Rethinking the Application-Database Interface

by

Alvin K. Cheung

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2015

Author . . . .
Department of Electrical Engineering and Computer Science
August 28, 2015

Certified by . . . .
Samuel Madden
Professor
Thesis Supervisor

Certified by . . . .
Armando Solar-Lezama
Associate Professor
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students

Rethinking the Application-Database Interface

by

Alvin K. Cheung

Submitted to the Department of Electrical Engineering and Computer Science
on August 28, 2015, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

Applications that interact with database management systems (DBMSs) are ubiquitous in our daily lives. Such database applications are usually hosted on an application server and perform many small accesses over the network to a DBMS hosted on a database server to retrieve data for processing. For decades, the database and programming systems research communities have worked on optimizing such applications from different perspectives: database researchers have built highly efficient DBMSs, and programming systems researchers have developed specialized compilers and runtime systems for hosting applications. However, there has been relative little work that examines the interface between these two software layers to improve application performance.

In this thesis, I show how making use of application semantics and optimizing across these layers of the software stack can help us improve the performance of database applications. In particular, I describe three projects that optimize database applications by looking at *both* the programming system and the DBMS in tandem. By carefully revisiting the interface between the DBMS and the application, and by applying a mix of declarative database optimization and modern program analysis and synthesis techniques, we show that multiple orders of magnitude speedups are possible in real-world applications. I conclude by highlighting future work in the area, and propose a vision towards automatically generating application-specific data stores.

Thesis Supervisor: Samuel Madden
Title: Professor

Thesis Supervisor: Armando Solar-Lezama
Title: Associate Professor

To my family, friends, and mentors.

Acknowledgments

Carrying out the work in this thesis would not be possible without the help of many others. First and foremost, I am very fortunate to have two great advisors who I interact with on a daily basis: Sam Madden and Armando Solar-Lezama. As an undergraduate student with close to zero knowledge of database systems, Sam courageously took me under his wings, and taught me everything from selecting optimal join orders to thinking positively in life. I am grateful to the amount of academic freedom that Sam allows me in exploring different areas in computer science, and from him I learned the importance of picking problems that are *both* interesting scientifically and matter to others outside of the computer science world. After moving out of Boston, I will miss his afternoon "wassup guys" (in a baritone voice) checkups where he would hang out in our office and discussed everything from the news to the best chocolate in town.

In addition to Sam, Armando has been my other constant source of inspirations. As someone who knows a great deal about programming systems, computer science, and life in general, I learned a lot from Armando from our many hours of discussions and working through various intricate proofs and subtle concepts. Armando has always been accessible and is someone who I can just knock on the door at any time to discuss any topic with. I will always remember the many sunrises that we see in Stata after pulling all-nighters due to conference deadlines. The amount of dedication that Armando puts into his students is simply amazing. I could not have asked for better advisors other than Armando and Sam.

I am also indebted to Andrew Myers, who introduced me to the exciting world of programming systems research. Andrew has always been extremely patient in explaining programming language concepts to a novice (me), and I learned so much about attention to details from his meticulous edits to manuscripts. Thank you also to Michael Stonebraker with his honest and insightful advice, and agreeing to be on my dissertation committee. I would also like to express gratitude to Martin Rinard and Daniel Jackson, who were members of my qualification exam committee. They have definitely shaped me into a better researcher throughout my graduate career. Finally, I am grateful to my mentor Rakesh Agrawal who introduced me to computer science research and told me to "go east!" and move to Boston for graduate studies.

My PhD journey would not be complete without the past and present illustrious members of the Database and Computer-Aided Programming groups. I have received so much intellectual and emotional support from all of them, along with all my friends on the top three floors of Stata: Ramesh Chandra, Shuo Deng, Shyam Gollakota, Shoaib Kamil, Eunsuk Kang, Taesoo Kim, Yandong Mao, Ravi Netravali, Jonathan Perry, Lenin Ravindranath, Meelap Shah, Anirudh Sivaraman, Arvind Thiagarajan, Xi Wang, and many other music group partners and friends on campus and around Boston who we shared laughs, music, and food together: Clement Chan, Godine Chan, Poting Cheung, Wang-Chi Cheung, Nicole Fong, Hee-Sun Han, Sueann Lee, Jenny Man, Chen Sun, Gary Tam, Liang Jie Wong, and I am sure I have missed many others.

Last but not least, I thank my parents and my family for their endless love and support throughout this journey. I owe all my accomplishments to them.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

From online shopping websites to banking applications, we interact with applications that store persistent data in database management systems (DBMSs) on a daily basis. As shown in Fig. 1-1, such database applications are often organized into three tiers: a database server that hosts a DBMS, an application server that hosts one part of the application, and a client that runs the end-user portion of the application. The application can be anything from a browser running on a laptop computer to a game running on a mobile phone that the end-user interacts with. To distinguish among the various parts of the application that are executed on the different servers, in this thesis I use the terms *client*, *server*, and *query component* to denote the software components that are executed on the client device, application server, and database server, respectively. This is depicted in Fig. 1-1.

Like many online transactional processing applications, developers of database applications have traditionally focused on improving throughput. In recent years, however, application latency has also become increasingly important. For instance, recent studies [11, 16] have shown that increasing application latency can have dramatic adverse effects on customer retainment for websites: as little as a half a second delay in webpage rendering can cause 20% drop in website traffic, and 57% of online shoppers will leave a website if they need to wait for 3 seconds or more in page rendering. As both the amount of persistent data and application complexity are strictly going to increase in the future, it is important to build database applications that have low latency and

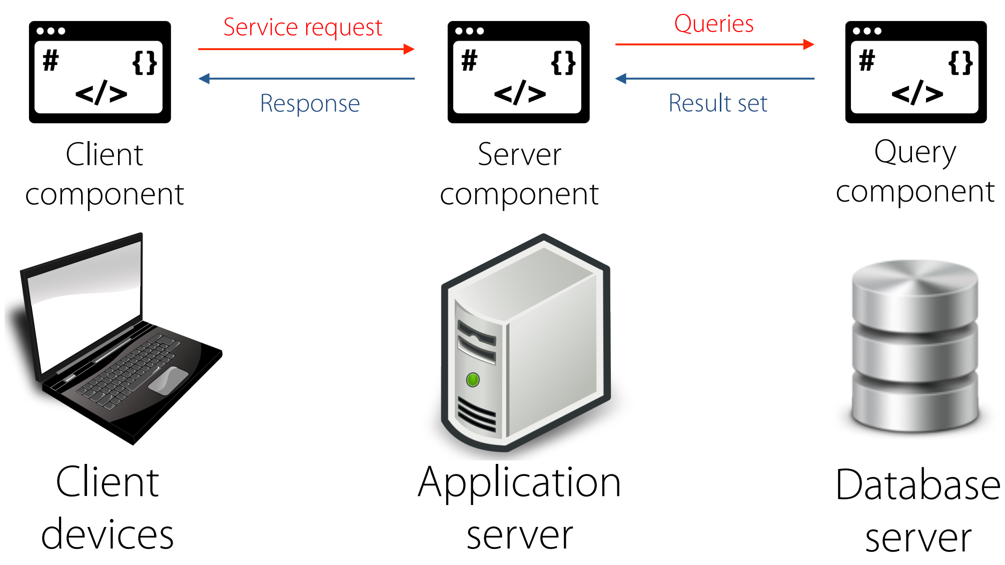


Figure 1-1: Software architecture of three-tier database applications

systems that can deliver high throughput.

Unfortunately for the developer, optimizing database applications is not an easy task. While dividing the application between the client, the application server, and database server simplifies application development by encouraging code modularity and separation of development efforts, it often results in applications that lack the desired performance. For example, the application compiler and query optimizer optimize distinct parts of the program, namely the application code and the queries that the application issues to the DBMS. Unfortunately, because the two components do not share any information, this often leads to missed opportunities to improve application performance. In the following, I first briefly review the structure of database applications. Then, through a series of projects, I discuss how using program analysis and program synthesis techniques can significantly improve the performance of database applications, without requiring developers to craft customized rewrites to the code in order to optimize their applications.

1.1 Architecture of Database Applications

As mentioned, database applications are typically hosted on an application and a database server. The application server is usually physically separated from the database server, although the two are usually in close proximity to each other (e.g., both within the same data center). Due to the close physical proximity between the application and database servers, there is usually low network latency and high transfer bandwidth between the application and database servers (on the order of microseconds for latency and gigabytes/second for bandwidth). In comparison, the network link between the client device and the application server can be substantially slower (on the order of milliseconds in terms of latency), with smaller bandwidth (on the order of megabytes/second).

Under this architecture, the way that database applications are executed is as follows. Upon receiving a request from the client component, the application server starts executing the appropriate server component of the application to handle the client's request. As shown in Fig. 1-2, the server component periodically issues requests (i.e., *queries*) to the query component hosted on DBMS as it executes in order to retrieve or manipulate data that are persistently stored. Upon receiving queries, the query component (e.g., a database driver) forwards them to the DBMS, which processes them and returns the results back to the server component. This process continues until the application finishes execution, for instance when the server component has gathered all the data needed to generate the requested web page, upon which it will forward the results back to the client component to be rendered on the client device.

1.2 Challenges in Implementing Database Applications

Unfortunately, the current way that database applications are structured and developed is often at odds with the goal of reducing latency of the application. For example, in order to optimize database applications programmers must manually make a number of decisions during application development, none of which is related to the application logic itself. First, developers need to determine where a piece of computation should be implemented (i.e., as the client, server, or the query component). Early database applications had no choice but to place most computations on

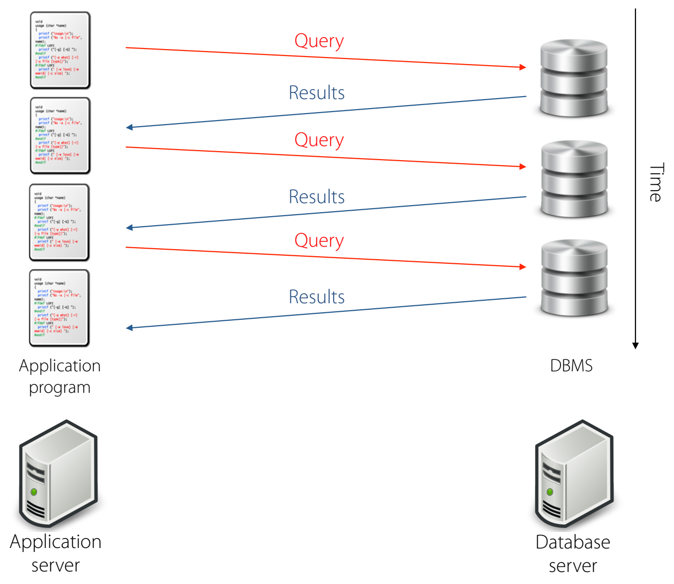


Figure 1-2: Interactions between application and database servers during database application execution

the server or query components, as clients had limited computational power available. Such limitations no longer exist given recent hardware advances. As a result, developers can now choose to implement their database application logic in any of the components. However, implementing application logic in each of the components often requires understanding different programming languages or paradigms. For instance, server components are often implemented in a general-purpose, imperative language such as Java, while database queries are written in a domain-specific query language such as SQL, and client components are often expressed using scripting languages such as Javascript. In addition to the language that is used to implement each of the components, another consequent of the above is that developers must decide where each computation is placed. For instance, DBMSs are capable of executing general-purpose code using stored procedures. Likewise, application servers can similarly execute relational operations on persistent data, for example by first fetching persistent data from the DBMS and computing on such data within the application server. Similarly, client components can be passive rendering engines of data received from the application server (e.g., a text terminal), or actively request data based on client interactions (e.g., a web browser). Finally, developers also need to be aware of the interactions among the different components of the application. For example, every interaction between the components (e.g., sending a query from the application server to the database server, and forwarding result sets from the database back to the application server) introduces a network round trip between servers, and each such interaction increases application latency. Worse yet, given the network variability across different links (e.g., the link between mobile devices and application server can be highly variable), it is very difficult to predict the performance of database applications during development.

One technique that developers employ to handle the issues listed above is to manually implement various optimizations in the code. For instance, they might implement relational operations in the server component if the results from such operations can be cached and reused in the application server. Unfortunately, while handcrafting such optimizations can yield significant performance improvements, these manual optimizations are done at the cost of code complexity, readability, and maintainability, not to mention breaking the separation of functionality between the different components in the application. Worse, these optimizations are brittle. Relatively small changes in control flow or workload pattern can have drastic consequences and cause the optimizations to hurt performance instead. Moreover, slight changes to the data table schema can break such manually optimized queries. As a result, such optimizations are effectively a black art, as they require a developer to reason about the behavior of the distributed implementation of the program across different platforms and programming paradigms, and have intimate knowledge about the implementation intricacies of the systems hosting each of the components.

These issues are further exacerbated by recent trends in database-backed application development. Specifically, the rise of Object-Relational Mapping (ORM) libraries such as Django [2], Hibernate [6], Rails [20], and query embedded languages such as LINQ [79] have further complicated cross-layer optimizations. Both mechanisms abstract away the DBMS into a series of functions represented by an API (in the case of ORMs) or new language constructs (in the case of query embedded languages). The advantage of using such mechanism is that developers can express their data requests in the same language used to implement the server component, without the need to implement a separate query component (using SQL for example). As the application executes, these function calls and language constructs are translated into queries (e.g., SQL) for the DBMS to process, and the returned results from the DBMS are converted into representations that can be processed by the server component.

Unfortunately, while using ORM libraries free developers from the need to learn separate query languages in implementing query components as they translate API calls directly to queries, they introduce other issues. For instance, as a static library that is linked with the application, ORM libraries lack high-level knowledge about the application, as they simply convert each library call into templatized queries. Although some embedded query languages postpone the conversion until runtime, they nonetheless do not take application semantics into account. As a result, the queries generated by such libraries and languages are often inefficient, as I will discuss in Ch. 2.

1.3 Thesis Contributions

Since the early days of relational DBMS development, the interface between the application and the DBMS has been one where the application sends SQL queries to the DBMS via an API function call, and blocks execution until the DBMS returns results. In this thesis, we focus on optimizing the server and query components of database applications by challenging this classical interface. We argue that this narrow interface does not allow the application to convey useful information to the DBMS beyond the explicit query to be executed, as a result forgoes many opportunities for optimizing application performance. Specifically, we revisit this interface between the application and the DBMS, and show that by moving non-query related code to the DBMS and vice versa to the application server, along with dynamically controlling when queries are executed by the DBMS and results returned to the application, we can achieve the gains of manual optimizations without incurring substantial burden on the developers. By analyzing the application program, the runtime system, and the DBMS as a whole, along with applying various program analysis and database optimization to drive program optimizations, we have shown order-of-magnitude speedups in real-world database applications. Not only that, we are able to achieve performance improvements while still allowing developers to use the same high-level programming model to develop such applications.

Concretely, I focus on three aspects of this problem in this thesis. In each case, we address each issue by designing systems that optimize both the runtime system and the DBMS as a whole by leveraging application semantics:

- First, we investigate the requirement that developers need to determine whether a piece of application logic should be implemented in the client, server, or the query component of the database application. In Ch. 2, I introduce QBS. QBS frees the developer from needing to determine whether computation should be implemented in imperative code or SQL by transforming imperative code fragments that are part of the server component into declarative queries that reside in the query component. Unlike traditional rule-based compilers, which rely on pattern-matching in order to find code fragments that can be converted, QBS formulates the code conversion problem as a code search. Furthermore, it uses program synthesis technology to prune down the search space and automatically verify that the conversion preserves semantics of the original code fragment. The query optimizer can then utilize various specialized implementations of relational operators and statistics about persistent data to determine the optimal way to execute the converted code.

- However, not all pieces of computation logic can be converted into declarative queries and take advantage of specialized implementations of query operators in the DBMS. Because of that, the database application will incur network round trips as it executes, either as a result of the server component issuing queries to the query component, or the query component returning results back to the server component. Yet, imperative code can still be transferred to the DBMS and executed as stored procedures. Unfortunately, transferring imperative code originally to be executed by the application server to the DBMS is often a double-edged sword: while doing so can reduce the number of network round trips incurred during program execution, it will also increase the load on the database server, and that might in turn increase the latency for query execution.

  To balance the two effects, I describe PYXIS in Ch. 3. PYXIS is a system that automatically partitions program logic statically across multiple servers for optimal performance. PYXIS works by first collecting a runtime profile of the application. Based on the collected profile, it partitions the original application program into the server component to be hosted on the application server, and the query component to be executed on the DBMS. PYXIS formulates code partitioning as an integer linear programming problem, with constraints being the amount of resources available on the servers (based on runtime profile and machine specifications), and the network bandwidth that is available between servers.

- While PYXIS reduces the number of network round trips between the database and application servers using static code partitioning, its results might be suboptimal as it can only estimate the amount of data transferred between the two servers using periodic profiling, and it will also take time before newly generated partitions will take effect. As an alternative to code partitioning, I introduce SLOTH in Ch. 4. SLOTH is a tool that reduces application latency by eliminating such network round trips between the two servers by batching multiple queries that are issued by the server component as the application executes. SLOTH batches queries by converting the application to execute under *lazy evaluation*—as the application executes, each time when it tries to issue a query the SLOTH runtime component intercepts and caches the query in a buffer rather than sending it to the DBMS. The application continues to execute as if the query results have been returned. Then, when it needs the results from a (seemingly) issued query, the SLOTH runtime will issue all the cached queries in a single batch, thus reducing the number of network round trips incurred.

We have achieved significant performance improvement on real-world database applications using our tools. Figure 1-3 shows an example code fragment, the converted version in SQL using QBS, and the resulting application latency improvement by multiple orders of magnitude compared to the original version. Figure 1-4 shows the latency of a database application (TPC-C in this case) implemented using JDBC and PYXIS, with the PYXIS version reducing application latency by 3×. The same figure highlights the timing experiment where some of the CPUs were shut down in the middle of the experiment (to simulate varying amount of resources available at the data center). The results show that the PYXIS-generated version of the application adapted to resource changes by switching to another pre-generated partitioning of the application. Finally, Fig. 1-5 shows the amount of load time speed up across 112 benchmarks from a real-world web-based database application (each benchmark represents a web page from the application), with performance improvement up to $2\times$ after using SLOTH. The same figure also shows the breakdown of time spent across the different servers as the benchmarks are loaded.

I describe each of the projects in detail in the following chapters, followed by an architecture for hosting and optimizing database applications by making use of the technologies discussed. Then, I highlight further opportunities in cross-system optimization in Ch. 6, followed by related work in Ch. 7, and conclude in Ch. 8.

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

```
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;
}
```

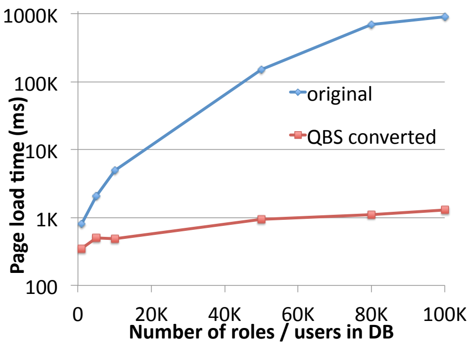
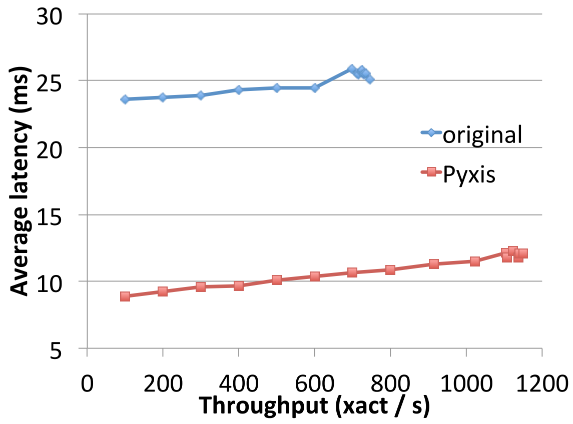


Figure 1-3: QBS results highlights: original code fragment (left); converted code fragment (middle); resulting performance comparison (right).



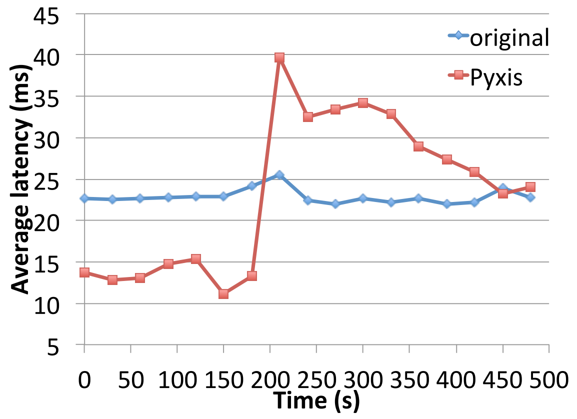
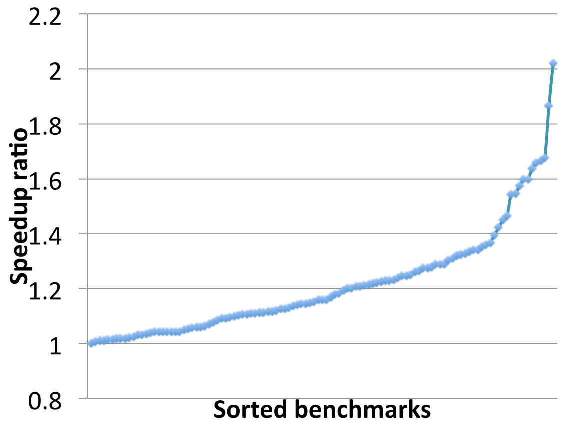


Figure 1-4: PYXIS results highlights: application latency with different throughput (left); application latency as the number of CPUs available decreased around 180ms (right).



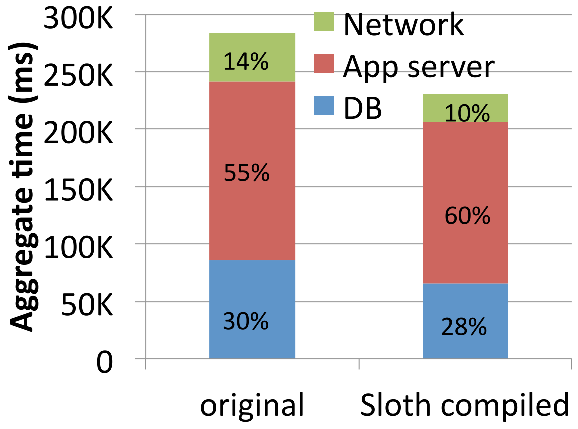


Figure 1-5: SLOTH results highlights: average page load speed up across 112 benchmarks (left); comparison of time breakdown in loading all benchmarks (right).

Chapter 2

QBS: Determining How to Execute

In this chapter I present QBS, a system that automatically transforms fragments of application logic into SQL queries. QBS differs from traditional compiler optimizations as it relies on synthesis technology to generate invariants and postconditions for a code fragment. The postconditions and invariants are expressed using a new theory of ordered relations that allows us to reason precisely about both the contents and order of the records produced complex code fragments that compute joins and aggregates. The theory is close in expressiveness to SQL, so the synthesized postconditions can be readily translated to SQL queries.

In this chapter, I discuss the enabling technologies behind QBS along with our initial prototype implementation. Using 75 code fragments automatically extracted from over 120k lines of open-source code written using the Java Hibernate ORM, our prototype can convert a variety of imperative constructs into relational specifications and significantly improve application performance asymptotically by orders of magnitude.[1]

2.1 Interacting with the DBMS

QBS (Query By Synthesis) is a new code analysis algorithm designed to make database-backed applications more efficient. Specifically, QBS identifies places where application logic can be

[1]Materials in this chapter are based on work published as Cheung, Solar-Lezama, and Madden, "Optimizing Database-Backed Applications with Query Synthesis," in proceedings of PLDI 13 [40].

converted into SQL queries issued by the application, and automatically transforms the code to do this. By doing so, QBS move functionalities that are implemented in the server component that is hosted on the application server to the query component to be executed in the DBMS. This reduces the amount of data sent from the database to the application, and it also allows the database query optimizer to choose more efficient implementations of some operations—for instance, using indices to evaluate predicates or selecting efficient join algorithms.

One specific target of QBS is programs that interact with the database through ORM libraries such as Hibernate for Java. Our optimizations are particularly important for such programs because ORM layers often lead programmers to write code that iterates over collections of database records, performing operations like filters and joins that could be better done inside of the database. Such ORM layers are becoming increasingly popular; for example, as of August, 2015, on the job board `dice.com` 13% of the 17,000 Java developer jobs are for programmers with Hibernate experience.

We are not the first researchers to address this problem; Wiedermann et al. [109, 110] identified this as the *query extraction problem*. However, our work is able to analyze a significantly larger class of source programs and generate a more expressive set of SQL queries than this prior work. Specifically, to the best of our knowledge, our work is the first that is able to identify joins and aggregates in general purpose application logic and convert them to SQL queries. Our analysis ensures that the generated queries are *precise* in that both the contents and the order of records in the generated queries are the same as those produced by the original code.

At a more foundational level, this work is the first to demonstrate the use of constraint-based synthesis technology to attack a challenging compiler optimization problem. Our approach builds on the observation by Iu et al. [67] that if we can express the postcondition for an imperative code block in relational algebra, then we can translate that code block into SQL. Our approach uses constraint-based synthesis to automatically derive loop invariants and postconditions, and then uses an SMT solver to check the resulting verification conditions. In order to make synthesis and verification tractable, we define a new *theory of ordered relations* (TOR) that is close in expressiveness to SQL, while being expressive enough to concisely describe the loop invariants necessary to verify the codes of interest. The postconditions expressed in TOR can be readily translated to

SQL, allowing them to be optimized by the database query planner and leading in some cases to orders of magnitude performance improvements.

At a high level, QBS makes the following contributions:

- We demonstrate a new approach to compiler optimization based on constraint-based synthesis of loop invariants and apply it to the problem of transforming low-level loop nests into high-level SQL queries.

- We define a theory of ordered relations that allows us to concisely represent loop invariants and postconditions for code fragments that implement SQL queries, and to efficiently translate those postconditions into SQL.

- We define a program analysis algorithm that identifies candidate code blocks that can potentially be transformed by QBS.

- We demonstrate our full implementation of QBS and the candidate identification analysis for Java programs by automatically identifying and transforming 75 code fragments in two large open source projects. These transformations result in order-of-magnitude performance improvements. Although those projects use ORM libraries to retrieve persistent data, our analysis is not specific to ORM libraries and is applicable to programs with embedded SQL queries.

2.2 QBS Overview

This section gives an overview of our compilation infrastructure and the QBS algorithm to translate imperative code fragments to SQL. I use as a running example a block of code extracted from an open source project management application [24] written using the Hibernate framework. The original code was distributed across several methods which our system automatically collapsed into a single continuous block of code as shown in Fig. 2-1. The code retrieves the list of users from the database and produces a list containing a subset of users with matching roles.

```
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 (Roles r : roles) {
      if (u.roleId().equals(r.roleId())) {
        User userok = u;
        listUsers.add(userok);
      }
    }
  }
  return listUsers;
}
```

Figure 2-1: Sample code that implements join operation in application code, abridged from actual source for clarity

```
List listUsers := [ ];
int i, j = 0;
List users := Query(SELECT * FROM users);
List roles = Query(SELECT * FROM roles);
while (i < users.size()) {
  while (j < roles.size()) {
    if (users[i].roleId = roles[j].roleId)
      listUsers := append(listUsers, users[i]);
    ++j;
  }
  ++i;
}
```

Figure 2-2: Sample code expressed in kernel language

Postcondition

$listUsers = \pi_{\ell}(\bowtie_{\varphi}(users, roles))$
where
$\varphi(e_{users}, e_{roles}) := e_{users}.roleId = e_{roles}.roleId$
ℓ contains all the fields from the *User* class

