UDF evaluation, the queries that invoke UDFs are kept simple so that UDF execution forms their main component. Evaluation of complex queries with UDFs is considered in Section 5.8.2.

Number of UDF invocations

The number of times a UDF is invoked as part of a query has a significant impact on the overall query performance. In order to compare the relationship between the number of UDF invocations and the corresponding performance gains, we consider a function F1 (which in turn calls another function F2). F1 and F2 are functions adapted from workload W1, and their definitions are given in Section 5.9. We use a simple query to invoke this UDF, of the form

select *dbo.F1(T.a, T.b)* **from** *T*

Since the UDF is invoked for every tuple in *T*, we can control the number of UDF invocations by varying the cardinality of *T*. Figure 5.8 shows the results of this experiment conducted with a warm cache. The x-axis denotes the cardinality of table *T* (and hence the number of UDF invocations), and the y-axis shows the time taken in seconds, in log scale. Note that in this experiment, the time shown in the y-axis does not include query compilation time, since the query plans were already present in the cache.

We vary the cardinality of *T* from 10 to 100000. With Froid disabled, we observe that the time taken grows with cardinality (the solid line in Figure 5.8). With Froid enabled, we see an improvement of one to three orders of magnitude (the dashed line). The advantages start to be noticeable right from a cardinality of 10.

Impact of parallelism

As described in this chapter, Froid brings the benefits of set-oriented plans, compiler optimizations, and parallelism to UDFs. In order to isolate the impact of parallelism from the rest of

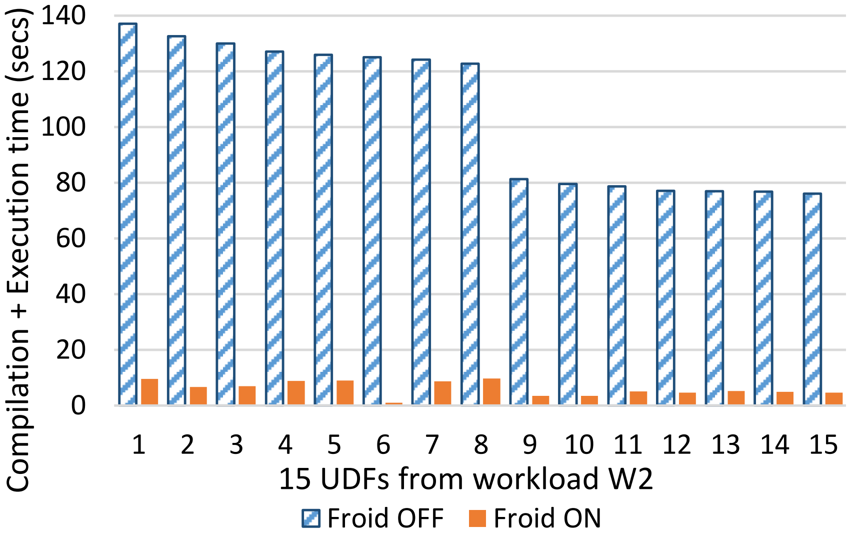


Figure 5.9: Elapsed time for Compilation and execution (using cold plan cache)

the optimizations (since enabling parallelism is a by-product of Froid's transformations), we conducted experiments where we enabled Froid but limited the Degree Of Parallelism (DOP). The dotted line in Figure 5.8 shows a result of this experiment. It includes all the optimizations of Froid, but forces the DOP to 1 using a query hint. For this particular UDF, SQL Server switches to a parallel plan when the cardinality of the table is greater than 10000 (indicated by the dashed line). The key observation we make here is that even without parallelism, Froid achieves improvements up to two orders of magnitude.

Compile time overhead

Since Froid is invoked during query compilation, there could be an increase in compilation time. This increase is not a concern as it is offset by the performance gains achieved. To quantify this, we measured the total elapsed time including compilation and execution by clearing the plan cache before running queries. This keeps the buffer pool warm, but the plan cache cold. The results of this experiment on 15 randomly chosen UDFs (sorted in descending order of elapsed time) of workload W2 are shown in Figure 5.9. The y-axis shows total elapsed time which includes compilation and execution. We observe gains of more than an order of magnitude for all these UDFs. Note that the compilation time of each of these UDFs is less than 10 seconds.

Complex Analytical Queries With UDFs

In the above experiments, we kept the queries simple so that the UDF forms the main component. To evaluate Froid in situations where the queries invoking UDFs are complex, we considered TPC-H [106] queries, and looked for opportunities where parts of queries could be expressed using scalar UDFs. We extracted several UDFs and then modified the queries to use these UDFs. The UDF definitions and rewritten queries are given in Section 5.11. Figure 5.10 shows the results on a 10GB TPC-H dataset with warm cache for 6 randomly chosen queries. For each query, we show the time taken for (a) the original query (without UDFs), (b) the rewritten query with UDFs (with Froid OFF), and (c) the rewritten query with Froid ON.

Observe that for all queries, Froid leads to improvements of multiple orders of magnitude (compare (b) vs. (c)). We also see that in most cases, there is no overhead to using UDFs when Froid is enabled (see (a) vs. (c)). These improvements are the outcome of all the optimizations

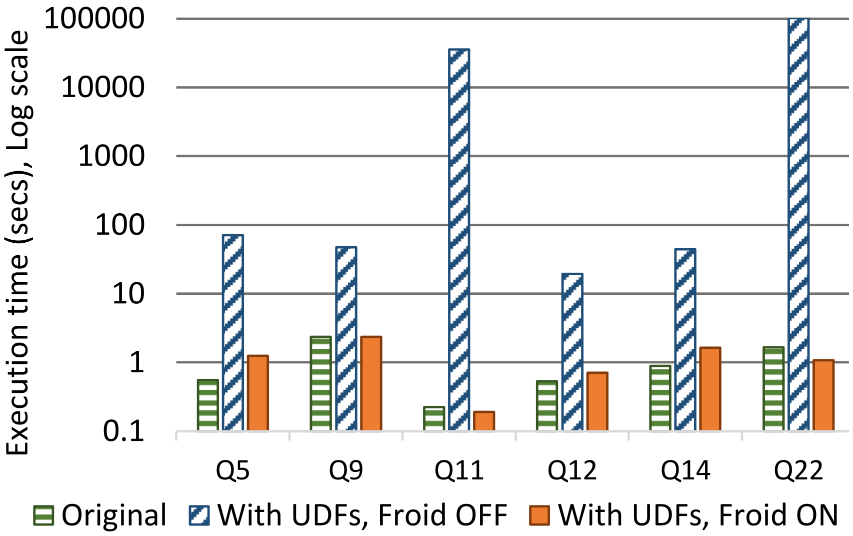


Figure 5.10: TPC-H queries using UDFs

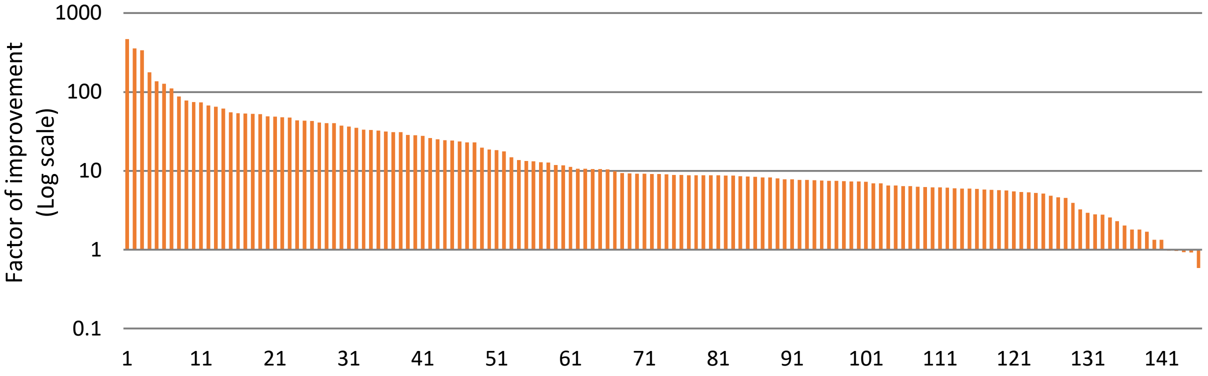


Figure 5.11: Improvement for UDFs in workload W1

that are enabled by Froid. For some queries (eg. Q5, Q14), there is a small overhead when compared with original queries. There are also cases (eg. Q11, Q22) where Froid does slightly better than the original. An analysis of query plans revealed that these are due to slight variations in the chosen plan as a result of Froid's transformations. The details of plan analysis are beyond the scope of this chapter.

Factor of improvement

We now consider the overall performance gains achieved due to Froid on workloads W1 and W2 (row store), shown in Figures 5.11 and 5.12. The size of table *T* was fixed at 100,000 rows, and queries were run with warm cache (averaged over 3 runs). In these figures, UDFs are plotted along the x-axis, ordered by the observed improvement with Froid (in descending order). The y-axis shows the factor of improvement (in log scale). We observe improvements in the range of 5x-1000x across both workloads. In total, there were 5 UDFs that showed no improvement or performed slightly worse due to Froid. One of the main reasons for this was the presence of complex recursive functions. These can be handled by appropriately tuning the constraints as described in Section 5.7.2. UDFs that invoke expensive TVFs was another reason. Since our implementation currently does not handle TVFs, such UDFs do not benefit from Froid.

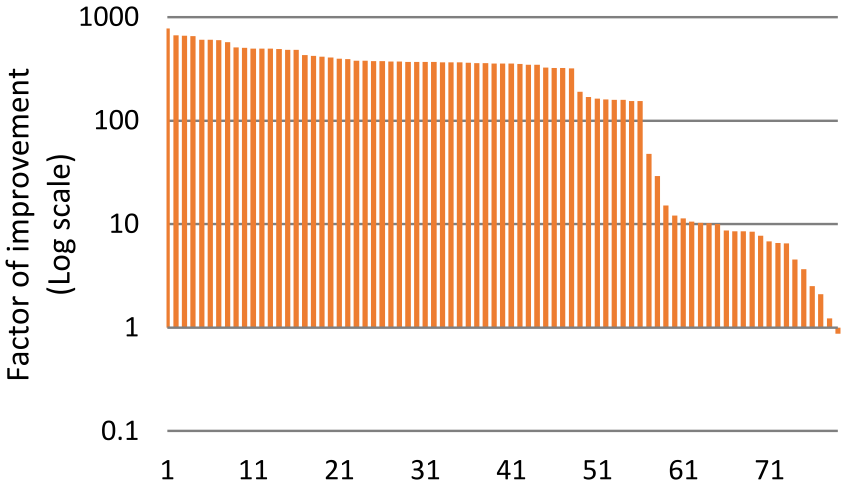


Figure 5.12: Improvement for UDFs in workload W2

```
create function discount_price(@price float, @disc float)
returns int as
begin
     return convert(int, @price * @disc);
end

Query:  select o_orderkey, c_name
from orders left outer join customer on o_custkey = c_custkey
where discount_price(o_totalprice, 0.1) > 50000;
```

Figure 5.13: Example for Section 5.8.2

Columnstore indexes

We now present the results of our experiments on column stores. Column-stores achieve better performance because of high compression rates, smaller memory footprint, and batch execution [35]. However, encapsulating aggregations and certain other operations inside a UDF prevents the optimizer from using batch mode for those operations. Froid brings the benefits of batch mode execution to UDFs. Consider a simple example based on the TPC-H schema as shown in Figure 5.13. The results of running this on a TPC-H 1GB database with a cold cache are shown in Table 5.4.

For this example, without Froid, using a clustered columnstore index (CCI) led to about 20% improvement in performance over row store. With Froid, however, we get about 5x improvement in performance by using column store over row store. Along with other reasons, the fact that the predicate and discount computation can now happen in batch mode contributes to the performance gains.

Configuration	**Froid OFF**	**Froid ON**
Row store	24241 ms	822 ms
Column store	19153 ms	155 ms

Table 5.4: Benefits of Froid on row and column stores (total elapsed time with cold cache) for the example in Figure 5.13 .

Configuration	Froid OFF	Froid ON
Query and UDF interpreted	41729 ms	2056 ms
Interpreted query, native UDF	27376 ms	NA
Native query, native UDF	9230 ms	2005 ms

Table 5.5: Benefits of Froid with native compilation (total elapsed time with warm cache) for the UDF in [88].

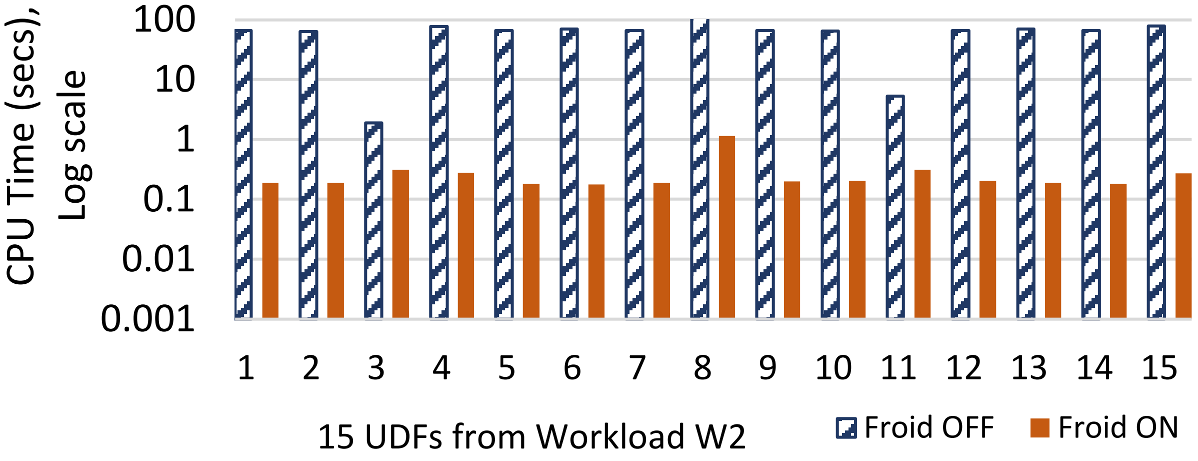


Figure 5.14: CPU time comparison

Natively compiled queries and UDFs

Hekaton, the memory-optimized OLTP engine in SQL Server performs native compilation of procedures [40], which allows more efficient query execution than interpreted T-SQL [76]. Due to its non-intrusive design, Froid seamlessly integrates with Hekaton and provides additional benefits. For this experiment, we considered the UDFs (*dbo.FarePerMile*) used in an MSDN article about native compilation [88] (the UDFs are reproduced in Section 5.10). We considered a memory optimized table with 3.5 million rows and 25 columns, with a CCI. The results of this experiment are shown in Table 5.5.