Translated code

```
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 2-3: Postcondition as inferred from Fig. 2-1 and code after query transformation

The example implements the desired functionality but performs poorly. Semantically, the code performs a relational join and projection. Unfortunately, due to the lack of global program information, the ORM library can only fetch all the users and roles from the database and perform the join in application code, without utilizing indices or efficient join algorithms the database system has access to. QBS fixes this problem by compiling the sample code to that shown at the bottom of Fig. 2-3. The nested loop is converted to an SQL query that implements the same functionality in the database where it can be executed more efficiently, and the results from the query are assigned to `listUsers` as in the original code. Note that the query imposes an order on the retrieved records; this is because in general, nested loops can constraint the ordering of the output records in ways that need to be captured by the query.

One way to convert the code fragment into SQL is to implement a syntax-driven compiler that identifies specific imperative constructs (e.g., certain loop idioms) and converts them into SQL based on pre-designed rules. Unfortunately, capturing all such constructs is extremely difficult and does not scale well to handle different variety of input programs. Instead, QBS solves the problem in three steps. First, it synthesizes a postcondition for a given code fragment, then it computes the necessary verification conditions to establish the validity of the synthesized postcondition, and it finally converts the validated postcondition into SQL. A postcondition is a Boolean predicate on the program state that is true after a piece of code is executed, and verification conditions are Boolean predicates that guarantees the correctness of a piece of code with respect to a given postcondition.

As an example, given a program statement s: `x = y;`, if we want to show that the postcondition `x = 10` holds after s is executed, then one way to do so is to check whether the predicate `y = 10` is true before executing the statement. In other words, if it is true that `y = 10` before executing s, then we can prove that the postcondition holds, assuming we know the semantics of s. In this case `y = 10` is the verification condition with respect to the postcondition `x = 10`. Note that there can be many postconditions for the same piece of code (e.g., `True` is an obvious but uninteresting one), and the corresponding verification conditions are different.

While a typical problem in program verification is to find the logically strongest postcondition[2] for a piece of code, in QBS our goal is to find a postcondition of the form `v = Q()`, where `Q` is a SQL query to be executed by the database, along with the verification conditions that establish the validity of the postcondition. Limiting the form of the postcondition greatly simplifies the search, and doing so also guarantees that the synthesized postcondition can be converted into SQL. Compared to syntax-driven approaches, solving the code conversion problem in this way allows us to handle a wide variety of code idioms. We are unaware of any prior work that uses both program synthesis and program verification to solve this problem, and I discuss the steps involved in the sections below.

2.2.1 QBS Architecture

I now discuss the architecture of QBS and describe the steps in inferring SQL queries from imperative code. The architecture of QBS is shown in Fig. 2-4.

Identify code fragments to transform. Given a database application written in Java, QBS first finds the persistent data methods in the application, which are those that fetch persistent data via ORM library calls. It also locates all entry points to the application such as servlet handlers. From each persistent data method that is reachable from the entry points, the system inlines a

[2] p is the strongest postcondition if there does not exist another postcondition p' such that $p' \rightarrow p$.

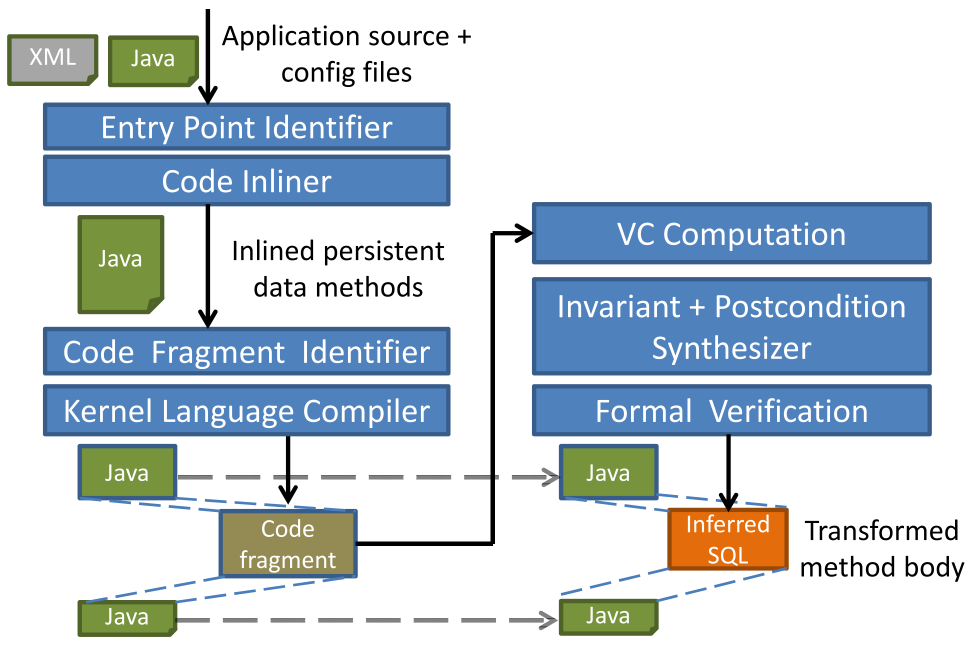


Figure 2-4: QBS architecture

neighborhood of calls, i.e., a few of the parent methods that called the persistent data method and a few of the methods called by them. If there is ambiguity as to the target of a call, all potential targets are considered up to a budget. A series of analyses is then performed on each inlined method body to identify a continuous code fragment that can be potentially transformed to SQL; ruling out, for example, code fragments with side effects. For each candidate code fragment, our system automatically detects the program variable that will contain the results from the inferred query (in the case of the running example it is `listUsers`) — we refer this as the "result variable."

In order to apply the QBS algorithm to perform the desired conversion, our system must be able to cope with the complexities of real-world Java code such as aliasing and method calls, which obscure opportunities for transformations. For example, it would not be possible to transform the code fragment in Fig. 2-1 without knowing that `getUsers` and `getRoles` execute specific queries on the database and return non-aliased lists of results, so the first step of the system is to identify promising code fragments and translate them into a simpler kernel language shown in Fig. 2-5.

The kernel language operates on three types of values: scalars, immutable records, and immutable lists. Lists represent the collections of records and are used to model the results that are

$$\begin{array}{rcl} c \in \text{constant} & ::= & \text{True} \mid \text{False} \mid \text{number literal} \mid \text{string literal} \\ e \in \text{expression} & ::= & c \mid [\,] \mid var \mid e.f \mid \{f_i = e_i\} \mid e_1\ op\ e_2 \mid \neg\, e \\ & \mid & \text{Query}(...) \mid \text{size}(e) \mid \text{get}_{e_r}(e_s) \\ & \mid & \text{append}(e_r, e_s) \mid \text{unique}(e) \\ c \in \text{command} & ::= & \text{skip} \mid var := e \mid \text{if}(e)\ \text{then}\ c_1\ \text{else}\ c_2 \\ & \mid & \text{while}(e)\ \text{do}\ c \mid c_1\ ;\ c_2 \mid \text{assert}\ e \\ op \in \text{binary op} & ::= & \wedge \mid \vee \mid > \mid = \end{array}$$

Figure 2-5: Abstract syntax of the kernel language

returned from database retrieval operations. Lists store either scalar values or records constructed with scalars, and nested lists are assumed to be appropriately flattened. The language currently does not model the three-valued logic of null values in SQL, and does not model updates to the database. The semantics of the constructs in the kernel language are mostly standard, with a few new ones introduced for record retrievals. Query(...) retrieves records from the database and the results are returned as a list. The records of a list can be randomly accessed using get, and records can be appended to a list using append. Finally, unique takes in a list and creates a new list with all duplicate records removed. Fig. 2-2 shows the example translated to the kernel language. At the end of the code identification process, QBS would have selected a set of code fragments that are candidates to be converted into SQL, and furthermore compile each of them into the kernel language as discussed.

Compute verification conditions. As the next step, the system computes the verification conditions of the code fragment expressed in the kernel language. The verification conditions are written using the predicate language derived from the theory of ordered relations to be discussed in Sec. 2.3. The procedure used to compute verification conditions is a fairly standard one [51, 57]; the only twist is that the verification condition must be computed in terms of an unknown postcondition and loop invariants. The process of computing verification conditions is discussed in more detail in Sec. 2.4.1.

Synthesize loop invariants and postconditions. The definitions of the postcondition and invariants need to be filled in and validated before translation can proceed. QBS does this using a synthesis-based approach that is similar to prior work [100], where a synthesizer is used to come up with a postcondition and invariants that satisfy the computed verification conditions. The synthesizer uses a symbolic representation of the space of candidate postconditions and invariants, and efficiently identifies candidates within the space that are correct according to a bounded verification procedure. It then uses a theorem prover (Z3 [80], specifically) to check if those candidates can be proven correct. The space of candidate invariants and postconditions is described by a template generated automatically by the compiler. To prevent the synthesizer from generating trivial postconditions (such as `True`), the template limits the synthesizer to only generate postconditions that can be translated to SQL as defined by our theory of ordered relations, such as that shown at the top of Fig. 2-3.

As mentioned earlier, we observe that it is not necessary to determine the strongest invariants or postconditions: we are only interested in finding postconditions that allow us transform the input code fragment into SQL. In the case of the running example, we are only interested in finding a postcondition of the form `listUsers =` *Query*(...), where *Query*(...) is an expression translatable to SQL. Similarly, we only need to discover loop invariants that are strong enough to prove the postcondition of interest. From the example shown in Fig. 2-1, our system infers the postcondition shown at the top of Fig. 2-3, where π, σ, and $\bowtie$ are ordered versions of relational projection, selection, and join, respectively to be defined in Sec. 2.3. The process of automatic template generation from the input code fragment and synthesis of the postcondition from the template are discussed in Sec. 2.4.

Unfortunately, determining loop invariants is undecidable for arbitrary programs [32], so there will be programs for which the necessary invariants fall outside the space defined by the templates. However, our system is significantly more expressive than the state of the art as demonstrated by our experiments in Sec. 2.7.

Convert to SQL. After the theorem prover verifies that the computed invariants and postcondition are correct, the input code fragment is translated to SQL, as shown in the bottom of Fig. 2-3. The predicate language defines syntax-driven rules to translate any expressions in the language into valid SQL. The details of validation is discussed in Sec. 2.5 while the rules for SQL conversion are introduced in Sec. 2.3.2. The converted SQL queries are patched back into the original code fragments and compiled as Java code.

2.3 Theory of Finite Ordered Relations

As discussed in Sec. 2.2, QBS synthesizes a postcondition and the corresponding verification conditions for a given code fragment before converting it into SQL. To do so, we need a language to express these predicates, and axioms that allow us to reason about terms in this language. For this purpose, QBS uses a theory of finite ordered relations. The theory is defined to satisfy four main requirements: precision, expressiveness, conciseness, and ease of translation to SQL. For precision, we want to be able to reason about *both* the contents and order of records retrieved from the database. This is important because in the presence of joins, the order of the result list will not be arbitrary even when the original list was arbitrary, and we do not know what assumptions the rest of the program makes on the order of records. The theory must also be expressive enough not just to express queries but also to express invariants, which must often refer to partially constructed lists. For instance, the loop invariants for the sample code fragment in Fig. 2-1 must express the fact that `listUsers` is computed from the first `i` and `j` records of `users` and `roles` respectively. Conciseness, e.g., the number of relational operators involved, is important because the complexity of synthesis grows with the size of the synthesized expressions, so if we can express invariants succinctly, we will be able to synthesize them more efficiently. Finally, the inferred postconditions must be translatable to standard SQL.

There are many ways to model relational operations (see Ch. 7), but we are not aware of any that fulfills all of the criteria above. For example, relational algebra is not expressive enough to describe sufficiently precise loop invariants. Defined in terms of sets, relational algebra cannot naturally express concepts such as "the first `i` elements of the list." First order logic (FOL), on the other hand, is very expressive, but it would be hard to translate arbitrary FOL expressions into SQL.

$$
\begin{array}{rcl}
c \in \text{constant} & ::= & \text{True} \mid \text{False} \mid \text{number literal} \mid \text{string literal} \\
e \in \text{expression} & ::= & c \mid [\,] \mid \text{program var} \mid \{f_i = e_i\} \mid e_1 \; op \; e_2 \mid \neg \, e \\
 & \mid & \text{Query}(...) \mid \text{size}(e) \mid \text{get}_{e_s}(e_r) \mid \text{top}_{e_s}(e_r) \\
 & \mid & \pi_{[f_{i_1},\ldots,f_{i_N}]}(e) \mid \sigma_{\varphi_\sigma}(e) \mid \bowtie_{\varphi_\bowtie}(e_1, e_2) \\
 & \mid & \text{sum}(e) \mid \text{max}(e) \mid \text{min}(e) \\
 & \mid & \text{append}(e_r, e_s) \mid \text{sort}_{[f_{i_1},\ldots,f_{i_N}]}(e) \mid \text{unique}(e) \\
op \in \text{binary op} & ::= & \wedge \mid \vee \mid > \mid = \\
\varphi_\sigma \in \text{select func} & ::= & p_{\sigma_1} \wedge \ldots \wedge p_{\sigma_N} \\
p_\sigma \in \text{select pred} & ::= & e.f_i \; op \; c \mid e.f_i \; op \; e.f_j \mid \text{contains}(e, e_r) \\
\varphi_\bowtie \in \text{join func} & ::= & p_{\bowtie_1} \wedge \ldots \wedge p_{\bowtie_N} \\
p_\bowtie \in \text{join pred} & ::= & e_1.f_i \; op \; e_2.f_j
\end{array}
$$

Figure 2-6: Abstract syntax for the predicate language based on the theory of ordered relations

2.3.1 Basics

Our theory of finite ordered relations is essentially relational algebra defined in terms of lists instead of sets. The theory operates on three types of values: scalars, records, and ordered relations of finite length. Records are collections of named fields, and an ordered relation is a finite list of records. Each record in the relation is labeled with an integer index that can be used to fetch the record. Figure 2-6 presents the abstract syntax of the theory and shows how to combine operators to form expressions.

The semantics of the operators in the theory are defined recursively by a set of axioms; a sample of which is shown in Fig. 2-7. get and top take in an ordered relation e_r and return the record stored at index e_s or all the records from index 0 up to index e_s respectively. The definitions for π, σ and $\bowtie$ are modeled after relational projection, selection, and join respectively, but they also define an order for the records in the output relation relative to those in the input relations. The projection operator π creates new copies of each record, except that for each record only those fields listed in $[f_{i_1}, \ldots, f_{i_N}]$ are retained. Like projection in relational algebra, the same field can be replicated multiple times. The σ operator uses a selection function φ_σ to filter records from the input relation. φ_σ is defined as a conjunction of predicates, where each predicate can compare the value of a

$$\boxed{\mathsf{size}}$$

$$\frac{r = [\,]}{\mathsf{size}(r) = 0} \qquad \frac{r = h : t}{\mathsf{size}(r) = 1 + \mathsf{size}(t)}$$

$$\boxed{\mathsf{get}}$$

$$\frac{i = 0 \quad r = h : t}{\mathsf{get}_i(r) = h} \qquad \frac{i > 0 \quad r = h : t}{\mathsf{get}_i(r) = \mathsf{get}_{i-1}(t)}$$

$$\boxed{\mathsf{append}}$$

$$\frac{r = [\,]}{\mathsf{append}(r, t) = [t]} \qquad \frac{r = h : t}{\mathsf{append}(r, t') = h : \mathsf{append}(t, t')}$$

$$\boxed{\mathsf{top}}$$

$$\frac{r = [\,]}{\mathsf{top}_r(i) = [\,]} \qquad \frac{i = 0}{\mathsf{top}_r(i) = [\,]} \qquad \frac{i > 0 \quad r = h : t}{\mathsf{top}_r(i) = h : \mathsf{top}_t(i-1)}$$

$$\boxed{\mathsf{join}\ (\bowtie)}$$

$$\frac{r_1 = [\,]}{\bowtie_\varphi(r_1, r_2) = [\,]} \qquad \frac{r_2 = [\,]}{\bowtie_\varphi(r_1, r_2) = [\,]}$$

$$\frac{r_1 = h : t}{\bowtie_\varphi(r_1, r_2) = \ \mathsf{cat}(\bowtie'_\varphi(h, r_2), \bowtie_\varphi(t, r_2))}$$

$$\frac{r_2 = h : t \quad \varphi(e, h) = \mathsf{True}}{\bowtie'_\varphi(e, r_2) = (e, h) : \bowtie'_\varphi(e, t)} \qquad \frac{r_2 = h : t \quad \varphi(e, h) = \mathsf{False}}{\bowtie'_\varphi(e, r_2) = \bowtie'_\varphi(e, t)}$$

$$\boxed{\mathsf{projection}\ (\pi)}$$

$$\frac{r = [\,]}{\pi_\ell(r) = [\,]} \qquad \frac{r = h : t \quad f_i \in \ell \quad h.f_i = e_i}{\pi_\ell(r) = \{f_i = e_i\} : \pi_\ell(t)}$$

$$\boxed{\mathsf{selection}\ (\sigma)}$$

$$\frac{r = [\,]}{\sigma_\varphi(r) = [\,]}$$

$$\frac{r = h : t \quad \varphi(h) = \mathsf{True}}{\sigma_\varphi(r) = h : \sigma_\varphi(t)} \qquad \frac{r = h : t \quad \varphi(h) = \mathsf{False}}{\sigma_\varphi(r) = \sigma_\varphi(t)}$$

$$\boxed{\mathsf{sum}}$$

$$\frac{r = [\,]}{\mathsf{sum}(r) = 0} \qquad \frac{r = h : t}{\mathsf{sum}(r) = h + \mathsf{sum}(t)}$$

$$\boxed{\mathsf{max}}$$

$$\frac{r = [\,]}{\mathsf{max}(r) = -\infty}$$

$$\frac{r = h : t \quad h > \mathsf{max}(t)}{\mathsf{max}(r) = h} \qquad \frac{r = h : t \quad h \leq \mathsf{max}(t)}{\mathsf{max}(r) = \mathsf{max}(t)}$$

$$\boxed{\mathsf{min}}$$

$$\frac{r = [\,]}{\mathsf{min}(r) = \infty}$$

$$\frac{r = h : t \quad h < \mathsf{min}(t)}{\mathsf{min}(r) = h} \qquad \frac{r = h : t \quad h \geq \mathsf{min}(t)}{\mathsf{min}(r) = \mathsf{min}(t)}$$

$$\boxed{\mathsf{contains}}$$

$$\frac{r = [\,]}{\mathsf{contains}(e, r) = \mathsf{False}}$$

$$\frac{e = h \quad r = h : t}{\mathsf{contains}(e, r) = \mathsf{True}} \qquad \frac{e \neq h \quad r = h : t}{\mathsf{contains}(e, r) = \mathsf{contains}(e, t)}$$

Figure 2-7: Axioms that define the theory of ordered relations

record field and a constant, the values of two record fields, or check if the record is contained in another relation e_r using contains. Records are added to the resulting relation if the function returns True. The $\bowtie$ operator iterates over each record from the first relation and pairs it with each record from the second relation. The two records are passed to the join function $\varphi_{\bowtie}$. Join functions are similar to selection functions, except that predicates in join functions compare the values of the fields from the input ordered relations. The axioms that define the aggregate operators max, min, and sum assume that the input relation contains only one numeric field, namely the field to aggregate upon.

The definitions of unique and sort are standard; in the case of sort, $[f_{i_1}, \ldots, f_{i_N}]$ contains the list of fields to sort the relation by. QBS does not actually reason about these two operations in terms of their definitions; instead it treats them as uninterpreted functions with a few algebraic properties, such as

$$\bowtie_{\varphi}(\mathsf{sort}_{\ell_1}(r_1), \mathsf{sort}_{\ell_2}(r_2)) = \mathsf{sort}_{\mathsf{cat}(\ell_1,\ell_2)}(\bowtie_{\varphi}(r_1, r_2)).$$

(where cat concatenates two lists together) Because of this, there are some formulas involving sort and unique that we cannot prove, but we have not found this to be significant in practice (see Sec. 2.7 for details).

2.3.2 Translating to SQL

The expressions defined in the predicate grammar can be converted into semantically equivalent SQL queries. In this section we prove that any expression that does not use append or unique can be compiled into an equivalent SQL query. We prove this in three steps; first, we define base and sorted expressions, which are formulated based on SQL expressions without and with ORDER BY clauses respectively. Next, we define *translatable expressions* and show that any expression that does not use append or unique can be converted into a translatable expression. Then we show how to produce SQL from translatable expressions.

Definition 1 (Translatable Expressions). *Any* transExp *as defined below can be translated into SQL:*

$$\begin{aligned} b \in \mathsf{baseExp} &::= \mathsf{Query}(...) \mid \mathsf{top}_e(s) \mid \bowtie_{\mathsf{True}}(b_1, b_2) \mid \mathsf{agg}(t) \\ s \in \mathsf{sortedExp} &::= \pi_{\ell_\pi}(\mathsf{sort}_{\ell_s}(\sigma_\varphi(b))) \\ t \in \mathsf{transExp} &::= s \mid \mathsf{top}_e(s) \end{aligned}$$

where the term agg *in the grammar denotes any of the aggregation operators (*min, max, sum, size*).*

Theorem 1 (Completeness of Translation Rules). *All expressions in the predicate grammar in Fig. 2-6, except for those that contain* append *or* unique*, can be converted into translatable expressions.*

The theorem is proved by defining a function Trans that maps any expression to a translatable expression and showing that the mapping is semantics preserving. The definition of Trans relies on a number of TOR expression equivalences:

Theorem 2 (Operator Equivalence). *The following equivalences hold, both in terms of the contents of the relations and also the ordering of the records in the relations:*

- $\sigma_\varphi(\pi_\ell(r)) = \pi_\ell(\sigma_\varphi(r))$
- $\sigma_{\varphi_2}(\sigma_{\varphi_1}(r)) = \sigma_{\varphi'}(r)$*, where* $\varphi' = \varphi_2 \wedge \varphi_1$
- $\pi_{\ell_2}(\pi_{\ell_1}(r)) = \pi_{\ell'}(r)$*, where* ℓ' *is the concatenation of all the fields in* ℓ_1 *and* ℓ_2*.*
- $\mathsf{top}_e(\pi_\ell(r)) = \pi_\ell(\mathsf{top}_e(r))$
- $\mathsf{top}_{e_2}(\mathsf{top}_{e_1}(r)) = \mathsf{top}_{\mathsf{max}(e_1,e_2)}(r)$
- $\bowtie_\varphi(r_1, r_2) = \sigma_{\varphi'}(\bowtie_{\mathsf{True}}(r_1, r_2))$*, i.e., joins can be converted into cross products with selections with proper renaming of fields.*
- $\bowtie_\varphi(\mathsf{sort}_{\ell_1}(r_1), \mathsf{sort}_{\ell_2}(r_2)) = \mathsf{sort}_{\ell_1:\ell_2}(\bowtie_\varphi(r_1, r_2))$

- $\bowtie_\varphi (\pi_{\ell_1}(r_1), \pi_{\ell_2}(r_2)) = \pi_{\ell'}(\bowtie_\varphi (r_1, r_2))$, *where* ℓ' *is the concatenation of all the fields in* ℓ_1 *and* ℓ_2.

Except for the equivalences involving sort, the other ones can be proven easily from the axiomatic definitions.

Given the theorem above, the definition of Trans is shown in Fig. 2-8. Semantic equivalence between the original and the translated expression is proved using the expression equivalences listed in Thm. 2. Using those equivalences, for example, we can show that for $s \in \mathsf{sortedExp}$ and $b \in \mathsf{baseExp}$:

$$\begin{aligned}
\mathsf{Trans}(\sigma_{\varphi'_\sigma}(s)) &= \mathsf{Trans}(\sigma_{\varphi'}(\pi_{\ell_\pi}(\mathsf{sort}_{\ell_s}(\sigma_\varphi(b))))) && \text{[sortedExp definition]} \\
&= \pi_{\ell_\pi}(\mathsf{sort}_{\ell_s}(\sigma_{\varphi_\sigma \wedge \varphi'_\sigma}(b))) && \text{[Trans definition]} \\
&= \pi_{\ell_\pi}(\sigma_{\varphi'_\sigma}(\mathsf{sort}_{\ell_s}(\sigma_{\varphi_\sigma}(b)))) && \text{[expression equivalence]} \\
&= \sigma_{\varphi'_\sigma}(\pi_{\ell_\pi}(\mathsf{sort}_{\ell_s}(\sigma_{\varphi_\sigma}(b)))) && \text{[expression equivalence]} \\
&= \sigma_{\varphi'_\sigma}(s) && \text{[sortedExp definition]}
\end{aligned}$$

Thus the semantics of the original TOR expression is preserved.

Translatable expressions to SQL. Following the syntax-directed rules in Fig. 2-9, any translatable expression can be converted into an equivalent SQL expression. Most rules in Fig. 2-9 are direct translations from the operators in the theory into their SQL equivalents.

One important aspect of the translation is the way that ordering of records is preserved. Ordering is problematic because although the operators in the theory define the order of the output in terms of the order of their inputs, SQL queries are not guaranteed to preserve the order of records from nested sub-queries; e.g., the ordering imposed by an ORDER BY clause in a nested query is not guaranteed to be respected by an outer query that does not impose any ordering on the records.

To solve this problem, the translation rules introduce a function Order—defined in Fig. 2-10—which scans a translatable expression t and returns a list of fields that are used to order the subexpressions in t. The list is then used to impose an ordering on the outer SQL query with an

$\boxed{\mathsf{Query}(...)}$

$$\mathsf{Trans}(\mathsf{Query}(...))) = \pi_\ell(\mathsf{sort}_{[\,]}(\sigma_{\mathsf{True}}(\mathsf{Query}(...))))$$

where ℓ projects all the fields from the input relation.

$\boxed{\pi_{\ell_2}(t)}$

$$\begin{aligned}\mathsf{Trans}(\pi_{\ell_2}(s)) &= \pi_{\ell'}(\mathsf{sort}_{\ell_s}(\sigma_\varphi(b)))\\ \mathsf{Trans}(\pi_{\ell_2}(\mathsf{top}_e(s))) &= \mathsf{top}_e(\pi_{\ell'}(\mathsf{sort}_{\ell_s}(\sigma_\varphi(b))))\end{aligned}$$

where ℓ' is the composition of ℓ_π and ℓ_2.

$\boxed{\sigma_{\varphi_2}(t)}$

$$\mathsf{Trans}(\sigma_{\varphi_2}(s)) = \pi_{\ell_\pi}(\mathsf{sort}_{\ell_s}(\sigma_{\varphi\wedge\varphi_2}(b)))$$

$$\mathsf{Trans}(\sigma_{\varphi_2}(\mathsf{top}_e(s))) \\ = \mathsf{top}_e(\pi_{\ell_\pi}(\mathsf{sort}_{\ell_s}(\sigma_{\varphi\wedge\varphi_2}(b))))$$

$\boxed{\bowtie_{\varphi_\bowtie}(t_1, t_2))}$

$$\mathsf{Trans}(\bowtie_{\varphi_\bowtie}(s_1, s_2)) \\ = \pi_{\ell'_\pi}(\mathsf{sort}_{\ell'_s}(\sigma_{\varphi'_\sigma}(\bowtie_{\mathsf{True}}(b_1, b_2))))$$

where $\varphi'_\sigma = \varphi_{\sigma_1} \wedge \varphi_{\sigma_2} \wedge \varphi_\bowtie$ with field names properly renamed, $\ell'_s = \mathsf{cat}(\ell_{s_1}, \ell_{s_2})$, and $\ell'_\pi = \mathsf{cat}(\ell_{\pi_1}, \ell_{\pi_2})$.

$$\mathsf{Trans}(\bowtie_\varphi(\mathsf{top}_e(s_1), \mathsf{top}_e(s_2))) \\ = \pi_\ell(\mathsf{sort}_{[\,]}(\sigma_\varphi(\bowtie_{\mathsf{True}}(\mathsf{top}_e(s_1), \mathsf{top}_e(s_2)))))$$

where ℓ contains all the fields from s_1 and s_2.

$\boxed{\mathsf{top}_{e_2}(t)}$

$$\begin{aligned}\mathsf{Trans}(\mathsf{top}_{e_2}(s)) &= \mathsf{top}_{e_2}(s)\\ \mathsf{Trans}(\mathsf{top}_{e_2}(\mathsf{top}_{e_1}(s))) &= \mathsf{top}_{e'}(s)\end{aligned}$$

where e' is the minimum value of e_1 and e_2.

$\boxed{\mathsf{agg}(t)}$

$$\begin{aligned}\mathsf{Trans}(\mathsf{agg}(s)) &= \pi_\ell(\mathsf{sort}_{[\,]}(\sigma_{\mathsf{True}}(\mathsf{agg}(s))))\\ \mathsf{Trans}(\mathsf{agg}(\mathsf{top}_e(s))) &= \pi_\ell(\mathsf{sort}_{[\,]}(\sigma_{\mathsf{True}}(\mathsf{agg}(s))))\end{aligned}$$

where ℓ contains all the fields from s.

$\boxed{\mathsf{sort}_{\ell_{s_2}}(t)}$

$$\begin{aligned}\mathsf{Trans}(\mathsf{sort}_{\ell_{s_2}}(s)) &= \pi_{\ell_\pi}(\mathsf{sort}_{\ell'_s}(\sigma_\varphi(b)))\\ \mathsf{Trans}(\mathsf{sort}_{\ell_{s_2}}(\mathsf{top}_e(s))) &= \mathsf{top}_e(\pi_{\ell_\pi}(\mathsf{sort}_{\ell'_s}(\sigma_\varphi(b))))\end{aligned}$$

where $\ell'_s = \mathsf{cat}(\ell_s, \ell_{s_2})$.

Let $s = \pi_{\ell_\pi}(\mathsf{sort}_{\ell_s}(\sigma_\varphi(b)))$. Trans is defined on expressions whose subexpressions (if any) are in translatable form, so we have to consider cases where the sub-expressions are either s or $\mathsf{top}_e(s)$. Each case is defined above.

Figure 2-8: Definition of Trans

$$
\begin{array}{rcl}
[\![\mathsf{Query}(\mathit{string})]\!] & = & (\ \mathit{string}\ ) \\
[\![\mathsf{top}_e(s)]\!] & = & \mathsf{SELECT}\ *\ \mathsf{FROM}\ [\![s]\!]\ \mathsf{LIMIT}\ [\![e]\!] \\
[\![\bowtie_{\mathsf{True}}(t_1, t_2)]\!] & = & \mathsf{SELECT}\ *\ \mathsf{FROM}\ [\![t_1]\!], [\![t_2]\!] \\
[\![\mathsf{agg}(t)]\!] & = & \mathsf{SELECT}\ \mathsf{agg}(\mathit{field})\ \mathsf{FROM}\ [\![t]\!] \\
[\![\pi_{\ell_1}(\mathsf{sort}_{\ell_2}(\sigma_{\varphi_\sigma}(t)))]\!] & = & \mathsf{SELECT}\ [\![\ell_1]\!]\ \mathsf{FROM}\ [\![t]\!]\ \mathsf{WHERE}\ [\![\varphi_\sigma]\!] \\
 & & \mathsf{ORDER\ BY}\ [\![\ell_2]\!], \mathsf{Order}(t) \\
[\![\mathsf{unique}(t)]\!] & = & \mathsf{SELECT\ DISTINCT}\ *\ \mathsf{FROM}\ [\![t]\!] \\
 & & \mathsf{ORDER\ BY}\ \mathsf{Order}(t) \\
[\![\varphi_\sigma(e)]\!] & = & [\![e]\!].f_1\ \mathit{op}\ [\![e]\!]\ \mathsf{AND}\ \ldots\ \mathsf{AND}\ [\![e]\!].f_N\ \mathit{op}\ [\![e]\!] \\
[\![\mathsf{contains}(e, t)]\!] & = & [\![e]\!]\ \mathsf{IN}\ [\![t]\!] \\
[\![[f_{i_1}, \ldots, f_{i_N}]]\!] & = & f_{i_1}, \ldots, f_{i_N}
\end{array}
$$

Figure 2-9: Syntactic rules to convert translatable expressions to SQL

ORDER BY clause. One detail of the algorithm not shown in the figure is that some projections in the inner queries need to be modified so they do not eliminate fields that will be needed by the outer ORDER BY clause, and that we assume Query(...) is ordered by the order in which the records are stored in the database (unless the query expression already includes an ORDER BY clause).

Append and Unique. The append operation is not included in translatable expressions because there is no simple means to combine two relations in SQL that preserves the ordering of records in the resulting relation.[3] We can still translate unique, however, using the SELECT DISTINCT construct at the outermost level, as Fig. 2-9 shows. Using unique in nested expressions, however, can change the semantics of the results in ways that are difficult to reason about (e.g., $\mathsf{unique}(\mathsf{top}_e(r))$ is not equivalent to $\mathsf{top}_e(\mathsf{unique}(r))$). Thus, the only expressions with unique that we translate to SQL are those that use it at the outermost level. In our experiments, we found that omitting those two operators did not significantly limit the expressiveness of the theory.

With the theory in mind, I now turn to the process of computing verification conditions of the input code fragments.

[3]One way to preserve record ordering in list append is to use case expressions in SQL, although some database systems such as SQL Server limit the number of nested case expressions.

$$\begin{aligned}\mathsf{Order}(\mathsf{Query}(...)) &= \text{[record order in DB]} & \mathsf{Order}(\mathsf{agg}(e)) &= [\,]\\ \mathsf{Order}(\mathsf{top}_i(e)) &= \mathsf{Order}(e) & \mathsf{Order}(\mathsf{unique}(e)) &= \mathsf{Order}(e)\\ \mathsf{Order}(\pi_\ell(e)) &= \mathsf{Order}(e) & \mathsf{Order}(\sigma_\varphi(e)) &= \mathsf{Order}(e)\end{aligned}$$

$$\begin{aligned}\mathsf{Order}(\bowtie_\varphi(e_1, e_2)) &= \mathsf{cat}(\mathsf{Order}(e_1), \mathsf{Order}(e_2))\\ \mathsf{Order}(\mathsf{sort}_\ell(e)) &= \mathsf{cat}(\ell, \mathsf{Order}(e))\end{aligned}$$

Figure 2-10: Definition of Order

$$i\ op \left\{\begin{array}{c} i \mid \mathsf{size}(users) \mid \mathsf{size}(roles) \mid \mathsf{size}(listUsers) \mid \\ \mathsf{sum}(\pi_\ell(users) \mid \mathsf{sum}(\pi_\ell(roles) \mid \mathsf{max}(\pi_\ell(users) \mid \\ \text{[other relational expressions that return a scalar value]} \end{array}\right\} \wedge$$

$$listUsers = \left\{\begin{array}{c} listUsers \mid \sigma_\varphi(users) \mid \\ \pi_\ell(\bowtie_\varphi(\mathsf{top}_{e_1}(users), \mathsf{top}_{e_2}(roles))) \mid \\ \pi_\ell(\bowtie_{\varphi_3}(\sigma_{\varphi_1}(\mathsf{top}_{e_1}(users), \sigma_{\varphi_2}(\mathsf{top}_{e_2}(roles))))) \mid \\ \text{[other relational expressions that return an ordered list]} \end{array}\right\}$$

Figure 2-11: Space of possible invariants for the outer loop of the running example.

2.4 Synthesis of Invariants and Postconditions

Given an input code fragment in the kernel language, the next step in QBS is to come up with an expression for the result variable of the form $resultVar = e$, where e is a translatable expression as defined in Sec. 2.3.2.

2.4.1 Computing Verification Conditions

In order to infer the postcondition, we compute *verification conditions* for the input code fragment using standard techniques from axiomatic semantics [63]. As in traditional Hoare style verification, computing the verification condition of the while statements involves a loop invariant. Unlike traditional computation of verification conditions, however, both the postcondition and the loop invariants are unknown when the conditions are generated. This does not pose problems for QBS as we simply treat invariants (and the postcondition) as unknown predicates over the program

Verification conditions for the outer loop (oInv = outerLoopInvariant, iInv = innerLoopInvariant, pcon = postCondition)	
initialization	$\mathsf{oInv}(0, \mathit{users}, \mathit{roles}, [\,])$
loop exit	$i \geq \mathsf{size}(\mathit{users}) \land \mathsf{oInv}(i, \mathit{users}, \mathit{roles}, \mathit{listUsers}) \rightarrow \mathsf{pcon}(\mathit{listUsers}, \mathit{users}, \mathit{roles})$
perservation	(same as inner loop initialization)
Verification conditions for the inner loop	
initialization	$i < \mathsf{size}(\mathit{users}) \land \mathsf{oInv}(i, \mathit{users}, \mathit{roles}, \mathit{listUsers}) \rightarrow \mathsf{iInv}(i, 0, \mathit{users}, \mathit{roles}, \mathit{listUsers})$
loop exit	$j \geq \mathsf{size}(\mathit{roles}) \land \mathsf{iInv}(i, j, \mathit{users}, \mathit{roles}, \mathit{listUsers}) \rightarrow \mathsf{oInv}(i+1, \mathit{users}, \mathit{roles}, \mathit{listUsers})$
preservation	$j < \mathsf{size}(\mathit{roles}) \land \mathsf{iInv}(i, j, \mathit{users}, \mathit{roles}, \mathit{listUsers}) \rightarrow (\mathsf{get}_i(\mathit{users}).\mathit{id} = \mathsf{get}_j(\mathit{roles}).\mathit{id} \;\land\; \mathsf{iInv}(i, j+1, \mathit{users}, \mathit{roles}, \mathsf{append}(\mathit{listUsers}, \mathsf{get}_i(\mathit{users})))) \lor (\mathsf{get}_i(\mathit{users}).\mathit{id} \neq \mathsf{get}_j(\mathit{roles}).\mathit{id} \;\land\; \mathsf{iInv}(i, j+1, \mathit{users}, \mathit{roles}, \mathit{listUsers}))$

Figure 2-12: Verification conditions for the running example

variables that are currently in scope when the loop is entered.

As an example, Fig. 2-12 shows the verification conditions that are generated for the running example. In this case, the verification conditions are split into two parts, with invariants defined for both loops.

The first two assertions describe the behavior of the outer loop on line 5, with the first one asserting that the outer loop invariant must be true on entry of the loop (after applying the rule for the assignments prior to loop entry), and the second one asserting that the postcondition for the loop is true when the loop terminates. The third assertion states that the inner loop invariant is true when it is first entered, given that the outer loop condition and loop invariant are true. The preservation assertion is the inductive argument that the inner loop invariant is preserved after executing one iteration of the loop body. The list *listUsers* is either appended with a record from $\mathsf{get}_i(\mathit{users})$, or remains unchanged, depending on whether the condition for the if statement, $\mathsf{get}_i(\mathit{users}).\mathit{id} = \mathsf{get}_j(\mathit{roles}).\mathit{id}$, is true or not. Finally, the loop exit assertion states that the outer loop invariant is valid when the inner loop terminates.

2.4.2 Constraint based synthesis

The goal of the synthesis step is to derive postcondition and loop invariants that satisfy the verification conditions generated in the previous step. We synthesize these predicates using the SKETCH constraint-based synthesis system [99]. In general, SKETCH takes as input a program with "holes" and uses a counterexample guided synthesis algorithm (CEGIS) to efficiently search the space of all possible completions to the holes for one that is correct according to a bounded model checking procedure. For QBS, the program is a simple procedure that asserts that the verification conditions hold for all possible values of the free variables within a certain bound. For each of the unknown predicates, the synthesizer is given a *sketch* (i.e., a template) that defines a space of possible predicates which the synthesizer will search. The sketches are automatically generated by QBS from the kernel language representation.

2.4.3 Inferring the Space of Possible Invariants

Recall that each invariant is parameterized by the current program variables that are in scope. Our system assumes that each loop invariant is a conjunction of predicates, with each predicate having the form $lv = e$, where lv is a program variable that is modified within the loop, and e is an expression in TOR.

The space of expressions e is restricted to expressions of the same static type as lv involving the variables that are in scope. The system limits the size of expressions that the synthesizer can consider, and incrementally increases this limit if the synthesizer fails to find any candidate solutions (to be explained in Sec. 2.4.5).

Fig. 2-11 shows a stylized representation of the set of predicates that our system considers for the outer loop in the running example. The figure shows the potential expressions for the program variable i and $listUsers$. One advantage of using the theory of ordered relations is that invariants can be relatively concise. This has a big impact for synthesis, because the space of expressions grows exponentially with respect to the size of the candidate expressions.

2.4.4 Creating Templates for Postconditions

The mechanism used to generate possible expressions for the result variable is similar to that for invariants, but we have stronger restrictions, since we know the postcondition must be of the form $resultVar = e$ in order to be translatable to SQL, and the form is further restricted by the set of translatable expressions discussed in Sec. 2.3.2.

For the running example, QBS considers the following possible set of postconditions:

$$listUsers = \left\{ \begin{array}{c} users \mid \sigma_{\varphi}(users) \mid \mathrm{top}_{e}(users) \mid \\ \pi_{\ell}(\bowtie_{\varphi} (\mathrm{top}_{e_1}(users), \mathrm{top}_{e_2}(roles))) \mid \\ \pi_{\ell}(\bowtie_{\varphi_3} (\sigma_{\varphi_1}(\mathrm{top}_{e_1}(users), \sigma_{\varphi_2}(\mathrm{top}_{e_2}(roles))))) \mid \\ \text{[other relational expressions that return an ordered list]} \end{array} \right\}$$

2.4.5 Optimizations

The basic algorithm presented above for generating invariant and postcondition templates is sufficient but not efficient for synthesis. In this section I describe two optimizations that improve the synthesis efficiency.

Incremental solving. As an optimization, the generation of templates for invariants and postconditions is done in an iterative manner: QBS initially scans the input code fragment for specific patterns and creates simple templates using the production rules from the predicate grammar, such as considering expressions with only one relational operator, and functions that contains one Boolean clause. If the synthesizer is able to generate a candidate that can be used to prove the validity of the verification conditions, then our job is done. Otherwise, the system repeats the template generation process, but increases the complexity of the template that is generated by considering expressions consisting of more relational operators, and more complicated Boolean functions. Our evaluation using real-world examples shows that most code examples require only a few (< 3) iterations before finding a candidate solution. Additionally, the incremental solving process can be run in parallel.

Breaking symmetries. Symmetries have been shown to be one of sources of inefficiency in constraint solvers [104, 50]. Unfortunately, the template generation algorithm presented above can generate highly symmetrical expressions. For instance, it can generate the following potential candidates for the postcondition:

$$\sigma_{\varphi_2}(\sigma_{\varphi_1}(\mathit{users}))$$
$$\sigma_{\varphi_1}(\sigma_{\varphi_2}(\mathit{users}))$$

Notice that the two expressions are semantically equivalent to the expression $\sigma_{\varphi_1 \wedge \varphi_2}(\mathit{users})$. These are the kind of symmetries that are known to affect solution time dramatically. The template generation algorithm leverages known algebraic relationships between expressions to reduce the search space of possible expressions. For example, our algebraic relationships tell us that it is unnecessary to consider expressions with nested σ like the ones above. Also, when generating templates for postconditions, we only need to consider translatable expressions as defined in Sec. 2.3.2 as potential candidates. Our experiments have shown that applying these symmetric breaking optimizations can reduce the amount of solving time by half.

Even with these optimizations, the spaces of invariants considered are still astronomically large; on the order of 2^{300} possible combinations of invariants and postconditions for some problems. Thanks to these optimizations, however, the spaces can be searched very efficiently by the constraint based synthesis procedure.

2.5 Formal Validation and Source Transformation

After the synthesizer comes up with candidate invariants and postconditions, they need to be validated using a theorem prover, since the synthesizer used in our prototype is only able to perform bounded reasoning as discussed earlier. We have implemented the theory of ordered relations in the Z3 [80] prover for this purpose. Since the theory of lists is not decidable as it uses universal quantifiers, the theory of ordered relations is not decidable as well. However, for practical purposes we have not found that to be limiting in our experiments. In fact, given the appropriate invariants and postconditions, the prover is able to validate them within seconds by making use of the axioms that are provided.

Type	Expression inferred
outer loop invariant	$i \leq \mathsf{size}(users) \ \wedge \ listUsers = \pi_{\ell}(\bowtie_{\varphi}(\mathsf{top}_i(users), roles))$
inner loop invariant	$i < \mathsf{size}(users) \wedge j \leq \mathsf{size}(roles) \wedge$ $listUsers = \mathsf{append}(\pi_{\ell}(\bowtie_{\varphi}(\mathsf{top}_i(users), roles)), \pi_{\ell}(\bowtie_{\varphi}(\mathsf{get}_i(users), \mathsf{top}_j(roles)))$
postcondition	$listUsers = \pi_{\ell}(\bowtie_{\varphi}(users, roles))$

where $\varphi(e_{users}, e_{roles}) \quad := \quad e_{users}.roleId \quad = \quad e_{roles}.roleId$,
ℓ contains all the fields from the $User$ class

Figure 2-13: Inferred expressions for the running example

If the prover can establish the validity of the invariants and postcondition candidates, the postcondition is then converted into SQL according to the rules discussed in Sec. 2.3.2. For instance, for the running example our algorithm found the invariants and postcondition as shown in Fig. 2-13, and the input code is transformed into the results in Fig. 2-3.

If the prover is unable to establish validity of the candidates (detected via a timeout), QBS asks the synthesizer to generate other candidate invariants and postconditions after increasing the space of possible solutions as described in Sec. 2.4.5. One reason that the prover may not be able to establish validity is because the maximum size of the relations set for the synthesizer was not large enough. For instance, if the code returns the first 100 elements from the relation but the synthesizer only considers relations up to size 10, then it will incorrectly generate candidates that claim that the code was performing a full selection of the entire relation. In such cases our algorithm will repeat the synthesis process after increasing the maximum relation size. If the verification is successful, the inferred queries are merged back into the code fragment. Otherwise QBS will invoke the synthesizer to generate another candidate. The process continues until a verified one is found or a time out happens.