First, in the classic mode of interpreted T-SQL, we see a 20x improvement due to Froid. Next, we natively compiled the UDF, but ran the query in interpreted mode. This results in a 1.5x improvement compared to the fully interpreted mode with Froid disabled. Froid is not applicable here since a compiled module cannot be algebrized.

Finally, we natively compiled both the UDF and the query, and ran it with and without Froid enabled. With Froid disabled, we see the full benefits of native compilation over interpreted mode, with a 4.5x improvement. With Froid enabled, we get the combined benefits of algebrization and native compilation. Froid first inlines the UDF, and then the resulting query is natively compiled, *giving an additional 4.6x improvement over native compilation*. Although native compilation makes UDFs faster, the benefits are limited as the query still invokes the UDF for each tuple. Froid removes this fundamental limitation and hence combining Froid with native compilation leads to more gains.

5.8.3 Resource consumption

In addition to significant performance gains, our techniques offer an additional advantage – they significantly reduce the resources consumed by such queries. The reduction in CPU time due

```
create function total_price(@key int) returns varchar(100) as
begin
  declare @price float;
  select @price = sum(o_totalprice) from orders where o_custkey = @key
  return convert(varchar(20), @price) + 'USD';
end
```

Query: `select c_custkey, total_price(c_custkey) from customer`

Figure 5.15: Example for I/O measurements

to Froid is shown in Figure 5.14. We show the results for a randomly chosen subset of UDFs from workload W2; the results were similar across all the workloads we evaluated. Observe that Froid reduces the CPU time by 1-3 orders of magnitude for all UDFs. This reduction is due to elimination of expensive context-switches (see Section 5.2.2), and also due to optimizations such as set-oriented evaluation, folding and slicing.

Due to the above-mentioned reasons, Froid also reduces I/O costs. The I/O metric is dependent upon the nature of operations in the UDF. For UDFs that perform data access, our transformations will lead to reductions in logical reads as it avoids repetition of data access for every invocation of the UDF. Consider a simple UDF such as the one in Figure 5.15. With Froid, the query requires about 3300 logical reads, whereas without Froid, it issued close to 5 million logical reads on a 1GB TPC-H dataset with cold cache. Such improvements lead to significant cost savings for our customers, especially for users of cloud databases, since they are billed for resources they consume.

5.9 Real-World UDF Examples

In this section, we provide some examples adapted from real world UDFs. They give an idea of the kinds of UDFs that are commonly encountered in practice. These have been modified to preserve anonymity, while retaining program structure. All the UDFs given in this section are inlineable by Froid. As it can be seen, Froid can handle a fairly large class of UDFs encountered in practice.

```
create function dbo.F1(@p1 int, @p2 int)
returns bit as
begin
  if EXISTS
    (SELECT 1 FROM View1 WHERE col1 = 0
    AND col2 = @p1
    AND ((col2 = 2) OR (col3 = 2))
    AND dbo.F2(col4,@p2,0)=1 AND dbo.F2(col5,@p2,0)=1
    AND dbo.F2(col6,@p2,0)=1 AND dbo.F2(col7,@p2,0)=1
    AND dbo.F2(col8,@p2,0)=1 AND dbo.F2(col9,@p2,0)=1
    AND dbo.F2(col10,@p2,0)=1 AND dbo.F2(col11,@p2,0)=1
    AND dbo.F2(col12,@p2,0)=1 AND dbo.F2(col13,@p2,0)=1
    AND dbo.F2(col14,@p2,0)=1 AND dbo.F2(col15,@p2,0)=1)
  return 1
 return 0
end
```

```
create function dbo.F2(@p1 int,@p2 int, @flag1 int = 0)
returns bit AS
begin
 DECLARE @Flag bit
 IF @flag1=0 BEGIN
  IF EXISTS (SELECT 1 FROM Table1
    WHERE col1=@p1 AND col2=@p2)
  OR @p1 Is Null
   SET @Flag= 1
  ELSE  SET @Flag= 0
 END
 ELSE BEGIN
  IF EXISTS (SELECT 1 FROM Table1 T1
    INNER JOIN Table2 T2
    ON T1.col1=T2.col2
    WHERE T2.col2=@p1 AND T1.col2=@p2)
  OR @p1 Is Null
   SET @Flag= 1
  ELSE SET @Flag= 0
 END
 return @Flag
end
```

```
create function dbo.DayOfWeek(@d datetime) returns int as
begin
 return (DATEPART(dw, @d) + @@DATEFIRST -1) % 7
end
```

```
create function dbo.BeginOfHour(@d datetime)
returns datetime as
begin
 declare @DayBeginUTC datetime
 set @DayBeginUTC = convert(datetime, convert(nvarchar, @d, 112))
 return dateadd(hh, datepart(hh, @d), @DayBeginUTC)
end
```

```
create function BeginOfMonth(@d datetime) returns datetime as
begin
  declare @DayUserLocal datetime, @DayFirst datetime
set @DayUserLocal = dbo.UTCToLocalTime(@d)
  declare @m = datepart(mm, @DayUserLocal)
  set @DayFirst = dbo.FirstDayOfMonth(@DayUserLocal, @m)
  return dbo.LocalTimeToUTC(@DayFirst)
end
```

```
CREATE  FUNCTION dbo.RptBracket(@MyDiff int, @NDays int)
RETURNS nvarchar(10) AS
BEGIN
  if(@MyDiff >= 5*@NDays)
  begin
    RETURN ( Cast(5 * @NDays as nvarchar(5)) + N’+’)
  end

  RETURN ( Cast(Floor(@MyDiff / @NDays) * @NDays as nvarchar(5))
    + N’ - ’
    + Cast(Floor(@MyDiff / @NDays + 1) * @NDays - 1 as nvarchar(5)))
END
```

```
create function dbo.FirstDayOfMonth (@d datetime, @Month int)
returns datetime as
begin
  declare @Result datetime
  set @Result = dateadd( day, 1 - datepart( day, @d ), @d )
  if datepart( month, @Result ) <> datepart( month, @d )
    set @Result = NULL

  declare @mdiff int = @Month - datepart(mm, @Result);
  set @Result = dateadd( mm, @mdiff, @Result)
  return (convert(datetime, convert(nvarchar, @Result, 112)))
end
```

```
create function dbo.VersionAsFloat(@v nvarchar(96))
returns float as
begin
  if @v is null return null
  declare @first int, @second int;
  declare @major nvarchar(6), @minor nvarchar(10);
  set @first = charindex('.', @v, 0);
  if @first = 0
    return CONVERT(float, @v);

  set @major = SUBSTRING(@v, 0, @first);
  set @second = charindex('.', @v, @first + 1);
  if @second = 0
    set @minor=SUBSTRING(@v, @first+1, len(@v)-@first)
  else
    set @minor=SUBSTRING(@v, @first+1, @second-@first-1);

  set @minor = CAST(CAST(@minor AS int) AS varchar);
  return CONVERT(float, @major + '.' + @minor);
end
```

```
create function dbo.fn_FindBusinessGuid()
returns uniqueidentifier as
begin
  declare @userGuid uniqueidentifier
  declare @businessguid uniqueidentifier

  if (is_member('SomeRole') | is_member('SomeGroup')) = 1
  begin
    select @userGuid = cast(context_info() as uniqueidentifier)
    if @userGuid is not null
    begin
      select @businessguid = s.col4
      from T1 s
      where s.col1 = @userGuid
      return @businessguid
    end
  end

  select @businessguid = s.col3
  from T1 s
  where s.col1 = SUSER_SNAME()
  return @businessguid
end
```

```
create function dbo.fn_FindUserGuid()
returns uniqueidentifier as
begin
  declare @userGuid uniqueidentifier
  if (is_member('AppReaderRole') | is_member('db_owner')) = 1
  begin
    select @userGuid = cast(context_info() as uniqueidentifier)
  end

  if @userGuid is null
  begin
    select @userGuid = s.SystemUserId
    from SystemUserBase s
    where s.DomainName = SUSER_SNAME()
  end
  return @userGuid
end
```

5.10 Natively compiled UDFs

These UDFs are borrowed from an MSDN article [88] about the benefits of Hekaton. As mentioned earlier, Hekaton, the memory-optimized OLTP engine in SQL Server performs native compilation of procedures [40], which allows faster data access and more efficient query execution than interpreted T-SQL [76]. We have used the following UDFs along with Froid, to measure the additional benefits that we achieve using our techniques.

Here is the simple UDF in T-SQL interpreted form:

```
  CREATE FUNCTION dbo.FarePerMile ( @Fare MONEY, @Miles INT )
  RETURNS MONEY
  WITH SCHEMABINDING
  AS
  BEGIN
    DECLARE @retVal MONEY = ( @Fare / @Miles );
    RETURN @retVal;
  END;
```

Here is the simple UDF written as a native compiled version:

```
  CREATE FUNCTION dbo.FarePerMile_native (@Fare money, @Miles int)
  RETURNS MONEY
  WITH NATIVE_COMPILATION, SCHEMABINDING, EXECUTE AS OWNER
  AS
  BEGIN ATOMIC
  WITH (TRANSACTION ISOLATION LEVEL = SNAPSHOT, LANGUAGE = Nus_english)

    DECLARE @retVal money = ( @Fare / @Miles)
    RETURN @retVal
  END
```

5.11 TPC-H Queries with UDFs

In this section, we first show some scalar UDFs extracted from TPC-H queries, and then show the queries rewritten to use these scalar UDFs. Observe that there are some UDFs that are used in multiple queries, highlighting the benefits of code reuse due to the use of UDFs. Without Froid, these rewritten queries with UDFs exhibit poor performance as shown in Section 5.8.

5.11.1 Scalar UDF Definitions

```
create function dbo.discount_price(@extprice decimal(12,2),
                                   @disc decimal(12,2))
returns decimal(12,2) as
begin
  return @extprice*(1-@disc);
end
```

```
create function dbo.discount_taxprice(@extprice decimal(12,2),
                                     @disc decimal(12,2),
                                     @tax decimal(12,2))
returns decimal(12,2) as
begin
  return dbo.discount_price(@extprice, @disc) * (1+@tax);
end
```

```
create function dbo.profit_amount(@extprice decimal(12,2),
                    @discount decimal(12,2),
                    @suppcost decimal(12,2),
                    @qty int)
returns decimal(12,2) as
begin
  return @extprice*(1-@discount)-@suppcost*@qty;
end
```

```
create function dbo.isShippedBefore(@shipdate date,
                                  @duration int,
                                  @stdatechar varchar(10))
returns int as
begin
  declare @stdate date = cast(@stdatechar as date);
  declare @newdate date =  dateadd(dd, @duration, @stdate);
  if(@shipdate > @newdate)
    return 0;
  return 1;
end
```

```
create function dbo.checkDate(@d varchar(10),
                                @odate date,
                                @shipdate date)
returns int as
begin
  if(@odate < @d AND @shipdate > @d)
    return 1;
  return 0;
end
```

```
create function dbo.q3conditions(@cmkt varchar(10),
                                @odate date,
                                @shipdate date)
returns int as
begin
  declare @thedate varchar(10) = ’1995-03-15’;
  if(@cmkt <> ’BUILDING’)
    return 0;
  if(dbo.checkDate(@thedate, @odate, @shipdate) = 0)
    return 0;
  if(dbo.isShippedBefore(@shipdate, 122, @thedate) = 0)
    return 0;
  return 1;
end
```

```
create function dbo.q5Conditions(@rname char(25),
                                @odate date)
returns int as
```

```
begin
  declare @beginDatechar varchar(10) = '1994-01-01';
  declare @beginDate date = cast(@beginDatechar as date);
  declare @newdate date;

  if(@rname <> 'ASIA')
    return 0;
  if(@odate < @beginDate)
    return 0;

  set @newdate = DATEADD(YY, 1, @beginDate);
  if(@odate >= @newdate)
    return 0;

  return 1;
end
```

```
create function dbo.q6conditions(@shipdate date,
                                 @discount decimal(12,2),
                                 @qty int)
returns int as
begin
  declare @stdateChar varchar(10) = '1994-01-01';
  declare @stdate date = cast(@stdateChar as date);
  declare @newdate date = dateadd(yy, 1, @stdate);

  if(@shipdate < @stdateChar)
    return 0;

  if(@shipdate >= @newdate)
    return 0;

  if(@qty >= 24)
    return 0;

  declare @val decimal(12,2) = 0.06;
  declare @epsilon decimal(12,2) = 0.01;
  declare @lowerbound decimal(12,2), @upperbound decimal(12,2);
  set @lowerbound = @val - @epsilon;
  set @upperbound = @val + @epsilon;
  if(@discount >= @lowerbound AND @discount <= @upperbound)
    return 1;

  return 0;
end
```

```
 create function dbo.q7conditions(@n1name varchar(25),
                                  @n2name varchar(25),
                                  @shipdate date)
returns int as
 begin
  if(@shipdate NOT BETWEEN ’1995-01-01’ AND ’1996-12-31’)
    return 0;

  if(@n1name = ’FRANCE’ AND @n2name = ’GERMANY’)
    return 1;
  else if(@n1name = ’GERMANY’ AND @n2name = ’FRANCE’)
    return 1;

  return 0;
 end
```

```
create function dbo.q10conditions(@odate date, @retflag char(1))
returns int as
begin
  declare @stdatechar varchar(10) = ’1993-10-01’;
  declare @stdate date = cast(@stdatechar as date);
  declare @newdate date = dateadd(mm, 3, @stdate);

  if(@retflag <> ’R’)
    return 0;
  if(@odate >= @stdatechar AND @odate < @newdate)
    return 1;

  return 0;
end
```

```
create function dbo.total_value() returns decimal(12,2) as
begin
```

```
  return (SELECT SUM(PS_SUPPLYCOST*PS_AVAILQTY) * 0.0001000000
      FROM PARTSUPP, SUPPLIER, NATION
      WHERE PS_SUPPKEY = S_SUPPKEY
      AND S_NATIONKEY = N_NATIONKEY AND N_NAME = 'GERMANY');
end
```

```
create function dbo.line_count(@oprio char(15), @mode varchar(4))
returns int as
begin
  declare @val int = 0;
  if(@mode = 'high')
  begin
    if(@oprio = '1-URGENT' OR @oprio = '2-HIGH')
      set @val = 1;
  end
  else if(@mode = 'low')
  begin
    if(@oprio = '1-URGENT' AND @oprio = '2-HIGH')
      set @val = 1;
  end
  return @val;
end
```

```
create function dbo.q12conditions(@shipmode char(10),
                                 @commitdate date,
                                 @receiptdate date,
                                 @shipdate date)
returns int as
begin
  if(@shipmode = 'MAIL' OR @shipmode ='SHIP')
  begin
    declare @stdatechar varchar(10) = '1995-09-01';
    declare @stdate date = cast(@stdatechar as date);
    declare @newdate date = dateadd(mm, 1, @stdate);

    if(@receiptdate < '1994-01-01')
      return 0;
    if(@commitdate < @receiptdate AND @shipdate < @commitdate
        AND @receiptdate < @newdate)
      return 1;
  end
  return 0;
end
```

```
create function dbo.promo_disc(@ptype varchar(25),
                        @extprice decimal(12,2),
                        @disc decimal(12,2))
returns decimal(12,2) as
begin
  declare @val decimal(12,2);

  if(@ptype LIKE 'PROMO%%')
     set @val = dbo.discount_price(@extprice, @disc);
  else
    set @val = 0.0;
  return @val;
end
```

```
create function dbo.q19conditions(@pcontainer char(10),
                  @lqty int,
                  @psize int,
                  @shipmode char(10),
                  @shipinst char(25),
                  @pbrand char(10))
returns int as
begin
  declare @val int = 0;
  if(@shipmode IN('AIR', 'AIR REG')
      AND @shipinst = 'DELIVER IN PERSON')
  begin
    if(@pbrand = 'Brand#12'
      AND @pcontainer
          IN ('SM CASE', 'SM BOX', 'SM PACK', 'SM PKG')
      AND @lqty >= 1 AND @lqty <= 1 + 10
      AND @psize BETWEEN 1 AND 5)
        set @val = 1;

    if(@pbrand = 'Brand#23'
      AND @pcontainer
          IN ('MED BAG', 'MED BOX', 'MED PKG', 'MED PACK')
      AND @lqty >= 10 AND @lqty <= 10 + 10
      AND @psize BETWEEN 1 AND 10)
        set @val = 1;

    if(@pbrand = 'Brand#34'
      AND @pcontainer
          IN ('LG CASE', 'LG BOX', 'LG PACK', 'LG PKG')
      AND @lqty >= 20 AND @lqty <= 20 + 10
      AND @psize BETWEEN 1 AND 15)
        set @val = 1;
  end
  return @val
end
```

```
create function dbo.avg_actbal() returns decimal(12,2) as
begin
  return (SELECT AVG(C_ACCTBAL) FROM CUSTOMER
    WHERE C_ACCTBAL > 0.00
    AND SUBSTRING(C_PHONE,1,2)
    IN ('13', '31', '23', '29', '30', '18', '17'));
end
```

5.11.2 TPC-H Queries Rewritten using UDFs

```
-- Query 1
SELECT L_RETURNFLAG, L_LINESTATUS, SUM(L_QUANTITY) AS SUM_QTY,
 SUM(L_EXTENDEDPRICE) AS SUM_BASE_PRICE,
 SUM(dbo.discount_price(L_EXTENDEDPRICE, L_DISCOUNT))
                     AS SUM_DISC_PRICE,
 SUM(dbo.discount_taxprice(L_EXTENDEDPRICE, L_DISCOUNT, L_TAX))
                       AS SUM_CHARGE,
 AVG(L_QUANTITY) AS AVG_QTY,
 AVG(L_EXTENDEDPRICE) AS AVG_PRICE, AVG(L_DISCOUNT) AS AVG_DISC,
 COUNT(*) AS COUNT_ORDER
FROM LINEITEM
WHERE dbo.isShippedBefore(L_SHIPDATE, -90, '1998-12-01') = 1
GROUP BY L_RETURNFLAG, L_LINESTATUS
ORDER BY L_RETURNFLAG,L_LINESTATUS
```

```
-- Query 3
SELECT TOP 10 L_ORDERKEY,
  SUM(dbo.discount_price(L_EXTENDEDPRICE, L_DISCOUNT)) AS REVENUE,
  O_ORDERDATE, O_SHIPPRIORITY
FROM CUSTOMER, ORDERS, LINEITEM
WHERE C_CUSTKEY = O_CUSTKEY AND L_ORDERKEY = O_ORDERKEY
AND dbo.q3conditions(C_MKTSEGMENT, O_ORDERDATE, L_SHIPDATE) = 1
GROUP BY L_ORDERKEY, O_ORDERDATE, O_SHIPPRIORITY
ORDER BY REVENUE DESC, O_ORDERDATE
```

```
-- Query 5
SELECT N_NAME,
SUM(dbo.discount_price(L_EXTENDEDPRICE, L_DISCOUNT)) AS REVENUE
FROM CUSTOMER, ORDERS, LINEITEM, SUPPLIER, NATION, REGION
WHERE C_CUSTKEY = O_CUSTKEY AND L_ORDERKEY = O_ORDERKEY
AND L_SUPPKEY = S_SUPPKEY AND C_NATIONKEY = S_NATIONKEY
AND S_NATIONKEY = N_NATIONKEY AND N_REGIONKEY = R_REGIONKEY
AND dbo.q5Conditions(R_NAME, O_ORDERDATE) = 1
GROUP BY N_NAME
ORDER BY REVENUE DESC
```

```
-- Query 6
SELECT SUM(L_EXTENDEDPRICE*L_DISCOUNT) AS REVENUE
FROM LINEITEM
WHERE dbo.q6conditions(L_SHIPDATE, L_DISCOUNT, L_QUANTITY) = 1;
```

```
-- Query 7
SELECT SUPP_NATION, CUST_NATION, L_YEAR, SUM(VOLUME) AS REVENUE
FROM ( SELECT N1.N_NAME AS SUPP_NATION, N2.N_NAME AS CUST_NATION,
 datepart(yy, L_SHIPDATE) AS L_YEAR,
 L_EXTENDEDPRICE*(1-L_DISCOUNT) AS VOLUME
 FROM SUPPLIER, LINEITEM, ORDERS, CUSTOMER, NATION N1, NATION N2
 WHERE S_SUPPKEY = L_SUPPKEY AND O_ORDERKEY = L_ORDERKEY
 AND C_CUSTKEY = O_CUSTKEY
 AND S_NATIONKEY = N1.N_NATIONKEY
 AND C_NATIONKEY = N2.N_NATIONKEY
 AND dbo.q7conditions(N1.N_NAME, N2.N_NAME, L_SHIPDATE) = 1 )
                    AS SHIPPING
GROUP BY SUPP_NATION, CUST_NATION, L_YEAR
ORDER BY SUPP_NATION, CUST_NATION, L_YEAR
```

```
-- Query 9
SELECT NATION, O_YEAR, SUM(AMOUNT) AS SUM_PROFIT
FROM (SELECT N_NAME AS NATION,
 datepart(yy, O_ORDERDATE) AS O_YEAR,
 dbo.profit_amount(L_EXTENDEDPRICE, L_DISCOUNT, PS_SUPPLYCOST, L_QUANTITY)
                        AS AMOUNT
 FROM PART, SUPPLIER, LINEITEM, PARTSUPP, ORDERS, NATION
 WHERE S_SUPPKEY = L_SUPPKEY AND PS_SUPPKEY= L_SUPPKEY
 AND PS_PARTKEY = L_PARTKEY AND P_PARTKEY= L_PARTKEY
 AND O_ORDERKEY = L_ORDERKEY AND S_NATIONKEY = N_NATIONKEY AND
 P_NAME LIKE '%%green%%') AS PROFIT
GROUP BY NATION, O_YEAR
ORDER BY NATION, O_YEAR DESC
```

```
-- Query 10
SELECT TOP 20 C_CUSTKEY, C_NAME,
 SUM(dbo.discount_price(L_EXTENDEDPRICE, L_DISCOUNT)) AS REVENUE,
 C_ACCTBAL, N_NAME, C_ADDRESS, C_PHONE, C_COMMENT
FROM CUSTOMER, ORDERS, LINEITEM, NATION
WHERE C_CUSTKEY = O_CUSTKEY AND L_ORDERKEY = O_ORDERKEY AND
dbo.q10conditions(O_ORDERDATE, L_RETURNFLAG) = 1
AND C_NATIONKEY = N_NATIONKEY
GROUP BY C_CUSTKEY, C_NAME, C_ACCTBAL, C_PHONE,
          N_NAME, C_ADDRESS, C_COMMENT
ORDER BY REVENUE DESC
```

```
-- Query 11
SELECT PS_PARTKEY, SUM(PS_SUPPLYCOST*PS_AVAILQTY) AS VALUE
FROM PARTSUPP, SUPPLIER, NATION
WHERE PS_SUPPKEY = S_SUPPKEY AND S_NATIONKEY = N_NATIONKEY
AND N_NAME = 'GERMANY'
GROUP BY PS_PARTKEY
HAVING SUM(PS_SUPPLYCOST*PS_AVAILQTY) > dbo.total_value()
ORDER BY VALUE DESC
```

```
-- Query 12
SELECT L_SHIPMODE,
SUM(dbo.line_count(O_ORDERPRIORITY, 'high')) AS HIGH_LINE_COUNT,
SUM(dbo.line_count(O_ORDERPRIORITY, 'low')) AS LOW_LINE_COUNT
FROM ORDERS, LINEITEM
```

```
WHERE O_ORDERKEY = L_ORDERKEY AND
dbo.q12conditions(L_SHIPMODE, L_COMMITDATE,
          L_RECEIPTDATE, L_SHIPDATE) = 1
GROUP BY L_SHIPMODE
ORDER BY L_SHIPMODE
```

```
-- Query 14
SELECT 100.00 *
  SUM(dbo.promo_disc(P_TYPE, L_EXTENDEDPRICE, L_DISCOUNT))
  / SUM(dbo.discount_price(L_EXTENDEDPRICE,L_DISCOUNT))
        AS PROMO_REVENUE
FROM LINEITEM, PART
WHERE L_PARTKEY = P_PARTKEY AND L_SHIPDATE >= ’1995-09-01’
AND L_SHIPDATE < dateadd(mm, 1, ’1995-09-01’)
```

```
-- Query 19
SELECT SUM(dbo.discount_price(L_EXTENDEDPRICE, L_DISCOUNT))
      AS REVENUE
FROM LINEITEM join PART on L_PARTKEY = P_PARTKEY
WHERE dbo.q19conditions(P_CONTAINER, L_QUANTITY, P_SIZE,
      L_SHIPMODE, L_SHIPINSTRUCT, P_BRAND ) = 1;
```

```
-- Query 22
SELECT CNTRYCODE,
    COUNT(*) AS NUMCUST, SUM(C_ACCTBAL) AS TOTACCTBAL
FROM (SELECT SUBSTRING(C_PHONE,1,2) AS CNTRYCODE, C_ACCTBAL
 FROM CUSTOMER WHERE SUBSTRING(C_PHONE,1,2)
      IN (’13’, ’31’, ’23’, ’29’, ’30’, ’18’, ’17’)
    AND C_ACCTBAL > dbo.avg_actbal()
    AND NOT EXISTS ( SELECT * FROM ORDERS
        WHERE O_CUSTKEY = C_CUSTKEY)) AS CUSTSALE
GROUP BY CNTRYCODE
ORDER BY CNTRYCODE
```

5.12 Related Work

In Section 5.2.4, we discussed existing techniques (such as [99] and our techniques from Chapter 3) that can be adapted for UDF inlining, and motivated the need for the approach we take in Froid. In this section, we compare and contrast Froid with other related work.

Optimization of SQL queries containing sub-queries is well-studied. There have been several techniques proposed over the years [66, 50, 93, 49, 37, 42, 77], and many RDBMSs can optimize nested sub-queries. Complementarily, there has been a lot of work spanning multiple decades, on optimization of imperative programs in the compilers community [17, 75, 64]. UDFs are similar to nested sub-queries, but contain imperative constructs. Hence, they lie in the intersection of these two streams of work; however, they have received little attention from either community.