```
List objs1 = fetchRecordsFromDB();
List results1 = new ArrayList();

for (Object o : objs1) {
  if (f(o))
    results1.add(o);
}

List results2 = new ArrayList();
for (Object o : objs1) {
  if (g(o))
    results2.add(o);
}
```

Figure 2-14: Code fragment with alias in results

2.5.1 Object Aliases

Implementations of ORM libraries typically create new objects from the records that are fetched, and our current implementation will only transform the input source into SQL if all the objects involved in the code fragment are freshly fetched from the database, as in the running example. In some cases this may not be true, as in the code fragment in Fig. 2-14.

Here, the final contents of `results1` and `results2` can be aliases to those in `objs1`. In that case, rewriting `results1` and `results2` into two SQL queries with freshly created objects will not preserve the alias relationships in the original code. Our current implementation will not transform the code fragment in that case, and we leave sharing record results among multiple queries as future work.

2.6 Preprocessing of Input Programs

In order to handle real-world Java programs, QBS performs a number of initial passes to identify the code fragments to be transformed to kernel language representation before query inference. The code identification process makes use of several standard analysis techniques, and in this section I describe them in detail.

2.6.1 Generating initial code fragments

As discussed in Sec. 2.2, code identification first involves locating application entry point methods and data persistent methods. From each data persistent method, our system currently inlines a neighborhood of 5 callers and callees. QBS only inline callees that are defined in the application, and provide models for native Java API calls. For callers QBS only inline those that can be potentially invoked from an entry point method. The inlined method bodies are passed to the next step of the process. Inlining improves the precision of the points-to information for our analysis. While there are other algorithms that can be used to obtain such information [112, 108], we chose inlining for ease of implementation and is sufficient in processing the code fragments used in the experiments.

2.6.2 Identifying code fragments for query inference

Given a candidate inlined method for query inference, QBS next identifies the code fragment to transform to the kernel language representation. While QBS can simply use the entire body of the inlined method for this purpose, we would like to limit the amount of code to be analyzed, since including code that does not manipulate persistent data will increase the difficulty in synthesizing invariants and postconditions with no actual benefit. QBS accomplishes this goal using a series of analyses. First, QBS runs a flow-sensitive pointer analysis [91] on the body of the inlined method. The results of this analysis is a set of points-to graphs that map each reference variable to one or more abstract memory locations at each program point. Using the points-to information, QBS performs two further analyses on the inlined method.

Location tainting. QBS runs a dataflow analysis that conservatively marks values that are derived from persistent data retrieved via ORM library calls. This analysis is similar to taint analysis [105], and the obtained information allows the system to remove regions of code that do not manipulate persistent data and thus can be ignored for our purpose. For instance, all reference variables and list contents in Fig. 2-1 will be tainted as they are derived from persistent data. The results from this analysis are used to identify the boundaries of the code fragment to be analyzed and converted.

Type analysis. As Java uses dynamic dispatch to resolve targets of method calls, we implemented class analysis to determine potential classes for each object in a given code fragment. If there are multiple implementations of the same method depending on the target's runtime type, then QBS will inline all implementations into the code fragment, with each one guarded by a runtime type lookup. For instance, if object `o` can be of type `Bar` or a subtype `Baz`, and that method `foo` is implemented by both classes, then inlining `o.foo()` will result in:

```
if (o instanceof Baz) {
  // implementation of Baz.foo()
} else if (o instanceof Bar) {
  // implementation of Bar.foo()
} else throw new RuntimeException(); // should not reach here
```

This process is applied recursively to all inlined methods.

Def-use analysis. For each identified code fragment, QBS runs a definition-use analysis to determine the relationship among program variables. The results are used after ensure that none of the variables that are defined in the code fragment to be replaced is used in the rest of the inlined method after replacement. And the checking is done before replacing the code fragment with a verified SQL query.

Value escapement. After that, QBS performs another dataflow analysis to check if any abstract memory locations are reachable from references that are outside of the inlined method body. This analysis is needed because if an abstract memory location `m` is accessible from the external environment (e.g., via a global variable) after program point `p`, then converting `m` might break the semantics of the original code, as there can be external references to `m` that rely on the contents of `m` before the conversion. This analysis is similar to classical escape analysis [108]. Specifically, we define an abstract memory location `m` as having escaped at program point `p` if any of the following is true:

- It is returned from the entry point method.
- It is assigned to a global variable that persists after the entry point method returns (in the web application context, these can be variables that maintain session state, for instance).
- It is assigned to a `Runnable` object, meaning that it can be accessed by other threads.
- It is passed in as a parameter into the entry point method.
- It can be transitively reached from an escaped location.

With that in mind, we define the beginning of the code fragment to pass to the QBS algorithm as the program point `p` in the inlined method where tainted data is first retrieved from the database, and the end as the program point `p'` where tainted data first escapes, where `p'` appears after `p` in terms of control flow. For instance, in Fig. 2-1 the `return` statement marks the end of the code fragment, with the result variable being the value returned.

2.6.3 Compilation to kernel language

Each code fragment that is identified by the previous analysis is compiled to our kernel language. Since the kernel language is based on value semantics and does not model heap updates for lists, during the compilation process QBS translates list references to the abstract memory locations that they point to, using the results from earlier analysis. In general, there are cases where the preprocessing step fails to identify a code fragment from an inlined method (e.g., persistent data values escape to multiple result variables under different branches, code involves operations not supported by the kernel language, etc.), and QBS will simply skip such cases. However, the number of such cases is relatively small as our experiments show.

2.7 Experiments

In this section I report our experimental results. The goal of the experiments is twofold: first, to quantify the ability of our algorithm to convert Java code into real-world applications and measure the performance of the converted code fragments, and second to explore the limitations of the current implementation.

We have implemented a prototype of QBS. The source code analysis and computation of verification conditions are implemented using the Polyglot compiler framework [84]. We use Sketch as the synthesizer for invariants and postconditions, and Z3 for validating the invariants and postconditions.

2.7.1 Real-World Evaluation

In the first set of experiments, we evaluated QBS using real-world examples from two large-scale open-source applications, Wilos and itracker, written in Java. Wilos (rev. 1196) [24] is a project management application with 62k LOC, and itracker (ver. 3.0.1) [5] is a software issue management system with 61k LOC. Both applications have multiple contributors with different coding styles, and use the Hibernate ORM library for data persistence operations. We passed in the entire source code of these applications to QBS to identify code fragments. The preprocessor initially found 120 unique code fragments that invoke ORM operations. Of those, it failed to convert 21 of them into the kernel language representation, as they use data structures that are not supported by our prototype (such as Java arrays), or access persistent objects that can escape from multiple control flow points and hence cannot be converted.

Meanwhile, upon manual inspection, we found that those 120 code fragments correspond to 49 distinct code fragments inlined in different contexts. For instance, if A and C both call method B, our system automatically inlines B into the bodies of A and C, and those become two different code fragments. But if all persistent data manipulation happens in B, then we only count one of the two as part of the 49 distinct code fragments. QBS successfully translated 33 out of the 49 distinct code fragments (and those 33 distinct code fragments correspond to 75 original code fragments). The results are summarized in Fig. 2-15, and the details can be found in Fig. 2-16.

This experiment shows that QBS can infer relational specifications from a large fraction of candidate fragments and convert them into SQL equivalents. For the candidate fragments that are reported as translatable by QBS, our prototype was able to synthesize postconditions and invariants,

Application	# persistent data code fragments	translated	failed
Wilos	33	29	4
itracker	16	12	4
Total	**49**	**41**	**7**

Figure 2-15: Real-world code fragments experiment

wilos code fragments

#	Java Class Name	Line	Oper.	Status	Time (s)
17	ActivityService	401	A	†	–
18	ActivityService	328	A	†	–
19	AffectedtoDao	13	B	✓	72
20	ConcreteActivityDao	139	C	*	–
21	ConcreteActivityService	133	D	†	–
22	ConcreteRoleAffectationService	55	E	✓	310
23	ConcreteRoleDescriptorService	181	F	✓	290
24	ConcreteWorkBreakdownElementService	55	G	†	–
25	ConcreteWorkProductDescriptorService	236	F	✓	284
26	GuidanceService	140	A	†	–
27	GuidanceService	154	A	†	–
28	IterationService	103	A	†	–
29	LoginService	103	H	✓	125
30	LoginService	83	H	✓	164
31	ParticipantBean	1079	B	✓	31
32	ParticipantBean	681	H	✓	121
33	ParticipantService	146	E	✓	281
34	ParticipantService	119	E	✓	301
35	ParticipantService	266	F	✓	260
36	PhaseService	98	A	†	–
37	ProcessBean	248	H	✓	82
38	ProcessManagerBean	243	B	✓	50
39	ProjectService	266	K	*	–
40	ProjectService	297	A	✓	19
41	ProjectService	338	G	†	–
42	ProjectService	394	A	✓	21
43	ProjectService	410	A	✓	39
44	ProjectService	248	H	✓	150
45	RoleDao	15	I	*	–
46	RoleService	15	E	✓	150
47	WilosUserBean	717	B	✓	23
48	WorkProductsExpTableBean	990	B	✓	52
49	WorkProductsExpTableBean	974	J	✓	50

itracker code fragments

#	Java Class Name	Line	Operation	Status	Time (s)
1	EditProjectFormActionUtil	219	F	✓	289
2	IssueServiceImpl	1437	D	✓	30
3	IssueServiceImpl	1456	L	*	–
4	IssueServiceImpl	1567	C	*	–
5	IssueServiceImpl	1583	M	✓	130
6	IssueServiceImpl	1592	M	✓	133
7	IssueServiceImpl	1601	M	✓	128
8	IssueServiceImpl	1422	D	✓	34
9	ListProjectsAction	77	N	*	–
10	MoveIssueFormAction	144	K	*	–
11	NotificationServiceImpl	568	O	✓	57
12	NotificationServiceImpl	848	A	✓	132
13	NotificationServiceImpl	941	H	✓	160
14	NotificationServiceImpl	244	O	✓	72
15	UserServiceImpl	155	M	✓	146
16	UserServiceImpl	412	A	✓	142

where:

A:	selection of records
B:	return literal based on result size
C:	retrieve the max / min record by first sorting and then returning the last element
D:	projection / selection of records and return results as a set
E:	nested-loop join followed by projection
F:	join using contains
G:	type-based record selection
H:	check for record existence in list
I:	record selection and only return the one of the records if multiple ones fulfill the selection criteria
J:	record selection followed by count
K:	sort records using a custom comparator
L:	projection of records and return results as an array
M:	return result set size
N:	record selection and in-place removal of records
O:	retrieve the max / min record

✓ indicates those that are translated by. QBS
* indicates those that QBS failed to find invariants for.
† indicates those that are rejected by QBS due to TOR / pre-processing limitations.

Figure 2-16: Details of the 49 distinct code fragments. The times reported correspond to the time required to synthesize the invariants and postconditions. The time taken for the other initial analysis and SQL translation steps are negligible.

and also validate them using the prover. Furthermore, the maximum time that QBS takes to process any one code fragment is under 5 minutes (with an average of 2.1 minutes). In the following, I broadly describe the common types of relational operations that our QBS prototype inferred from the fragments, along with some limitations of the current implementation.

Projections and Selections. A number of identified fragments perform relational projections and selections in imperative code. Typical projections include selecting specific fields from the list of records that are fetched from the database, and selections include filtering a subset of objects using field values from each object (e.g., user ID equals to some numerical constant), and a few use criteria that involve program variables that are passed into the method.

One special case is worth mentioning. In some cases only a single field is projected out and loaded into a set data structure, such as a set of integer values. One way to translate such cases is to generate SQL that fetches the field from all the records (including duplicates) into a list, and eliminate the duplicates and return the set to the user code. Our prototype, however, improves upon that scheme by detecting the type of the result variable and inferring a postcondition involving the unique operator, which is then translated to a SELECT DISTINCT query that avoids fetching duplicate records from the database.

Joins. Another set of code fragments involve join operations. I summarize the join operations in the application code into two categories. The first involves obtaining two lists of objects from two base queries and looping through each pair of objects in a nested for or while loop. The pairs are filtered and (typically) one of the objects from each pair is retained. The running example in Fig. 2-1 represents such a case. For these cases, QBS translates the code fragment into a relational join of the two base queries with the appropriate join predicate, projection list, and sort operations that preserve the ordering of records in the results.

Another type of join also involves obtaining two lists of objects from two base queries. Instead of a nested loop join, however, the code iterates through each object e from the first list, and searches if e (or one of e's fields) is contained in the second. If true, then e (or some of its fields) is appended to the resulting list. For these cases QBS converts the search operation into a contains expression in the predicate language, after which the expression is translated into a correlated subquery in the form of SELECT * FROM r1, r2 WHERE r1 IN r2, with r1 and r2 being the base queries.

QBS handles both join idioms mentioned above. However, the loop invariants and postconditions involved in such cases tend to be more complex as compared to selections and projections, as illustrated by the running example in Fig. 2-13. Thus, they require more iterations of synthesis and formal validation before finding a valid solution, with up to 5 minutes in the longest case. Here, the majority of the time is spent in synthesis and bounded verification. We are not aware of any prior techniques that can be used to infer join queries from imperative code, and we believe that more optimizations can be devised to speed up the synthesis process for such cases.

Aggregations. Aggregations are used in fragments in a number of ways. The most straightforward ones are those that return the length of the list that is returned from an ORM query, which are translated into COUNT queries. More sophisticated uses of aggregates include iterating through all records in a list to find the max or min values, or searching if a record exists in a list. Aggregates such as maximum and minimum are interesting as they introduce loop-carried dependencies [28], where the running value of the aggregate is updated conditionally based on the value of the current record as compared to previous ones. By using the top operator from the theory of ordered relations, QBS is able to generate a loop invariant of the form $\mathsf{v} = \mathsf{agg}(\mathsf{top}_{\mathsf{i}}(\mathsf{r}))$, where agg represents an aggregate operation, and then translate the postcondition into the appropriate SQL query.

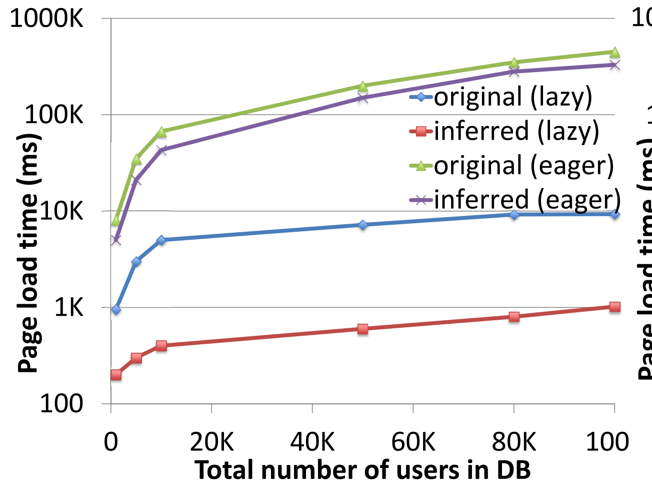
As a special case, a number of fragments check for the existence of a particular record in a relation by iterating over all records and setting a result Boolean variable to be true if it exists. In such cases, the generated invariants are similar to other aggregate invariants, and our prototype translates such code fragments into SELECT COUNT(*) > 0 FROM ... WHERE e, where e is the expression to check for existence in the relation. We rely on the database query optimizer to further rewrite this query into the more efficient form using EXISTS.

Limitations. We have verified that in all cases where the generated template is expressive enough for the invariants and postconditions, our prototype does indeed find the solution within a preset timeout of 10 minutes. However, there are a few examples from the two applications where our prototype either rejects the input code fragment or fails to find an equivalent SQL expression from the kernel language representation. Fragments are rejected because they involve relational update operations that are not handled by TOR. Another set of fragments include advanced use of types, such as storing polymorphic records in the database, and performing different operations based on the type of records retrieved. Incorporating type information in the theory of ordered relations is an interesting area for future work. There are also a few that QBS fails to translate into SQL, even though we believe that there is an equivalent SQL query without updates. For instance, some fragments involve sorting the input list by `Collections.sort`, followed by retrieving the last record from the sorted list, which is equivalent to `max` or `min` depending on the sort order. Including extra axioms in the theory would allow us to reason about such cases.

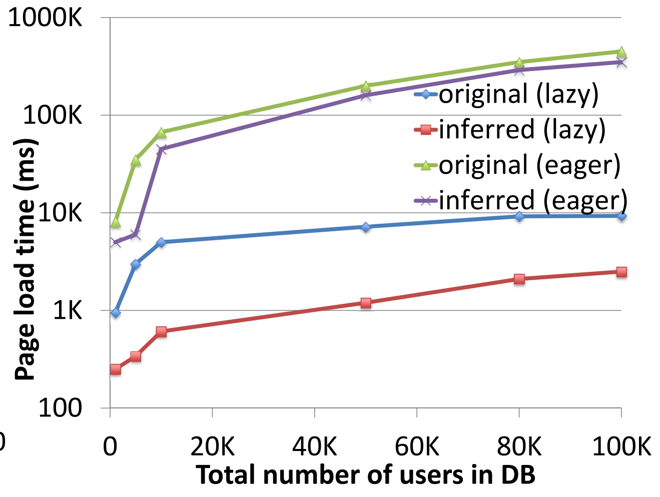
2.7.2 Performance Comparisons

Next, we quantify the amount of performance improvement as a result of query inference. To do so, we took a few representative code fragments for selection, joins, and aggregation, and populated databases with different number of persistent objects. We then compared the performance between the original code and our transformed versions of the code with queries inferred by QBS. Since Hibernate can either retrieve all nested references from the database (eager) when an object is fetched, or only retrieve the top level references (lazy), we measured the execution times for both modes (the original application is configured to use the lazy retrieval mode). The results shown in Fig. 2-17 compare the time taken to completely load the webpages containing the queries between the original and the QBS inferred versions of the code.

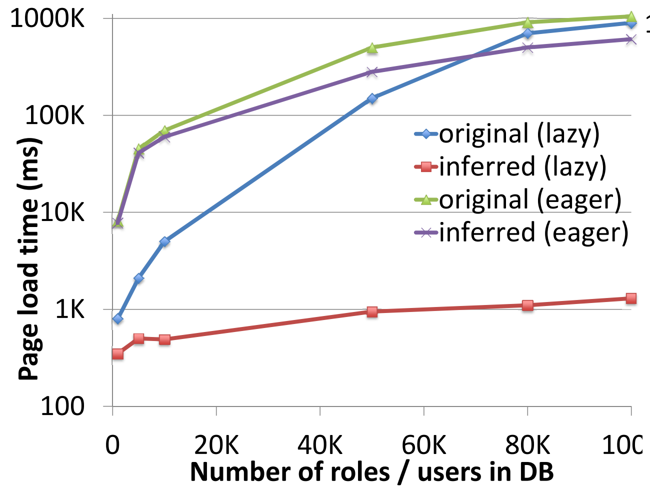
Selection Code Fragment. Fig. 2-17a and Fig. 2-17b show the results from running a code fragment that includes persistent data manipulations from fragment #40 in Fig. 2-16. The fragment returns the list of unfinished projects. Fig. 2-17a shows the results where 10% of the projects stored are unfinished, and Fig. 2-17b shows the results with 50% unfinished projects. While the original version performs the selection in Java by first retrieving all projects from the database, QBS inferred a selection query in this case. As expected, the query inferred by QBS outperforms the original fragments in all cases as it only needs to retrieve a portion (specifically 10 and 50%)



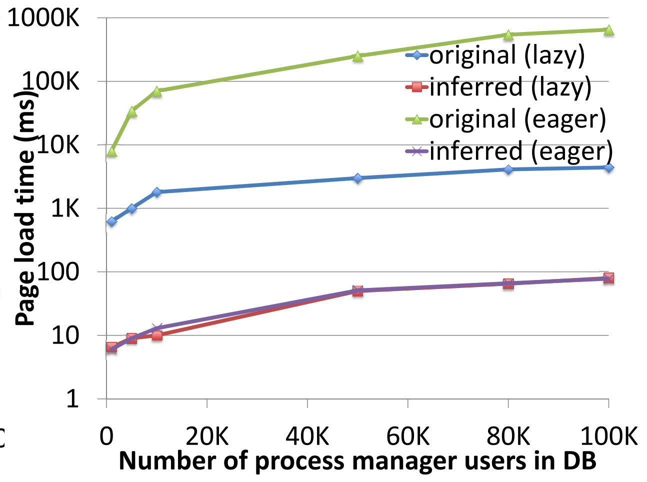
(a) Selection with 10% selectivity



(b) Selection with 50% selectivity



(c) Join code fragment



(d) Aggregation code fragment

Figure 2-17: Webpage load times comparison of representative code fragments

of all persistent objects from the database.

Join Code Fragment. Fig. 2-17c shows the results from a code fragment with contents from fragment #46 in Fig. 2-16 (which is the same as the example from Fig. 2-1). The fragment returns the projection of `User` objects after a join of `Roles` and `Users` in the database on the `roleId` field. The original version performs the join by retrieving all `User` and `Role` objects and joining them in a nested loop fashion as discussed in Sec. 2.2. The query inferred by QBS, however, pushes the join and projection into the database. To isolate the effect of performance improvement due to query selectivity (as in the selection code fragment), we purposefully constructed the dataset so that the query returns all `User` objects in the database in all cases, and the results show that the query inferred by QBS still has much better performance than the original query. This is due to two reasons. First, even though the number of `User` objects returned in both versions are the same, the QBS version does not need to retrieve any `Role` objects since the projection is pushed into the database, unlike the the original version. Secondly, thanks to the automatically created indices on the `Role` and `User` tables by Hibernate, the QBS version essentially transforms the join implementation from a nested loop join into a hash join, i.e., from an $O(n^2)$ to an $O(n)$ implementation, thus improving performance asymptotically.

Aggregation Code Fragment. Finally, Fig. 2-17d shows the results from running code with contents from fragment #38, which returns the number of users who are process managers. In this case, the original version performs the counting by bringing in all users who are process managers from the database, and then returning the size of the resulting list. QBS, however, inferred a COUNT query on the selection results. This results in multiple orders of magnitude performance improvement, since the QBS version does not need to retrieve any objects from the database beyond the resulting count.

2.7.3 Advanced Idioms

In the final set of experiments, we used synthetic code fragments to demonstrate the ability of our prototype to translate more complex expressions into SQL. Although we did not find such examples in either of our two real-world applications, we believe that these can occur in real applications.

Hash Joins. Beyond the join operations that we encountered in the applications, we wrote two synthetic test cases for joins that join relations `r` and `s` using the predicate `r.a = s.b`, where `a` and `b` are integer fields. In the first case, the join is done via hashing, where we first iterate through records in `r` and build a hashtable, whose keys are the values of the `a` field, and where each key maps to a list of records from `r` that has that corresponding value of `a`. We then loop through each record in `s` to find the relevant records from `r` to join with, using the `b` field as the look up key. QBS currently models hashtables using lists, and with that our prototype is able recognize this process as a join operation and convert the fragment accordingly, similar to the join code fragments mentioned above.

Sort-Merge Joins. Our second synthetic test case joins two lists by first sorting `r` and `s` on fields `a` and `b` respectively, and then iterating through both lists simultaneously. We advance the scan of `r` as long as the current record from `r` is less than (in terms of fields `a` and `b`) the current record from `s`, and similarly advance the scan of `s` as long as the current `s` record is less than the current `r` record. Records that represent the join results are created when the current record from `r` equals to that from `s` on the respective fields. Unfortunately, our current prototype fails to translate the code fragment into SQL, as the invariants for the loop cannot be expressed using the current the predicate language, since that involves expressing the relationship between the current record from `r` and `s` with all the records that have been previously processed.

Iterating over Sorted Relations. We next tested our prototype with two usages of sorted lists. We created a relation with one unsigned integer field `id` as primary key, and sorted the list using the `sort` method from Java. We subsequently scanned through the sorted list as follows:

```
List records = Query("SELECT id FROM t");
List results = new ArrayList();
Collections.sort(records); // sort by id
for (int i = 0; i < 10; ++i) {
  results.add(records.get(i));
}
```

Our prototype correctly processes this code fragment by translating it into `SELECT id FROM t ORDER BY id LIMIT 10`. However, if the loop is instead written as follows:

```
List records = Query("SELECT id FROM t");
List results = new ArrayList();
Collections.sort(records); // sort by id
int i = 0;
while (records.get(i).id < 10) {
  results.add(records.get(i));
  ++i;
}
```

The two loops are equivalent since the `id` field is a primary key of the relation, and thus there can at most be 10 records retrieved. However, our prototype is not able to reason about the second code fragment, as that requires an understanding of the schema of the relation, and that iterating over `id` in this case is equivalent to iterating over `i` in the first code fragment. Both of which require additional axioms to be added to the theory before such cases can be converted.

2.8 Summary

In this chapter, I discussed QBS, a system for inferring relational specifications from imperative code that retrieves data using ORM libraries. Our system automatically infers loop invariants and postconditions associated with the source program, and converts the validated postcondition into SQL queries. Our approach is both sound and precise in preserving the ordering of records. We developed a theory of ordered relations that allows efficient encoding of relational operations into a predicate language, and we demonstrated the applicability using a set of real-world code examples. The techniques developed in this system are applicable to the general problem of inferring high-level structure from low-level code representations, a problem I discuss that further in Ch. 6.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 3

PYXIS: Deciding Where to Execute

In addition to determining how to a piece of computation should be represented, placing computation on the appropriate server is another factor that affects database application performance. While the number of ways computation can be placed is limited in previous-generation database-backed application architectures as certain software components are installed on dedicated servers (e.g., the database engine is available only on servers with large amount of disk storage and physical memory), this is no longer true as most software components can now be executed on commodity hardware. Thus, there are many more ways in which computation can be placed on different servers.

While QBS attempts to convert imperative code blocks into SQL queries, there are certain cases where QBS fails to convert even though a valid rewrite exists, as described in Sec. 2.7.1. In such cases, it might still be beneficial to move computation from the application server to the database server to reduce the amount of data to be transferred from the database back to the application server. Doing so requires a system that can automatically partition a database application into server and query components. In this chapter, I describe a new system called PYXIS that addresses this code placement problem by making use of profile-driven code partitioning during compilation.

The goal of PYXIS is to automatically partition database applications into the server and query components to be executed on the application and database servers, respectively. To do so, PYXIS first profiles the application and server loads, statically analyzes the code's dependencies, and produces a partitioning that minimizes the number of control transfers as well as the amount of data sent during each transfer. Our experiments using TPC-C and TPC-W show that PYXIS is able to generate partitions with up to $3\times$ reduction in latency and $1.7\times$ improvement in throughput when compared to a traditional non-partitioned implementation and has comparable performance to that of a custom stored procedure implementation.[1]

3.1 Code Partitioning Across Servers

Many database applications are extremely latency sensitive for two reasons. First, in many transactional applications (e.g., database-backed websites), there is typically a hard response time limit of a few hundred milliseconds, including the time to execute application logic, retrieve query results, and generate HTML. Saving even a few tens of milliseconds of latency per transaction can be important in meeting these latency bounds. Second, longer-latency transactions hold locks longer, which can severely limit maximum system throughput in highly concurrent systems.

Stored procedures are a widely used technique for improving the latency of database applications. The idea behind stored procedures is to rewrite sequences of application logic that are interleaved with database commands (e.g., SQL queries) into parameterized blocks of code that are stored on the database server. The application then sends commands to the database server, typically on a separate physical machine, to invoke these blocks of code.

Stored procedures can significantly reduce transaction latency by avoiding round trips between the application and database servers. These round trips would otherwise be necessary in order to execute the application logic found between successive database commands. The resulting speedup can be substantial. For example, in a Java implementation of a database application similar to TPC-C [22] (a common benchmark suite for transactional database applications)—which has relatively little application logic—running each TPC-C transaction as a stored procedure can offer up to a $3\times$ reduction in latency versus running each SQL command via a separate JDBC call. This reduction results in a $1.7\times$ increase in overall transaction throughput on this benchmark.

[1]Materials in this chapter are based on work published as Cheung, Arden, Madden, and Myers, “Automatic Partitioning of Database Applications,” in PVLDB 5:11 [36].

However, stored procedures have several disadvantages:

- **Portability and maintainability**: Stored procedures break a straight-line application into two distinct and logically disjoint code bases. These code bases are usually written in different languages and must be maintained separately. Stored-procedure languages are often database-vendor specific, making applications that use them less portable between databases. Programmers are less likely to be familiar with or comfortable in low-level—even arcane—stored procedure languages like PL/SQL or TransactSQL, and tools for debugging and testing stored procedures are less advanced than those for more widely used languages.

- **Conversion effort:** Identifying sections of application logic that are good candidates for conversion into stored procedures is tricky. In order to design effective stored procedures, programmers must identify sections of code that make multiple (or large) database accesses and can be parameterized by relatively small amounts of input. Weighing the relative merits of different designs requires programmers to model or measure how often a stored procedure is invoked and how much parameter data need to be transferred, both of which are nontrivial tasks.

- **Dynamic server load:** Running parts of the application as stored procedures is not always a good idea. If the database server is heavily loaded, pushing more computation into it by calling stored procedures will hurt rather than help performance. Besides, a database server's load tends to change over time, depending on the workload and the utilization of the applications it is supporting, so it is difficult for developers to predict the resources available on the servers hosting their applications. Even with accurate predictions they have no easy way to adapt their programs' use of stored procedures to a dynamically changing server load.

We propose that these disadvantages of manually generated stored procedures can be avoided by automatically identifying and extracting application code to be shipped to the database server. We implemented this new approach in PYXIS, a system that automatically partitions a database application into two pieces, one deployed on the application server as the server component, and the other in the database server as the query component implemented using stored procedures.

The two programs communicate with each other via remote procedure calls (RPCs) to implement the semantics of the original application. In order to generate a partition, PYXIS first analyzes application source code using static analysis and then collects dynamic information such as runtime profile and machine loads. The collected profile data and results from the analysis are then used to formulate a linear program whose objective is to minimize, subject to a maximum CPU load, the overall latency due to network round trips between the application and database servers as well as the amount of data sent during each round trip. The solved linear program then yields a fine-grained, statement-level partitioning of the application's source code. The partitioned code is split into two components and executed on the application and database servers using the PYXIS runtime.

The main benefit of our approach is that the developer does not need to manually decide which part of her program should be executed where. PYXIS identifies good candidate code blocks for conversion to stored procedures and automatically produces the two distinct pieces of code from the single application codebase. When the application is modified, PYXIS can automatically regenerate and redeploy this code. By periodically re-profiling their application, developers can generate new partitions as load on the server or application code changes. Furthermore, the system can switch between partitions as necessary by monitoring the current server load.

PYXIS makes several contributions:

1. We present a formulation for automatically partitioning programs into stored procedures that minimize overall latency subject to CPU resource constraints. Our formulation leverages a combination of static and dynamic program analysis to construct a linear optimization problem whose solution is our desired partitioning.

2. We develop an execution model for partitioned applications where consistency of the distributed heap is maintained by automatically generating custom synchronization operations.

3. We implement a method for adapting to changes in real-time server load by dynamically switching between pre-generated partitions created using different resource constraints.

4. We evaluate our PYXIS implementation on two popular transaction processing benchmarks,

TPC-C and TPC-W, and compare the performance of our partitions to the original program and versions using manually created stored procedures. Our results show PYXIS can automatically partition database programs to get the best of both worlds: when CPU resources are plentiful, PYXIS produces a partition with comparable performance to that of hand-coded stored procedures; when resources are limited, it produces a partition comparable to simple client-side queries.

The rest of the chapter is organized as follows. I start with an architectural overview of PYXIS in Sec. 3.2. I describe how PYXIS programs execute and synchronize data in Section Sec. 3.3. I present the optimization problem and describe how solutions are obtained in Sec. 3.4. Sec. 3.5 explains the generation of partitioned programs, and Sec. 3.6 describes the PYXIS runtime system. Sec. 3.7 shows our experimental results, followed by a summary of PYXIS in Sec. 3.8.

3.2 Overview

Figure 3-1 shows the architecture of PYXIS. PYXIS starts with an application written in Java that uses JDBC to connect to the database and performs several analyses and transformations.[2] The analysis used in PYXIS is general and does not impose any restrictions on the programming style of the application. The final result is two separate programs, one that runs on the application server and one that runs on the database server. These two programs communicate with each other as necessary to implement the original application's semantics. Execution starts at the partitioned program on the application server but periodically switches to its counterpart on the database server, and vice versa. We refer to these switches as *control transfers*. Each statement in the original program is assigned a *placement* in the partitioned program on either the application server or the database server. Control transfers occur when a statement with one placement is followed by a statement with a different placement. Following a control transfer, the calling program blocks until the callee returns control. Hence, a single thread of control is maintained across the two servers.

[2]We chose Java due to its popularity in writing database applications. Our techniques can be applied to other languages as well.

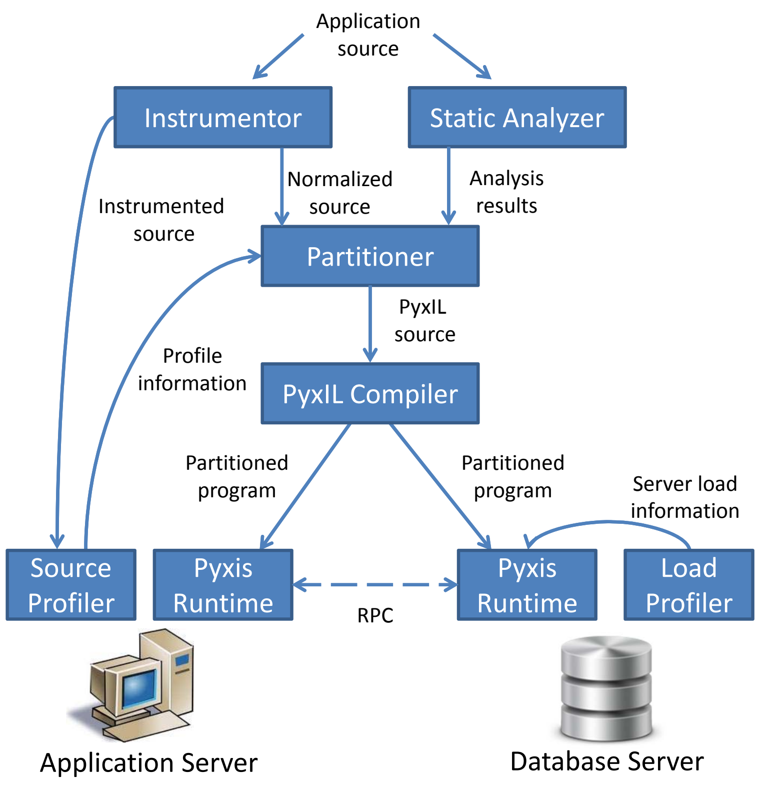


Figure 3-1: PYXIS architecture

Although the two partitioned programs execute in different address spaces, they share the same logical heap and execution stack. This program state is kept in sync by transferring heap and stack updates during each control transfer or by fetching updates on demand. The execution stack is maintained by the PYXIS runtime, but the program heap is kept in sync by explicit heap synchronization operations, generated using a conservative static program analysis.

Static dependency analysis. The goal of partitioning is to preserve the original program semantics while achieving good performance. This is done by reducing the number of control transfers and amount of data sent during transfers as much as possible. The first step is to perform an interprocedural static dependency analysis that determines the data and control dependencies between program statements. The data dependencies conservatively capture all data that may be necessary to send if a dependent statement is assigned to a different partition. The control dependencies capture the necessary sequencing of program statements, allowing the PYXIS code generator to find the best program points for control transfers.

The results of the dependency analysis are encoded in a graph form that we call a *partition graph*. It is a program dependence graph (PDG)[3] augmented with extra edges representing additional information. A PDG-like representation is appealing because it combines both data and control dependencies into a single representation. Unlike a PDG, a partition graph has a weight that models the cost of satisfying the edge's dependencies if the edge is partitioned so that its source and destination lie on different machines. The partition graph is novel; previous automatic partitioning approaches have partitioned control-flow graphs [114, 42] or dataflow graphs [82, 113]. Prior work in automatic parallelization [92] has also recognized the advantages of PDGs as a basis for program representation.

Profile data collection. Although the static dependency analysis defines the structure of dependencies in the program, the system needs to know how frequently each program statement is executed in order to determine the optimal partition; placing a "hot" code fragment on the database server may increase server load beyond its capacity. In order to get a more accurate picture of the runtime behavior of the program, statements in the partition graph are weighted by an estimated execution count. Additionally, each edge is weighted by an estimated latency cost that represents the communication overhead for data or control transfers. Both weights are captured by dynamic profiling of the application.

For applications that exhibit different operating modes, such as the browsing versus shopping mix in TPC-W, each mode could be profiled separately to generate partitions suitable for it. The PYXIS runtime includes a mechanism to dynamically switch between different partitionings based on current CPU load.

Optimization as integer programming. Using the results from the static analysis and dynamic profiling, the PYXIS partitioner formulates the placement of each program statement and each data field in the original program as an integer linear programming problem. These placements then drive the transformation of the input source into the intermediate language *PyxIL* (for PYXis Intermediate Language). The PyxIL program is very similar to the input Java program except that each statement is annotated with its placement, `:APP:` or `:DB:`, denoting the server and query

[3]Since it is interprocedural, it actually is closer to a *system dependence graph* [53, 64] than a PDG.

components of the database application, to be executed respectively on the application or database server. Thus, PyxIL compactly represents a distributed program in a single unified representation. PyxIL code also includes explicit heap synchronization operations, which are needed to ensure the consistency of the distributed heap.

In general, the partitioner generates several different partitionings of the program using multiple *server instruction budgets* that specify upper limits on how much computation may be executed at the database server. Generating multiple partitions with different resource constraints enables automatic adaptation to different levels of server load.

Compilation from PyxIL to Java. For each partitioning, the PyxIL compiler translates the PyxIL source code into two Java programs, one for each runtime. These programs are compiled using the standard Java compiler and linked with the PYXIS runtime. The database partition program is run in an unmodified Java Virtual Machine (JVM) colocated with the database server, and the application partition is similarly run on the application server. While not exactly the same as running traditional stored procedures inside a DBMS, our approach is similar to other implementations of stored procedures that provide a foreign language interface such as PL/Java [18] and execute stored procedures in a JVM external to the DBMS. We find that running the program outside the DBMS does not significantly hurt performance as long as it is colocated. With more engineering effort, the database partition program could run in the same process as the DBMS.

Executing partitioned programs. The runtimes for the application server and database server communicate over TCP sockets using a custom remote procedure call mechanism. The RPC interface includes operations for control transfer and state synchronization. The runtime also periodically measures the current CPU load on the database server to support dynamic, adaptive switching among different partitionings of the program. The runtime is described in more detail in Sec. 3.6.

3.3 Running PyxIL Programs

Figure 3-2 shows a running example used to explain PYXIS throughout this chapter. It is meant to resemble a fragment of the new-order transaction in TPC-C, modified to exhibit relevant features of

```
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 3-2: Running example

PYXIS. The transaction retrieves the order that a customer has placed, computes its total cost, and deducts the total cost from the customer's account. It begins with a call to `placeOrder` on behalf of a given customer `cid` at a given discount `dct`. Then `computeTotalCost` extracts the costs of the items in the order using `getCosts`, and iterates through each of the costs to compute a total and record the discounted cost. Finally, control returns to `placeOrder`, which updates the customer's account. The two operations `insertNewLineItem` and `updateAccount` update the database's contents while `getCosts` retrieves data from the database. If there are N items in the order, the example code incurs N round trips to the database from the `insertNewLineItem` calls, and two more from `getCosts` and `updateAccount`.

There are multiple ways to partition the fields and statements of this program. An obvious partitioning is to assign all fields and statements to the application server. This would produce the same number of remote interactions as in the standard JDBC-based implementation. At the other extreme, a partitioning might place all statements on the database server, in effect creating a stored procedure for the entire `placeOrder` method. As in a traditional stored procedure, each time `placeOrder` is called the values `cid` and `dct` must be serialized and sent to the remote runtime. Other partitionings are possible. Placing only the loop in `computeTotalCost` on the database would save N round trips if no additional communication were necessary to satisfy data dependencies. In general, assigning more code to the database server can reduce latency, but it also can put additional load on the database. PYXIS aims to choose partitionings that achieve the smallest latency possible using the current available resources on the server.

3.3.1 A PyxIL Partitioning

Figure 3-3 shows PyxIL code for one possible partitioning of our running example. PyxIL code makes explicit the placement of code and data, as well as the synchronization of updates to a distributed heap, but keeps the details of control and data transfers abstract. Field declarations and statements in PyxIL are annotated with a placement label (`:APP:` or `:DB:`). The placement of a statement indicates where the instruction is executed. For field declarations, the placement indicates where the authoritative value of the field resides. However, a copy of a field's value may be found on the remote server. The synchronization protocols using a conservative program analysis ensure that this copy is up to date if it might be used before the next control transfer. Thus, each object apparent at the source level is represented by two objects, one at each server. We refer to these as the APP and DB parts of the object.

In the example code, the field `id` is assigned to the application server, indicated by the `:APP:` placement label, but field `totalCost` is assigned to the database, indicated by `:DB:`. The array allocated on line 21 is placed on the application server. All statements are placed on the application server except for the `for` loop in `computeTotalCost`. When control flows between two statements with different placements, a *control transfer* occurs. For example, on line 24 in Fig. 3-3, execution is suspended at the application server and resumes at line 26 on the database server.

Arrays are handled differently from objects. The placement of an array is defined by its allocation site: that is, the placement of the statement allocating the array. This approach means the contents of an array may be assigned to either partition, but the elements are not split between them. Additionally, since the stack is shared by both partitions, method parameters and local variable declarations do not have placement labels in PyxIL.

```
class Order {
  :APP: int id;
  :APP: double[] realCosts;
  :DB: double totalCost;

  Order(int id) {
    :APP: this.id = id;
    :APP: sendAPP(this);
  }

  void placeOrder(int cid, double dct) {
    :APP: totalCost = 0;
    :APP: sendDB(this);
    :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);
    :APP: sendNative(realCosts,costs);
    :APP: i = 0;

    for (:DB: itemCost : costs) {
      double realCost;
      :DB: realCost = itemCost * dct;
      :DB: totalCost += realCost;
      :DB: sendDB(this);
      :DB: realCosts[i++] = realCost;
      :DB: sendNative(realCosts);
      :DB: insertNewLineItem(id, realCost);
    }
  }
}
```

Figure 3-3: A PyxIL version of the Order class

3.3.2 State Synchronization

Although all data in PyxIL has an assigned placement, remote data may be transferred to or updated by any host. Each host maintains a local heap for fields and arrays placed at that host as well as a *remote cache* for remote data. When a field or array is accessed, the current value in the local heap or remote cache is used. Hosts synchronize their heaps using *eager batched updates*; modifications are aggregated and sent on each control transfer so that accesses made by the remote host are up to date. The host's local heap is kept up to date whenever that host is executing code. When a host executes statements that modify remotely partitioned data in its cache, those updates must be transferred on the next control transfer. For the local heap, however, updates only need to be sent to the remote host before they are accessed. If static analysis determines that no such access occurs, then no update message is required.

In some scenarios, eagerly sending updates may be suboptimal. If the amount of latency incurred by transferring unused updates exceeds the cost of an extra round trip communication, it may be better to request the data *lazily* as needed. In PYXIS, we only generate PyxIL programs that send updates eagerly, but investigating hybrid update strategies is an interesting future direction.

PyxIL programs maintain the above heap invariants using explicit synchronization operations. Recall that classes are partitioned into two partial classes, one for `APP` and one for `DB`. The `sendAPP` operation sends the `APP` part of its argument to the remote host. In Fig. 3-3, line 8 sends the contents of `id` while line 30 sends `totalCost` and the array reference `realCosts`. Since arrays are placed dynamically based on their allocation site, a reference to an array may alias both locally and remotely partitioned arrays. On line 23, the contents of the array `realCosts` allocated at line 21 are sent to the database server using the `sendNative` operation.[4] The `sendNative` operation is also used to transfer unpartitioned native Java objects that are serializable. Like arrays, native Java objects are assigned locations based on their allocation site.

Send operations are batched together and executed at the next control transfer. Even though the `sendDB` operation on line 30 and the `sendNative` operation on line 32 are inside a `for` loop, the

[4]For presentation purposes, the contents of `costs` is also sent here. This operation would typically occur in the body of `getCosts()`.

updates will only be sent when control is transferred back to the application server.

3.4 Partitioning PYXIS code

PYXIS finds a partitioning for a user program by generating partitions with respect to a representative workload that a user wishes to optimize. The program is profiled using this workload, generating inputs that partitioning is based upon.

3.4.1 Profiling

PYXIS profiles the application in order to be able to estimate the size of data transfers and the number of control transfers for any particular partitioning. To this end, statements are instrumented to collect the number of times they are executed, and assignment expressions are instrumented to measure the average size of the assigned objects. The application is then executed for a period of time to collect data. Data collected from profiling is used to set weights in the partition graph.

This profile need not perfectly characterize the future interactions between the database and the application, but a grossly inaccurate profile could lead to suboptimal performance. For example, with inaccurate profiler data, PYXIS could choose to partition the program where it expects few control transfers, when in reality the program might exercise that piece of code very frequently. For this reason, developers may need to re-profile their applications if the workload changes dramatically and have PYXIS dynamically switch among the different partitions. A future work would be to automatically detect significant changes in workload and trigger re-profiling.

3.4.2 The Partition Graph

After profiling, the normalized source files are submitted to the partitioner to assign placements for code and data in the program. First, the partitioner performs an object-sensitive points-to analysis [81] using the Accrue Analysis Framework [21]. This analysis approximates the set of objects that may be referenced by each expression in the program. Using the results of the points-to analysis, an interprocedural definition-use analysis links together all assignment statements (defs) with expressions that may observe the value of those assignments (uses). Next, a control dependency analysis [27] links statements that cause branches in the control flow graph (i.e., conditionals, loops, and calls) with the statements whose execution depends on them. For instance, each statement in the body of a loop has a control dependency on the loop condition and is therefore linked to the loop condition.

The precision of these analyses can affect the quality of the partitions found by PYXIS and therefore performance. To preserve soundness, the analysis is *conservative*, which means that some dependencies identified by the analysis may not be necessary. Unnecessary dependencies result in inaccuracies in the cost model and superfluous data transfers at run time. For this work we used a "2full+1H" object-sensitive analysis as described in prior work [96].

Dependencies. Using these analyses, the partitioner builds the partition graph, which represents information about the program's dependencies. The partition graph contains nodes for each statement in the program and edges for the dependencies between them.

In the partition graph, each statement and field in the program is represented by a node in the graph. Dependencies between statements and fields are represented by edges, and edges have weights that model the cost of satisfying those dependencies. Edges represent different kinds of dependencies between statements:

- A *control edge* indicates a control dependency between two nodes in which the computation at the source node influences whether the statement at the destination node is executed.
- A *data edge* indicates a data dependency in which a value assigned in the source statement influences the computation performed at the destination statement. In this case the source statement is a *definition* (or *def*) and the destination is a *use*.
- An *update edge* represents an update to the heap and connects field declarations to statements that update them.

Part of the partition graph for our running example is shown in Fig. 3-4. Each node in the graph is labeled with the corresponding line number from Fig. 3-2. Note, for example, that while lines

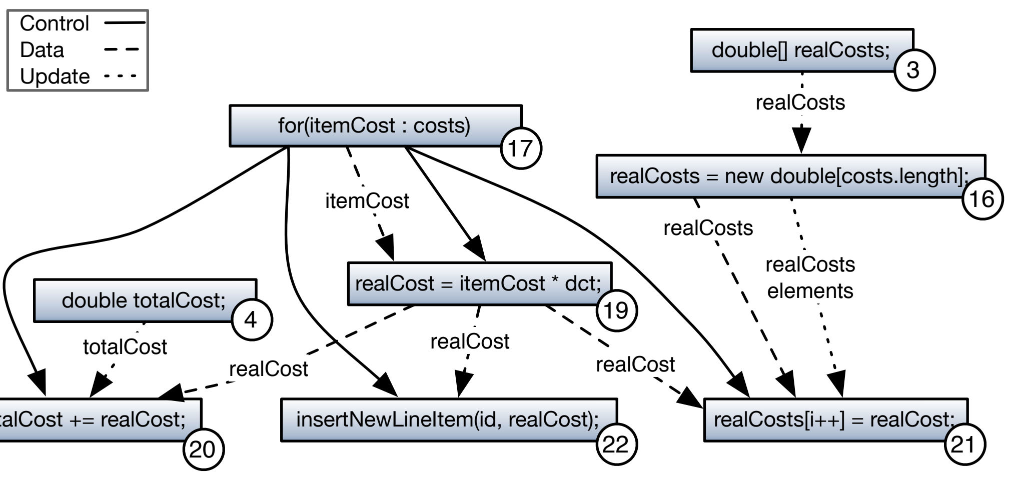


Figure 3-4: A partition graph for part of the code from Fig. 3-2

24–26 appear sequentially in the program text, the partition graph shows that these lines can be safely executed in any order, as long as they follow line 23. The partitioner adds additional edges (not shown) for output dependencies (write-after-write) and anti-dependencies (read-before-write), but these edges are currently only used during code generation and do not affect the choice of node placements. For each JDBC call that interacts with the database, we also insert control edges to nodes representing "database code."

Edge weights. The tradeoff between network overhead and server load is represented by weights on nodes and edges. Each statement node assigned to the database results in additional estimated server load in proportion to the execution count of that statement. Likewise, each dependency that connects two statements (or a statement and a field) on separate partitions incurs estimated network latency proportional to the number of times the dependency is satisfied. Formally, let $\mathsf{cnt}(s)$ be the number of times statement s was executed in the profile. Given a control or data edge from statement src to statement dst, we approximate the number of times the edge e was satisfied as $\mathsf{cnt}(e) = \min(\mathsf{cnt}(src), \mathsf{cnt}(dst))$.

Furthermore, let $\mathsf{size}(def)$ represent the average size of the data that is assigned by statement def. Then, given an average network latency LAT and a bandwidth BW, the partitioner assigns weights to edges e and nodes s as follows:

- Control edge e: LAT $\cdot\ \mathsf{cnt}(e)$
- Data edge e: $\dfrac{\mathsf{size}(src)}{\text{BW}} \cdot \mathsf{cnt}(e)$
- Update edge e: $\dfrac{\mathsf{size}(src)}{\text{BW}} \cdot \mathsf{cnt}(dst)$
- Statement node s: $\mathsf{cnt}(s)$
- Field node: 0

Note that some of these weights, such as those on control and data edges, represent times. The partitioner's objective is to minimize the sum of the weights of edges cut by a given partitioning. The weight on statement nodes is used separately to enforce the constraint that the total CPU load on the server does not exceed a given maximum value.

The formula for data edges charges for bandwidth but not latency. For all but extremely large objects, the weights of data edges are therefore much smaller than the weights of control edges. By setting weights this way, we leverage an important aspect of the PYXIS runtime: satisfying a data dependency does not necessarily require separate communication with the remote host. As described in Sec. 3.3.2, PyxIL programs maintain consistency of the heap by batching updates and sending them on control transfers. Control dependencies between statements on different partitions inherently require communication in order to transfer control. However, since updates to data can be piggy-backed on a control transfer, the marginal cost to satisfy a data dependency is proportional to the size of the data. For most networks, bandwidth delay is much smaller than propagation delay, so reducing the number of messages a partition requires will reduce the average latency even though the size of messages increases. Furthermore, by encoding this property of data dependencies as weights in the partition graph, we influence the choice of partition made by the solver; cutting control edges is usually more expensive than cutting data edges.

Our simple cost model does not always accurately estimate the cost of control transfers. For example, a series of statements in a block may have many control dependencies to code outside the block. Cutting all these edges could be achieved with as few as one control transfer at runtime, but in the cost model, each cut edge contributes its weight, leading to overestimation of the total

$$\begin{aligned} \text{Minimize:} \quad & \Sigma_{e_i \in \text{Edges}} \; e_i \cdot w_i \\ & n_j - n_k - e_i \leq 0 \\ \text{Subject to:} \quad & n_k - n_j - e_i \leq 0 \\ & \vdots \\ & \Sigma_{n_i \in \text{Nodes}} \; n_i \cdot w_i \leq \text{Budget} \end{aligned}$$

Figure 3-5: Partitioning problem

partitioning cost. Also, fluctuations in network latency and CPU utilization could also lead to inaccurate average estimates and thus result in suboptimal partitions. We leave more accurate cost estimates to future work.

3.4.3 Optimization Using Integer Programming

The weighted graph is then used to construct a Binary Integer Programming problem [106]. For each node we create a binary variable $n \in$ Nodes that has value 0 if it is partitioned to the application server and 1 if it is partitioned to the database. For each edge we create a variable $e \in$ Edges that is 0 if it connects nodes assigned to the same partition and 1 if it is *cut*; that is, the edge connects nodes on different partitions. This problem formulation seeks to minimize network latency subject to a specified budget of instructions that may be executed on the server. In general, the problem has the form shown in Figure 3-5. The objective function is the summation of edge variables e_i multiplied by their respective weights, w_i. For each edge we generate two constraints that force the edge variable e_i to equal 1 if the edge is cut. Note that for both of these constraints to hold, if $n_j \neq n_k$ then $e_i = 1$, and if $n_j = n_k$ then $e_i = 0$. The final constraint in Figure 3-5 ensures that the summation of node variables n_i, multiplied by their respective weights w_i, is at most the "budget" given to the partitioner. This constraint limits the load assigned to the database server.

In addition to the constraints shown in Figure 3-5, we add placement constraints that pin certain nodes to the server or the client. For instance, the "database code" node used to model the JDBC driver's interaction with the database must always be assigned to the database, and similarly assign code that prints on the user's console to the application server.

The placement constraints for JDBC API calls are more interesting. Since the JDBC driver maintains unserializable native state regarding the connection, prepared statements, and result sets used by the program, all API calls must occur on the same partition. While these calls could also be pinned to the database, this could result in low-quality partitions if the partitioner has very little budget. Consider an extreme case where the partitioner is given a budget of 0. Ideally, it should create a partition equivalent to the original program where all statements are executed on the application server. Fortunately, this behavior is easily encoded in our model by assigning the same node variable to all statements that contain a JDBC call and subsequently solving for the values of the node variables. This encoding forces the resulting partition to assign all such calls to the same partition.

After instantiating the partitioning problem, PYXIS invokes the solver. If the solver returns with a solution, PYXIS applies it to the partition graph by assigning all nodes a location and marking all edges that are cut. Our implementation currently supports lpsolve [76] and Gurobi Optimizer[61].

Finally, the partitioner generates a PyxIL program from the partition graph. For each field and statement, the code generator emits a placement annotation `:APP:` or `:DB:` according to the solution returned by the solver. For each dependency edge between remote statements, the code generator places a heap synchronization operation after the source statement to ensure the remote heap is up to date when the destination statement is executed. Synchronization operations are always placed after statements that update remotely partitioned fields or arrays.

3.4.4 Statement Reordering

A partition graph may generate several valid PyxIL programs that have the same cost under the cost model since the execution order of some statements in the graph is ambiguous. To eliminate unnecessary control transfers in the generated PyxIL program, the code generator performs a reordering optimization to create larger contiguous blocks with the same placement, reducing control transfers. Because it captures all dependencies, the partition graph is particularly well suited to this transformation. In fact, PDGs have been applied to similar problems such as vectorizing program statements [53]. The reordering algorithm is simple. Recall that in addition to control, data, and update edges, the partitioner includes additional (unweighted) edges for output dependencies (for ordering writes) and anti-dependencies (for ordering reads before writes) in the partition graph. We can usefully reorder the statements of each block without changing the semantics of the program [53] by topologically sorting the partition graph while ignoring back-edges and interprocedural edges.[5]

The topological sort is implemented as a breadth-first traversal over the partition graph. Whereas a typical breadth-first traversal would use a single FIFO queue to keep track of nodes not yet visited, the reordering algorithm uses two queues, one for `:DB:` statements and one for `:APP:` statements. Nodes are dequeued from one queue until it is exhausted, generating a sequence of statements that are all located in one partition. Then the algorithm switches to the other queue and starts generating statements for the other partition. This process alternates until all nodes have been visited.

3.4.5 Insertion of Synchronization Statements

The code generator is also responsible for placing heap synchronization statements to ensure consistency of the PYXIS distributed heap. Whenever a node has outgoing data edges, the code generator emits a `sendAPP` or `sendDB` depending on where the updated data is partitioned. At the next control transfer, the data for all such objects so recorded is sent to update of the remote heap.

Heap synchronization is conservative. A data edge represents a definition that may reach a field or array access. Eagerly synchronizing each data edge ensures that all heap locations will be up to date when they are accessed. However, imprecision in the reaching definitions analysis is unavoidable, since predicting future accesses is undecidable. Therefore, eager synchronization is sometimes wasteful and may result in unnecessary latency from transferring updates that are never used.

The PYXIS runtime system also supports lazy synchronization in which objects are fetched from the remote heap at the point of use. If a use of an object performs an explicit fetch, data edges to that use can be ignored when generating data synchronization. Lazy synchronization makes sense for large objects and for uses in infrequently executed code. In the current PYXIS

[5]Side-effects and data dependencies due to calls are summarized at the call site.

```java
public class Order {
  ObjectId oid;
  class Order_app { int id; ObjectId realCostsId; }
  class Order_db { double totalCost; }
  ...
}
```

Figure 3-6: Partitioning fields into APP and DB objects

```
stack locations for computeTotalCost:
stack[0] = oid
stack[1] = dct
stack[2] = object ID for costs
stack[3] = costs.length
stack[4] = i
stack[5] = loop index
stack[6] = realCost
```

```java
computeTotalCost0:
  pushStackFrame(stack[0]);
  setReturnPC(computeTotalCost1);
  return getCosts0; // call this.getCosts()

computeTotalCost1:
  stack[2] = popStack();
  stack[3] = APPHeap[stack[2]].length;
  oid = stack[0];
  APPHeap[oid].realCosts = nativeObj(new dbl[stack[3]);
  sendAPP(oid);
  sendNative(APPHeap[oid].realCosts, stack[2]);
  stack[4] = 0;
  return computeTotalCost2; // control transfer to DB
```

```java
computeTotalCost2:
  stack[5] = 0; // i = 0
  return computeTotalCost3; // start the loop

computeTotalCost3:
  if (stack[5] < stack[3]) // loop index < costs.length
        return computeTotalCost4; // loop body
  else return computeTotalCost5; // loop exit

computeTotalCost4:
  itemCost = DBHeap[stack[2]][stack[5]];
  stack[6] = itemCost * stack[1];
  oid = stack[0];
  DBHeap[oid].totalCost += stack[6];
  sendDB(oid);
  APPHeap[oid].realCosts[stack[4]++] = stack[6];
  sendNative(APPHeap[oid].realCosts);
  ++stack[5];
  pushStackFrame(APPHeap[oid].id, stack[6]);
  setReturnPC(computeTotalCost3);
  return insertNewLineItem0; // call insertNewLineItem

computeTotalCost5:
  return returnPC;
```

Figure 3-7: Running example: APP code (left) and DB code (right). For simplicity, execution blocks are presented as a sequence of statements preceded by a label.

implementation, lazy synchronization is not used in normal circumstances. We leave a hybrid eager/lazy synchronization scheme to future work.

3.5 PyxIL compiler

The PyxIL compiler translates PyxIL source code into two partitioned Java programs that together implement the semantics of the original application when executed on the PYXIS runtime. To illustrate this process, we give an abridged version of the compiled `Order` class from the running example. Fig. 3-6 shows how `Order` objects are split into two objects of classes `Order_app` and `Order_db`, containing the fields assigned to `APP` and `DB` respectively in Fig. 3-3. Both classes contain a field `oid` (line 2) which is used to identify the object in the PYXIS-managed heaps (denoted by `DBHeap` and `APPHeap`). In essence, the fields in the original class are split across two embedded classes in the compiled code, each containing the fields that are assigned to the runtime hosted on the application server or the database. Because each object instance from the original program is split into two, all objects on the heap are referenced with heap object IDs rather than the actual object itself (e.g., `realCostsId` field on line 3).

Additionally, an *entry point* (see Sec. 3.5.2) is generated for those methods that have public visibility in the PyxIL source (lines 3–8). In the following I describe the compiled constructs in detail.

3.5.1 Representing Execution Blocks

In order to arbitrarily assign program statements to either the application or the database server, the runtime needs to have complete control over program control flow as it transfers between the servers. PYXIS accomplishes this by compiling each PyxIL method into a set of *execution blocks*, each of which corresponds to a straight-line PyxIL code fragment. For example, Fig. 3-7 shows the code generated for the method `computeTotalCost` in the class. This code includes execution blocks for both the `:APP:` and `:DB:` partitions. Note that local variables in the PyxIL source are translated into indices of an array `stack` that is explicitly maintained in the Java code and is used to model the stack frame that the method is currently operating on.

The key to managing control flow is that each execution block ends by returning the identifier of the next execution block. The PYXIS runtime, upon receiving the identifier of the next execution block, will either start executing the new block if it is assigned the same partition as the runtime, or the runtime will block and send a control transfer message to the remote runtime along with the block identifier. This style of code generation is similar to that used by the SML/NJ compiler [98] to implement continuations; indeed, code generated by the PyxIL compiler is essentially in continuation-passing style [48].

For instance, in Fig. 3-7, execution of `computeTotalCost` starts with block `computeTotalCost0` at line 1. After creating a new stack frame and passing in the object ID of the receiver, the block asks the runtime to execute `getCosts0` with `computeTotalCost1` recorded as the return address. The runtime then executes the execution blocks associated with `getCosts()`. When `getCosts()` returns, it jumps to `computeTotalCost1` to continue the method, where the result of the call is popped into `stack[2]`.

Next, control is transferred to the database server on line 14, because block `computeTotalCost1` returns the identifier of a block that is assigned to the database server (`computeTotalCost2`). This implements the transition in Fig. 3-3 from (APP) line 24 to (DB) line 26. Execution then continues with `computeTotalCost3`, which implements the evaluation of the loop condition in Fig. 3-3.

This example shows how the use of execution blocks gives the PYXIS partitioner complete freedom to place each piece of code and data to the servers. Furthermore, because the runtime regains control after every execution block, it has the ability to perform other tasks between execution blocks or while waiting for the remote server to finish its part of the computation such as garbage collection on local heap objects.

This approach to code execution has a number of advantages. First, as I explain in Sec. 3.6, the transfer of heap objects can be piggybacked on control transfers. Second, with explicit heap synchronization embedded in the code, we no longer need to augment the original method to transfer heap objects needed by remote code. Finally, because control transfer is implemented by informing the remote runtime of next execution block ID to run and returning to the caller, long chains of handler threads are not spawned, which avoids incurring substantial overhead in maintaining thread pools.

3.5.2 Interoperability with Existing Modules

PYXIS does not require that the whole application be partitioned. This is useful, for instance, if the code to be partitioned is a library used by other, non-partitioned code. Another benefit is that code without an explicit main method can be partitioned, such as a servlet whose methods are invoked by the application server.

To use this feature, the developer indicates the *entry points* within the source code to be partitioned, i.e., methods that the developer exposes to invocations from non-partitioned code. The

PyxIL compiler automatically generates a wrapper for each entry point, such as the one shown in Fig. 3-8, that does the necessary stack setup and teardown to interface with the PYXIS runtime. Non-PyxIL code can simply invoke the generated wrappers as if they are standard Java code.

3.6 PYXIS runtime system

The PYXIS runtime executes the compiled PyxIL program. The runtime is a Java program that runs in unmodified JVMs on each server. In this section I describe its operation and how control transfer and heap synchronization is implemented.

3.6.1 General Operations

The PYXIS runtime maintains the program stack and distributed heap. Each execution block discussed in Sec. 3.5.1 is implemented as a Java class with a `call` method that implements the program logic of the given block. When execution starts from any of the entry points, the runtime invokes the `call` method on the block that was passed in from the entry point wrapper. Each `call` method returns the next block for the runtime to execute next, and this process continues on the local runtime until it encounters an execution block that is assigned to the remote runtime. When that happens, the runtime sends a *control transfer message* to the remote runtime and waits until the remote runtime returns control, informing it of the next execution block to run on its side.

Multi-threading support. PYXIS does not currently support thread instantiation or shared-memory multithreading, but a multithreaded application can first instantiate threads outside of PYXIS and then have PYXIS manage the code to be executed by each of the threads.

Exception handling. The current implementation supports exception handling across runtimes. As a result of analyzing the original source code, each PYXIS execution block is annotated with register and un-register statements for exception handlers. When a register statement is encountered during program execution, the PYXIS runtime will add to its exception table the type of exception that is handled, and the the identifier of the execution block that corresponds to the start of the exception handling code. The record is removed when the corresponding un-register statement is subsequently encountered. As in standard JVM, when an exception is thrown during program execution, the runtime first consults its exception table to see if the exception is handled, and if so it will jump to the execution block corresponding to the beginning of the handler. If the exception is handled but the beginning of the execution handler block is assigned to the remote runtime, then the runtime will send a control transfer message to the remote runtime with the exception information. Finally, if the exception is not handled (i.e., no corresponding entry found in the exception table), then the PYXIS runtime exits and propagates the exception to the underlying JVM.

Memory management. Since the PYXIS runtime manages its own program heap, we have implemented a trace-based, generational garbage collector for its managed program heap [78]. When the number of free slots in the heap drops below a threshold, the garbage collector will pause the runtime's execution and mark all live objects in the heap. However, since each object is physically partitioned between the two heaps, an object is only garbage only if *both* halves are labeled as garbage by the two runtimes. Thus, actual memory reclamation is not done until both runtimes have completed tracing live objects, upon which they will compare the two lists of potential garbage and only remove those objects that appear in both lists.