Some databases perform sub-program inlining, which applies only to nested function calls [80]. This technique works by replacing the call to a function with the function body. Another technique is to cache function results [89], which is useful only when there are repeated UDF invocations with identical parameter values. Unlike Froid, none of these techniques offer a complete solution that addresses all drawbacks of UDF evaluation listed in Section 5.2.3.

There have been recent efforts that use programming languages techniques to optimize database-backed applications. Cheung et al. [33] consider applications written using object-relational mapping libraries and transforms fragments of code into SQL using Query-By-Synthesis (QBS). The goals of QBS and Froid are similar, but the approaches are entirely different. QBS is based on program synthesis, whereas Froid uses a program transformation based approach. Although QBS is a powerful technique, it is limited in its scalability to large functions. We have manually analyzed all code fragments used in [33] (given in Appendix A of [33]), and found that none of those are larger than 100 lines of code. Even for these small code fragments, QBS suffers from potentially very long optimization times due to the space-exploration involved. They use a preset timeout of 10 mins in their experiments. Froid overcomes both these limitations – it can handle UDFs with 1000s of statements, and can transform them in less than 10 seconds (see Section 5.8.2).

The StatusQuo system [30] includes (a) a program analysis that identifies blocks of imperative logic that can be translated to SQL and (b) a program partitioning method to move application logic into imperative stored procedures. The SQL translation in StatusQuo uses QBS[33] to extract equivalent SQL. The program partitioning is orthogonal to our work. Once such partitioning is done, the resulting imperative procedures can be optimized using Froid.

5.13 Summary

While declarative SQL and procedural extensions are both supported by RDBMSs, their primary focus has been the efficient evaluation of declarative SQL. Although imperative UDFs and procedures offer many advantages and are preferred by many users, their poor performance is a major concern. Often, using UDFs is discouraged for this reason.

In this chapter, we address this important problem using novel techniques that automatically transform imperative programs into relational expressions. This enables us to leverage sophisticated query optimization techniques thereby resulting in efficient, set-oriented, parallel plans for queries invoking UDFs. Froid, our extensible, language-agnostic optimization framework built into Microsoft SQL Server, not only overcomes current drawbacks in UDF evaluation, but also offers the benefits of many compiler optimization techniques with no additional effort. The benefits of our framework are demonstrated by our evaluation on customer workloads, showing significant gains. We believe that our work will enable and encourage the wider use of UDFs to build modular, reusable and maintainable applications without compromising performance.

Chapter 6

Other Applications of Static Analysis

In this chapter, we present other applications of static analysis that we have explored, for optimizing and testing data access in database applications.

6.1 Rewriting ORDER BY Queries

In this section, we propose a data flow analysis, which we call *live order analysis*, for rewriting database applications and embedded queries to remove unnecessary ordering of query results[1]. Often, database developers use predefined ORDER BY queries and ordered data structures (such as a Java `List`) to store such query results, for computations that do not require the query results to be ordered. This is particularly prevalent in applications using ORMs, as developers typically use the ORM framework APIs along with a fixed set of well tuned queries for data access.

Manually identifying by code inspection, whether the specified ordering is actually required, and rewriting the application to use precise queries and data structures is non-trivial and error prone. *Live order analysis* can automatically analyze the program along with queries and collections used for storing query results, and identify whether or not a collection needs to be ordered.

6.1.1 Introduction

Developers of database applications often use ORDER BY queries whose results are stored in an ordered collection such as a list or a sorted set. This is typically the case in applications using ORM frameworks where the framework automatically translates the ResultSet (returned by the JDBC driver) into an ordered collection. However, the imposed order of elements in the collection may not always be necessary. In such cases, it is preferable to use a query without ORDER BY, which allows the database engine to potentially choose a cheaper plan that does not require an ordering of the query results.

For example, consider the program shown in Fig. 6.1, which is adapted from Broadleaf [1], a real world e-commerce application that is widely used. The function `getCustomerPayments`

[1] This is joint work with Pooja Agrawal. Pooja worked along with me on intra-procedural analysis. Later, we significantly revised the presentation and extended the approach to enable inter-procedural analysis.

```
//File: CustomerVariableExpression.java
List getCustomerPayments() {
  Customer customer = CustomerState.getCustomer();
  List tmp = customerPaymentService.fetchPaymentsById(customer.getId());
  List customerPayments = new ArrayList<Object>().addAll(tmp);
  Collections.sort(customerPayments, new Comparator(){...});
  return customerPayments;
}

//File: CustomerPaymentDaoImpl.java
List fetchPaymentsById(Long customerId) {
  Query q = em.createQuery("SELECT * FROM CustomerPayment cp, Customer c
   WHERE cp.customer_id = c.customer_id AND cp.customer_id = :customerId
   ORDER BY cp.id");
  q.setParameter("customerId", customerId);
  List res = q.getResultList();
  return res;
}
```

Figure 6.1: Custom sorting of ordered query results

fetches the payments for a customer by calling the `fetchPaymentsById` function, and the payments are sorted using a custom comparator. Although the query used to retrieve the payments from the database contains an `ORDER BY cp.id` clause, this ordering is irrelevant. Thus, the query along with the data structures storing query results can be rewritten to remove unnecessary ordering, for potential benefits.

Manually identifying for each query, whether the order of query results (if any) is necessary is tedious and impractical for large programs. Further, maintaining multiple versions of queries with and without ordering is error prone, and the performance hit may not be evident when testing with small datasets. In some cases, the queries are fine tuned by database administrators (DBAs) and developers are discouraged from modifying the queries. For these reasons, developers prefer to write queries that can be reused in multiple places. In this section, we discuss techniques that can automatically rewrite the program to use precise queries and data structures. This allows developers to use queries and code that is easy to maintain, without compromising on performance.

Given a program with embedded queries, our techniques identify collections that store query results, and rewrite the queries and collections to remove unnecessary ordering. Such a rewrite is non-trivial because real world programs contain complex control flow including branching, loops, and function calls. We develop a program analysis, which we call *live order analysis*, that can identify whether or not the ordering of elements in a collection is necessary in the program. We focus on collections that store query results (directly or indirectly). Using the results of the analysis, if we determine that the ordering is not necessary in an ordered collection (such as a list), we automatically rewrite the program to use an unordered collection (such as a multiset). The corresponding queries are also modified to remove ordering of results.

In the next section (Section 6.1.2), we present the necessary background on data flow analysis. We then formulate live order analysis (in Section 6.1.3) and discuss an algorithm (in Section 6.1.4) to rewrite the program using the results of live order analysis.

6.1.2 Background: Data Flow Analysis

Data flow analysis is a program analysis technique that is used to derive information about the run time behavior of a program [65]. Data flow analysis can be a *forward* analysis or a *backward* analysis. In forward analysis, information is propagated along the direction of control flow in the program. In a *backward* analysis, where information is propagated against the direction of control flow in the program. The choice of forward or backward analysis depends on the problem at hand. For example, *strongly live variables analysis* [65] is a data flow analysis that derives information about whether the value of a particular variable at a location in the program is used in the future. This requires identifying the uses of a variable at a program point, and propagating the information to an earlier program point (backward analysis).

For a given program entity e, such as an assignment statement `y = a + b`, data flow analysis involves two steps:

1. Discovering the effect of individual program statements on e (called *local* data flow analysis). This is expressed in terms of sets Gen_n and $Kill_n$ for each node n in the CFG (CFG, or control flow graph of a program has been described in Section 3.3.1). Gen_n denotes the data flow information generated within node n. $Kill_n$ denotes the information that becomes invalid in node n.

   For example, in strongly live variables analysis, the set Gen_n for the above statement will contain the variables `a` and `b` as the values of `a` and `b` are read in the statement. The set $Kill_n$ will contain the variable `y` as it is (re-)assigned (thus the value of `y` from earlier in the program is overwritten/killed). The values of Gen_n and $Kill_n$ are computed once per node, and remain unchanged.

2. Relating these effects across statements in the program (called *global* data flow analysis) by propagating data flow information from one node to another. This is expressed in terms of sets In_n and Out_n, which represent the data flow information at $Entry(n)$ and $Exit(n)$ respectively (*Entry* and *Exit* have been described in Section 3.3.1).

The specific definitions of sets Gen_n, $Kill_n$, In_n and Out_n depend upon the analysis, and we define them for our analysis in Section 6.1.3. The relationship between local and global data flow information is captured by a system of *data flow equations*. The function that represents the transformation of data flow values at a basic block[2] is called as a *flow function*. The nodes of the CFG are traversed and these equations are iteratively solved until the system stabilizes, i.e., reaches a fix point. Data flow analysis captures all the necessary inter-statement data and control dependencies about e through the sets In_n and Out_n. The results of the analysis are then used to infer information about e.

6.1.3 Data Flow Equations for Live Order Analysis

Our approach for live order analysis draws intuition from strongly live variables analysis. Strongly live variables analysis identifies whether the value of a variable is used after a program point; this analysis is typically used for dead code elimination as the variables that are not live can be removed. Live order analysis identifies whether the order of a collection is used

[2] Some approaches consider a basic block region as a sequence of statements. In this chapter, we consider each statement as a basic block, and treat a sequence of statements as consisting of multiple basic blocks with sequential control flow between them. In our implementation, we use an intermediate representation of bytecode [102], where each statement is represented using a three-address code [17].

after a program point. Our analysis differs from strongly live variable analysis in the following aspects: (a) our goal is to rewrite the program using more precise queries and collections, not dead code elimination (b) we compute and propagate data flow information for ordered collections as against variables. The scope of this analysis is intra-procedural i.e., we use this analysis to find order liveness within a procedure. (We discuss extensions to our approach for handling inter-procedural order liveness analysis in Section 6.1.4). We now formally define our analysis:

DEFINITION 1: *The order of elements in a collection* c *is live at a program point* p *if at least one path from* p *to* End *contains an ordered use of the elements in* c*, and the use is not preceded by any statement that defines* c *or modifies the order of elements in* c. □

Live order analysis is a data flow framework with ordered collections being the data flow values (program entities of interest). All required data flow information for this analysis are compactly represented using bit vectors, where each bit represents an ordered collection. For an ordered collection c, we define local data flow information in terms of the sets Gen_n and $Kill_n$ as follows:

- Gen_n contains the collection c if n contains a use of c that respects the order of elements in c. For example, if a statement n extracts the i'th element from a list c, then Gen_n for that statement will contain c.

- $Kill_n$ contains the collection c if n contains a definition of c or n modifies the order of elements in c. For example, if a statement n sorts a collection c such as in line 4 of Figure 6.1, then $Kill_n$ for that statement will contain c.

In order to check whether a particular use of a collection c in a statement n respects the order of elements in c, we examine the operations performed on c in n. Our analysis assumes that every operation on a collection c respects the order of elements in c, except for the set of operations that we explicitly identify as *order irrelevant operations*, which do not respect the order of elements in a collection. Examples of order irrelevant operations include sort, set insertion, scalar operations such as addition, etc. (Note that scalar operations are performed on individual elements in a collection, which is an indirect use of the collection. We discuss how we handle this in Section 6.1.4.) Similarly, we assume that every collection needs to be ordered unless the analysis determines that it can be unordered. These conservative assumptions ensure that our analysis does not incorrectly identify a collection as unordered in the presence of unknown or custom operations and types.

Live order analysis requires propagation of information against the direction of control flow, i.e., backward data flow analysis. The data flow information at *Exit(n)* is computed by merging information at the *Entry* of all successors of n. The data flow equations for live order analysis are:

$$In_n = (Out_n - Kill_n) \cup X$$

$$\text{where } X = \begin{cases} Gen_n & \text{if } Out_n \neq \phi \text{ and } Kill_n \subseteq\ Out_n \\ \phi & \text{otherwise} \end{cases} \tag{6.1}$$

$$Out_n = \begin{cases} \phi & \text{if } n \text{ is the } End \text{ node} \\ \bigcup_{s \in succ(n)} In_s & \text{otherwise} \end{cases} \tag{6.2}$$

Equation 6.1 defines In_n in terms of Out_n, Gen_n and $Kill_n$. Out_n is defined in Equation 6.2 by merging the In values of all successors of n using set union ($\cup$) as the merge operator. Out_{End} is initialized to be ϕ as order of collections is not used at $Exit(End)$. We use $\cup$ to capture the notion that the order of a collection is live at Out_n if it is used along *any path* from n to *End*. In_n and Out_n for all other nodes are initialized to ϕ (zeros).

6.1.4 Algorithm for Program Rewriting

We now discuss the algorithm to rewrite a program to remove unnecessary ordering, using the results of live order analysis. In our approach, we assume that statements have no hidden side-effects, i.e., all reads and writes performed by a statement are captured in Gen_n and $Kill_n$ sets. We also assume that there are no aliases or global variables except function return values that may be used at the caller location. Our technique is summarized in Algorithm 2.

Algorithm 2 Remove Unnecessary Ordering

Input: A control flow graph (G)

Precondition: There should be no *lcfd* edge in G other than the ones due to accumulator variables

Output: G, modified to remove unnecessary ordering of query results

```
procedure REMOVEORDERING(G)
    C ← all ordered collections in G
    Perform LiveOrderAnalysis on G w.r.t. C

    candidates ← {}                                  ▷ Set of <collection, node> pairs
    for each collection $c \in C$ do
        for each node $n \in G$ do
            if n defines c and $c \notin Out_n$ then
                candidates.add(<c, n>)
            end if
        end for
    end for

    REWRITEQUERIES(candidates)                       ▷ Remove ORDER BY
    MIGRATETYPES(candidates)                         ▷ Change collection types
end procedure
```

Algorithm 2 accepts the CFG of a program as input and returns a modified CFG, where unnecessary ordering of query results has been removed. The algorithm internally uses Live Order Analysis (Section 6.1.3) to identify the collections whose order of elements is unused and the program points where these collections are defined (i.e., assigned) as potential candidates for rewriting. The procedure REWRITEQUERIES examines the candidates, and if the definition of a candidate collection contains a query execution statement with an ORDER BY query, it rewrites the query without ORDER BY. Further, the procedure MIGRATETYPES modifies the type of collections storing results of the rewritten queries to an unordered type.