3.6.2 Program State Synchronization

When a control transfer happens, the local runtime needs to communicate with the remote runtime about any changes to the program state (i.e., changes to the stack or program heap). Stack changes are always sent along with the control transfer message and as such are not explicitly encoded in the PyxIL code. However, requests to synchronize the heaps are explicitly embedded in the PyxIL code, allowing the partitioner to make intelligent decisions about what modified objects need to be sent and when. As mentioned in Sec. 3.4.5, PYXIS includes two heap synchronization routines: `sendDB` and `sendAPP`, depending on which portion of the heap is to be sent. In the implementation, the heap objects to be sent are simply piggy-backed onto the control transfer messages (just like stack updates) to avoid initiating more round trips. We measure the overhead of heap synchronization and discuss the results in Sec. 3.7.3.

3.6.3 Selecting a Partitioning Dynamically

The runtime also supports dynamically choosing between partitionings with different CPU budgets based on the current load on the database server. It uses a feedback-based approach in which the PYXIS runtime on the database server periodically polls the CPU utilization on the server and communicates that information to the application server's runtime.

At each time t when a load message arrives with server load S_t, the application server computes an exponentially weighted moving average (EWMA) of the load, $L_t = \alpha L_{t-1} + (1 - \alpha)S_t$. Depending on the value of L_t, the application server's runtime dynamically chooses which partition to execute at each of the entry points. For instance, if L_t is high (i.e., the database server is currently loaded), then the application server's runtime will choose a partitioning that was generated using a low CPU budget until the next load message arrives. Otherwise the runtime uses a partitioning that was generated with a higher CPU-budget since L_t indicates that CPU resources are available on the database server. The use of EWMA here prevents oscillations from one deployment mode to another. In our experiments with TPC-C (Sec. 3.7.1), we used two different partitions and set the threshold between them to be 40% (i.e., if $L_t > 40$ then the runtime uses a lower CPU-budget partition). Load messages were sent every 10 seconds with α set to 0.2. The values were determined after repeat experimentation.

This simple, dynamic approach works when separate client requests are handled completely independently at the application server, since the two instances of the program do not share any state. This scenario is likely in many server settings; e.g., in a typical web server, each client is handled completely independent of other clients. Generalizing this approach so that requests sharing state can utilize this adaptation feature is future work.

```
public class Order {
  ...
  public void computeTotalCost(double dct) {
    pushStackFrame(oid, dct);
    execute(computeTotalCost0);
    popStackFrame();
    return; // no value to be returned
  }
}
```

Figure 3-8: Wrapper for interfacing with regular Java

3.7 Experiments

In this section I report experimental results. The goals of the experiments are:

1. To evaluate PYXIS's ability to generate partitions of an application under different server loads and input workloads.
2. To measure the performance of those partitionings as well as the overhead of performing control transfers between runtimes.

PYXIS is implemented in Java using the Polyglot compiler framework [85] with Gurobi and lpsolve as the linear program solvers. The experiments used mysql 5.5.7 as the DBMS with buffer pool size set to 1GB, hosted on a machine with 16 2.4GHz cores and 24GB of physical RAM. We used Apache Tomcat 6.0.35 as the web server for TPC-W, hosted on a machine with eight 2.6GHz cores and 33GB of physical RAM. The disks on both servers are standard serial ATA disks. The two servers are physically located in the same data center and have a ping round trip time of 2ms. All reported performance results are the average of three experimental runs.

For TPC-C and TPC-W experiments below, we implemented three different versions of each benchmark and measured their performance as follows:

- **JDBC**: This is a standard implementation of each benchmark where program logic resides completely on the application server. The program running on the application server connects to the remote DBMS using JDBC and makes requests to fetch or write back data to the

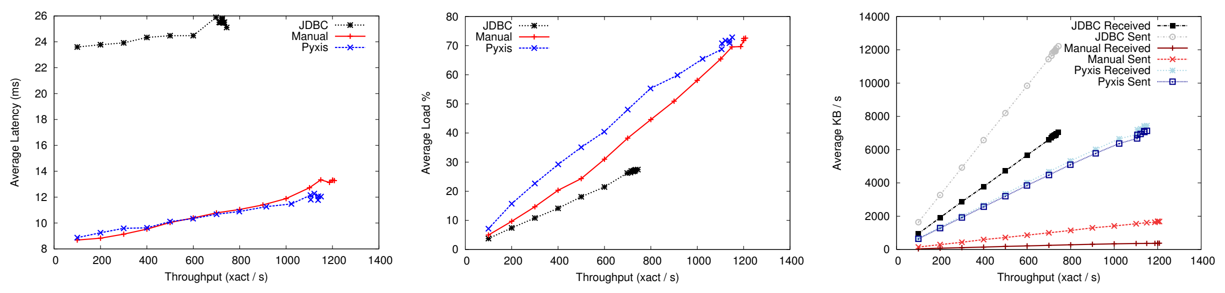
DBMS using the obtained connection. A round trip is incurred for each database operation between the two servers.

- **Manual**: This is an implementation of the benchmarks where all the program logic is manually split into two halves: the "database program," which resides on the JVM running on the database server, and the "application program," which resides on the application server. The application program is simply a wrapper that issues RPC calls via the Java Remote Method Invocation (RMI) interface to the database program for each type of transaction, passing along with it the arguments to each transaction type. The database program executes the actual program logic associated with each type of transaction and opens local connections to the DBMS for the database operations. The final results are returned to the application program. This is the exact opposite from the JDBC implementation with all program logic residing on the database server. Here, each transaction only incurs one round trip. For TPC-C we also implemented a version that implements each transaction as a MySQL user-defined function rather than issuing JDBC calls from a Java program on the database server. We found that this did not significantly impact the performance results.

- **Pyxis**: To obtain instruction counts, we first profiled the JDBC implementation under different target throughput rates for a fixed period of time. We then asked PYXIS to generate different partitions with different CPU budgets. We deployed the two partitions on the application and database servers using PYXIS and measured their performance.

3.7.1 TPC-C Experiments

In the first set of experiments we implemented the TPC-C workload in Java. Our implementation is similar to an "official" TPC-C implementation but does not include client think time. The database contains data from 20 warehouses (initial size of the database is 23GB), and for the experiments we instantiated 20 clients issuing new order transactions simultaneously from the application server to the database server with 10% transactions rolled back. We varied the rate at which the clients issued the transactions and then measured the resulting system's throughput and average latency



(a) Latency (b) CPU utilization (c) Network utilization

Figure 3-9: TPC-C experiment results on 16-core database server

of the transactions for 10 minutes.

Full CPU Setting

In the first experiment we allowed the DBMS and JVM on the database server to use all 16 cores on the machine and gave PYXIS a large CPU budget. Fig. 3-9(a) shows the throughput versus latency. Fig. 3-9(b) and (c) show the CPU and network utilization under different throughputs.

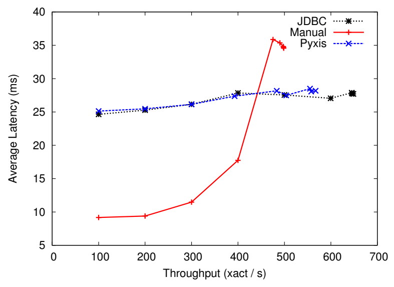
The results illustrate several points. First, the Manual implementation was able to scale better than JDBC both in terms of achieving lower latencies, and being able to process more transactions within the measurement period (i.e., achieve a high overall throughput). The higher throughput in the Manual implementation is expected since each transaction takes less time to process due to fewer round trips and incurs less lock contention in the DBMS due to locks being held for less time.

For Pyxis, the resulting partitions for all target throughputs were very similar to the Manual implementation. Using the provided CPU budget, PYXIS assigned most of the program logic to be executed on the database server. This is the desired result; since there are CPU resources available on the database server, it is advantageous to push as much computation to it as possible to achieve maximum reduction in the number of round trips between the servers. The difference in performance between the Pyxis and Manual implementations is negligible (within the margin of error of the experiments due to variation in the TPC-C benchmark's randomly generated transactions).

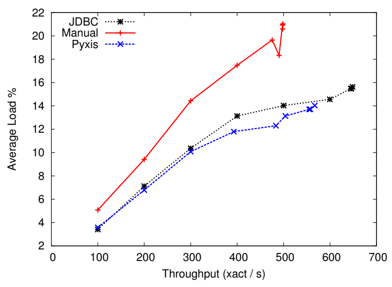
There are some differences in the operations of the Manual and Pyxis implementations, however. For instance, in the Manual implementation, only the method arguments and return values are communicated between the two servers, whereas the Pyxis implementation also needs to transmit changes to the program stack and heap. This can be seen from the network utilization measures in Fig. 3-9(c), which show that the Pyxis implementation transmits more data compared to the Manual implementation due to synchronization of the program stack between the runtimes. We experimented with various ways to reduce the number of bytes sent, such as with compression and custom serialization, but found that they used more CPU resources and increased latency. However, Pyxis sends data only during control transfers, which are fewer than database operations. Consequently, Pyxis sends less data than the JDBC implementation.

Limited CPU Setting

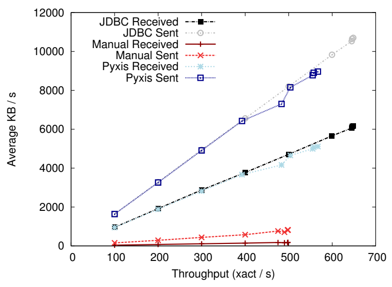
Next we repeated the same experiment, but this time limited the DBMS and JVM (applicable only to Manual and Pyxis implementations) on the database server to use a maximum of three CPUs and gave PYXIS a small CPU budget for partitioning purposes. This experiment was designed to emulate programs running on a highly contended database server or a database server serving load on behalf of multiple applications or tenants. The resulting latencies are shown in Fig. 3-10(a) with CPU and network utilization shown in Fig. 3-10(b) and (c). The Manual implementation has lower latencies than Pyxis and JDBC when the throughput rate is low, but for higher throughput values the JDBC and Pyxis implementations outperforms Manual. With limited CPUs, the Manual implementation uses up all available CPUs when the target throughput is sufficiently high. In contrast, all partitions produced by PYXIS for different target throughput values resemble the JDBC implementation in which most of the program logic is assigned to the application server. This configuration enables the Pyxis implementation to sustain higher target throughputs and deliver lower latency when the database server experiences high load. The resulting network and CPU utilization are similar to those of JDBC as well.



(a) Latency



(b) CPU utilization



(c) Network utilization

Figure 3-10: TPC-C experiment results on 3-core database server

Dynamically Switching Partitions

In the final experiment we enabled the dynamic partitioning feature in the runtime as described in Sec. 3.6.3. The two partitionings used in this case were the same as the partitionings used in the previous two experiments (i.e., one that resembles Manual and another that resembles JDBC). In this experiment, however, we fixed the target throughput to be 500 transactions / second for 10 minutes since all implementations were able to sustain that amount of throughput. After three minutes elapsed we loaded up most of the CPUs on the database to simulate the effect of limited CPUs. The average latencies were then measured during each 30-second period, as shown in Fig. 3-11. For the Pyxis implementation, we also measured the proportion of transactions executed using the JDBC-like partitioning within each 1-minute period. Those are plotted next to each data point.

As expected, the Manual implementation had lower latencies when the server was not loaded. For JDBC the latencies remain constant as it does not use all available CPUs even when the server is loaded. When the server is unloaded, however, it had higher latency compared to the Manual implementation. The Pyxis implementation, on the other hand, was able to take advantage of the two implementations with automatic switching. In the ideal case, Pyxis's latencies should be the minimum of the other two implementations at all times. However, due to the use of EWMA, it took a short period of time for Pyxis to adapt to load changes, although it eventually settled to an all-application (JDBC-like) deployment as shown by the proportion numbers. This experiment illustrates that even if the developer was not able to predict the amount of available CPU resources, PYXIS can generate different partitions under various budgets and automatically choose the best one given the actual resource availability.

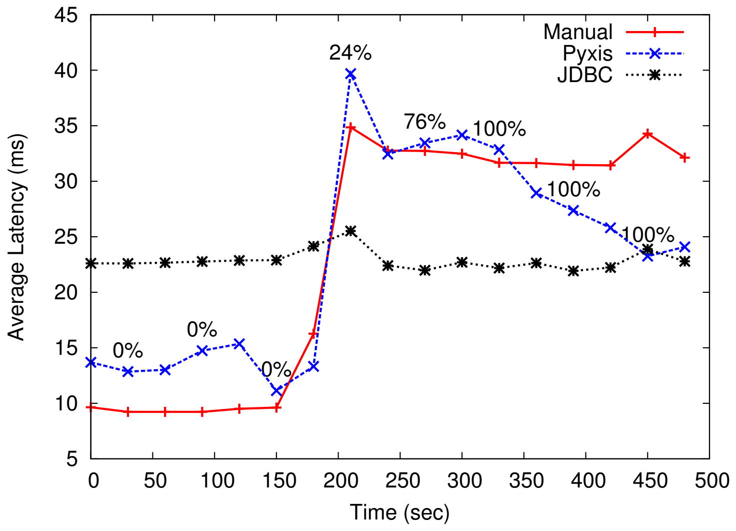


Figure 3-11: TPC-C performance results with dynamic switching

3.7.2 TPC-W

In the next set of experiments, we used a TPC-W implementation written in Java. The database contained 10,000 items (about 1GB on disk), and the implementation omitted the thinking time. We drove the load using 20 emulated browsers under the browsing mix configuration and measured the average latencies at different target Web Interactions Per Seconds (WIPS) over a 10-minute period. We repeated the TPC-C experiments by first allowing the JVM and DBMS to use all 16 available cores on the database server followed by limiting to three cores only. Fig. 3-12 and Fig. 3-13 show the latency results. The CPU and network utilization are similar to those in the TPC-C experiments and are not shown.

Compared to the TPC-C results, we see a similar trend in latency. However, since the program logic in TPC-W is more complicated than TPC-C, the Pyxis implementation incurs a bit more overhead compared to the Manual implementation.

One interesting aspect of TPC-W is that unlike TPC-C, some web interactions do not involve database operations at all. For instance, the order inquiry interaction simply prints out a HTML form. PYXIS decides to place the code for those interactions entirely on the application server even when the server budget is high. This choice makes sense since executing such interactions on the application server does not incur any round trips to the database server. Thus the optimal decision, also found by PYXIS, is to leave the code on the application server rather than pushing it to the database server as stored procedures.

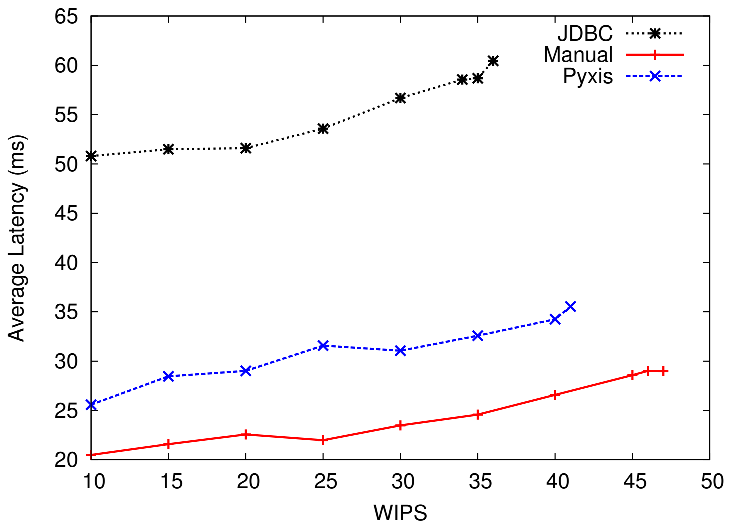


Figure 3-12: TPC-W latency experiment using 16 cores

3.7.3 Microbenchmark 1

In our third experiment we compared the overhead of PYXIS to native Java code. We expected code generated by PYXIS to be slower because all heap and stack variable manipulations in PYXIS are managed through special PYXIS objects. To quantify this overhead we implemented a linked list and assigned all the fields and code to be on the same server. This configuration enabled a fair comparison to a native Java implementation. The results show that the PYXIS implementation has an average overhead of 6× compared to the Java implementation. Since the experiment did not involve any control transfers, the overhead is entirely due to the time spent running execution blocks and bookkeeping in the PYXIS managed program stack and heap. This benchmark is a worst-case

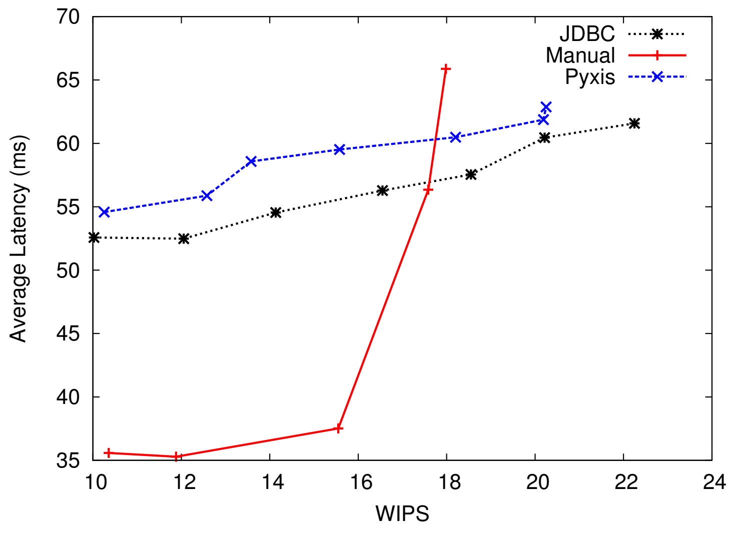


Figure 3-13: TPC-W latency experiment using 3 cores

scenario for PYXIS since the application is not distributed at all. We expect that programmers will ask PYXIS to partition fragments of an application that involve database operations and will implement the rest of their program as native Java code. As noted in Sec. 3.5.2, this separation of distributed from local code is fully supported in our implementation; programmers simply need to mark the classes or functions they want to partition. Additionally, results in the previous sections show that allowing PYXIS to decide on the partitioning for distributed applications offers substantial benefits over native Java (e.g., the JDBC and Manual implementations) even though PYXIS pays some local execution overhead.

The overhead of PYXIS is hurt by the mismatch between the Java Virtual Machine execution model and PYXIS's execution model based on execution blocks. Java is not designed to make this style of execution fast. One concrete example of this problem is that we cannot use the Java stack to store local variables. Targeting a lower-level language should permit lowering the overhead substantially.

CPU Load	**APP**	**APP|DB**	**DB**
No load	7m37s	7m	**6m39s**
Partial load	8m55s	**8m**	8m11s
Full load	**16m36s**	17m30s	17m44s

Figure 3-14: Microbenchmark 2 results

3.7.4 Microbenchmark 2

In the final experiment, we compare the quality of the generated partitions under different CPU budgets by using a simple microbenchmark designed to have several interesting partitionings. This benchmark runs three tasks in order: it issues a 100k small select queries, performs a computationally intensive task (compute SHA1 digest 500k times), and issues another 100k select queries. We gave three different CPU budget values (low, medium, high) to represent different server loads and asked PYXIS to produce partitions. Three different partitions were generated: one that assigns all logic to application server when a low budget was given (**APP**); one that assigns the query portions of the program to the database server and the computationally intensive task to the application when a middle range budget was given (**APP|DB**); and finally one that assigns everything to the database server (**DB**). We measured the time taken to run the program under different real server loads, and the results are shown in Fig. 3-14.

The results show that PYXIS was able to generate a partition that fits different CPU loads in terms of completion time (highlighted in Fig. 3-14). While it might be possible for developers to avoid using the PYXIS partitioner and manually create the two extreme partitions (**APP** and **DB**), this experiment shows that doing so would miss the "middle" partitions (such as **APP|DB**) and lead to suboptimal performance. Using PYXIS, the developer only needs to write the application once, and PYXIS will automatically produce partitions that are optimized for different server loads.

3.8 Summary

In this chapter I presented Pyxis, a system for partitioning database applications across an application and database server. The goal of this partitioning is to automatically determine optimal placement of computation across multiple servers. In this process, we are able to eliminate the many small, latency-inducing round trips between the application and database servers by pushing blocks of code into the database as stored procedures.

Benefits of our approach include that users are not required to manually convert their code into the stored procedure representation, and that our system can dynamically identify the best decomposition of the application based on variations in server load. Our key technical contributions include a cost-based profiling and analysis framework, as well as a compilation strategy for converting the output of our analysis into valid executables. This work opens many new problems in adapting programs to changing environments, a topic that I will discuss in Ch. 6.

Chapter 4

SLOTH: Reducing Communication Overhead

In the previous chapter, I described the design of PYXIS, a system that automatically partitions database applications into the server and query components. Although one of the goals of static code partitioning is to reduce the number of network round trips between the application and database servers, it is inevitable that the two servers will need to communicate as the application executes. Unfortunately, these network roundtrips increase the latency of database applications and represent a significant fraction of overall execution time for many applications.

One way to eliminate more of these round trips is to employ *batching*, i.e., group together SQL statements into groups that can be executed on the database server in a single round trip. Unfortunately, prior work are limited by 1) a requirement that developers manually identify batching opportunities in the source code, or 2) the fact that they employ static analysis techniques that cannot exploit many opportunities for batching due to imprecision of the analysis.

In this chapter, I describe SLOTH, a new system that makes use of *lazy evaluation* to expose query batching opportunities during application execution, even across loops, branches, and method boundaries. By using lazy evaluation, we allow the application to dynamically decide when and how to batch network round trips. We evaluated SLOTH using over 100 benchmarks from two large-scale open-source applications, and achieved up to a 3$\times$ reduction in page load time by delaying computation as much as possible.[1]

[1]Materials in this chapter are based on work published as Cheung, Madden, and Solar-Lezama, "Sloth: Being Lazy

4.1 Introduction

As discussed in Ch. 1, the server and query components of many database applications are located on physically separated from the servers hosting the application. Even though the two machines tend to reside in close proximity, a typical page load spends a significant amount of time issuing SQL queries and waiting for network round trips to complete, with a consequent impact on application latency. The situation is exacerbated by object-relational mapping (ORM) frameworks such as Hibernate and Django, which access the database by manipulating native objects rather than issuing SQL queries. These frameworks automatically translate accesses to objects into SQL, often resulting in multiple database queries (and round trips) to reconstruct a single object. For example, even with two machines in the same data center, we found that many pages spend 50% or more of their time waiting on network communication.

As noted in Ch. 1, latency is important for a variety of reasons. First, in web applications, even hundreds of milliseconds of additional latency can dramatically increase the dissatisfaction of users. For example, a 2010 study by Akamai suggested that 57% of users will abandon a web page that takes more than three seconds to load [16]. As another example, Google reported in 2006 that an extra half second of latency reduced the overall traffic by 20% [11]. These numbers are likely worse on modern computers where pages load faster and faster. Given that database load times are only a fraction of total page rendering times, it is important that database query time be minimized. Second, ORM frameworks such as Hibernate can greatly amplify load times by performing additional round trips for each of many items found by an initial lookup, such that a few 10's of milliseconds per object can turn into seconds of additional latency for an entire web page [23, 4, 13, 19]. Though some techniques (such as "eager fetching" in Hibernate) are available to mitigate this, they are far from perfect as we discuss below. Finally, decreasing latency often increases throughput since each request takes less time, and ties up fewer resources while waiting for a query to complete.

There are two general approaches to cope with this problem: i) programs can hide this latency by overlapping communication and computation, ii) programs can reduce the number of round trips

is a Virtue (When Issuing Database Queries)," in proceedings of SIGMOD 14 [38].

by fetching more data (or issuing multiple queries) in each one, or iii) using tools such as PYXIS to statically distribute computation between the application and database servers with the aim to reduce the number of round trips between servers. Latency hiding is most commonly achieved by prefetching query results so that the communication time overlaps with computation, and the data is ready when the application really needs it. All of these techniques have been explored in prior research. Latency hiding, which generally takes the form of asynchronously "prefetching" query results so that they are available when needed by the program, was explored by Ramachandra et al. [88], where they employed static analysis to identify queries that will be executed unconditionally by a piece of code. The compiler can then transform the code so that these queries are issued as soon as their query parameters are computed, and before their results are needed. Unfortunately, for many web applications there is not enough computation to perform between the point when the query parameters are available and the query results are used, which reduces the effectiveness of the technique. Also, if the queries are executed conditionally, prefetching queries requires speculation about program execution and can end up issuing additional useless queries.

In contrast to prefetching, most ORM frameworks provide some mechanism to reduce the number of round trips, by allowing the user to specify "fetching strategies" that describe when an object or members of a collection should be fetched from the database. The default strategy is usually "lazy fetching," where each object is fetched from the database only when it is used by the application. This means that there is a round trip for every object, but the only object fetched are those that are certainly used by the application. The alternative "eager" strategy causes the all objects related to an object (e.g., that are part of the same collection, or that are referenced by a foreign key) to be fetched as soon as the object is requested. The eager strategy reduces the number of round trips to the database by combining the queries involved in fetching multiple entities (e.g., using joins). Of course, this eager strategy can sometimes result in objects being fetched that are not needed, and, in some cases, can actually incur *more* round trips than lazy fetching. For this reason, deciding when to label entities as "eager" is a non-trivial task, as evidenced by the number of questions on online forums regarding when to use which strategy, with "it depends" being the most common answer. In addition, for large-scale projects that involve multiple developers, it is difficult for the designer of the data access layer to predict how entities will be accessed in the application and therefore which strategy should be used. Finally, approaches based on fetching strategies are very specific to ORM frameworks and fail to address the general problem which is also present in non-ORM applications. Finally, while code partitioning tools such as PYXIS is effective in reducing round trips, they rely on statically approximation of the amount of communication between the server and query components of the application, such as using profiling. As such, their effectiveness depend on the precision of the analysis, and accuracy of using profiling as a representative of the actual workload.

Rather than using static analysis to determine when to submit queries to the database server (and hence when to receive the results subsequently), or partition code into server and query components, we postpone such decision until application execution time by making use of dynamic analysis. In this chapter, I describe this new approach for reducing the latency of database applications that combines many of the features of the two strategies described earlier. The goal is to reduce the number of round trips to the database by *batching* queries issued by the application. The key idea is to collect queries by relying on a new technique which we call *extended lazy evaluation* (or simply "lazy evaluation" in the rest of the chapter.) As the application executes, queries are batched into a query store instead of being executed right away. In addition, non-database related computation is delayed until it is absolutely necessary. As the application continues to execute, multiple queries are accumulated with the query store. When a value that is derived from query results is finally needed (say, when it is displayed by the client component), then all the queries that are registered with the query store are executed by the database in a single batch, and the results are then used to evaluate the outcome of the computation. The technique is conceptually related to the traditional lazy evaluation as supported by functional languages such as Haskell, Miranda, Scheme and OCaml [70]. In traditional lazy evaluation, there are two classes of computations; those that can be delayed, and those that force the delayed computation to take place because they must be executed eagerly. In our extended lazy evaluation, queries constitute a third kind of computation because even though their actual execution is delayed, they must eagerly register themselves with the batching mechanism so they can be issued together with other queries in the batch.

Compared to query extraction using static analysis, our approach batches queries dynamically as the program executes, and defers computation as long as possible to maximize the opportunity to overlap query execution with program evaluation. As a result, it is able to batch queries across branches and even method calls, which results in larger batch sizes and fewer database round trips. Unlike the approach based on fetching strategies, our approach is not fundamentally tied to ORM frameworks. Moreover, we do not require developers to label entities as eager or lazy, as our system only brings in entities from the database as they are originally requested by the application. Note that our approach is orthogonal to other multi-query optimization approaches that optimize batches of queries [55, 33]; we do not merge queries to improve their performance, or depend on many concurrent users to collect large batches. Instead, we optimize applications to extract batches from a *single* client, and issue those in a single round trip to the database (which still executes the individual query statements.)

We have implemented this approach in a new system called SLOTH. The system is targeted towards applications written in Java that use databases for persistent storage. SLOTH consists of two components: a compiler and a number of libraries for runtime execution. Unlike traditional compilers, SLOTH compiles the application source code to execute using lazy evaluation strategy. In summary, SLOTH makes the following contributions:

- We devise a new mechanism to batch queries in applications that use databases for persistent storage based on a combination of a new "lazy-ifying" compiler and dynamic program analysis to generate the queries to be batched. Our transformation preserves the semantics of the original program, including transaction boundaries.

- We propose a number of optimizations to improve the quality of the compiled lazy code.

- We built and evaluated SLOTH using real-world web applications totaling over 300k lines of code. Our results show that SLOTH achieves a median speedup between $1.3\times$ and $2.2\times$ (depending on network latency), with maximum speedups as high as $3.1\times$. Reducing latency also improves maximum throughput of our applications by $1.5\times$.

In the following, I first describe how SLOTH works through a motivating example in Sec. 4.2.

```
ModelAndView handleRequest(...) {
  Map model = new HashMap<String, Object>();
  Object o = request.getAttribute("patientId");
  if (o != null) {
    Integer patientId = (Integer) o;
    if (!model.containsKey("patient")) {
      if (hasPrivilege(VIEW_PATIENTS)) {
         Patient p = getPatientService().getPatient(patientId);
         model.put("patient", p);
         ...
         model.put("patientEncounters",
                    getEncounterService().getEncountersByPatient(p));
         ...
         List visits = getVisitService().getVisitsByPatient(p);
         CollectionUtils.filter(visits, ...);
         model.put("patientVisits", visits);
         model.put("activeVisits", getVisitService().getActiveVisitsByPatient(p));
         ...
      }
    }

  return new ModelAndView(portletPath, "model", model);
}
```

Figure 4-1: Code fragment abridged from OpenMRS

Then, I conceptually explain our compilation strategy in Sec. 4.3, followed by optimizations to improve generated code quality in Sec. 4.6. Sec. 4.7 then describes our prototype implementation of SLOTH, and I report our experimental results using both benchmark and real-world applications in Sec. 4.8.

4.2 Overview

In this section I give an overview of SLOTH using the code fragment shown in Fig. 4-1. The fragment is abridged from OpenMRS [15], a real-world patient record web application written in Java that uses Spring as the web framework and Hibernate ORM library to manage persistent data. The application has been deployed in numerous countries worldwide since 2006.

The application is structured using the Model-View-Control (MVC) pattern, and the code fragment is part of a controller that builds a model to be displayed by the view after construction. The controller is invoked by the web framework when a user logs-in to the application and tries to view the dashboard for a particular patient. After logging in, the controller first creates a model (a `HashMap` object), populates it with appropriate patient data based on the logged-in user's privileges, and returns the populated model to the web framework. The web framework then passes the partially constructed model to other controllers which may add additional data, and finally to the view creator to generate HTML output.

As written, this code fragment can issue up to four queries; the queries are issued by calls of the form `getXXX` on the data access objects, i.e., the `Service` objects, following the web framework's convention. The first query in Line 8 fetches the `Patient` object that the user is interested in displaying and adds it to the model. The code then issues queries on Lines 12 and 14, and Line 17 to fetch different types of data associated with the patient of interest, and adds this data to the model as well. It is important to observe that of the four round trips that this code can incur, only the first one is essential—without the result of that first query, the other queries cannot be constructed. In fact, the results from the other queries are only stored in the model and not used until the view is actually rendered. This means that in principle, the developer could have collected the last three queries in a single batch and sent it to the database in a single round trip. The developer could have gone even further, collecting in a single batch all the queries involved in building the model until the data from any of the queries in the batch is really needed—either because the model needs to be displayed, or because the data is needed to construct a new query. Manually transforming the code in this way would have a big impact in the number of round trips incurred by the application, but would also impose an unacceptable burden on the developer. In the rest of the section, I describe how SLOTH automates such a transformation with only minimal changes to the original code, and requires no extra work from the developer.

An important ingredient to automatically transform the code to batch queries is *lazy evaluation*. In most traditional programming languages, the evaluation of a statement causes that statement to execute, so any function calls made by that statement are executed before the program proceeds to the evaluating the next statement. In lazy evaluation, by contrast, the evaluation of a statement does not cause the statement to execute; instead, the evaluation produces a *Thunk*: a place-holder that stands for the result of that computation, and it also remembers what the computation was. The only statements that are executed immediately upon evaluation are those that produce output (e.g., printing on the console), or cause an externally visible side effect (e.g., reading from files). When such a statement executes, the thunks corresponding to all the values that flow into that statement will be *forced*, meaning that the delayed computation they represented will finally be executed.

The key idea behind our approach is to modify the basic machinery of lazy evaluation so that when a thunk is created, any queries performed by the statement represented by the thunk are added to a *query store* kept by the runtime to batch queries. Because the computation has been delayed, the results of those queries are not yet needed, so the queries can accumulate in the query store until any thunk that requires the result of such queries is forced; at that point, the entire batch of queries is sent to the database for processing in a single round trip. This process is illustrated in Figure 4-2; during program execution, Line 8 issues a call to fetch the `Patient` object that corresponds to `patientId` (Q1). Rather than executing the query, SLOTH compiles the call to register the query with the query store instead. The query is recorded in the current batch within the store (Batch 1), and a thunk is returned to the program (represented by the gray box in Fig. 4-2). In Line 12, the program needs to access the patient object `p` to generate the queries to fetch the patient's encounters (Q2) followed by visits in Line 14 (Q3). At this point the thunk `p` is forced, Batch 1 is executed, and its results (`rs1`) are recorded in the query cache in the store. A new non-thunk object `p'` is returned to the program upon deserialization from `rs1`, and `p'` is memoized in order to avoid redundant deserializations. After this query is executed, Q2 and Q3 can be generated using `p'` and are registered with the query store in a new batch (Batch 2). Unlike the patient query, however, Q2 and Q3 are not executed within `handleRequest` since their results are not used (thunks are stored in the model map in Lines 12 and 16). Note that even though Line 15 uses the results of Q3 by filtering it, our analysis determines that the operation does not have externally visible side effects and is thus delayed, allowing Batch 2 to remain unexecuted. This leads to batching another query in Line 17 that fetches the patient's active visits (Q4), and the method returns.

Depending on subsequent program path, Batch 2 might be appended with further queries. Q2, Q3, and Q4 may be executed later when the application needs to access the database to get the value from a registered query, or they might not be executed at all if the application has no further need to access the database.

This example shows how SLOTH is able to perform much more batching than either the existing "lazy" fetching mode of Hibernate or prior work using static analysis [88]. Hibernate's lazy fetching mode would have to evaluate the results of the database-accessing statements such as `getVisitsByPatient(p)` on Line 14 as its results are needed by the filtering operation, leaving no opportunity to batch. In contrast, SLOTH places thunks into the model and delays the filtering operation, which avoid evaluating any of the queries. This enables more queries to be batched and executed together in a subsequent trip to the database. Static analysis also cannot perform any batching for these queries, because it cannot determine what queries need to be evaluated at compile time as the queries are parameterized (such as by the specific patient id that is fetched in Line 8), and also because they are executed conditionally only if the logged-in user has the required privilege.

There are some languages such as Haskell that execute lazily by default, but Java (and other languages that are routinely used for developing database applications) has no such support. Furthermore, we want to have significant control over how the lazy evaluation takes place so that we can calibrate the tradeoffs between execution overhead and the degree of batching achieved by the system. We would not have such tight control if we were working under an existing lazy evaluation framework. Instead, we rely on our own SLOTH compiler to transform the code for lazy evaluation. At runtime, the transformed code relies on the SLOTH runtime to maintain the query store. The runtime also includes a custom JDBC driver that allows multiple queries to be issued to the database in a single round trip, as well as extended versions of the web application framework, ORM library, and application server that can process thunks (we currently provided extensions to the Spring application framework, the Hibernate ORM library, and the Tomcat application server, to be described in Sec. 4.7). For monolithic applications that directly use the JDBC driver to interact with the database, developers just need to change such applications to use the SLOTH batch JDBC driver instead. For applications hosted on application servers, developers only need to host

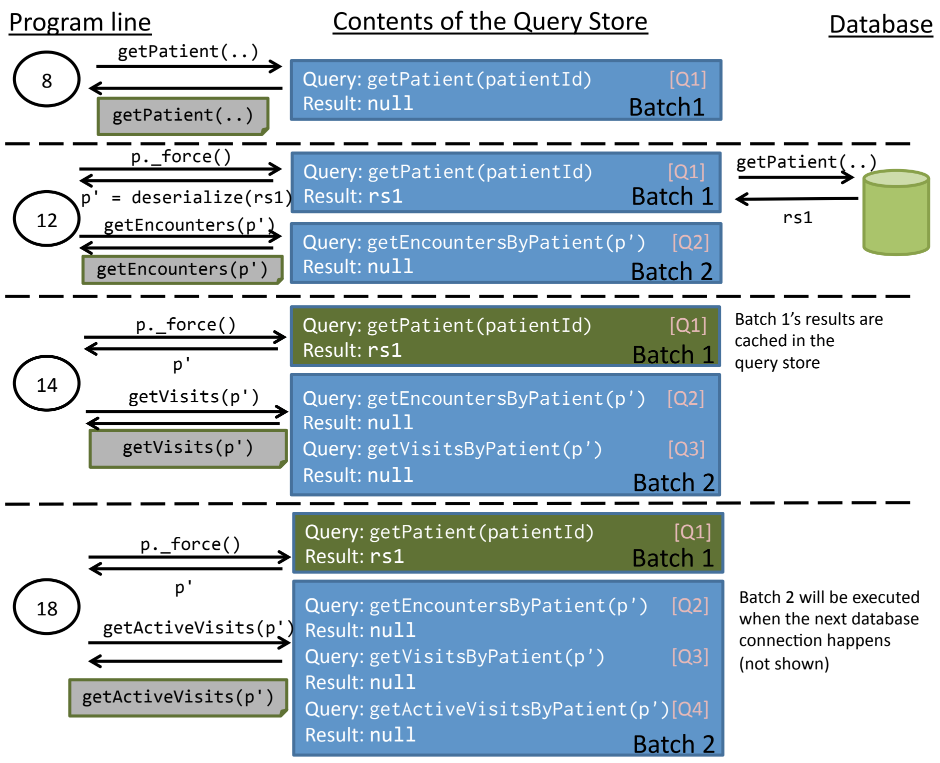

Program line
Contents of the Query Store
Database
8
getPatient(..)
Query: getPatient(patientId) [Q1]
Result: null
Batch1
12
p._force()
p' = deserialize(rs1)
getEncounters(p')
Query: getPatient(patientId) [Q1]
Result: rs1
Batch 1
Query: getEncountersByPatient(p') [Q2]
Result: null
Batch 2
getPatient(..)
rs1
14
p._force()
p'
getVisits(p')
Query: getPatient(patientId) [Q1]
Result: rs1
Batch 1
Batch 1's results are cached in the query store
Query: getEncountersByPatient(p') [Q2]
Result: null
Query: getVisitsByPatient(p') [Q3]
Result: null
Batch 2
18
p._force()
p'
getActiveVisits(p')
Query: getPatient(patientId) [Q1]
Result: rs1
Batch 1
Query: getEncountersByPatient(p') [Q2]
Result: null
Query: getVisitsByPatient(p') [Q3]
Result: null
Query: getActiveVisitsByPatient(p')[Q4]
Result: null
Batch 2
Batch 2 will be executed when the next database connection happens (not shown)

Figure 4-2: Operational diagram of the example code fragment

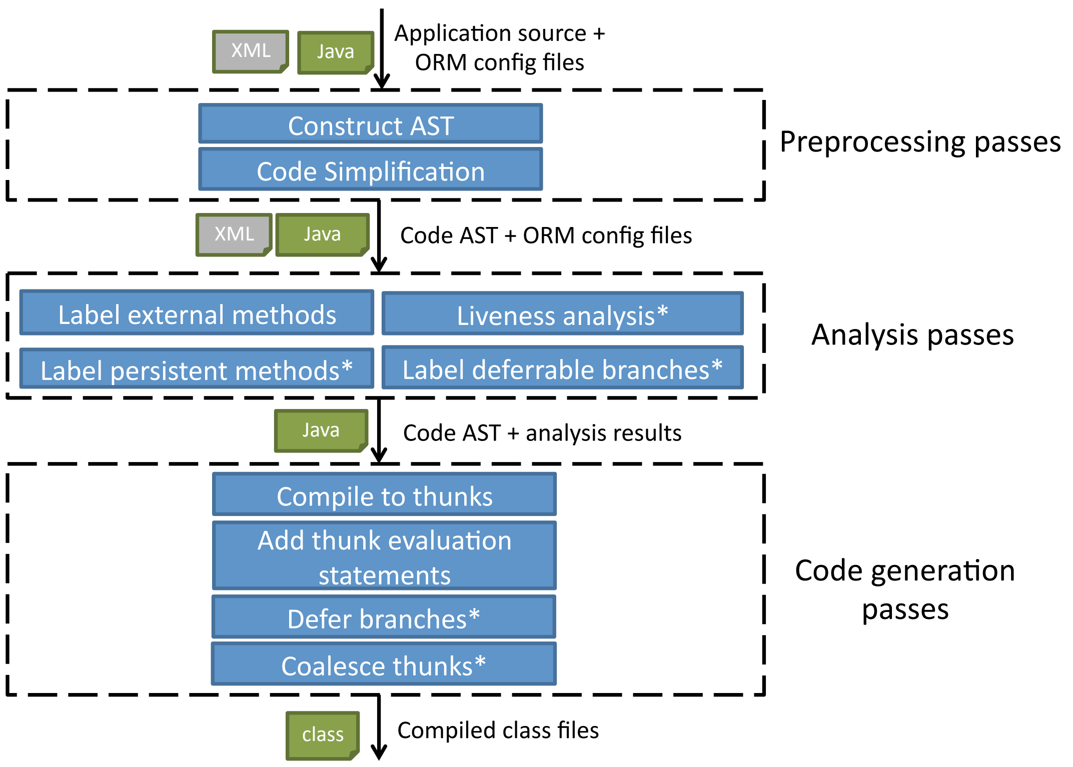


Figure 4-3: Architecture of the SLOTH Compiler, with * marking those components used for optimization

them on the SLOTH extended application server instead after compiling their application with the SLOTH compiler.

4.3 Compiling to Lazy Semantics

In this section I describe the compilation process SLOTH uses to compile application source code to be evaluated lazily. Figure 4-3 shows the overall architecture of the SLOTH compiler, the details of which are described in this section and next.

4.3.1 Code Simplification

The SLOTH compiler is a source-to-source translator that converts code intended to be executed under eager evaluation into lazy evaluation instead. To ease the implementation, SLOTH first per-

$$\begin{array}{rcl}
v \in \text{program variable} & & \\
t \in \text{assign target} & ::= & v \mid v.f \mid v[v] \\
a \in \text{assign value} & ::= & v \mid uop\ v \mid v\ op\ v \mid v.f \mid v[v] \\
s \in \text{statement} & ::= & t = a; \mid \text{if}(v)\ \text{then}\ s_1\ \text{else}\ s_2 \mid \text{while}\{\text{True}\}\{s\} \mid s_1\ ;\ s_2 \mid \text{throw}\ v \mid \\
 & & v\ \text{instanceof}\ C \mid \text{return}\ v; \mid \text{break;} \mid \text{continue;} \mid \text{synchronized}(v)\{s\} \\
uop \in \text{unary op} & ::= & ! \mid - \\
binop \in \text{binary op} & ::= & + \mid - \mid * \mid / \mid \&\& \mid || \mid < \mid =
\end{array}$$

Figure 4-4: Concrete syntax for the simplified input language used by the SLOTH compiler

forms a number of simplification of the input source code. For example, it converts all looping constructs to `while (true)`, where the original loop condition is converted into branches with control flow statements in their bodies, and it breaks down assignment statements to have at most one operation on their right-hand-side. Thus an assignment such as `x = a + b + c` will be translated to `t = a + b; x = t + c;`, with `t` being a temporary variable. The compiler also eliminates type parameters (generics) and extract inner and anonymous classes into stand-alone classes. Figure 4-4 describes the simplified input language that is used by the rest of the SLOTH compiler toolchain.

4.3.2 Thunk Conversion

After simplification, the SLOTH compiler converts each statement of the source code into extended lazy semantics. For clarity, in the following I present the compilation through a series of examples using concrete Java syntax. However, beyond recognizing methods that issue queries (such as those in the JDBC API), our compilation is not Java-specific, and we formalize the compilation process in Sec. 4.4 using an abstract kernel language.

In concrete syntax, each statement in the original program is replaced with an allocation of an anonymous class derived from the abstract `Thunk` class after compilation, with the code for the original statement placed inside a new `_force` class method. To "evaluate" the thunk, we invoke this method, which executes the original program statement and returns the result (if any). For example, the following statement:

```
int x = c + d;
```

is compiled into lazy semantics as:

```
Thunk<Integer> x =
  new Thunk<Integer>() {
    Integer _force() { return c._force() + d._force(); }
  };
```

There are a few points to note in the example. First, all variables are converted into `Thunk` types after compilation. For instance, `x` has type `Thunk<Integer>` after compilation, and likewise for `c` and `d`. As a consequence, all variables need to be evaluated before carrying out the actual computation inside the body of `_force`. Secondly, to avoid redundant evaluations, the runtime memoizes the return value of `_force` so that subsequent calls to `_force` will return the memoized value instead (the details of which are not shown).

While the compilation is relatively straightforward, the mechanism presented above can incur substantial runtime overhead, as the compiled version of each statement incurs allocation of a `Thunk` object, and all computation are converted to method calls. Sec. 4.6 describes several optimizations that we have devised to reduce the overhead. Sec. 4.8.6 quantifies the overhead of lazy semantics, which shows that despite some overhead, it is generally much less than the savings we obtain from reducing round trips.

4.3.3 Compiling Query Calls

Method calls that issue database queries, such as JDBC `executeQuery` calls, and calls to ORM library APIs that retrieve entities from persistent storage are compiled differently from ordinary method calls. In particular, we want to extract the query that would be executed from such calls and record it in the query store so it can be issued when the next batch is sent to the database. To facilitate this, the query store consists of the following components: a) a buffer that stores the current batch of queries to be executed and associates a unique id with each query, and b) a result store that contains the results returned from batches previously sent to the database; the result store is a map from the unique identifier of a query to its result set. The query store API consists of two methods:

- `QueryId registerQuery(String sql)`: Add the `sql` query to the current batch of queries and return a unique identifier to the caller. If `sql` is an `INSERT`, `UPDATE`, `ABORT`, or `COMMIT`, the current batch will be immediately sent to the database to ensure these updates to the database are not left lingering in the query store. On the other hand, the method avoids introducing redundant queries into the batch, so if `sql` matches another query already in the query buffer, the identifier of the first query will be returned.

- `ResultSet getResultSet(QueryId id)`: Check if the result set associated with `id` resides in the result store; if so, return the cached result. Otherwise, issue the current batch of queries in a single round trip, process the result sets by adding them to the result store, and return the result set that corresponds to `id`.

To make use of the query store, method calls that issue database queries are compiled to a thunk that passes the SQL query to be executed in the constructor. The thunk registers the query to be executed with the query store using `registerQuery` in its constructor and stores the returned `QueryId` as its member field. The `_force` method of the thunk then calls `getResultSet` to get the corresponding result set. For ORM library calls, the result sets are passed to the appropriate deserialization methods in order to convert the result set into heap objects that are returned to the caller.

Note that when such thunks are created, it will evaluate any other thunks that are needed to construct the SQL query to register with the query store. For example, consider Line 8 of Fig. 4-1, which makes a call to the database to retrieve a particular patient's data:

```
Patient p = getPatientService().getPatient(patientId);
```

In the lazy version of this fragment, `patientId` is converted to a thunk that is evaluated before the SQL query can be passed to the query store:

```
Thunk<Patient> p = new Thunk<Patient>(patientId) {
  { this.id = queryStore.regQuery(getQuery(patientId._force())); }
  Patient _force() {
    return deserialize(queryStore.getResultSet(id));
  }
}
```

Here, `getQuery` calls an ORM library method to generate the SQL string and substitutes the evaluated `patientId` in it, and `deserialize` reconstructs an object from a SQL result set.

4.3.4 Compiling Method Calls

In the spirit of being as lazy as possible, it would be ideal to delay executing method calls as long as possible (in the best case, the result from the method call might never be needed and therefore we do not need to execute the call). However, method calls might have side effects that change the program heap, for instance changing the values of heap objects that are externally visible outside of the application (such as a global variable). Or the target of the call might be a class external to the application, meaning that our compiler does not have access to its source code such as a method of the standard JDK (otherwise we call such methods "internal"). Because of that, method calls are compiled differently according to the type of the called method as follows.

Internal methods without side effects. This is the ideal situation where we can delay the executing the method call. In this case the call is compiled to a thunk with the method call as the body of the `_force` method. If there is any return value to the method then it gets assigned to the generated thunk. For example, if `int foo(Object x, Object y)` is an internal method with no side effects, then:

```
int r = foo(x, y);
```

is compiled to:

```
Thunk<Integer> r = new Thunk<Integer>(x, y) {
    int _force() { return this.foo(x._force(), y._force()); }
  };
```

Internal methods with externally visible side effects. In this case, we cannot defer the execution of the method due to its externally visible side effects. However, we can still defer the evaluation of its arguments until necessary inside the method body. Thus, the SLOTH compiler generates a special version of the method where its parameters are thunk values, and the original call site will be compiled to calling the special version of the method instead. For example, if `int bar(Object x)` is such a method, then:

```
int r = bar(x);
```

is compiled to:

```
Thunk<Integer> r = bar_thunk(x);
```

with the declaration of `bar_thunk` as `Thunk<Integer> bar_thunk(Thunk<Object> x)`.

External methods. We cannot defer the execution of external methods unless we know that they are side effect free. Since we do not have access to their source code, the SLOTH compiler does not change the original method call during compilation, except for forcing the arguments and receiver objects as needed. As an example, Line 3 in Fig. 4-1:

```
Object o = request.getAttribute("patientId");
```

is compiled to:

```
Thunk<Object> o = new LiteralThunk(request._force().getAttribute("patientId"));
```

As discussed earlier, since the types of all variables are converted to thunks, the (non-thunk) return value of external method calls are stored in `LiteralThunk` objects that simply returns the non-thunk value when `_force` is called, as shown in the example above.

Method labeling is done as part of the analysis passes in the SLOTH compiler, and the thunk conversion pass makes use of this information when method calls are encountered during compilation.

4.3.5 Class Definitions and Heap Operations

For classes that are defined by the application, the SLOTH compiler changes its field and method definitions. First, the type of each member field is changed to `Thunk` in order to facilitate accesses to field values under lazy evaluation. For each publicly accessible final field, the compiler adds an extra field with the original type, with its value set to the evaluated result of the corresponding thunk-ified version of the field. These fields are created in case they are accessed from external methods. Publicly accessible non-final fields cannot be made lazy.

In addition, for each method, the SLOTH compiler changes the type of each parameter to `Thunk` objects in order to facilitate method call conventions discussed in Sec. 4.3.4. Like public fields, since public methods can potentially be invoked by external code (e.g., the web framework that hosts the application, or by JDK methods such as calling `equals` while searching for a key within a `HashMap` object), the SLOTH compiler generates a "dummy" method that has the same declaration (in terms of method name and parameter types) as the original method, The body of such dummy methods simply invokes the thunk-converted version of the corresponding method. If the method has a return value, then it is evaluated on exit. For instance, the following method:

```
public Foo bar (Baz b) { ... }
```

is compiled to two methods by the SLOTH compiler:

```
// to be called from internal code
public Thunk<Foo> bar_thunk (Thunk<Baz> b) { ... }
// to be called from external code
public Foo bar (Baz b) { return bar_thunk(new LiteralThunk(b))._force(); }
```

With that in mind, the compiler translates reads of object fields to simply return the thunk fields. However, updates to heap objects are not delayed in order to ensure that subsequent heap reads are consistent with the semantics of the original program. In order to carry out the write, however, the receiver object needs to be evaluated if it is a thunk. Thus statements such as:

```
Foo obj = ...
obj.field = x;
```

are compiled to:

```
Thunk<Foo> obj = ...
obj._force().field = x;
```

Notice that while the *target* of the heap write is evaluated (`obj` in the example), the *value* that is written (`x` in the example) can be a thunk object, meaning that it can represent computation that has not been evaluated yet.

4.3.6 Evaluating Thunks

In the previous sections, I discussed the basic compilation of statements into lazy semantics using thunks. In this section I describe when thunks are evaluated, i.e., when the original computation that they represent is actually carried out, some of which has already been discussed.

As mentioned in the last section, the target object in field reads and writes are evaluated when encountered. However, the value of the field and the object that is written to the field can still be thunks. The same is applied to array accesses and writes, where the target array and index are evaluated before the operation.

In method calls where the execution is not delayed, as discussed in Sec. 4.3.4, the target object is evaluated prior to the call if the called method is non-static. While our compiler could have deferred the evaluation of the target object by converting all member methods into static class methods, it is likely that the body of such methods (or further methods that are invoked inside the body) accesses some fields of the target object and hence will evaluate the target object. Thus there is unlikely to be significant savings in delaying such evaluation. Also, when calling external methods all parameters are evaluated as discussed.

In the basic compiler, all branch conditions are evaluated when `if` statements are encountered. Recall that all loops are canonicalized into `while (true)` loops with the loop condition rewritten using branches. I present an optimization to this restriction in Sec. 4.6.2 below. Similarly, statements that throw exceptions, obtain locks on objects (`synchronized`), and that spawn new threads of control are not deferred, in order to preserve the control flow of the original program. Finally, thunk evaluations can also happen when compiling statements that issue queries, as discussed in Sec. 4.3.3.

4.3.7 Limitations

There are two limitations that the SLOTH compiler does not currently handle. First, because of delayed execution, exceptions that are thrown by the original program might not occur at the same program point in the compiled version. For instance, the original program might throw an exception in a method call, but in the SLOTH-compiled version, the call might be deferred until later when the thunk corresponding to the call is evaluated. While the exception will still be thrown eventually, the SLOTH-compiled program might have executed more code than the original program before hitting the exception.

Second, since the SLOTH compiler changes the representation of member fields in each internal class, it currently does not support custom deserializers. For instance, one of the applications used in our experiments reads in an XML file that contains the contents of an object before the application source code is compiled by SLOTH. As a result, the compiled application fails to re-create the object as its representation has changed. We manually fixed the XML file to match the expected types in our benchmark. In general, we do not expect this to be common practice, given that Java already provides its own object serialization routines.

4.4 Formal Semantics

I now formally define the extended lazy evaluation outlined by the compilation process described above. For the sake of presentation, I describe the semantics in terms of an abstract language shown in Fig. 4-5. The language is simple and is based on the concrete syntax shown in Fig. 4-4, but will help us illustrate the main principles behind extended lazy evaluation in a way that can be easily applied not just to Java but to any other object-oriented languages.

In the abstract language, we model unstructured control flow statements such as `break` and `continue` using Boolean variables to indicate if control flow has been altered. Such unstructured control flow statements are then translated into Boolean variable assignments, and the statements that can be affected by unstructured control flow (e.g., statements that follow a `break` statement inside a loop) are wrapped into a conditional block guarded by the appropriate indicator Boolean

$$
\begin{array}{rcl}
c \in \mathsf{constant} & ::= & \mathsf{True} \mid \mathsf{False} \mid \mathsf{literal} \\
e_l \in \mathsf{assignExpr} & ::= & x \mid e.f \\
e \in \mathsf{expr} & ::= & c \mid e_l \mid \{f_i = e_i\} \mid e_1 \; op \; e_2 \mid \neg \, e \mid f(e) \mid e_a[e_i] \mid R(e) \\
c \in \mathsf{command} & ::= & \mathsf{skip} \mid e_l := e \mid \mathsf{if}(e) \; \mathsf{then} \; c_1 \; \mathsf{else} \; c_2 \mid \\
 & & \mathsf{while}(\mathsf{True}) \; \mathsf{do} \; c \mid W(e) \mid c_1 \; ; \; c_2 \\
op \in \mathsf{binary\ op} & ::= & \wedge \mid \vee \mid > \mid < \mid =
\end{array}
$$

Figure 4-5: Abstract language for formalizing lazy evaluation used in SLOTH

variable(s).

In addition, the language models the return value of each method using the special variable `@`. Statements such as return e are translated into two variable assignments: one that assigns to `@`, and another that assigns to an indicator Boolean variable to represent the fact that control flow has been altered. Statements that follow the original return e statement are wrapped into conditional blocks similar to that discussed above for loops.

Other than that, the main constructs to highlight in the language are the expression $R(e)$ which issues a database *read* query derived from the value of expression e, and $W(e)$ a statement that issues a query that can mutate the database, such as `INSERT` or `UPDATE`.

4.4.1 Program Model

We model the execution model as a tuple $\langle D, \sigma, h \rangle$ for programs written in the abstract language and executed under standard evaluation, and $\langle Q, D, \sigma, h \rangle$ for those to be executed under extended lazy evaluation. to model the program state under standard and extended lazy evaluation. In particular:

- $D : e \to e$ represents the database. It is modeled as a map that takes in an expression representing the SQL query, and returns another expression, which is either a single value (e.g., the value of an aggregate) or an array of expressions (e.g., a set of records stored in the database).

- $\sigma : e \rightarrow e$ represents the environment that maps program variables to expressions. It is used to look up variable values during program evaluation.

- $h : e \rightarrow e$ represents the program heap that maps a heap location to the expression stored at that location. We use the notation $h[e]$ to represent looking up value stored at location e, and $h[e_1, e_2]$ to denote looking up the value stored at heap location e_1 with offset e_2 (e.g., a field dereference), and the same notation is used to denote array value lookups as well.

- $Q : id \rightarrow e$ represents the query store under extended lazy semantics. It maps unique query identifiers (as discussed in Sec. 4.3.3) to the query expressions that were registered with the query store when the respective identifier was initially created.

In addition, we define the auxillary function `update` : $D \times e \rightarrow D$ to model database modifications (e.g., as a result of executing an `UPDATE` query). It takes in a database D and a write query e, and returns a new database. Since the language semantics are not concerned with the details of database modifications, we model such operations as a pure function that returns a new database state.

4.4.2 Standard Evaluation Semantics

Given the above, we first define the semantics of standard evaluation for each of the program constructs in the abstract language. The evaluation rules are shown below, where the $\emptyset$ symbol below represents the null value.

Semantics of expressions:

$$\frac{}{[\![\langle D, \sigma, h\rangle, x]\!] \rightarrow \langle D, \sigma, h\rangle, \sigma[x]} \quad \text{[Variable]}$$

$$\frac{[\![\langle D, \sigma, h\rangle, e]\!] \rightarrow \langle D', \sigma, h'\rangle, v}{[\![\langle D, \sigma, h\rangle, e.f]\!] \rightarrow \langle D', \sigma, h'\rangle, h'[v, f]} \quad \text{[Field dereference]}$$

$$\frac{}{[\![\langle D, \sigma, h\rangle, c]\!] \rightarrow \langle D, \sigma, h\rangle, c} \quad \text{[Constant]}$$

$$\frac{h' = h[v \to \{f_i = \emptyset\}],\ v \text{ is a fresh location}}{[\![\langle D, \sigma, h\rangle, \{f_i = e_i\}]\!] \to \langle D, \sigma, h'\rangle, v} \quad \text{[Object allocation]}$$

$$\frac{[\![\langle D, \sigma, h\rangle, e_1]\!] \to \langle D', \sigma, h'\rangle, v_1 \quad [\![\langle D', \sigma, h\rangle, e_2]\!] \to \langle D'', \sigma, h''\rangle, v_2 \quad v_1\ op\ v_2 = v}{[\![\langle D, \sigma, h\rangle, e_1\ op\ e_2]\!] \to \langle D'', \sigma, h''\rangle, v} \quad \text{[Binary]}$$

$$\frac{[\![\langle D, \sigma, h\rangle, e]\!] \to \langle D', \sigma, h'\rangle, v_1 \quad uop\ v_1 = v}{[\![\langle D, \sigma, h\rangle, uop\ e]\!] \to \langle D', \sigma, h'\rangle, v} \quad \text{[Unary]}$$

$$\frac{[\![\langle D, \sigma, h\rangle, e]\!] \to \langle D', \sigma, h'\rangle, v \quad [\![\langle D', [x \to v], h'\rangle, s]\!] \to \langle D'', \sigma'', h''\rangle \quad s \text{ is the body of } f(x)}{[\![\langle D, \sigma, h\rangle, f(e)]\!] \to \langle D'', \sigma, h''\rangle, \sigma[@]} \quad \text{[Method]}$$

$$\frac{[\![\langle D, \sigma, h\rangle, e_i]\!] \to \langle D', \sigma, h'\rangle, v_i \quad [\![\langle D', \sigma, h'\rangle, e_a]\!] \to \langle D'', \sigma, h''\rangle, v_a}{[\![\langle D, \sigma, h\rangle, e_a[e_i]]\!] \to \langle D'', \sigma, h''\rangle, h''[v_a, v_i]} \quad \text{[Array deference]}$$

$$\frac{[\![\langle D, \sigma, h\rangle, e]\!] \to \langle D', \sigma, h'\rangle, v}{[\![\langle D, \sigma, h\rangle, R(e)]\!] \to \langle D', \sigma, h'\rangle, D'[v]} \quad \text{[Read query]}$$

Semantics of statements:

$$\frac{}{[\![\langle D, \sigma, h\rangle, \mathsf{skip}]\!] \to \langle D, \sigma, h\rangle} \quad \text{[Skip]}$$

$$\frac{[\![\langle D, \sigma, h\rangle, e]\!] \to \langle D', \sigma, h'\rangle, v \quad [\![\langle D', \sigma, h'\rangle, e_l]\!] \to \langle D'', \sigma, h''\rangle, v_l}{[\![\langle D, \sigma, h\rangle, e_l := e]\!] \to \langle D'', \sigma[v_l \to v], h''\rangle} \quad \text{[Assignment]}$$

$$\frac{[\![\langle D, \sigma, h\rangle, e]\!] \to \langle D', \sigma, h', \mathsf{True}\rangle \quad [\![\langle D', \sigma, h'\rangle, s_1]\!] \to \langle D'', \sigma', h''\rangle}{[\![\langle D, \sigma, h\rangle, \mathsf{if}(e)\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2]\!] \to \langle D'', \sigma', h''\rangle} \quad \text{[Conditional–true]}$$

$$\frac{[\![\langle D, \sigma, h\rangle, e]\!] \to \langle D', \sigma, h', \mathsf{False}\rangle \quad [\![\langle D', \sigma, h'\rangle, s_2]\!] \to \langle D'', \sigma', h''\rangle}{[\![\langle D, \sigma, h\rangle, \mathsf{if}(e)\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2]\!] \to \langle D'', \sigma', h''\rangle} \quad \text{[Conditional–false]}$$

$$\frac{[\![\langle D, \sigma, h\rangle, s]\!] \to \langle D', \sigma', h'\rangle}{[\![\langle D, \sigma, h\rangle, \mathsf{while(True)\ do}\ s]\!] \to \langle D', \sigma', h'\rangle} \quad \text{[Loop]}$$

$$\frac{[\![\langle D, \sigma, h\rangle, e]\!] \to \langle D', \sigma, h'\rangle, v \quad \mathsf{update}(D', v) = D''}{[\![\langle D, \sigma, h\rangle, W(e)]\!] \to \langle D'', \sigma, h'\rangle} \quad \text{[Write query]}$$

$$\frac{[\![\langle D, \sigma, h\rangle, s_1]\!] \to \langle D', \sigma', h'\rangle \quad [\![\langle D', \sigma', h'\rangle, s_2]\!] \to \langle D'', \sigma'', h''\rangle}{[\![\langle D, \sigma, h\rangle, s_1\ ;\ s_2]\!] \to \langle D'', \sigma'', h''\rangle} \quad \text{[Sequence]}$$

Each of the evaluation rules describes the result of evaluating a program construct in the language (shown below the line), given the intermediate steps that are taken during evaluation (shown above the line). The rules for standard evaluation are typical of imperative languages. In general, each of the rules for expressions returns a (possibly changed) state, along with the result of the evaluation.

As an example, the rule to evaluate the binary expression $e_1\ op\ e_2$ is repeated below.

$$\frac{\langle s, e_1\rangle \to \langle s', v_1\rangle \quad \langle s', e_2\rangle \to \langle s'', v_2\rangle \quad v_1\ op\ v_2 \to v}{\langle s, e_1\ op\ e_2\rangle \to \langle s'', v\rangle}$$

The notation above the line describes how the subexpressions e_1 and e_2 are evaluated to values v_1 and v_2 respectively. The result of evaluating the overall expression is shown below the line and it is the result of applying op to v_1 and v_2, together with the state as transformed by the evaluation of the two subexpressions.

As another example, the evaluation of a read query $R(e)$ must first evaluate the query e to a query string v, and then return the result of consulting the database D' with this query string. Note that the evaluation of e might itself modify the database, for example if e involves a function call that internally issues an update query, so the query v must execute on the database as it is left after the evaluation of e:

$$\frac{\langle (D, \sigma, h), e\rangle \rightarrow \langle (D', \sigma, h'), v\rangle}{\langle (D, \sigma, h), R(e)\rangle \rightarrow \langle (D', \sigma, h'), D'[v]\rangle}$$

Meanwhile, the rules for evaluating statements return only a new program state and no values. Note that to evaluate a write query $W(e)$, we use the update function to perform the change on the database after evaluating the query expression. The changed database is then included as part of the modified state.

4.4.3 Semantics of Extended Lazy Evaluation

To describe lazy evaluation, we augment the state tuple s with the query store Q, which maps a query identifier to a pair (q, rs) that represents the SQL query q and its corresponding result set rs. rs is initially set to null ($\emptyset$) when the pair is created. We model thunks using the pair (σ, e), where σ represents the environment for looking up variables during thunk evaluation, and e is the expression to evaluate. In our Java implementation the environment is implemented as fields in the `Thunk` class and e is the expression in the body of the `_force` method.

The evaluation rules for extended lazy evaluation are shown below.