In the presence of loops, the use of a collection is indirect, as the collection is typically accessed one element at a time. For a variable that is updated inside the loop (accumulator variable), there may be a loop carried flow dependency (*lcfd*) due to read/write of the variable across various iterations of the loop. The precondition in Algorithm 2 checks that there is no other *lcfd* in the loop other than the ones due to accumulator variables. This ensures that any operation on an individual element of a collection (such as within a single iteration of a loop) can be treated as an operation on the entire collection itself. The preconditions are similar to the

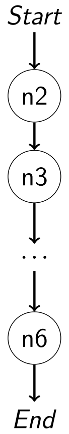


Figure 6.2: CFG for the function `getCustomerPayments` from Figure 6.1

.

preconditions in the algorithm for converting loops to F-IR (Algorithm 1 from Section 3.4.2), and can be checked using the data dependence graph (DDG) of the program.

The techniques presented in Algorithm 2 rewrite queries and collections within a single method. In the case of nested function calls with the method, we modify the algorithm as follows (inter procedural analysis). Intuitively, we first compute order liveness information for the caller, and use the information at each call site to compute order liveness in the corresponding callee.

Let f be a function that contains other function invocations. (Step 0): Set $root = f$. (Step 1): Let $H = \{h_1, h_2, ...\}$ be functions that are called from within *root* (we assume there are no recursive calls). Let $L = \{l_1, l_2, ...\}$ denote locations within *root* where l_i is the call site of h_i in *root*. (Step 2): Perform live order analysis (lines 2-3 of Algorithm 2) on CFG(f). (Step 3): Then, for each $h_i \in H$ run live order analysis using $Out_{End(h_i)} = Out_{Exit(l_i)}$, where $End(h_i)$ denotes the *End* node in the CFG of h_i and $Exit(l_i)$ denotes the program point immediately after l_i in *root*. (Step 4): If h_i contains other nested function calls within it, then set $root = h_i$ and go to step 1. If not, go to step 4. (Step 4) Use the results of the analysis to rewrite the functions (lines 4-13 of Algorithm 2) by considering them in reverse topological order. Typically, a callee function may be called from more than one different caller functions, so we modify a copy of the callee to rewrite it, and modify the caller to invoke the rewritten function.

Example

We now illustrate the working of our algorithm using the example code from Figure 6.1. We first consider the function `getCustomerReports` for live order analysis. The control flow graph for `getCustomerReports` is a sequence of nodes, as shown in Figure 6.2. Each node *ni* in the CFG denotes the statement on line *i* in the program.

We have two collections to be considered for live order analysis: `tmp` and `customerPayments`. Consequently, the bit vector for our data flow analysis for the function contains two bits – {*tmp, customerPayments*} – one for each of the above collections, respectively.

The values Gen_n and $Kill_n$ for each node are computed first. Gen_n and $Kill_n$ for *Start* and *End* nodes are set as 00 as no computation happens in these nodes. The value Out_{End} is initialized as 01 as the order of the variable `customerPayments` is live after the end of the method by virtue of being the function return value. Using these values, Out_n and In_n values are computed for all other nodes using the dataflow equations from Section 6.1.3. The results of performing live order analysis for the function `getCustomerPayments` are shown in Table 6.1 (top). Since

Node	Local Information		Global Information	
			Iteration #1	
	Gen_n	$Kill_n$	Out_n	In_n
`getCustomerPayments`				
End	00	00	01	01
n_6	01	00	01	01
n_5	00	01	01	00
n_4	10	01	00	00
n_3	00	10	**0**0	00
n_2	00	00	00	00
Start	00	00	00	00
`fetchPaymentsById`				
End	0	0	**0**	0
n_{12}	1	0	0	0
n_{11}	0	1	0	0
n_{10}	0	0	0	0
n_9	0	0	0	0
Start	0	0	0	0

Table 6.1: Live order analysis for the program from Figure 6.1

there are no branches/loops, our analysis reaches the fixpoint after one iteration.

The highlighted bit (see Out_{n3} in Table 6.1) denotes that at the end of line 3 (where `fetchPaymentsById` is called), the order liveness value for the collection `tmp` is 0, i.e., order is not live. We now use this information to set the Out_{End} bit corresponding to the return value of `fetchPaymentsById` as 0, and compute order liveness for `fetchPaymentsById`. In this case, the bit vector contains only one bit corresponding to the collection `res`. The results are shown towards the bottom of Table 6.1.

Using the results of live order analysis, the following *candidates* are identified for rewriting (refer Algorithm 2 for details) in `fetchPaymentsById`:

{ <res, n11>}

The rewritten function is shown in Figure 6.3. Note that the query has been modified to remove the unused ORDER BY, and the type of collection `res` is migrated to a `MultiSet` instead of a `List`[3]. The function signature has also been modified to rename the function and change its return type. Similarly, the following *candidates* are identified in `getCustomerPayments`:

{ <tmp, n5>, <customerPayments, n6>}

The rewritten function is shown in Figure 6.3. The type of `tmp` has been migrated to `MultiSet` based on the modified return type of the callee. In our current implementation, we conservatively skip type migration for variables that are used for holding ordered collections at some program points and unordered collections at other program points. For example, although live order analysis determines that the variable `customerPayments` at line 4 in Figure 6.1 can be unordered, the same variable is used to store ordered (sorted) elements in line 5 (this is identified by live order analysis in the value of Out_{n5}), so we skip type migration for `customerPayments`.

[3] The API for the object-relational mapping framework (JPA) used in the original program does not support retrieving query results as a `MultiSet`. So, we use a custom function `Utils.executeQuery` that achieves this.

```
List getCustomerPayments() {
 Customer customer = CustomerState.getCustomer();
 MultiSet tmp = customerPaymentService.fetchPaymentsById_unordered(customer.getId());
 List customerPayments = new ArrayList<Object>().addAll(tmp);
 Collections.sort(customerPayments, new Comparator(){...});
 return customerPayments;
}

MultiSet fetchPaymentsById_unordered(Long customerId) {
 Query q = em.createQuery("SELECT * FROM CustomerPayment cp, Customer c WHERE cp.customer_id = c.customer_id AND cp.customer_id = :customerId");
 q.setParameter("customerId", customerId);
 MultiSet res = Utils.executeQuery(q);
 return res;
}
```

Figure 6.3: Rewritten program after removing unused ordering from Figure 6.1

6.1.5 Summary

In this section, we presented live order analysis, which is a data flow analysis technique to identify whether the order of elements in a collection in a program is necessary. Using live order analysis, we presented an algorithm to rewrite queries in database applications to remove unnecessary ORDER BY clause, along with collections that store the results of these queries. Future work includes quantifying the applicability and potential performance benefits of our techniques.

6.2 Test Data Generation for Database Applications

In this thesis so far, we have discussed techniques for optimizing data access in database applications, which may contain embedded queries. We argued that traditional approaches for optimizing database applications, which treat queries/data access instructions as black boxes, may not perform optimizations that a database aware compiler can perform. In this section, we extend this argument to testing of database applications.

Conventionally, approaches for testing database applications have been classified into black box and white box testing methods [22]. In black box testing approaches, test cases are generated independently of the database, and often act as a guide for programmers to develop an application. White box testing methods (such as path coverage, statement testing etc.) aim to test coverage of parts of a program; however, in the case of database applications, white box approaches still treat embedded SQL queries as black boxes.

There has been work on test data generation for standalone SQL queries. The XData system [24] for automatic grading of student queries generates test data to test the correctness of a given (student) SQL query against a known correct (instructor) query. Given a correct query, XData generates multiple test datasets that give different results on the correct query and a potential incorrect query. Student queries are then checked for correctness against the generated test datasets. For further details on the techniques underlying XData, we refer the reader to [94, 24].

In this section, we propose an approach for test data generation for embedded queries based on program regions to identify queries in different paths of the program, and generate test data for the queries and related program variables by leveraging the XData system. Our techniques consider a large number of query mutations and can handle complex control flow, making them suitable for test data generation for complex real world database applications. The contents of this section have been published in [16].

6.2.1 Introduction

Testing of database applications containing embedded queries along with imperative code is crucial to ensure the correctness of enterprise applications. Existing approaches for testing these applications run the application to assert expected program behavior either by (i) loading a copy of the actual data; this approach is fraught with privacy, security and maintenance concerns, or (ii) loading a (developer/tester designed) sample dataset; however, manually designed datasets may often miss some errors due to data that is not represented in the synthesized sample. In our work, we propose a solution to this problem as follows:

- Automatically extract queries, query parameters, and conditions in each program path.
- Generate test data based on the extracted information, using the XData system.
- Create unit/functional test cases using the generated test data to test the correctness of functions containing queries.

In this chapter, we will focus on extraction of queries and unit test generation and refer the reader to [23] for details on test data generation.

Automatically extracting queries and related information from database application programs for test data generation is non-trivial due to the complexity of queries, the imperative program, and their interaction. For example, consider the program shown in shown in Figure 6.4a, which adapted from a real world application that was in production use at our organization[4]. The function `getNumVenues` returns a list of buildings along with the number of venues in the building that are at least of the given `size`. Using Figure 6.4a as an example, we now discuss the challenges involved in query extraction:

1. Queries in database applications are intertwined with imperative code.
2. Queries may contain parameters, which could be program variables, expressions, user inputs, or results of other queries. For example, in Figure 6.4a, the second parameter to query `q2` (line 17) is `group_id`, which is obtained by executing query `q1` (line 4).
3. Queries are determined dynamically based on the program path. For example, the same variable `q2` (line 19) may refer to different queries based on whether the `if` condition (line 5) evaluated to true or false.
4. There is no specification for the correct query. In Figure 6.4a, in case a building has no venues of the required size, instead of returning that building with count 0, the query `q2`, which uses an inner join, omits that building altogether. This can be identified by running the program against a dataset that is cleverly designed to make this distinction. For example, with the database instance shown in Figure 6.4b, and with function inputs: `size` = 10 and a user corresponding to `group_id` = 3, the query will not list the Nilgiri

[4] In Figure 6.4a, we used pseudo code and made a few modifications to the original code for ease of presentation. Our implementation uses the actual code.

```
1   getNumVenues(user_id, size) {
2     q1=Query("select group_id from users where user_id=?");
3     q1.setParam(1, user_id);
4     group_id = q1.executeQuery();
5     if(group_id==0) { //admin user
6       q2=Query("select b.b_name,count(venue_id) from
7         building b inner join venue v
8         on(b.b_name=v.b_name and v.size>=?)
9          group by b_name");
10      q2.setParam(1, size);
11    } else {
12      q2=Query("select b.b_name,count(venue_id) from
13        building b inner join venue v
14        on (b.b_name=v.b_name and v.size>=?)
15         where b.group_id=? group by b_name");
16      q2.setParam(1, size);
17      q2.setParam(2, group_id);
18    }
19    return(q2.executeQuery());
20  }
```

___	- - -	....
Basic block region *B1*:2-4, *B2*:6-10, *B3*:12-17, *B4*:19	Conditional region *C1*:5-18	Sequential region *S1*:2-19

(a) Function to find number of venues

b_name	group_id
Himalaya	3
Nilgiri	3

(a) building instance

venue_id	b_name	size
21	Himalaya	10
11	Nilgiri	5

(b) venue instance

building_name	count
Himalaya	1

(c) Result of function getNumVenues

building_name	count
Himalaya	1
Nilgiri	0

(d) Expected result

(b) Database instances and results for size=10 and a user with group_id=3

Figure 6.4: Motivating example for automatic test data generation

building which has no room of size at least 10. It is possible that the developer made an error in the query or that this was indeed the intent of the developer.

In the rest of this chapter, we discuss our test data generation for programs with embedded queries. Figure 6.5 summarizes our approach. Given an input program, the system first constructs an intermediate representation of the program, using which queries, parameters and path conditions are extracted. This information is passed to XData for test data generation. Using the generated test data and a user feedback system that records whether the function generates

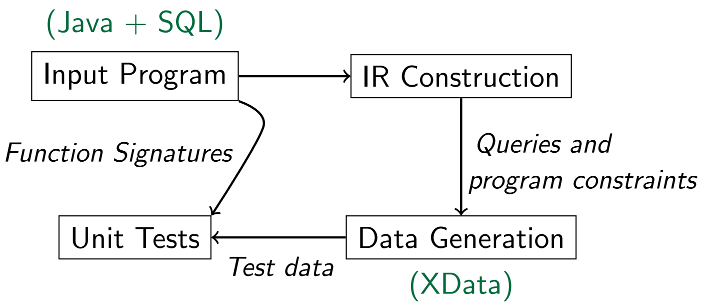
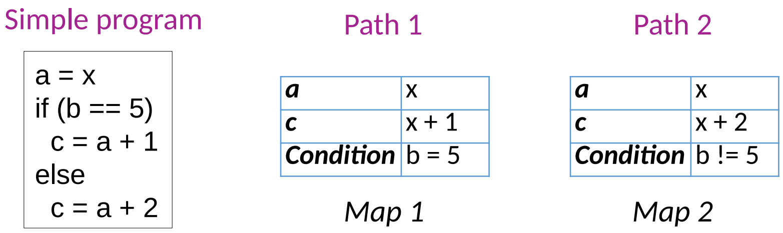


Figure 6.5: Test data generation architecture

.



```
a = x
if (b == 5)
  c = a + 1
else
  c = a + 2
```

a	x
c	x + 1
Condition	b = 5

a	x
c	x + 2
Condition	b != 5

Figure 6.6: Intermediate representation for extracting query information

the expected result on the generated data, unit tests are generated.