Semantics of expressions:

$$\frac{}{[\![\langle Q, D, \sigma, h\rangle, x]\!] \rightarrow \langle Q, D, \sigma, h\rangle, ([x \rightarrow \sigma[x]], x)} \quad \text{[Variable]}$$

$$\frac{[\![\langle Q, D, \sigma, h\rangle, e]\!] \rightarrow \langle Q', D', \sigma, h'\rangle, (\sigma', e) \quad \text{force}(Q, D, (\sigma', e)) \rightarrow v}{[\![\langle Q, D, \sigma, h\rangle, e.f]\!] \rightarrow \langle Q', D', \sigma, h'\rangle, h'[v, f]} \quad \text{[Field deference]}$$

$$\frac{}{[\![\langle Q, D, \sigma, h\rangle, c]\!] \rightarrow \langle Q, D, \sigma, h\rangle, ([\,], c)} \quad \text{[Constant]}$$

$$\frac{h' = h[v \rightarrow \{f_i = \emptyset\}],\ v \text{ is a fresh location}}{[\![\langle Q, D, \sigma, h\rangle, \{f_i = e_i\}]\!] \rightarrow \langle Q, D, \sigma, h'\rangle, ([\,], v)} \quad \text{[Object allocation]}$$

$$\frac{\begin{array}{c}[\![\langle Q, D, \sigma, h\rangle, e_1]\!] \rightarrow \langle Q', D', \sigma, h'\rangle, (\sigma', e_1) \\ [\![\langle Q', D', \sigma, h'\rangle, e_2]\!] \rightarrow \langle Q'', D'', \sigma, h''\rangle, (\sigma'', e_2)\end{array}}{[\![\langle Q, D, \sigma, h\rangle, e_1\ op\ e_2]\!] \rightarrow \langle Q'', D'', \sigma, h''\rangle, (\sigma' \cup \sigma'', e_1\ op\ e_2)} \quad \text{[Binary]}$$

$$\frac{[\![\langle Q, D, \sigma, h\rangle, e]\!] \rightarrow \langle Q', D', \sigma, h'\rangle, (\sigma, e)}{[\![\langle Q, D, \sigma, h\rangle, uop\ e]\!] \rightarrow \langle Q', D', \sigma, h'\rangle, (\sigma, uop\ e)} \quad \text{[Unary]}$$

$$\frac{[\![\langle Q, D, \sigma, h\rangle, e]\!] \rightarrow \langle Q', D', \sigma, h'\rangle, (\sigma', e)}{[\![\langle Q, D, \sigma, h\rangle, f(e)]\!] \rightarrow \langle Q', D', \sigma, h'\rangle, ([x \rightarrow (\sigma', e)], f(x))} \quad \text{[Method–internal \& pure]}$$

$$\frac{\begin{array}{c}[\![\langle Q, D, \sigma, h\rangle, e]\!] \rightarrow \langle Q', D', \sigma, h'\rangle, (\sigma', e) \\ [\![\langle Q', D', [x \rightarrow (\sigma', e)], h'\rangle, s]\!] \rightarrow \langle Q'', D'', \sigma'', h''\rangle \\ s \text{ is the body of } f(x)\end{array}}{[\![\langle Q, D, \sigma, h\rangle, f(e)]\!] \rightarrow \langle Q'', D'', \sigma'', h''\rangle, \sigma''[@]} \quad \text{[Method–internal \& impure]}$$

$$\frac{\begin{array}{c}[\![\langle Q, D, \sigma, h\rangle, e]\!] \rightarrow \langle Q', D', \sigma, h'\rangle, (\sigma', e) \\ [\![\langle Q', D', [x \rightarrow (\sigma', e)], h'\rangle, s]\!] \rightarrow \langle Q''', D''', \sigma'', h''\rangle \\ \mathsf{force}(Q', D', (\sigma', e)) \rightarrow Q'', D'', v \\ s \text{ is the body of } f(x)\end{array}}{[\![\langle Q, D, \sigma, h\rangle, f(e)]\!] \rightarrow \langle Q''', D''', \sigma'', h''\rangle, \sigma''[@]} \quad \text{[Method–external]}$$

$$\frac{\begin{array}{c}[\![\langle Q, D, \sigma, h\rangle, e_i]\!] \rightarrow \langle Q', D', \sigma, h'\rangle, (\sigma', e_i) \\ [\![\langle Q', D', \sigma, h'\rangle, e_a]\!] \rightarrow \langle Q'', D'', \sigma, h''\rangle, (\sigma'', e_a) \\ \mathsf{force}(Q'', D'', (\sigma', e_i)) \rightarrow Q''', D''', v_i \\ \mathsf{force}(Q''', D''', (\sigma'', e_a)) \rightarrow Q'''', D'''', v_a\end{array}}{[\![\langle Q, D, \sigma, h\rangle, e_a[e_i]]\!] \rightarrow \langle Q'''', D'''', \sigma, h''\rangle, h''[v_a, v_i]} \quad \text{[Array deference]}$$

$$\frac{\begin{array}{c}[\![\langle Q, D, \sigma, h\rangle, e]\!] \rightarrow \langle Q', D', \sigma, h'\rangle, (\sigma', e) \quad Q''' = Q''[id \rightarrow (v, \emptyset)] \\ \mathsf{force}(Q', D', (\sigma', e)) \rightarrow Q'', D'', v \qquad id \text{ is a fresh identifier}\end{array}}{[\![\langle Q, D, \sigma, h\rangle, R(e)]\!] \rightarrow \langle Q''', D'', \sigma, h'\rangle, ([\,], id)} \quad \text{[Read query]}$$

Semantics of statements:

$$\frac{}{[\![\langle Q, D, \sigma, h\rangle, \mathsf{skip}]\!] \rightarrow \langle Q, D, \sigma, h\rangle} \quad \text{[Skip]}$$

$$\frac{[\![\langle Q, D, \sigma, h\rangle, e]\!] \rightarrow \langle Q', D', \sigma, h'\rangle, (\sigma', e) \quad [\![\langle Q', D', \sigma, h'\rangle, e_l]\!] \rightarrow \langle Q'', D'', \sigma, h''\rangle, v_l}{[\![\langle Q, D, \sigma, h\rangle, e_l := e]\!] \rightarrow \langle Q'', D'', \sigma[v_l \rightarrow (\sigma', e)], h''\rangle} \quad \text{[Assignment]}$$

$$\frac{\begin{array}{c}[\![\langle Q, D, \sigma, h\rangle, e]\!] \rightarrow \langle Q', D', \sigma, h'\rangle, (\sigma', e) \quad \mathsf{force}(Q', D', (\sigma', e)) \rightarrow Q'', D'', \mathsf{True} \\ [\![\langle Q'', D'', \sigma, h'\rangle, s_1]\!] \rightarrow \langle Q''', D'', \sigma', h''\rangle\end{array}}{[\![\langle Q, D, \sigma, h\rangle, \mathsf{if}(e) \mathsf{\ then\ } s_1 \mathsf{\ else\ } s_2]\!] \rightarrow \langle Q''', D''', \sigma', h''\rangle} \quad \text{[Conditional–true]}$$

$$\frac{\begin{array}{c}[\![\langle Q, D, \sigma, h\rangle, e]\!] \rightarrow \langle Q', D', \sigma, h'\rangle, (\sigma', e) \quad \mathsf{force}(Q', D', (\sigma', e)) \rightarrow Q'', D'', \mathsf{False} \\ [\![\langle Q'', D'', \sigma, h'\rangle, s_2]\!] \rightarrow \langle Q''', D'', \sigma', h''\rangle\end{array}}{[\![\langle Q, D, \sigma, h\rangle, \mathsf{if}(e) \mathsf{\ then\ } s_1 \mathsf{\ else\ } s_2]\!] \rightarrow \langle Q''', D''', \sigma', h''\rangle} \quad \text{[Conditional–false]}$$

$$\frac{[\![\langle Q, D, \sigma, h\rangle, s]\!] \rightarrow \langle Q', D', \sigma', h'\rangle}{[\![\langle Q, D, \sigma, h\rangle, \mathsf{while(True)\ do\ } s]\!] \rightarrow \langle Q', D', \sigma', h'\rangle} \quad \text{[Loop]}$$

$$\frac{\begin{array}{c}[\![\langle Q, D, \sigma, h\rangle, e]\!] \rightarrow \langle Q', D', \sigma, h'\rangle, (\sigma', e) \\ \mathsf{force}(Q', D', (\sigma', e)) \rightarrow Q'', D'', v \\ \mathsf{update}(D'', v) \rightarrow D''' \\ \forall id \in Q'' \,.\, Q'''[id] = \begin{cases} D'''[Q''[id].s] & \text{if } Q''[id].rs = \emptyset \\ Q''[id].rs & \text{otherwise} \end{cases}\end{array}}{[\![\langle Q, D, \sigma, h\rangle, W(e)]\!] \rightarrow \langle Q''', D''', \sigma, h'\rangle} \quad \text{[Write query]}$$

$$\frac{[\![\langle Q, D, \sigma, h\rangle, s_1]\!] \rightarrow \langle Q', D', \sigma', h'\rangle \quad [\![\langle Q, D', \sigma', h'\rangle, s_2]\!] \rightarrow \langle Q'', D'', \sigma'', h''\rangle}{[\![\langle Q, D, \sigma, h\rangle, s_1 \ ; \ s_2]\!] \rightarrow \langle Q'', D'', \sigma'', h''\rangle} \quad \text{[Sequence]}$$

As discussed in Sec. 4.3.2, to evaluate the expression $e_1 \ op \ e_2$ using lazy evaluation, we first create thunk objects v_1 and v_2 for e_1 and e_2 respectively, and then create another thunk object that represents the op. As shown in the above, this is formally described as:

$$\frac{\begin{array}{c}\langle s, e_1\rangle \rightarrow \langle s', v_1\rangle \quad \langle s', e_2\rangle \rightarrow \langle s'', v_2\rangle \\ v_1 = (s', e_1') \qquad v_2 = (s'', e_2') \\ v = (\sigma' \cup \sigma'', e_1' \ op \ e_2')\end{array}}{\langle s, e_1 \ op \ e_2\rangle \rightarrow \langle s'', v\rangle}$$

Note that the environment for v is the union of the environments from v_1 and v_2 since we might need to look up variables stored in either of them.

On the other hand, as discussed in Sec. 4.3.3, under lazy evaluation query calls are evaluated by first forcing the evaluation of the thunk that corresponds to the query string, and then registering the query with the query store.

For instance, in the rule for evaluating read queries ($\mathsf{R(e)}$), the force function is used to evaluate thunks, similar to that described in the examples above using Java. $\mathsf{force}(Q, D, t)$ takes in the current database D and query store Q and returns the evaluated thunk along with the modified query store and database. When force encounters an id in a thunk, it will check the query store to see if that id already has a result associated with it. If it does not, it will issue as a batch all the queries in the query store that do not yet have results associated with them, and will then assign those results once they arrive from the database.

Likewise, in the case of writes, we first evaluate the query expression thunk. Then, before executing the actual write query, we first execute all read queries that have been delayed and not executed in the query store due to extended lazy evaluation. This is shown in the evaluation rule by issuing queries to the database for all the query identifiers that have not been executed, and changing the contents of the query store as a result. After that, we execute the write query on the database, and use the update function to change to the database contents as in standard evaluation.

As discussed in Sec. 4.3.4, method calls are evaluated using three different rules based on the kind of method that is invoked. For methods that are internal and pure, we first evaluate the actual parameters of the call (by creating thunks to wrap each of the actuals), and then create another thunk with the delayed expression being the call itself. Meanwhile, since we cannot delay calls to external methods or methods that have side-effects, we evaluate such calls by first evaluating the parameters and then either calling the specialized method that takes thunk parameters (if the method is internal), or evaluating the parameter thunks and calling the external method directly.

Next, we define the force function that is used to evaluate thunks:

$$\frac{}{\mathsf{force}(Q, D, ([x \rightarrow \sigma[x]], x)) \rightarrow Q, D, \sigma[x]} \quad \text{[Variable]}$$

$$\frac{}{\mathsf{force}(Q, D, (\sigma, c)) \rightarrow Q, D, c} \quad \text{[Constant]}$$

$$\frac{\mathsf{force}(Q, D, (\sigma, e_1)) \rightarrow Q', D', v_1 \quad \mathsf{force}(Q', D', (\sigma, e_2)) \rightarrow Q'', D'', v_2 \quad v_1 \; op \; v_2 = v}{\mathsf{force}(Q, D, (\sigma, e_1 \; op \; e_2)) \rightarrow Q'', D'', v} \quad \text{[Binary]}$$

$$\frac{\mathsf{force}(Q, D, (\sigma, e)) \rightarrow Q', D', v_1 \quad uop \; v_1 \rightarrow v}{\mathsf{force}(Q, D, (\sigma, uop \; e)) \rightarrow Q', D', v} \quad \text{[Unary]}$$

$$\frac{\begin{array}{c}\mathsf{force}(Q, D, (\sigma, e)) \rightarrow Q', D', v \\ [\![\langle Q', D', [x \rightarrow v], h\rangle, s]\!] \rightarrow \langle Q'', D'', \sigma', h\rangle \\ s \text{ is the body of } f(x)\end{array}}{\mathsf{force}(Q, D, ([x \rightarrow (\sigma, e)], f(x)) \rightarrow Q'', D'', \sigma'[@]} \quad \text{[Method–internal \& pure]}$$

$$\frac{Q[id].rs \neq \emptyset}{\mathsf{force}(Q, D, (\sigma, id)) \rightarrow Q, D, Q[id].rs} \quad \text{[Issued query]}$$

$$\frac{Q[id].rs = \emptyset \quad \forall id \in Q \;.\; Q'[id] = \begin{cases} D[Q[id].s] & \text{if } Q[id].rs = \emptyset \\ Q[id].rs & \text{otherwise} \end{cases}}{force(Q, D, ([\,], id)) \to Q', D, Q'[id].rs} \quad \text{[Unissued query]}$$

The evaluation rules above show what happens when thunks are evaluated. The rules are defined based on the on the type of expression that is delayed. For instance, to evaluate a thunk with a variable expression, we simply look up the value of the variable from the environment that is embedded in the thunk. For thunks that contain method calls, we first use force to evaluate each of the parameters, then evaluate the method body itself as in standard evaluation. Note that since we only create thunks for pure method calls, the heap remains unchanged after the method returns. Finally, for read queries, force either returns the result set that is stored in the query store if the corresponding query has already been executed (as described in the rule [Issued query]), or issues all the unissued queries in the store as a batch before returning the results (as described in the rule [Unissued query]).

4.5 Soundness of Extended Lazy Evaluation

Given the formal language described above, we now show that extended lazy semantics preserves the semantics of the original program that is executed under standard evaluation, except for the limitations such as exceptions that are described in Sec. 4.3.7. We show that fact using the following theorem:

Theorem 3 (Semantic Correspondence). *Given a program p and initial states Q_0, D_0, σ_0 and h_0. If*

$[\![\langle D_0, \sigma_0, h_0\rangle, p]\!] \to \langle D_S, \sigma_S, h_S\rangle$ *under standard semantics, and*

$[\![\langle Q_0, D_0, \sigma_0, h_0\rangle, p]\!] \to \langle Q_E, D_E, \sigma_E, h_E\rangle$ *under extended lazy semantics, then*

$\forall x \in \sigma_S \;.\; \sigma_S[x] = \text{force}(Q_E, D_E, \sigma_E[x])$ *and* $\forall x \in h_S \;.\; h_S[x] = \text{force}(Q_E, D_E, h_E[x])$.

For presentation purposes, we use $P(S_S, S_L)$ to denote that the state $S_S = \langle D_S, \sigma_S, h_S \rangle$ under standard evaluation and the state $S_L = \langle Q_L, D_L, \sigma_L, h_L \rangle$ under extended lazy evaluation satisfy the correspondence property above. We use structural induction on the input language to prove the theorem. Since most of the proof is mechanical, we do not include the entire proof. Instead we highlight a few representative cases below.

Constants. The rules for evaluating constants do not change the state under both standard and extended lazy evaluation. Thus, the the correspondence property is satisfied.

Variables. Using the evaluation rules, suppose $[\![S_S, x]\!] \to S'_S, \sigma[x]$, and $[\![S_L, x]\!] \to S'_L, ([x \to \sigma[x]], x)$. From the induction hypothesis, assume that $P(S_S, S_L)$ and $P(S'_S, S'_L)$ are true. Given that, we need to show that $P(S'_S, S''_L)$ is true, where S''_L is the state that results from evaluating the thunk $([x \to \sigma[x]], x)$. This is obvious since the extended lazy evaulation rule for variables change neither the database, environment, or the heap. Furthermore, the definition of force shows that evaluating the thunk $([x \to \sigma[x]], x)$ returns $\sigma[x]$ as in standard evalation, hence proving the validity of $P(S'_S, S''_L)$.

Unary expressions. Like the case above, suppose $[\![S_S, uop\ e]\!] \to S'_S, v_S$ under standard evaluation, and $[\![S_L, uop\ e]\!] \to S'_L, (\sigma, uop\ e)$ under extended lazy evaluation. Let S''_L be the state that results from evaluating the thunk $(\sigma, uop\ e)$. From the induction hypothesis, we assume that $P(S_S, S_L)$ and $P(S'_S, S'_L)$ are true. We need to show that $P(S'_S, S''_L)$ is true. First, from the definition of force, we know that evaluating the thunk $(\sigma, uop\ e)$ results in the same value as v_S. Next, we need to show that $D''_L = D'_S$ as a result of evaluating the thunk. Note that there are three possible scenarios that can happen as a result of evaluating $(\sigma, uop\ e)$. First, if e does not contain a query then obviously $D''_L = D'_S$. Next, if e contains a query, then it is either a write query or a read query. If it is a write query, then the query is executed immediately as in standard evaluation, thus $D''_L = D'_S$. Otherwise, it is a read query. Since read queries do not change the state of the database, hence $D''_L = D'_S$ as well.

Binary expressions. Binary expressions of the form $e_1\ op\ e_2$ are similar to unary expressions, except that we need to consider the case when both expressions contain queries and the effects on the state of the database when they are evaluated (otherwise it is the same situation as unary expressions). For binary expressions, the extended lazy evaluation rule and the definition of force prescribe that e_1 is first evaluated prior to e_2. If e_1 contains a write query, then it would already been executed during lazy evaluation, since write queries are not deferred. e_2 will be evaluated using the database as a result of evaluating e_1, and thus evaluated the same way as in standard evaluation. The situation is similar if e_1 contains a read query and e_2 contains a write query. On the other hand, if both e_1 and e_2 contain only read queries, then evaluating them do not change the state of the database, and the correspondence property is satisfied.

Read queries. If the query has previously been evaluated, then the cached result is returned and no change is induced on the program state, and the property is satisfied. Otherwise, it is a new read query and a new query identifier is created as a result of extended lazy evaluation. Evaluating the query identifier using force will execute the read query against the database. Since the database state is not changed as a result of read queries, the correspondence property is preserved.

Method calls. Calls to methods that are either external or internal with side-effects are not deferred under extended lazy evaluation and are thus equivalent to standard evaluation. For the pure function calls that are deferred, the definition of force for evaluating such thunks is exactly the same as those for evaluating method calls under standard evaluation (except for the evaluation of thunk parameters), and thus does not alter program semantics.

Field accesses, array dereferences, and object allocations. Since extended lazy evaluation does not defer evaluation of these kinds of expressions, the correspondence property is satisfied.

Statements. Since evaluation of statements are not deferred under extended lazy evaluation, the correspondence property for statements is satisfied as it follows from the proof for expressions.

4.6 Being Even Lazier

In the previous sections, I described how SLOTH compiles source code into lazy semantics. However, as noted in Sec. 4.3.2, there can be substantial overhead if we follow the compilation procedure naively. In this section, I describe three optimizations we developed. The goal of these optimizations is to generate more efficient code and to further defer computation. As discussed in Sec. 4.2, deferring computation delays thunk evaluations, which in turn increases the chances of achieving larger query batch sizes during execution time. Like the previous section, I describe the optimizations using concrete Java syntax for clarity, although they can all be formalized using the language described in Fig. 4-5.

4.6.1 Selective Compilation

The goal of compiling to lazy semantics is to enable query batching dynamically as the application executes. Obviously the benefits are observable only for the parts of the application that actually issue queries, and simply adds runtime overhead for the remaining parts of the application. Thus, the SLOTH compiler analyzes each method to determine whether it can possibly access the database. The analysis is a conservative one that labels a method as using persistent data if it:

- Issues a query in its method body.
- Calls another method that uses persistent data. Because of dynamic dispatch, if the called method is overridden by any of its subclasses, SLOTH checks if any of the overridden versions is persistent, and if so it labels the call to be persistent.
- Accesses object fields that are stored persistently. This is done by examining the static object types that are used in each method, and checking whether it uses an object whose type is persistently stored. The latter is determined by checking for classes that are populated by query result sets in its constructor (in the case of JDBC), or by examining the object mapping configuration files for ORM frameworks.

The analysis is implemented as an inter-procedural, flow-insensitive dataflow analysis [72]. It first identifies the set of methods m containing statements that perform any of the above. Then, any method that calls m is labeled as persistent. This process continues until all methods that can possibly be persistent are labeled. For methods that are not labeled as persistent, the SLOTH compiler does not convert their bodies into lazy semantics — they are compiled as is. For the two applications used in our experiments, our results show about 17% and 21% of the methods do not use persistent data, and those are mainly methods that handle application configuration and output page formatting (see Sec. 4.8.5 for details).

4.6.2 Deferring Control Flow Evaluations

In the naive compiler, all branch conditions are evaluated when an `if` statement is encountered, as discussed in Sec. 4.3.6. The rationale is that the outcome of the branch affects subsequent program path. We can do better, however, based on the intuition that if neither branch of the condition creates any changes to the program state that are externally visible outside of the application, then the entire branch statement can be deferred as a thunk like other simple statements. Formally, if none of the statements within the branch contains: i) calls that issue queries; or ii) thunk evaluations as discussed in Sec. 4.3.6 (recall that thunks need to be evaluated when their values are needed in operations that cannot be deferred, such as making changes to the program state that are externally visible), then the entire branch statement can be deferred. For instance, in the following code fragment:

```
if (c) {
  a = b;
}
else {
  a = d;
}
```

The naive compiler would compile the code fragment into:

```
if (c._force()) {
  a = new Thunk0(b) { ... };
}
else {
  a = new Thunk1(d) { ... };
}
```

which could result in queries being executed as a result of evaluating c. However, since the bodies of the branch statements do not make any externally visible state changes, the whole branch statement can be deferred as:

```
ThunkBlock tb = new ThunkBlock2(b, d) {
  void _force () {
    if (c._force()) { a = b._force(); }
    else { a = d._force(); }
  }
};
Thunk<int> a = tb.a();
```

where the evaluation of c is further delayed. The ThunkBlock class is similar to the Thunk class except that it defines methods (not shown above) that return thunk variables defined within the block, such as a in the example. Calling _force on any of the thunk outputs from a thunk block will evaluate the entire block, along with all other output objects that are associated with that thunk block. In sum, this optimization allows us to delay thunk evaluations, which in turn might increase query batches sizes.

To implement this optimization, the SLOTH compiler first iterates through the body of the if statement to determine if any thunk evaluation takes place, and all branches that are deferrable are labeled. During thunk generation, deferrable branches are translated to thunk objects, with the original statements inside the branches constituting the body of the _force methods. Variables defined inside the branch are assigned to output thunks as described above. The same optimization is applied to defer loops as well. Recall from Sec. 4.3.1 that all loops are converted into while (true) loops with embedded control flow statements (break and continue) inside their bodies. Using similar logic, a loop can be deferred if all statements inside the loop body can be deferred.

4.6.3 Coalescing Thunks

The compilation approach described in Sec. 4.3.2 results in new Thunk objects being created for each computation that is delayed. Due to the temporary variables that are introduced as a result of code simplification, the number of operations (and thus the number of Thunk objects) can be much larger than the number of lines of Java code. This can substantially slow down the compiled application due to object creation and garbage collection. As an optimization, the thunk coalescing pass merges consecutive statements into thunk blocks to avoid allocation of thunks. The idea is that if there are two consecutive statements s_1 and s_2, and that s_1 defines a variable v that is used in s_2 and not anywhere after in the program, then we can combine s_1 and s_2 into a thunk block with s_1 and s_2 inside its _force method (provided that both statements can be deferred as discussed in Sec. 4.3). This way, SLOTH avoids creating the thunk object for v that would be created under basic compilation. As an illustrative example, consider the following code fragment:

```
int foo (int a, int b, int c, int d) {
  int e = a + b;
  int f = e + c;
  int g = f + d;
  return g;
}
```

Using the compilation procedure discussed earlier, the code fragment will compile to:

```
1 Thunk<int> foo (Thunk<int> a, b, c, d) {
2   Thunk<int> e = new Thunk0(a, b) { ... }
3   Thunk<int> f = new Thunk1(e, c) { ... }
4   Thunk<int> g = new Thunk2(f, d) { ... }
5   return g;
6 }
```

Notice that three thunk objects are created, with the additions in the original code performed in the _force methods inside the definitions of classes Thunk0, Thunk1 and Thunk2, respectively. However, in this case the variables e and f are not used anywhere, i.e., they are no longer *live*, after Line 4. Thus SLOTH can combine the first three statements into a single thunk, resulting in the following:

```
Thunk<int> foo (Thunk<int> a, b, c, d) {
  ThunkBlock tb = new ThunkBlock3(a, b, c, d) { ... }
  Thunk<int> g = tb.g();
  return g;
}
```

The optimized version reduces the number of object allocations from 3 to 2: one allocation for ThunkBlock3 and another one for the Thunk object representing g that is created within the thunk block. In this case, the _force method inside the ThunkBlock3 class consists of statements that perform the addition in the original code. As described earlier, the thunk block keeps track of all thunk values that need to be output, in this case the variable g.

This optimization is implemented in multiple steps in the SLOTH compiler. First, SLOTH identifies variables that are live at each program statement. Live variables are computed using a dataflow analysis that iterates through program statements in a backwards manner to determine the variables that are used at each program statement (and therefore must be live).

After thunks are generated, the compiler then iterates through each method and attempts to combine consecutive statements into thunk blocks. The process continues until no statements can be further combined within each method. After that, the compiler examines the _force method of each thunk block and records the set of variables that are defined. For each such variable v, the compiler checks to see if all statements that use of v are also included in the same thunk block by making use of the liveness information. If that is the case, then it does not need to create a thunk object for v. This optimization significantly reduces the number of thunk objects that need to be allocated and thus improves the efficiency of the generated code as shown in Sec. 4.8.5.

4.7 Implementation

We have implemented a prototype of SLOTH. The SLOTH compiler is built using Polyglot [84]. We have implemented a query store for the thunk objects to register and retrieve query results. To issue the batched queries in a single round trip, we extended the MySQL JDBC driver to allow executing multiple queries in one `executeQuery` call, and the query store uses the batch query driver to issue queries to the database. Once received by the database, our extended driver executes all read queries in parallel. In addition, we have also made the following changes to the application hosting framework to enable them process thunk objects that are returned by the hosted application. Our extensions are not language specific and can be applied to other ORM and web hosting frameworks as well.

JPA Extensions. We extended the Java Persistence API (JPA) [8] to allow returning thunk objects from calls that retrieve objects from the database. For example, the JPA defines a method `Object find(Class, id)` that fetches the persistently stored object of type `Class` with `id` as its object identifier. We extended the API with a new method `Thunk<Object> find_thunk(Class, id)` that performs the same functionality except that it returns a thunk rather than the requested object. We implemented this method in the Hibernate ORM library. The implementation first generates a SQL query that would be issued to fetch the object from the database, registers the query with the query store, and returns a thunk object to the caller. Invoking the `_force` method on the returned thunk object forces the query to be executed, and Hibernate will then deserialize the result into an object of the requested type before returning to the caller. Similar extensions are made to other JPA methods. Note that our extensions are targeted to the JPA not Hibernate — we decided to implement them within the Hibernate ORM framework as it is a popular open-source implementation of JPA and is also used by the applications in our experiments.

Spring Extensions. We extended the Spring web application framework to allow thunk objects be stored and returned during model construction within the MVC pattern, as OpenMRS is hosted with Spring and implements the MVC pattern. This is a minor change the consists of about 100 lines of code.

JSP API Extensions. We extended the JavaServer Pages (JSP) API [7] to enable thunk operations. In particular, we allow thunk objects to be returned while evaluating the JSP Expression Language on the model that is constructed by the web application framework. We also extended the `JspWriter` class from the JSP API that generates the output HTML page when a JSP is requested.

The class provides methods to write different types of objects to the output stream. We extended the class with a `writeThunk` method that writes thunk objects to the output stream. `writeThunk` stores the thunk to be written in a buffer, and thunks in the buffer are not evaluated until the writer is flushed by the web server (which typically happens when the entire HTML page is generated). We have implemented our JSP API extensions in Tomcat, which is a popular open-source implementation of the JSP API. This is also a minor change that consists of about 200 lines of code.

4.8 Experiments

In this section I report our experimental results. The goals of these experiments are: i) evaluate the effectiveness of SLOTH at batching queries, ii) quantify the change in application load times, and iii) measure the overhead of running applications using lazy evaluation. All experiments were performed using Hibernate 3.6.5, Spring 3.0.7, and Tomcat 6 with the extensions mentioned above. The web server and applications were hosted on a machine with 8GB of RAM and 2.8GHz processor, and data was stored in an unmodified MySQL 5.5 database with 47GB of RAM and 12 2.4GHz processors. Unless stated there was a 0.5ms round trip delay between the two machines (this is the latency of the group cluster machines). We used the following applications for our experiments:

- itracker version 3.1.5 [5]: itracker is an open-source software issue management system. The system consists of a Java web application built on top of the Apache Struts framework and uses Hibernate to manage storage. The project has 10 contributors with 814 Java source files with a total of 99k lines of Java code.

- OpenMRS version 1.9.1 [15]: OpenMRS is an open-source medical record system that has been deployed in numerous countries. The system consists of a Java web application built on top of the Spring web framework and uses Hibernate to manage storage. The project has over 70 contributors. The version used consists of 1326 Java source files with a total of 226k lines of Java code. The system has been in active development since 2004 and the code illustrates various coding styles for interacting with the ORM.

We created benchmarks from the two applications by first manually examining the source code to locate all web page files (html and jsp files). Next, we analyzed the application to find the URLs that load each of the web pages. This resulted in 38 benchmarks for itracker, and 112 benchmarks for OpenMRS. Each benchmark was run by loading the extracted URL from the application server via a client that resides on the same machine as the application server.

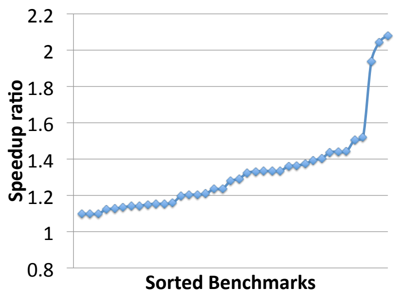
We also tested with TPC-C and TPC-W coded in Java [22]. Because the implementations display the query results immediately after issuing them, there is no opportunity for batching. We only use them to measure the runtime overhead of lazy evaluation as described below.

4.8.1 Page Load Experiments

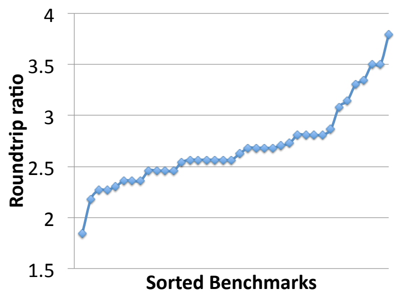
In the first set of experiments, we compared the time taken to load each benchmark from the original and the SLOTH compiled versions of the applications. For each benchmark, we started the web and database servers and measured the time taken to load the entire page. Each measurement was the average of 5 runs. For benchmarks that require user inputs (e.g., patient ID for the patient dashboard, project ID for the list of issues to be displayed), we filled the forms with valid values from the database. We restarted the database and web servers after each measurement to clear all cached objects. For OpenMRS, we used the sample database (2GB in size) provided by the application. For itracker, we created an artificial database (0.7GB in size) consisting of 10 projects and 20 users. Each project has 50 tracked issues, and none of the issues has attachments. We did not define custom scripts or components for the projects that we created. We also created larger versions of these databases (up to 25 GB) and report their performance on selected benchmarks in Sec. 4.8.4, showing that our gains continue to be achievable with much larger database sizes.

We loaded all benchmarks with the applications hosted on the unmodified web framework and application server, and repeated with the SLOTH compiled applications hosted on the SLOTH extended web framework using the ORM library and web server discussed in Sec. 4.7. For all benchmarks, we computed the speedup ratios as:

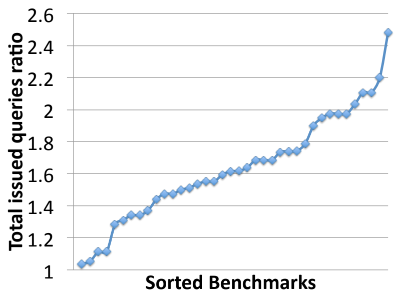
$$\frac{\text{load time of the original application}}{\text{load time of the S}\textsc{loth}\text{ compiled application}}$$



(a) load time ratios CDF

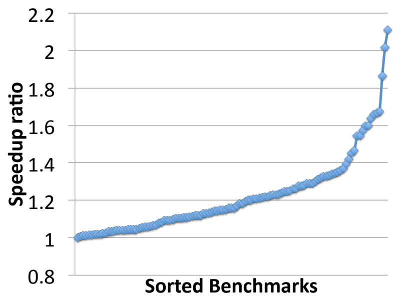


(b) round trip ratios CDF

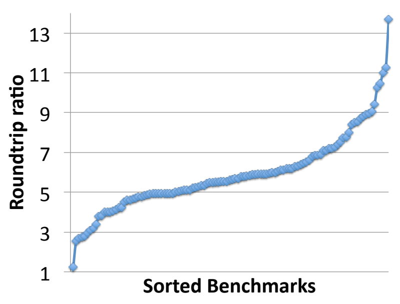


(c) queries issued ratios CDF

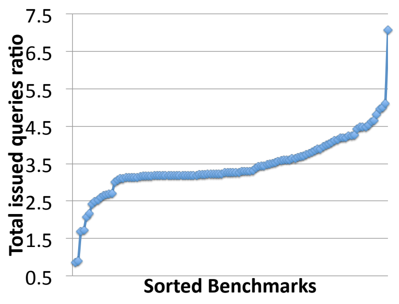
Figure 4-6: itracker benchmark experiment results



(a) load time ratios CDF



(b) round trip ratios CDF



(c) queries issued ratios CDF

Figure 4-7: OpenMRS benchmark experiment results

Benchmark	**Original application**		**SLOTH compiled application**			
	Time (ms)	# round trips	Time (ms)	# round trips	Max # queries in batches	Total queries issued
module-reports/list_reports.jsp	22199	74	15929	24	9	38
self_register.jsp	22776	59	16741	23	5	41
portalhome.jsp	22435	59	14888	25	10	56
module-searchissues/ search_issues_form.jsp	22608	59	16938	21	6	44
forgot_password.jsp	22968	59	15927	24	6	33
error.jsp	21492	59	15752	22	5	40
unauthorized.jsp	21266	59	17588	21	8	28
module-projects/move_issue.jsp	21655	59	18994	32	5	57
module-projects/list_projects.jsp	22390	59	15588	26	4	38
module-projects/ view_issue_activity.jsp	23001	76	17240	33	11	47
module-projects/view_issue.jsp	22676	85	10892	39	20	62
module-projects/edit_issue.jsp	22868	129	11181	34	5	52
module-projects/create_issue.jsp	22606	76	18898	23	14	40
module-projects/list_issues.jsp	22424	59	19492	25	4	40
module-admin/admin_report/ list_reports.jsp	21880	59	19481	23	5	35
module-admin/admin_report/ edit_report.jsp	21178	59	18354	22	6	46
module-admin/admin_configuration/ import_data_verify.jsp	22150	59	17163	22	4	38
module-admin/admin_configuration/ edit_configuration.jsp	22242	59	18486	24	3	28
module-admin/admin_configuration/ import_data.jsp	21719	59	16977	23	15	37
module-admin/admin_configuration/ list_configuration.jsp	21807	59	19100	25	16	53
module-admin/admin_workflow/ list_workflow.jsp	22140	59	19510	23	4	34
module-admin/admin_workflow/ edit_workflowscript.jsp	22221	59	18453	22	3	29
module-admin/admin_user/ edit_user.jsp	25297	59	23043	24	4	36

module-admin/admin_user/ list_users.jsp	23181	59	15245	24	10	53
module-admin/unauthorized.jsp	22619	59	16964	23	5	39
module-admin/admin_project/ edit_project.jsp	23934	59	17057	21	5	44
module-admin/admin_project/ edit_projectscript.jsp	22467	59	20480	26	6	35
module-admin/admin_project/ edit_component.jsp	21966	59	16596	21	7	45
module-admin/admin_project/ edit_version.jsp	22577	59	15666	23	6	35
module-admin/admin_project/ list_projects.jsp	23222	63	17445	22	4	39
module-admin/admin_attachment/ list_attachments.jsp	22370	63	18109	24	8	41
module-admin/admin_scheduler/ list_tasks.jsp	22282	61	16231	24	6	35
module-admin/adminhome.jsp	22138	73	11423	27	4	42
module-admin/admin_language/ list_languages.jsp	22585	77	19597	22	6	39
module-admin/admin_language/ create_language_key.jsp	22217	77	19174	23	5	39
module-admin/admin_language/ edit_language.jsp	23347	66	20747	21	5	30
module-preferences/ edit_preferences.jsp	22891	77	20845	22	4	39
module-help/show_help.jsp	22985	60	18598	22	6	40