Our implementation focuses on Java programs using JDBC or Hibernate for database access, but the techniques themselves are not tied to any programming language or data access framework. The front-end to our test generation tool is a plugin for the IntelliJ IDEA IDE. The plugin enables users to interact with our system through a simple graphical user interface. Details of the plugin can be found in [16]. We now discuss each step in our approach, in detail.

6.2.2 Query Extraction

In this section, we discuss our techniques that use static program analysis to identify queries and related information from a database application program. Real world programs can contain complex control flow including branching and loops. In our approach, we use the concept of *program regions* to systematically construct our IR for such complex programs. Program regions for Figure 6.4a are shown alongside the code. For further details on regions, refer Chapter 3.

Intermediate Representation (IR)

Our IR is based on the DAG based representation for database applications proposed in Chapter 3. The IR from Chapter 3 is essentially a variable to expression map. The expression represents the value of the variable at any point in the region/program in terms of the region/program inputs (intermediate assignments are bypassed). In this chapter, we use an array of such variable-expression maps, one map for each alternative execution path in the program. Each map is also annotated with a condition. The map is valid for the program execution path in which the annotated condition evaluates to true.

Figure 6.6 shows a toy program and our intermediate representation for the program. There are two paths in the program corresponding to the `if` and `else` branches respectively, so our IR contains two maps – one for each path.

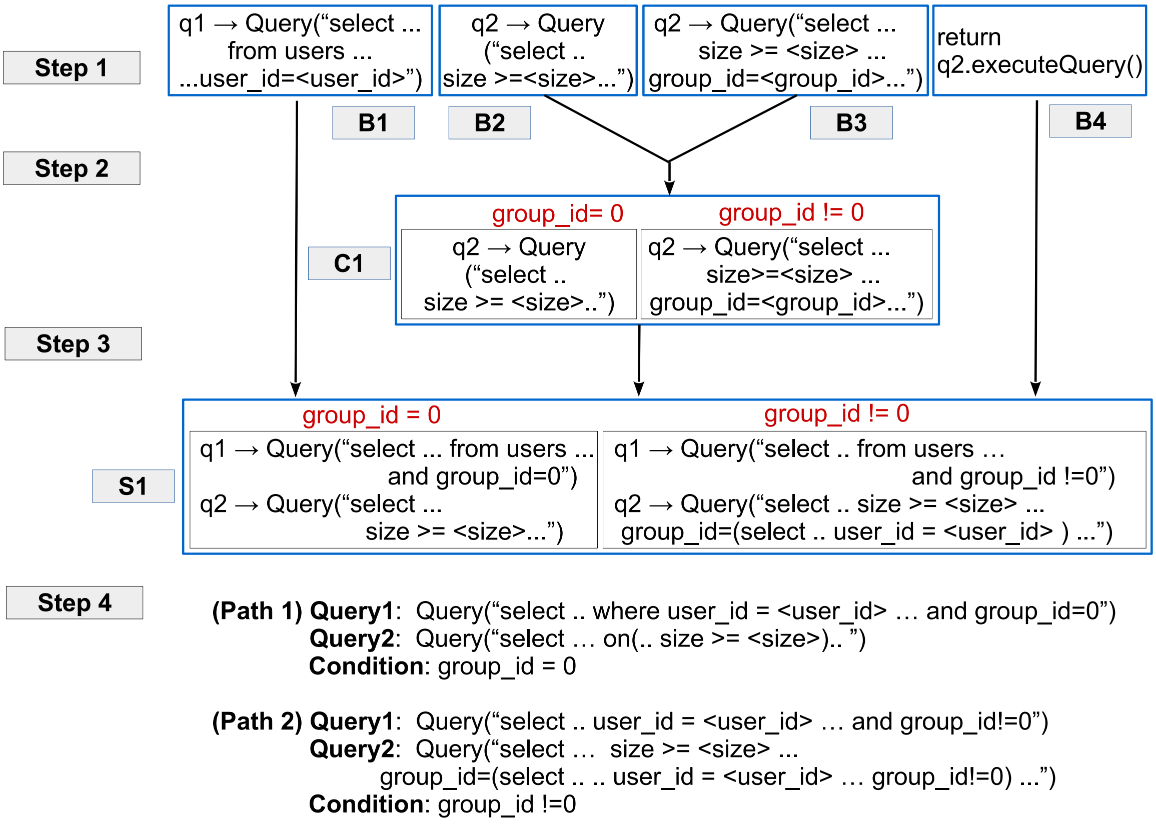

Step 1
q1 → Query("select ... from users ... ...user_id=<user_id>")
q2 → Query ("select .. size >=<size>...")
q2 → Query("select ... size >= <size> ... group_id=<group_id>...")
return q2.executeQuery()
B1
B2
B3
B4
Step 2
group_id= 0
group_id != 0
C1
q2 → Query ("select .. size >= <size>..")
q2 → Query("select ... size>=<size> ... group_id=<group_id>...")
Step 3
group_id = 0
group_id != 0
S1
q1 → Query("select ... from users ... and group_id=0")
q2 → Query("select ... size >= <size>...")
q1 → Query("select .. from users … and group_id !=0")
q2 → Query("select .. size >= <size> ... group_id=(select .. user_id = <user_id>) ...")
Step 4
(Path 1) Query1: Query("select .. where user_id = <user_id> … and group_id=0")
Query2: Query("select … on(.. size >= <size>).. ")
Condition: group_id = 0
(Path 2) Query1: Query("select .. user_id = <user_id> … and group_id!=0")
Query2: Query("select … size >= <size> ... group_id=(select .. .. user_id = <user_id> … group_id!=0) ...")
Condition: group_id !=0

Figure 6.7: Walk-through of IR construction

IR Construction using Regions

The IR construction algorithm is similar to the D-IR construction algorithm from Chapter 3 (reproduced below):

- 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.
- 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.

However, the aim in Chapter 3 was to build a single algebraic representation for the entire program. In this chapter, our aim is to build an array of IRs, one for each path in the program. So, we merge the sub-regions of a conditional region to obtain an array (size 2) of IRs – one for the true sub-region, and another for the false sub-region. Each of these IRs is then merged independently with regions that precede/follow the conditional region.

Figure 6.7 illustrates the IR construction for the program in Figure 6.4a. Each node in the IR in Figure 6.7 is annotated with its corresponding region, as marked in the program from Figure 6.4a. The first step is to construct IR for basic blocks. This is shown alongside *Step 1* in Figure 6.7. Note that the IR for each basic block consists of a single variable to expression map, and there are no conditions associated with the map. Merging the blocks B2 and B3 into conditional region C1 in step 2 gives us two maps, one corresponding to group_id=0 and the other corresponding to group_id!=0. Merging the blocks B1, C1 and B4 in step 3 gives us the final IR with maps and relevant conditions for each program execution path. Note that our approach for IR construction also performs constant folding for dynamically constructed queries.

Once we have the final IR, in step 4, we consider each path separately extract the queries and the conditions for the path. The extracted queries and conditions are then passed to XData for generating test data and unit tests to test each execution path. Note that for path 2 the *group_id* input of q2 depends on the result of query q1. We take this into account by expressing the *group_id* parameter in q2 in terms of the query q1.

Supported Program Constructs

Our system is able to extract queries and constraints from real world programs with complex control flow. The program constructs handled by our system include:

- Arbitrary levels of if-else branching, interspersed with straight line code.
- Arbitrary levels of nested function calls without recursion.
- Reuse and reassignment of variables. The same variable may be used to construct and execute multiple queries, at different program points. Our system is able to extract all such queries.
- Multiple queries in the same program execution path.
- Chained queries, where the results of one query are used (directly or indirectly) to construct another query.
- Constraints on query parameters and constraints on result set attributes.
- Loops: We only consider cursor loops with some restrictions, detailed below.

Restrictions on Loops

In general, the number of iterations in a loop is unknown at compile time. A special case of loops that iterate over a query result set/collection, which are called *cursor loops*, are widely used in database applications for iteratively processing query results. Our system supports test data generation for programs containing cursor loops.

When the loop body does not contain any branching, all the paths in the loop are covered by the following datasets: (i) empty dataset to cover the case with no iterations of the loop, and (ii) other datasets to cover the loop body. If the loop body has branching and if the branch conditions are all predicates of the current tuple or loop invariant variables only, we generate SQL queries such that generated datasets would be sufficient to cover every path present inside the loop at least once. For other cases of branching inside the loop body, the number of possible paths is not bounded by the program size, and it may not be possible to determine the sequence of paths using static program analysis techniques.

Applications using object-relational mappers (ORMs)

SQL queries are explicit in JDBC programs. However, in programs using ORMs (such as the Hibernate ORM [60]) joins may also be implicitly realized by specifying associations between attributes of mapped classes. DBridge is able to obtain explicit SQL queries in such cases (refer Chapter 3), from which XData can generate datasets.

Consider the following code snippet extracted from Wilos, an open source orchestration software.

```
  for(Project p:  getAllProjects())
   if(!(p.isFinished()))
    unfinP.add(p.getId());
```

The above code computes the set of projects whose status is marked as unfinished. The method `getAllProjects()` internally uses Hibernate API calls to fetch the list of all projects. This list is then filtered inside the application and a set of project id's satisfying the condition are returned.

Given such a program, our system first translates this program into an equivalent program that uses SQL queries, using techniques from Chapter 3. Chapter 3 contains techniques for translating relational operations such as projections, selections, joins and aggregations performed using loops in imperative code into a query. For instance, the above program is translated as follows:

```
  Query query = Utils.executeQuery
    ("SELECT id FROM Project WHERE isFinished <> 1");
```

The approach discussed earlier in this section (Section 6.2.2) can then be used to extract queries and relevant constraints.

6.2.3 Test Data and Unit Test Generation

Once the SQL query and relevant constraints from the program are obtained, we use the XData system [24, 94] for generating the test datasets. The datasets are designed to catch common errors in SQL queries. The errors in queries are modeled as query mutations. A dataset that is able to produce different results on the correct query and its mutant (thereby showing that the mutant is not equivalent to the correct query) is said to kill the mutations.

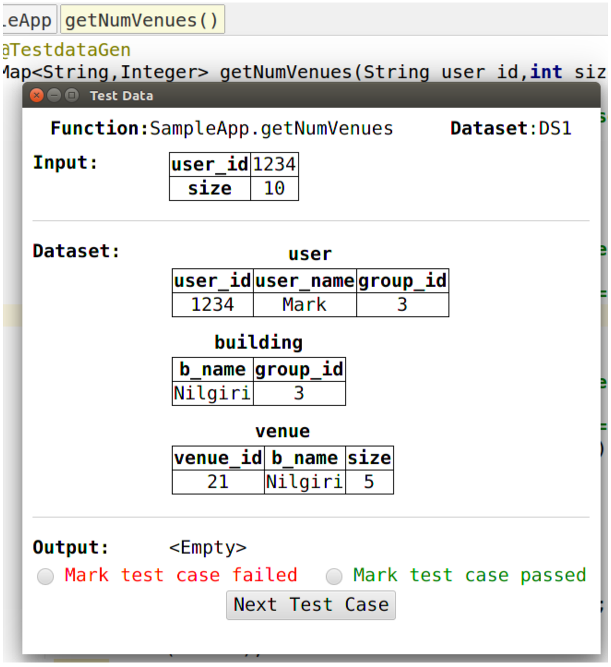
The type of mutations considered include join type mutations (inner/outer), join condition mutations, selection condition mutations, aggregate operator mutations, group by attribute mutations, mutations in string patterns, like clause mutations, distinct clause mutations, subquery connective mutations and set operator mutations, among others. XData generates several datasets for each query. Each dataset is targeted to kill one or more mutations. In order to kill a mutation we need to ensure that the dataset satisfies some constraints. XData encodes these constraints along with database constraints in the CVC3 [20] solver. XData then uses the solver to generate a dataset that satisfies the constraints.

In the case of testing applications with embedded queries, which is the focus of this chapter, there may be additional constraints due to the program in addition to the constraints imposed by the query. We appropriately encode any such program constraints into constraints that we pass to the solver. We also pass the program input parameters to the solver to get back values that may be used when invoking the program/interface for unit testing. For more details on modifications to XData for the purpose of test data generation for database applications, refer [23].

User Interaction Console

We mentioned in Section 6.2.1 that in the case of real world database applications, there is no specification of the correct query. In this section, we discuss a user feedback system to examine the correctness of queries in a function using the generated datasets.

Once the datasets for queries in the program have been generated, for each function containing queries, the generated datasets are loaded one at a time, the function is executed using the generated database, parameter and function input values, and the result is displayed to the user in the form of a user interaction window, as shown in Fig. 6.8. The window displays the



```
@TestdataGen
Map<String,Integer> getNumVenues(String user_id,int siz
```

Function: SampleApp.getNumVenues **Dataset:**DS1

Input:

user_id	1234
size	10

Dataset:

user

user_id	user_name	group_id
1234	Mark	3

building

b_name	group_id
Nilgiri	3

venue

venue_id	b_name	size
21	Nilgiri	5

Output: <Empty>

○ Mark test case failed ○ Mark test case passed

Figure 6.8: User interaction on test results

function name (getNumVenues), dataset id (DS1), generated function input parameter values (user_id:1234, size:10), and the generated dataset, along with the output of running the function using these values. Note that there may be multiple datasets generated by XData for a single query.

The user is asked to mark if the function's output matches the expected output for the given function inputs and the dataset. The user's response (match/not match, shown as passed/failed in Figure 6.8) is recorded and is used for asserting the result of the function in the corresponding unit test that will be generated. Once all the datasets have been marked for a function, unit tests are generated from a predefined template, using the function signature and details of the database containing generated datasets and parameter values. These unit tests are added to the test suite for use in future regression testing.

6.2.4 Related Work

Although mutation testing is a well known technique for testing applications in general, these techniques do not consider queries embedded in the application. Pan et al. [81] and Emmi et al. [46] focus on test data generation to ensure path coverage for database applications but do not take into account testing of SQL queries. Qex [113] generates a test database for a database application along with query parameters such that certain properties in the query results are satisfied (e.g. the query result is non-empty). Our techniques, which are based on the XData system, are able to consider a large space of mutations of queries, so the generated datasets can catch more errors.

Marcozzi et al. [70] propose an approach for test data generation for database applications written in an intermediate language (ImperDB) that models imperative program behaviors as well as database interactions. However, compiling programs written in other languages into ImperDB is not automated, hence the applicability of this approach is limited. Chan et al. [22] propose an approach for white box testing of programs with embedded queries. They propose techniques to translate SQL statements into imperative code, thereby exposing more paths to be

Description of control flow	# programs
Straight line code only	9
if-else branching (no loops)	30
Loops (with branching in body)	7
Maximum number of paths (excluding loops)	4

Figure 6.9: Characteristics of synthetic programs

considered for white box testing of such programs. However, they do not consider generation of test data, which is the focus of our system. Our approach can be used in conjunction with [22] for enhanced coverage and correctness testing.

6.2.5 Experiments

We evaluated our test data generation system on real world and synthetic programs. We used a JDBC program from a venue booking system earlier in use at IIT Bombay, and a Hibernate program from an open source application (Wilos [115]). The real world programs both contained simple if-else branching. We also considered 47 JDBC programs with simple and complex control flow, which we created to test the correctness and capabilities of our system. The distribution of control flow structures across the samples that we created is shown in Figure 6.9. Our system successfully generated test data for queries in each path, for all cases. Expanding our evaluation to include more real world programs is an area of future work.

6.2.6 Summary

We have described an approach based on program regions and mutation testing of queries to generate test data for SQL queries embedded in imperative code. Using the generated test data and a user feedback system, our techniques are able to generate unit/functional test cases for functions containing queries. Our framework can be used to complement the existing test cases so that both imperative code and database queries can be tested.

Chapter 7

Conclusions and Future Work

In this chapter, we present our conclusions for the techniques presented in this thesis, and then identify interesting directions for future work.

7.1 Conclusions

In this thesis, we presented novel techniques for optimization of imperative programs with embedded relational database accesses. Our techniques use static program analysis and program transformations to rewrite programs for more efficient data access. In particular, we focused on optimization of applications that use database abstractions, where data processing is split between the database and the application program, leading to inefficient data access.

In Chapter 3, we have described 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 chapter are widely applicable and useful in real world applications, and provide performance improvements that existing approaches cannot provide, on many programs.

In Chapter 4, 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 optimization of imperative programs, and extend the Volcano/Cascades framework for optimizing algebraic expressions, to optimize programs with regions. Our experimental evaluation on several real world application programs demonstrates the applicability and performance improvements due to cost-based rewriting.

In Chapter 5, we addressed the important problem of poor performance of UDFs in a relational database. We proposed techniques that automatically transform imperative programs into relational expressions. This enables us to leverage sophisticated query optimization techniques thereby resulting in efficient, set-oriented, parallel plans for queries invoking UDFs. Froid, our extensible, language-agnostic optimization framework built into Microsoft SQL Server, not only overcomes current drawbacks in UDF evaluation, but also offers the benefits of many compiler optimization techniques with no additional effort. The benefits of our framework are demonstrated by our evaluation on customer workloads, showing significant gains. We believe that our work will enable and encourage the wider use of UDFs to build modular, reusable and maintainable applications without compromising performance.

In Chapter 6, we discussed other applications of static analysis for optimizing and testing data access in database applications. In Section 6.1, We developed a data flow analysis framework called live order analysis to identify whether the order of elements in a collection is used in the program. Using live order analysis, we proposed techniques to rewrite ORDER BY queries and collections storing query results in database applications to remove unused ordering of query results. In Section 6.2, we have described a system that generates data to test SQL queries embedded in application code. Our approach is based on path testing, and generates data for queries and program variables so that each path in the program is covered at least once. Our techniques are able to handle (a) complex control flow in imperative code including branching, nested function calls, and cursor loops with some restrictions, and (b) complex queries including sub-queries and chained queries, which earlier techniques are unable to handle. This makes our system suitable for test data generation for real world applications.

Our implementation of the proposed techniques in Chapters 3, 4, and 6, focused on Java programs that use the JDBC/Hibernate APIs for data access. Our techniques in Chapter 5 have been discussed in the context of optimizing UDFs written in T-SQL. However, the techniques themselves are language agnostic, and can be applied in general for optimization and testing of programs/UDFs written using other languages and data access APIs. The program transformation techniques presented in this thesis add to the repertoire of *holistic* optimizations for database applications, and can be used independently or in conjunction with other techniques for further benefits.

7.2 Future Work

Although a lot of work has been done in the area of holistic optimization of database applications, there are many interesting and important problems to be addressed yet. The approaches to some of these problems can be extensions of the work presented in this thesis, and solutions for others may need different techniques. In Chapters 3, 4, 5, and 6, we discussed future directions and extensions to the work presented in the respective chapters. We now discuss more general future directions to our work.

7.2.1 Program transformations for ORM Applications

ORMs aim to address an important problem – the object-relational impedance mismatch. However, object persistence is a complex topic [21]. Consequently, most ORM frameworks used in real world applications are quite complex, with various components that address each subproblem in the paradigm mismatch such as data granularity, identity, associations between entities, and others. The details of each of these components are often not well understood by developers of ORM applications, justifiably so, owing to many intricacies. This provides many opportunities for an optimizing compiler that is database and ORM aware to automatically rewrite these programs for improved performance.

For example, caching of retrieved results plays an important role in ORM applications, and most ORMs provide cache capabilities out of the box. Not only does an ORM cache reduce access time by storing frequently accessed data locally, it also plays the role of a *logical database* against which all data access statements in a transaction are executed, before they are eventually flushed to the database. Existing techniques for optimizing ORM applications have largely ignored the impact due to the ORM cache. There is a need to explore potential compile-time and runtime techniques for optimizing ORM applications by understanding and modeling the working of cache in ORMs along with database accesses.

A related problem is the optimization of data access in applications using object document mappers (ODMs) for abstracting document store databases. Examples of ODMs include Mongoose [5] for Node.js [6], and Doctrine [4] for PHP [7]. Techniques for optimizing ORM applications can be adapted and extended for optimization of ODM applications.

7.2.2 Cost-based program transformations

In this thesis, we described the COBRA framework with a focus on cost-based program transformations for imperative programs with embedded data access. However, the framework can be used for other cost-based transformations in general, with an appropriate cost model.

For instance, modern processors may contain CPUs with many cores, GPUs (which are now being used increasingly for general purpose computing tasks), or a mix of CPUs and GPUs [72] (heterogenous architectures). These processing units have different characteristics – GPUs are more efficient for highly parallel tasks whereas CPUs perform better for sequential tasks. It has also been noted in the context of map-reduce programs, that different phases of a single application may be suitable for execution on CPU or GPU [72]. It will be interesting to explore if the same argument can be extended to regions within a program, and rewrite program regions in a cost-based manner using COBRA, to take advantage of the capabilities of the underlying processor.

The availability of scientific computing libraries such as *pandas* [9] and NumPy [8] enables research practitioners to program various data analysis tasks using these libraries. Often, the data being analyzed is huge, and may be available in files instead of a database. The developers of these programs are typically not database experts, so the programs developed may be functionally correct but may not necessarily provide the best performance. In such cases, the COBRA optimizer can rewrite it using program transformations for significant performance benefits, using cost estimates derived from sampling the input data.

7.2.3 Optimization of User Defined Functions

In this thesis, we have focused on optimization of scalar user defined functions without loops. Loop fission and statement reordering transformations, which have been proposed earlier [84] in the context of database application programs, can be used to isolate statements with cyclic dependencies in a separate loop, and the other parts of the loop along with the rest of the UDF can be inlined. Although this may not enable set-oriented execution of the UDF, it can provide some benefit due to partial inlining and reducing the number of statements executed in each iterative invocation of the UDF.

7.2.4 Compiling Over-specified Data Structures

The data flow analysis framework that we developed in this thesis – order liveness analysis – was restricted to collections in programs that store query results, and the analysis focused on testing the liveness of the order of elements in a collection. These techniques can be generalized to test the liveness of over-specified properties of data structures. In that sense, order liveness analysis is a special case where the data structures being analyzed are collections, and the property considered is order of elements in a collection. Examples where such a general analysis is applicable include use of arrays in place of lists, use of structures/classes with attributes instead of maps with static keys, and others.

7.2.5 Optimizing Interactions between Web Services and Clients

Modern JavaScript engines in web browsers are quite powerful; web browsers are able to seamlessly run programs in other languages compiled to JavaScript, in-browser databases, analytics tasks, and other heavy tasks that were earlier run on dedicated servers. Mobile phones hosting applications that access web services are powered by multi-core processors. In this context, program transformation techniques can be used to automatically partition web application programs to run partly on the web server and partly on the client to offload computation from the web server, to improve its throughput. Alternatively, holistic optimization techniques such as batching and prefetching proposed earlier can be adapted to automatically rewrite web application programs to reduce perceived latency while loading web pages and results of web service calls.

Bibliography

[1] Broadleaf commerce https://github.com/broadleafcommerce.

[2] Network Emulator Toolkit. `https://blog.mrpol.nl/2010/01/14/network-emulator-toolkit/`.

[3] TPC-DS specification. `http://www.tpc.org/`.

[4] *MongoDB Object Document Mapper*, accessed Dec 11, 2018. `https://www.doctrine-project.org/projects/mongodb-odm.html`.

[5] *Mongoose ODM*, accessed Dec 11, 2018. `https://mongoosejs.com/`.

[6] *Node.js*, accessed Dec 11, 2018. `https://nodejs.org/en/`.

[7] *PHP: Hypertext Preprocessor*, accessed Dec 11, 2018. `http://www.php.net/`.

[8] *NumPy*, accessed Dec 12, 2018. `http://www.numpy.org`.

[9] *Python Data Analysis Library*, accessed Dec 12, 2018. `https://pandas.pydata.org/`.

[10] *Hyperloop - Solving performance issues in your web application*, accessed Dec 5, 2018. `https://hyperloop-rails.github.io/`.

[11] *Django - Models and Databases*, accessed Dec 6, 2018. `https://docs.djangoproject.com/en/2.1/topics/db/`.

[12] *Entity Framework*, accessed Dec 6, 2018. `https://docs.microsoft.com/en-us/ef/`.

[13] *Ruby on Rails - Active Record Basics*, accessed Dec 6, 2018. `https://guides.rubyonrails.org/active_record_basics.html`.

[14] *ABAP Development*, accessed Dec 9, 2018. `https://www.sap.com/community/topics/abap.html`.

[15] *Language Integrated Query (LINQ)*, accessed Dec 9, 2018. `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/`.

[16] P. Agrawal, B. Chandra, K. V. Emani, N. Garg, and S. Sudarshan. Test data generation for database applications. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 1621–1624, 2018.

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

[18] Query processing architecture guide, https://msdn.microsoft.com/en-us/library/mt744587.aspx.

[19] AWS Network Latency Map `https://datapath.io/resources/blog/aws-network-latency-map/`.

[20] C. Barrett and C. Tinelli. CVC3. In *Computer Aided Verification (CAV)*, pages 298–302, 2007.

[21] C. Bauer, G. King, and G. Gregory. *Java Persistance with Hibernate*. Dreamtech Press, 2014.

[22] M.-Y. Chan and S.-C. Cheung. Testing database applications with sql semantics. In *CODAS*, volume 99, pages 363–374, 1999.

[23] B. Chandra. *Automatic Testing and Grading of SQL Queries*. Ph.D. thesis, Indian Institute of Technology, Bombay, 2019.

[24] B. Chandra, B. Chawda, B. Kar, K. V. M. Reddy, S. Shah, and S. Sudarshan. Data generation for testing and grading SQL queries. *The VLDB Journal*, 24(6), 2015.

[25] B. Chandra and S. Sudarshan. Runtime optimization of join location in parallel data management systems. *arXiv preprint arXiv:1703.01148*, 2017.

[26] M. Chavan, R. Guravannavar, K. Ramachandra, and S. Sudarshan. DBridge: A program rewrite tool for set-oriented query execution (demo). In *ICDE*, 2011.

[27] M. Chavan, R. Guravannavar, K. Ramachandra, and S. Sudarshan. Program transformations for asynchronous query submission. In *ICDE*, pages 375–386, 2011.

[28] T.-H. Chen. *Improving the performance of database-centric applications through program analysis*. PhD thesis, Queen's University, 2016.

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

[30] A. Cheung, O. Arden, S. Madden, A. Solar-Lezama, and A. C. Myers. Statusquo: Making familiar abstractions perform using program analysis. In *CIDR*. www.cidrdb.org, 2013.

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

[32] A. Cheung, S. Madden, and A. Solar-Lezama. Sloth: Being lazy is a virtue (when issuing database queries). *ACM Transactions on Database Systems (TODS)*, 41(2):8, 2016.

[33] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing database-backed applications with query synthesis. In *PLDI '13*, pages 3–14, 2013.

[34] CLR User-Defined Functions, https://msdn.microsoft.com/en-us/library/ms131077.aspx.

[35] Columnstore indexes guide, https://msdn.microsoft.com/en-us/library/gg492088.aspx.

[36] A. Dasgupta, V. Narasayya, and M. Syamala. A static analysis framework for database applications. In *ICDE '09*, pages 1403–1414, 2009.

[37] U. Dayal. Of Nests and Trees: A Unified approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers. In *VLDB*, 1987.

[38] 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.

[39] K. Delaney, B. Beuchemin, and C. Cunningham. *Microsoft SQL Server 2012 Internals*. 2013.

[40] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *ACM SIGMOD*, SIGMOD '13, 2013.

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

[42] M. Elhemali, C. A. Galindo-Legaria, T. Grabs, and M. M. Joshi. Execution Strategies for SQL Subqueries. In *ACM SIGMOD*, 2007.

[43] K. V. Emani, T. Deshpande, K. Ramachandra, and S. Sudarshan. DBridge: Translating Imperative Code to SQL. In *SIGMOD '17*, pages 1663–1666.

[44] K. V. Emani, K. Ramachandra, S. Bhattacharya, and S. Sudarshan. Extracting Equivalent SQL from Imperative Code in Database Applications. In *SIGMOD '16*.

[45] K. V. Emani and S. Sudarshan. Cobra: A framework for cost-based rewriting of database applications. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 689–700, 2018.

[46] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *ISSTA*, pages 151–162, 2007.

[47] Reduce vs foldleft. https://stackoverflow.com/a/25158790/1299738.

[48] Create Function (MSDN), https://msdn.microsoft.com/en-us/library/ms186755.aspx.

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

[50] R. A. Ganski and H. K. T. Wong. Optimization of Nested SQL Queries Revisited. In *ACM SIGMOD*, 1987.

[51] G. Giorgidze, T. Grust, T. Schreiber, and J. Weijers. Haskell boards the ferry. In *IFL*, pages 1–18. Springer, 2011.

[52] G. Graefe. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[53] G. Graefe and W. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Intl. Conf. on Data Engineering*, 1993.

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

[55] T. Grust, J. Rittinger, and T. Schreiber. Avalanche-safe linq compilation. *VLDB*, 3(1-2):162–172, 2010.

[56] R. Guravannavar and S. Sudarshan. Rewriting procedures for batched bindings. *PVLDB*, 1(1):1107–1123, 2008.

[57] M. W. Hall, B. R. Murphy, S. P. Amarasinghe, S.-W. Liao, and M. S. Lam. Interprocedural analysis for parallelization. In *LCPC*, pages 61–80, 1995.

[58] S. R. Hardikar and N. L. Sarda. Cobol program and test data generator. In *Masters Dissertation*. IIT Bombay, 1984.

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

[60] Hibernate. `http://www.hibernate.org`.

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

[62] Jasper Reports with Hibernate
http://jasperreports.sourceforge.net/sample.reference/hibernate/.

[63] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1-4):79–119, 1988.

[64] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., 2002.

[65] U. Khedker, A. Sanyal, and B. Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., 1st edition, 2009.

[66] W. Kim. On Optimizing an SQL-like Nested Query. In *ACM Trans. on Database Systems, Vol 7, No.3*, 1982.

[67] D. F. Lieuwen and D. J. DeWitt. Optimizing loops in database programming languages. In *Proceedings of the Third International Workshop on Database Programming Languages*, DBPL3, pages 287–305, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[68] Logical and Physical Operators Reference, https://technet.microsoft.com/en-us/library/ms191158(v=sql.105).aspx.

[69] A. Manjhi, C. Garrod, B. M. Maggs, T. C. Mowry, and A. Tomasic. Holistic query transformations for dynamic web applications. In *ICDE*, pages 1175–1178, 2009.

[70] M. Marcozzi, W. Vanhoof, and J.-L. Hainaut. Test input generation for database programs using relational constraints. In *Proceedings of the Fifth International Workshop on Testing Database Systems*, page 6. ACM, 2012.

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

[72] S. Mittal and J. S. Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Computing Surveys (CSUR)*, 47(4):69, 2015.

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

[74] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., 1997.

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

[76] Natively compiled stored procedures, https://msdn.microsoft.com/en-us/library/dn133184.aspx.

[77] T. Neumann and A. Kemper. Unnesting arbitrary queries. In *BTW*, 2015.

[78] Performance overhead of SQL user-defined functions, http://glennpaulley.ca/conestoga/2015/07/performanceoverhead-of-sql-user-defined-functions/.

[79] How Functions can Wreck Performance, http://www.oraclemagician.com/mag/magic9.pdf.

[80] Subprogram inlining in oracle, https://docs.oracle.com/cd/b28359_01/appdev.111/-b28370/inline_pragma.htm.

[81] K. Pan, X. Wu, and T. Xie. Generating program inputs for database application testing. In *ASE*, pages 73–82, 2011.

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

[83] K. Ramachandra. *Holistic Optimization of Database Application*. PhD thesis, Indian Institute of Technology, Bombay, Department of Computer Sc. & Engg., 2014.

[84] K. Ramachandra, M. Chavan, R. Guravannavar, and S. Sudarshan. Program transformations for asynchronous and batched query submission. *TKDE '15*, 27(2):531–544.

[85] K. Ramachandra and R. Guravannavar. Database-aware program optimization via static analysis. *IEEE Data Eng. Bull.*, 37(1):60–69, 2014.

[86] K. Ramachandra, K. Park, K. V. Emani, A. Halverson, C. A. Galindo-Legaria, and C. Cunningham. Froid: Optimization of Imperative Programs in a Relational Database. *PVLDB*, 11(4):432–444, 2017.

[87] K. Ramachandra and S. Sudarshan. Holistic optimization by prefetching query results. In *SIGMOD*, 2012.

[88] Soften the RBAR impact with Native Compiled UDFs, https://blogs.msdn.microsoft.com/sqlcat/2016/02/17/soften-the-rbar-impact-with-native-compiled-udfs-in-sql-server-2016.

[89] PL/SQL Function Result Cache, http://www.oracle.com/technetwork/issue-archive/2010/10sep/o57plsql088600.html.

[90] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and Extensible Algorithms for Multi Query Optimization. In *SIGMOD*, 2000.

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

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

[93] P. Seshadri, H. Pirahesh, and T. C. Leung. Complex Query Decorrelation. In *ICDE*, 1996.

[94] S. Shah, S. Sudarshan, S. Kajbaje, S. Patidar, B. P. Gupta, and D. Vira. Generating test data for killing SQL mutants: A constraint-based approach. In *ICDE*, 2011.

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

[96] B. Shneiderman and G. Thomas. An architecture for automatic relational database sytem conversion. *ACM TODS 1982*, 7(2):235–257.

[97] J. E. Shopiro. Theseus - a programming language for relational databeses. *ACM Trans. Database Syst.*, 4(4):493–517, Dec. 1979.

[98] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw Hill, 6th ed., 2010.

[99] 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.

[100] 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.

[101] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. *ACM Sigplan Notices*, 41(11):404–415, 2006.

[102] Soot: A Java Optimization Framework
`http://www.sable.mcgill.ca/soot`.

[103] Spring Framework. `https://spring.io/`.

[104] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: A new approach to optimization. In *POPL*, pages 264–276, 2009.

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

[106] TPC. TPC-H Benchmark Specification, 2005, http://www.tpc.org.

[107] Transact SQL
https://docs.microsoft.com/en-us/sql/t-sql/language-elements/language-elements-transact-sql.

[108] Performance overhead of sql user-defined functions, http://glennpaulley.ca/conestoga/2015/07/performance-overhead-of-sql-user-defined-functions.

[109] Tsql scalar functions are evil, http://sqlblogcasts.com/blogs/simons/archive/-2008/11/03/tsql-scalar-functions-are-evil-.aspx.

[110] Scalar functions, inlining, and performance, http://sqlblog.com/blogs/adam_machanic/archive/2006/-08/04/scalar-functions-inlining-and-performance-an-entertaining-title-for-a-boring-post.aspx.

[111] T-sql user-defined functions: the good, the bad, and the ugly, http://sqlblog.com/blogs/hugo_kornelis/archive/2012/05/20/t-sql-user-defined-functions-the-good-the-bad-and-the-ugly-part-1.aspx.

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

[113] M. Veanes, N. Tillmann, and J. de Halleux. Qex: Symbolic SQL query explorer. In *LPAR*, pages 425–446, 2010.

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

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

[116] C. Yan and A. Cheung. Leveraging lock contention to improve oltp application performance. *Proceedings of the VLDB Endowment*, 9(5):444–455, 2016.

[117] J. Yang, P. Subramaniam, S. Lu, C. Yan, and A. Cheung. Powerstation: Automatically detecting and fixing inefficiencies of database-backed web applications in ide. In *26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2018.

[118] J. Yang, C. Yan, P. Subramaniam, S. Lu, and A. Cheung. How not to structure your database-backed web applications: a study of performance bugs in the wild. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 800–810. IEEE, 2018.

[119] J. Yang, C. Yan, P. Subramaniam, S. Lu, and A. Cheung. Powerstation: automatically detecting and fixing inefficiencies of database-backed web applications in ide. In *FSE*, pages 884–887. ACM, 2018.

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

Appendix A

Translating Imperative Code to SQL

A.1 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 Algorithm 1, 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 $\textsc{LoopToFold}(R)$, 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 v 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,

$$
\begin{aligned}
v_{k+1} &= e'_{acc}(v_k, t_{k+1}) \\
&= e'_{acc}(fold[\, e'_{acc},\ v_0,\ Q_k\,], t_{k+1}) \\
&= fold[\, e'_{acc},\ v_0,\ Q_{k+1}\,] && \text{/* defn of fold */}
\end{aligned}
$$

Hence, proved.

Part (b): The procedure CONSTRUCTFIR first translates all sub-regions 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.

A.2 D-IR Construction

In Section 3.3.3, we gave an outline of D-IR construction for various types of regions. We now describe the algorithms for D-IR construction in detail.

A.2.1 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.

A.2.2 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.

A.2.3 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*.

A.2.4 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.

A.2.5 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.

A.2.6 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 A.2.3).

Publications based on this work

1. K. Venkatesh Emani, Karthik Ramachandra, Subhro Bhattacharya and S. Sudarshan, *Extracting Equivalent SQL From Imperative Code in Database Applications*, ACM Special Interest Group on Management of Data (SIGMOD) 2016.

2. K. Venkatesh Emani, Tejas Deshpande, Karthik Ramachandra and S. Sudarshan, *DBridge: Translating Imperative Code to SQL*, ACM Special Interest Group on Management of Data (SIGMOD) 2017 (Demo paper).

3. Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, Cesar Galindo-Legaria and Conor Cunningham, *Froid: Optimization of Imperative Programs in a Relational Database*, Proceedings of the VLDB Endowment (PVLDB) 2017.

4. K. Venkatesh Emani and S. Sudarshan, *Cobra: A Framework for Cost Based Rewriting of Database Applications*, IEEE International Conference on Data Engineering (ICDE) 2018.

5. Pooja Agrawal, Bikash Chandra, K. Venkatesh Emani, Neha Garg and S. Sudarshan, *Test Data Generation for Database Applications*, IEEE International Conference on Data Engineering (ICDE) 2018 (Demo paper).

Acknowledgments

I am extremely grateful to have Prof. Sudarshan as my advisor. This Ph.D. would not have been possible without his vision and guidance. His commitment to our work and constant support kept me motivated, and inspired me to keep striving to cross the occasional hurdles. Working with him has been a great learning experience, both professionally and personally.

I thank Prof. Uday Khedker, Prof. Amitabha Sanyal, and Prof. Krithi Ramamritham for their regular feedback and valuable suggestions at various stages of my research. I thank Karthik Ramachandra, who has been a mentor, friend and collaborator throughout my Ph.D. Thanks to Bikash for being a collaborator and travel companion for all the conferences that we attended together. I thank Ravindra Guravannavar and Prasanna Kumar for their inputs and our many discussions. It was a pleasure working with the graduate students at Infolab, especially Subhro, Tarun, Tejas, Mohit, Neha and Pooja; you made the lab a fun place to work in. Many thanks to the CSE department staff, especially Vijay, Sunanda and Rupali for their help in all administrative tasks.

My mother Indira and father Satyanarayana Murthy, both teachers, have been instrumental in motivating me towards academics, and I thank them for supporting my decision to return to university after a gap. I thank my wife Kaivalya for her unwavering support and encouragement, and for accommodating my work priorities in this endeavor. My brother Siva has been my constant companion and I am grateful for him. I thank my cousins for their support and the many wonderful times that kept me buoyed in this journey. I thank my uncles Dr. Ravikrishna Chebolu and Dr. Kameswara Rao Emani for inspiring me at various stages in my life to pursue a Ph.D. My in-laws have been very supportive and encouraging of my academic commitments, and I thank them for it.

I have been fortunate to find good friends who shared the ups and downs of Ph.D. along with me. I am grateful to Vrinda and Om for having my back through everything in the last five years. I thank Anshuj, Bharath, Chandra Prakash, Durgesh, Ramya, Saketh and Sridhar for all the fun times and great memories we have shared over the years. I would also like to express my gratitude to my colleagues at Flipkart – Ramakrishna, Neha, Ankur and Swagat – for encouraging me in my decision to pursue higher education.

I thank TCS for supporting me with a Ph.D. fellowship, and SIGMOD for the travel grant to attend and present at SIGMOD 2016.

K. Venkatesh Emani