Figure 4-8: iTracker benchmark timings

Benchmark	Original application		SLOTH compiled application			
	Time (ms)	# round trips	Time (ms)	# round trips	Max # queries in batches	Total queries issued
dictionary/conceptForm.jsp	2811	183	2439	36	3	38
dictionary/conceptStatsForm.jsp	5418	100	5213	82	16	112
dictionary/concept.jsp	1778	92	1626	36	3	38
optionsForm.jsp	1882	93	1196	19	5	30
help.jsp	1326	67	1089	21	2	27
admin/provider/providerAttributeTypeList.jsp	1988	102	1792	15	6	28
admin/provider/providerAttributeTypeForm.jsp	2000	88	1826	14	6	27
admin/provider/index.jsp	1845	99	1523	17	6	27
admin/provider/providerForm.jsp	1925	124	1893	11	5	25
admin/concepts/conceptSetDerivedForm.jsp	2055	89	1958	13	3	28
admin/concepts/conceptClassForm.jsp	2038	89	1724	12	4	28
admin/concepts/conceptReferenceTermForm.jsp	2252	120	2202	26	4	33
admin/concepts/conceptDatatypeList.jsp	2109	91	2071	24	3	29
admin/concepts/conceptMapTypeList.jsp	1997	119	1918	21	3	28
admin/concepts/conceptDatatypeForm.jsp	2178	148	1930	26	2	32
admin/concepts/conceptIndexForm.jsp	2072	119	1867	14	3	28
admin/concepts/conceptProposalList.jsp	2033	115	1920	16	3	29
admin/concepts/conceptDrugList.jsp	1933	102	1823	21	3	29
admin/concepts/proposeConceptForm.jsp	2406	89	1940	18	3	28
admin/concepts/conceptClassList.jsp	2072	91	1860	17	4	29
admin/concepts/conceptDrugForm.jsp	2535	133	2056	21	4	36
admin/concepts/conceptStopWordForm.jsp	1989	89	1803	18	5	28
admin/concepts/conceptProposalForm.jsp	2651	89	2103	18	6	28
admin/concepts/conceptSourceList.jsp	1897	94	1838	17	5	30
admin/concepts/conceptSourceForm.jsp	2215	92	2103	15	5	29

admin/concepts/ conceptReferenceTerms.jsp	2565	143	2030	16	6	28
admin/concepts/conceptStopWordList.jsp	2560	92	1939	19	6	29
admin/visits/visitTypeList.jsp	2220	89	1356	18	6	28
admin/visits/visitAttributeTypeForm.jsp	1865	88	1125	22	5	27
admin/visits/visitTypeForm.jsp	2304	88	1141	22	5	27
admin/visits/configureVisits.jsp	2214	100	1716	21	5	25
admin/visits/visitForm.jsp	2043	140	1805	18	6	28
admin/visits/visitAttributeTypeList.jsp	2125	109	1523	15	6	28
admin/patients/shortPatientForm.jsp	2552	136	1742	33	9	54
admin/patients/patientForm.jsp	4402	222	2641	34	10	54
admin/patients/mergePatientsForm.jsp	2845	149	2457	21	20	47
admin/patients/ patientIdentifierTypeForm.jsp	2269	118	1859	21	11	33
admin/patients/ patientIdentifierTypeList.jsp	1847	91	1773	18	8	29
admin/modules/ modulePropertiesForm.jsp	2033	89	1550	18	6	28
admin/modules/moduleList.jsp	2488	87	1819	15	5	27
admin/hl7/hl7SourceList.jsp	2599	89	1681	15	5	27
admin/hl7/hl7OnHoldList.jsp	2485	101	1829	14	5	27
admin/hl7/hl7InQueueList.jsp	2365	93	1779	15	5	27
admin/hl7/hl7InArchiveList.jsp	2273	93	1761	15	5	27
admin/hl7/hl7SourceForm.jsp	2053	87	1697	15	4	27
admin/hl7/hl7InArchiveMigration.jsp	2250	90	1698	15	5	25
admin/hl7/hl7InErrorList.jsp	2272	100	1894	17	4	27
admin/forms/addFormResource.jsp	1386	50	1065	18	8	58
admin/forms/formList.jsp	2167	89	1761	16	6	28
admin/forms/formResources.jsp	1320	50	1300	17	7	29
admin/forms/formEditForm.jsp	2966	219	1855	16	7	31
admin/forms/fieldTypeList.jsp	2082	89	1743	15	6	28
admin/forms/fieldTypeForm.jsp	1978	87	1894	17	7	27
admin/forms/fieldForm.jsp	2495	115	1845	18	10	33
admin/index.jsp	2782	91	2429	19	4	29

admin/orders/orderForm.jsp	2578	89	1919	21	4	28
admin/orders/orderList.jsp	2246	99	1966	21	4	33
admin/orders/orderTypeList.jsp	2077	89	1995	16	5	28
admin/orders/orderDrugList.jsp	1970	116	1233	15	8	34
admin/orders/orderTypeForm.jsp	1962	87	1774	17	6	27
admin/orders/orderDrugForm.jsp	2713	124	2296	21	6	30
admin/programs/programList.jsp	2013	89	1977	15	5	28
admin/programs/programForm.jsp	2248	121	1757	20	7	27
admin/programs/conversionForm.jsp	2318	89	1817	21	7	29
admin/programs/conversionList.jsp	1786	89	1689	13	11	28
admin/encounters/encounterRoleList.jsp	2034	97	2013	15	8	25
admin/encounters/encounterForm.jsp	2449	172	1587	19	30	52
admin/encounters/encounterTypeForm.jsp	2128	89	1855	16	5	27
admin/encounters/ encounterTypeList.jsp	1954	107	1803	20	5	28
admin/encounters/ encounterRoleForm.jsp	2076	87	1811	17	8	25
admin/observations/obsForm.jsp	2399	131	2397	28	11	49
admin/observations/personObsForm.jsp	3911	230	3126	86	25	137
admin/locations/hierarchy.jsp	2319	172	2121	43	11	64
admin/locations/ locationAttributeType.jsp	2344	88	1758	16	6	27
admin/locations/ locationAttributeTypes.jsp	1821	90	1766	18	6	28
admin/locations/addressTemplate.jsp	1939	90	1672	15	5	28
admin/locations/locationForm.jsp	2423	138	2129	36	5	52
admin/locations/locationTagEdit.jsp	2497	166	2274	41	5	51
admin/locations/locationList.jsp	2170	127	1924	41	6	47
admin/locations/locationTag.jsp	2152	139	1726	21	5	31
admin/scheduler/schedulerForm.jsp	1902	95	1710	18	5	27
admin/scheduler/schedulerList.jsp	2224	106	1989	18	6	28
admin/maintenance/ implementationIdForm.jsp	1979	124	1912	18	4	28
admin/maintenance/serverLog.jsp	1884	104	1628	17	4	27
admin/maintenance/localesAndThemes.jsp	2085	93	2033	17	4	30

admin/maintenance/currentUsers.jsp	1990	87	1799	16	5	27
admin/maintenance/settings.jsp	2179	92	2161	20	5	29
admin/maintenance/systemInfo.jsp	1902	90	1762	20	5	28
admin/maintenance/quickReport.jsp	2109	101	2021	17	3	28
admin/maintenance/globalPropsForm.jsp	3406	89	3042	17	4	28
admin/maintenance/ databaseChangesInfo.jsp	12555	88	6732	15	8	27
admin/person/addPerson.jsp	1870	89	1843	18	5	28
admin/person/relationshipTypeList.jsp	2170	89	1806	10	5	28
admin/person/relationshipTypeForm.jsp	2222	121	1808	11	5	27
admin/person/ relationshipTypeViewForm.jsp	1953	113	1825	11	7	28
admin/person/personForm.jsp	3154	149	1883	21	8	32
admin/person/ personAttributeTypeForm.jsp	1854	89	1834	18	6	27
admin/person/ personAttributeTypeList.jsp	2149	104	2021	19	6	33
admin/users/roleList.jsp	1892	113	1567	20	7	27
admin/users/privilegeList.jsp	2974	89	2051	18	8	27
admin/users/userForm.jsp	2443	126	1821	15	12	31
admin/users/users.jsp	2003	113	1921	15	11	27
admin/users/roleForm.jsp	2523	115	2060	11	13	27
admin/users/changePasswordForm.jsp	1481	105	1325	17	6	31
admin/users/alertForm.jsp	2595	113	1826	12	12	27
admin/users/privilegeForm.jsp	1905	87	1824	10	5	27
admin/users/alertList.jsp	12413	1705	9621	623	16	824
patientDashboardForm.jsp	7617	494	3610	95	15	190
encounters/encounterDisplay.jsp	9648	878	8242	260	68	406
forgotPasswordForm.jsp	1439	96	1128	12	12	27
feedback.jsp	1399	97	1121	11	5	27
personDashboardForm.jsp	2390	145	1965	17	3	32

Figure 4-9: OpenMRS benchmark timings

Figure 4-6(a) and Fig. 4-7(a) show the CDF of the results where we sorted the benchmarks according to their speedups for presentation purposes (and similarly for other experiments). The actual timings are shown in Fig. 4-8 and Fig. 4-9.

The results show that the SLOTH compiled applications loaded the benchmarks faster compared to the original applications, achieving up to 2.08× (median 1.27×) faster load times for itracker and 2.1× (median 1.15×) faster load times for OpenMRS. Figure 4-6(b) and Fig. 4-7(b) show the ratio of the number of round trips to the database, computed as:

$$\frac{\text{\# of database round trips in original application}}{\text{\# database round trips in SLOTH version of the application}}$$

For itracker, the minimum number of round trip reductions was 27 (out of 59 round trips) while the maximum reduction was 95 (out of 124 original round trips). For OpenMRS, the minimum number of reductions was 18 (out of 100 round trips) and the maximum number was 1082 (out of 1705 round trips). For both applications, SLOTH reduced the number of round trips required to load each of the benchmarks. Although these may seem like large numbers of round trips for a single web page, issues such as the $1 + N$ issue in Hibernate [23] make it quite common for developers to write apps that issue hundreds of queries to generate a web page in widely used ORM frameworks.

Finally, Fig. 4-6(c) and Fig. 4-7(c) show the CDF of the ratio of the total number of queries issued for the two applications. In OpenMRS, the SLOTH compiled application batched as many as 68 queries into a single batch. SLOTH was able to batch multiple queries in all benchmarks, even though the original applications already make extensive use of the eager and lazy fetching strategies provided by Hibernate. This illustrates the effectiveness of applying lazy evaluation in improving performance. Examining the generated query batches, we attribute the performance speedup to the following:

Avoiding unnecessary queries. For all benchmarks, the SLOTH compiled applications issued fewer total number of queries as compared to the original. The reduction is due to the developers' use of eager fetching to load entities in the original applications. Eager fetching incurs extra round

```
1  if (Context.isAuthenticated()) {
2    FormService fs = Context.getFormService();
3    for (Obs o : encounter.getObsAtTopLevel(true)) {
4      FormField ff = fs.getFormField(form, o.getConcept(),..);
5      ...
6      obsMapToReturn.put(ff, list);
7    }
8  }
9  map.put("obsMap", obsMapToReturn);
10 return map;
```

Figure 4-10: Code used to load patient observations

trips to the database to fetch the entities and increases query execution time, and is wasteful if the fetched entities were not ultimately used as our results show. As noted in Sec. 4.1, it is very difficult for developers to decide when to load objects eagerly during development. Using SLOTH, on the other hand, frees the developer from making such decisions while improving application performance.

Batching queries. The SLOTH compiled applications batched a significant number of queries together. For example, one of the OpenMRS benchmarks (`encounterDisplay.jsp`) loads observations about a patient's visit. Observations include height, blood pressure, etc, and there were about 50 observations on average fetched for each patient. Loading is done as shown in the code in Fig. 4-10: i) all observations are first retrieved from the database (Line 3); ii) each observation is iterated over and its corresponding `Concept` object (i.e., the textual explanation of the observed value) is fetched and stored into a `FormField` object (Line 4). The `FormField` object is then put into the model (Line 9) similar to that shown in Fig. 4-1. The model is returned at the end of the method and the fetched concepts are displayed in the view.

In the original application, the concept entities are lazily fetched by the ORM during view generation, and each fetch incurs a round trip to the database. It is difficult to statically analyze the code to extract the queries that would be executed in presence of the authentication check on Line 1, and traditional query batching techniques are inapplicable as the application does not issue subsequent queries until the results from the outstanding query are returned. On the other hand, since the fetched concepts are not used in the method, the SLOTH compiled application batches all the concept queries and issues them in a single batch along with others. This results in a dramatic reduction in the number of round trips and an overall reduction of 1.17$\times$ in page load time.

Finally, there are a few benchmarks where the SLOTH compiled application issued *more* queries than the original, as shown in Fig. 4-7(c) This is because the SLOTH compiled application registers queries to the query store whenever they are encountered during execution, and all registered queries are executed when a thunk that requires data to be fetched is subsequently evaluated. However, not all fetched data are used. The original application, with its use of lazy fetching, avoided issuing those queries and that results in fewer queries executed. In sum, while the SLOTH compiled application does not necessarily issue the minimal number of queries required to load each page, our results show that the benefits in reducing database round trips outweigh the costs of executing a few extra queries.

4.8.2 Throughput Experiments

Next, we compared the throughput of the SLOTH compiled applications and the original. We fixed the number of browser clients, and each client repeatedly loaded pages from OpenMRS for 10 minutes (clients wait til the previous load completes, and then load a new page.) As we did not have a workload, the pages were chosen at random from the list of benchmarks described earlier. We changed the number of clients in each run, and measured the resulting total throughput across all clients. The results (averaged across 5 runs) are shown in Fig. 4-11.

The results show that the application compiled with SLOTH has better throughput than the original, reaching about 1.5$\times$ the peak throughput of the original application. This is expected as the SLOTH version takes less time to load each page. Interestingly, the SLOTH version achieves its peak throughput at a lower number of clients compared to the original. This is because given our experiment setup, both the database and the web server were under-utilized when the number of clients is low and throughput is bound by network latency. Hence, reducing the number of round trips improves application throughput, despite the overhead incurred on the web server from lazy evaluation. However, as the number of clients increases, the web server becomes CPU-bound and

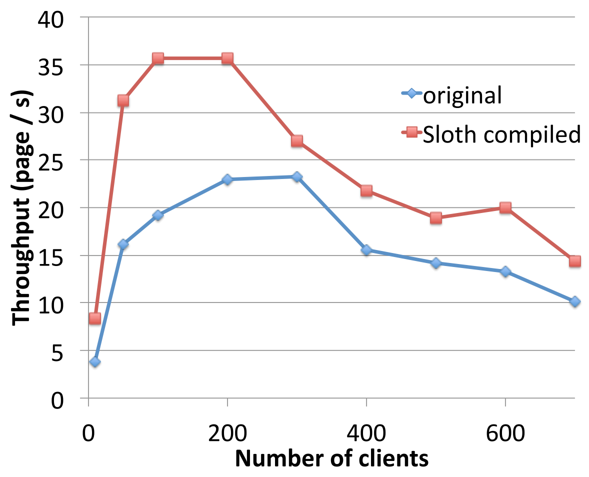


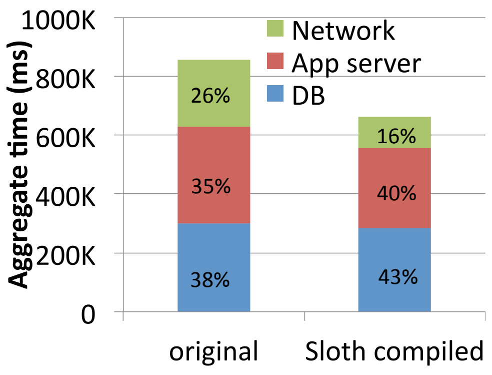
Figure 4-11: Throughput experiment results

throughput decreases. Since the original application does not incur any CPU overhead, it reaches the throughput at a higher number of clients, although the overall peak is lower due to network round trips.

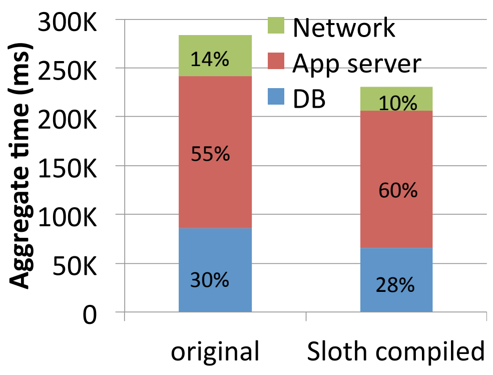
4.8.3 Time Breakdown Comparisons

Reducing the total number of queries issued by the application reduces one source of load time. However, there are other of sources of latency. To understand the issues, we measured the amount of time spent in the different processing steps of the benchmarks: application server processing, database query execution, and network communication. We first measured the overall load time for loading the entire page. Then, we instrumented the application server to record the amount of time spent in processing, and modified our batch JDBC driver to measure the amount of time spent in query processing on the database server. We attribute the remaining time as time spent in network communication. We ran the experiment across all benchmarks and measured where time was spent while loading each benchmark, and computed the sum of time spent in each phase across all benchmarks. The results for the two applications are shown in Fig. 4-12.

For the SLOTH compiled applications, the results show that the aggregate amount of time spent in network communication was significantly lower, reducing from 226k to 105k ms for itracker,



(a) itracker benchmarks



(b) OpenMRS benchmarks

Figure 4-12: Time breakdown of benchmark loading experiments

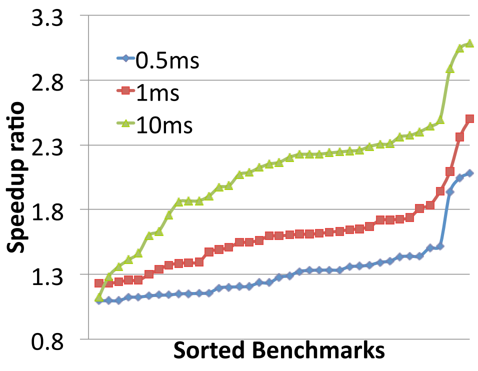
and 43k to 24k ms for OpenMRS. This is due to the reduction in network round trips. In addition, the amount of time spent in executing queries also decreased. We attribute that to the reduction in the number of queries executed, and to the parallel processing of batched queries on the database by our batch driver. However, the portion of time spent in the application server was higher for the SLOTH compiled versions due to the overhead of lazy evaluation, as expected.

4.8.4 Scaling Experiments

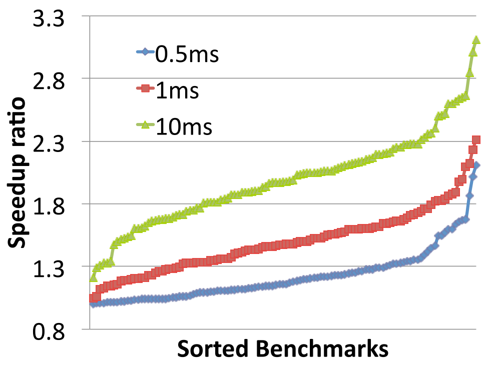
In the next set of experiments, we study the effects of round trip reduction on page load times. We ran the same experiments as in Sec. 4.8.1, but varied network delay from 0.5ms between the application and database servers (typical value for machines within the same data center), to 10ms (typical for machines connected via a wide area network and applications hosted on the cloud). Figure 4-13 shows the results for the two applications.

While the number of round trips and queries executed remained the same as before, the results show that the amount of speedup dramatically increases as the network round trip time increases (more than 3× for both applications with round trip time of 10ms). This indicates that reducing the number of network round trips is a significant factor in reducing overall load times of the benchmarks, in addition to reducing the number of queries executed.

Next, we measured the impact of database size on benchmark load times. In this experiment, we varied the database size (up to 25 GB) and measured the benchmark load times. Although

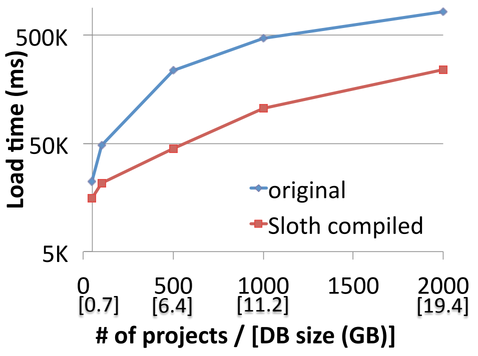


(a) itracker

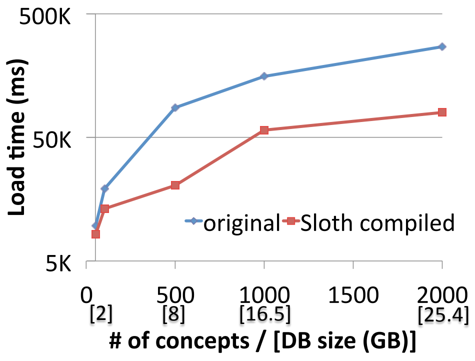


(b) OpenMRS

Figure 4-13: Network scaling experiment results



(a) itracker



(b) OpenMRS

Figure 4-14: Database scaling experiment results

the database still fits into the memory of the machine, we believe this is representative of the way that modern transactional systems are actually deployed, since if the database working set does not fit into RAM, system performance drops rapidly as the system becomes I/O bound. We chose two benchmarks that display lists of entities retrieved from the database. For itracker, we chose a benchmark that displays the list of user projects (`list_projects.jsp`) and varied the number of projects stored in the database; for OpenMRS, we chose a benchmark that shows the observations about a patient (`encounterDisplay.jsp`), a fragment of which was discussed in Sec. 4.8.1, and varied the number of observations stored. The results are shown in Fig. 4-14(a) and (b) respectively.

The SLOTH version of the applications achieved lower page load times in all cases, and they also scaled better as the number of entities increases. This is mostly due to query batching. For

Application	# persistent methods	# non-persistent methods
OpenMRS	7616	2097
itracker	2031	421

Figure 4-15: Number of persistent methods identified

instance, the OpenMRS benchmark batched a maximum of 68, 88, 480, 980, and 1880 queries as the number of database entities increased. Examining the query logs reveals that queries were batched in the manner discussed in Sec. 4.8.1. While the numbers of queries issued by two versions of the application are the same proportionally as the number of entities increases, the experiment shows that batching reduces the overall load time significantly, both because of the fewer round trips to the database, and the parallel processing of the batched queries. The itracker benchmark exhibits similar behavior.

4.8.5 Optimization Experiments

In this experiment we measured the effects of the optimizations presented in Sec. 4.6. First, we study the effectiveness of selective compilation. Figure 4-15 shows the number of methods that are identified as persistent in the two applications. As discussed in Sec. 4.6.1, non-persistent methods are not compiled to lazy semantics.

Next, we quantify the effects of optimizations by comparing the amount of time taken to load the benchmarks. We first measured the time taken to load all benchmarks from the SLOTH compiled applications with no optimizations. Next, we turned each of the optimizations on one at a time: selective compilation (SC), thunk coalescing (TC), and branch deferral (BD), in that order. We recompiled each time and Fig. 4-16 shows the resulting load time for all benchmarks as each optimization was turned on.

In both applications, branch deferral is the most effective in improving performance. This makes sense as both applications have few statements with externally visible side-effects, which increases the applicability of branch deferral. In addition, as discussed in Sec. 4.6.2, deferring control flow statements further delays the evaluation of thunks, which allows more query batching to take place.

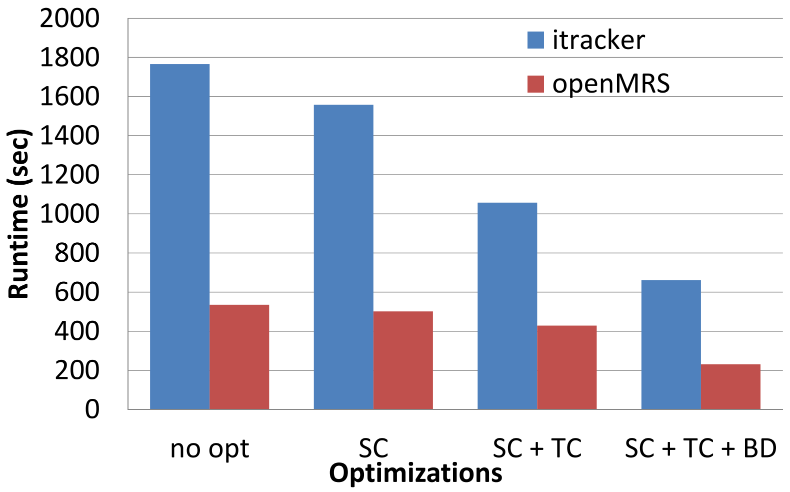


Figure 4-16: Performance of SLOTH on two benchmarks as optimizations are enabled. [SC=Selective computation, TC=Thunk Coalescing, BD=Branch Deferral.]

Overall, there was more than a 2× difference in load time between having none and all the optimizations for both applications. Without the optimizations, we would have lost all the benefits from database round trip reductions, i.e., the actual load times of the SLOTH compiled applications would have been slower than the original, due to the overhead of lazy evaluation.

4.8.6 Overhead Experiments

In the final experiment, we measured the overhead of lazy evaluation. We use TPC-C and TPC-W for this purpose. We chose implementations that use JDBC directly for database operations and do not cache query results. The TPC-W implementation is a standalone web application hosted on Tomcat. Since each transaction has very few queries, and the query results are used almost immediately after they are issued (e.g., printed out on the console in the case of TPC-C, and converted to HTML in the case of TPC-W), there are essentially no opportunities for SLOTH to improve performance, making these experiments a pure measure of overhead of executing under lazy semantics.

For the TPC-C experiment, we used a 20 warehouse database (initial size of the database is 23GB). We used 10 clients, with each client executing 10k transactions, and measured the time taken to finish all transactions. For the TPC-W experiment, the database contained 10,000 items

Transaction type	**Original time (s)**	**SLOTH time (s)**	**Overhead**
TPC-C			
New order	930	955	15.8%
Order status	752	836	11.2%
Stock level	420	459	9.4%
Payment	789	869	10.2%
Delivery	626	665	6.2%
TPC-W			
Browsing mix	1075	1138	5.9%
Shopping mix	1223	1326	8.5%
Ordering mix	1423	1600	12.4%

Figure 4-17: Overhead experiment results

(about 1 GB on disk), and the implementation omitted the think time. We used 10 emulated browsers executing 10k transactions each. The experiments were executed on the same machines as in the previous experiments, and all SLOTH optimizations were turned on. Figure 4-17 show the results.

As expected, the SLOTH compiled versions were 5-15% slower than the original, due to lazy semantics. However, given that the Java virtual machine is not designed for lazy evaluation, we believe that these overheads are reasonable, especially given the significant performance gains observed in real applications.

4.9 Summary

In this chapter I presented SLOTH, a new compiler and runtime framework that speeds up database applications by reducing the number of round trips between the application and query components in database applications. In constrast to prior work that uses purely static program analysis to rewrite the application source code, SLOTH instead makes use of a combination of static and dynamic techniques to reduce the number of round trips incurred during application execution. We showed that by delaying computation using lazy semantics, our system is able to reduce database round trips substantially by batching together multiple queries from the application and issuing them in a single batch. Along with a number of optimization techniques, we evaluated our framework on a variety of real-world applications. Our results show that our technique outperforms existing approaches in query batching using static analysis, and delivers substantial reduction (up to 3$\times$) in application execution time with modest worst-case runtime overheads.

Chapter 5

Bringing it Together: A Framework for Optimizing and Hosting Database Applications

As discussed in Ch. 1, due to the way that most database applications are developed, there is a separation between application logic and data access logic. As mentioned, the separation can be physical as well, as the application and database components of database applications are often hosted on separate servers. The hard separation between the application and the database makes development difficult and often results in applications that lack the desired performance, as have been demonstrated in previous chapters.

Not only that, reliability and security are also affected by the logical separation between the database and application. For example, the coexistence of application data structures and their persistent relational representations can make it difficult to reason about consistency and therefore ensure the integrity of transactions. Security, in turn, can be compromised by any inconsistency between application-level authorization mechanisms and the corresponding database-level access controls. The developer may have an end-to-end security policy in mind, but its piecemeal implementation can be complex and limiting.

There have been many efforts to create a programming environment that unifies the database and application views of data and removes the "impedance mismatch" [46] arising from the logical and physical separation between the two. This was a goal of the object-oriented database (OODB) movement [29, 49], which has seen a resurgence in recent years due to the widely used ORM frameworks such as Hibernate for Java and Django for Python. Many modern database-backed applications are developed using these frameworks rather than embedded SQL queries. As discussed in Ch. 2, a straightforward translation of operations on persistent objects into SQL queries can use the database inefficiently, and programmers may implement relational operations in imperative code due to a lack of understanding. All of these lead to inefficient applications.

In the previous chapters, I have described a number of technologies that can help to alleviate different aspects of the problems mentioned above. In this chapter, I proposal STATUSQUO,[1] a new programming system that utilizes the technologies described in this thesis in a platform for optimizing and hosting database applications. In contrast to traditional ORM frameworks and OODBs, STATUSQUO explicitly avoids forcing developers to use any one programming style. Instead, STATUSQUO allow developers to use the programming style—imperative, declarative, or mixed—that most naturally and concisely describes the intended computations. In addition, programmers are not required to think about how computation over data will be mapped onto the physical machines. Instead, STATUSQUO automatically decides where computation and data should be placed, and optimizes the entire application by employing various program and query optimization techniques. STATUSQUO also *adaptively* alters the placement of code and data in response to changes in the environment in which the application runs, such as machine load. Because STATUSQUO sees the *whole* system with information collected during execution time, it can do a better job of optimizing performance and of enforcing other end-to-end properties of the application, as compared to a system that only has partial program information during compile time, such as traditional ORM frameworks.

To do this, STATUSQUO must be able to move data and computation between the application server and the database server, and translate logic between the declarative and imperative programming models. The translation is done transparently by the compiler and runtime system—without requiring the programmer to rewrite their programs. This is made possible through techniques described in the earlier chapters for program analysis and transformation which ensure that programs

[1]Thus named as the system preserves the way that programmers currently use to develop database applications, yet delivers substantial performance gain over similar frameworks that are in use.

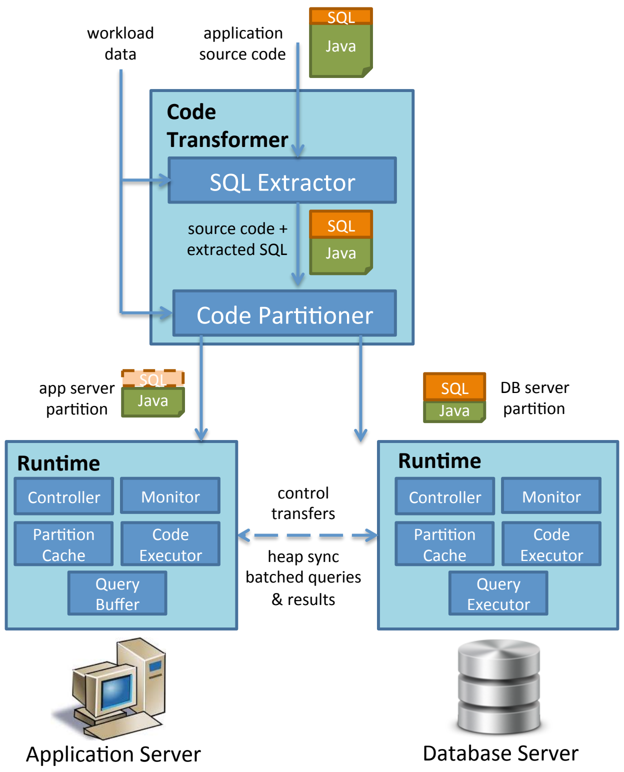


Figure 5-1: StatusQuo system design

continue to function correctly even as they are substantially transformed.

Implementing StatusQuo is beyond the scope of this thesis. In this chapter, I focus on the design of StatusQuo and how the components in the framework make use of previously described technologies. The overall proposed design of StatusQuo is shown in Fig. 5-1, consisting of a compiler and a runtime component installed on the application and database servers respectively. In the following I discuss each of them in detail.[2]

5.1 Compiling Database Applications

The StatusQuo compiler is responsible for compiling database application source code to be executed by the StatusQuo runtime components. Compilation takes place in two steps: collecting

[2]The basic design of the framework is published earlier as Cheung, Arden, Madden, Solar-Lezama, and Myers, "StatusQuo: Making Familiar Abstractions Perform Using Program Analysis," in proceedings of CIDR 13 [37].

profile information, and lifting portions of the input imperative code into declarative SQL queries to be executed by the database server.

As mentioned, StatusQuo allows developers to write their applications using imperative (e.g., Java or Python), declarative (e.g., SQL), or a mix of the two different paradigms. In addition, they are free to implement their data access logic using different interfaces, such as database drivers (e.g., JDBC [9], ODBC [14]) if they prefer to express data accesses as explicitly SQL queries, interfaces provided by ORM frameworks (e.g., JPA [10], Rails [20]), and query-embedded languages (e.g., LINQ [79]). Given the application source code, the StatusQuo compiler instruments it and deploys the instrumented version on the application and database servers to be executed for a short period of time (not shown in the figure). Inspired by Pyxis, the goal of instrumentation is to collect a workload profile that captures information such as execution frequency for methods and loops. After such information has been collected, the code is passed to the **SQL Extractor** component, which uses the Qbs algorithm to convert specific blocks of imperative code into declarative SQL queries. As discussed in Ch. 2, converting code into declarative form gives the system flexibility regarding how and where to implement the given functionality, and frees the developer from making such choices during implementation.

After converting portions of the input code to SQL queries, the source is given to the **Partitioner** component based on Pyxis. The partitioner uses the workload data collected during the profile run to split the source code into two programs: one to be executed on the database server as stored procedures, and another to be executed on the application server. As in Pyxis, the goal of partitioning is to reduce the amount of data transferred and the number of round trips between the two servers based on the collected profile. For declarative code fragments, the partitioner may decide to execute such fragments as queries on the database server or convert them (back) into (potentially optimized) imperative code. Another option would be to directly execute the declarative code on the application server using a declarative runtime engine, as shown with the dotted lines in Fig. 5-1. In addition to splitting program logic, the partitioner also adds the necessary housekeeping code such as heap synchronizations in order to preserve the semantics of the application when it is subsequently executed in a distributed manner. During compilation, the partitioner generates several different partitions under a range of load conditions, with the partitions differing in the amount of program logic allocated to each server.

At the end of the compilation process, the STATUSQUO compiler converts each of the generated partitions into binaries and sends them to the runtime components on the two servers to be executed.

5.2 Runtime for Database Applications

Unlike traditional application runtime systems, the STATUSQUO runtime executes code using continuation-passing style and lazy evaluation for database queries, as inspired by PYXIS and SLOTH. Each of the components in the runtime is described below.

The STATUSQUO runtime module consists of multiple components for hosting database applications. First, the **Partition Cache** holds the generated binaries received from the STATUSQUO compiler. As the application executes, the controller component in the runtime selects a partition from the set of available partitions based on the current server load.

The **Code Executor** component is responsible for running each partition. It uses extended lazy evaluation to execute each of the partitions (the partitioner is aware of this execution model when generating partitions). Similar to SLOTH, for queries that are issued by partitions allocated to the application server, they are first collected into the **Query Buffer** in the STATUSQUO runtime installed on the application server, as shown in Fig. 5-1. As the program executes, the executors periodically communicate with each other to transfer the control flow of the program (called *control transfers*), exchange heap data, and forward the buffered queries and result sets between the two servers. Upon receiving the batched queries, the **Query Executor** component on the database server runtime sends the queries to the DBMS to be executed, and the results are stored on the database server runtime to be forwarded to the application server runtime on the next control transfer.

Meanwhile, the **Monitor** component measures various aspects of the servers during application execution: the amount of resources (CPU and memory) available, the network bandwidth, and the number of control transfers, including the actual amount of the data transmitted and the number of queries buffered. This is performed so that if any of the servers exceed or otherwise deviate from the workload that the partition in use was designed for during initial profiling (e.g., caused by other applications that are hosted on the same machine or changes in the input workload), the executor informs the controller to dynamically switch to another partition that is better suited to the new load.

Finally, the **Controller** checks to see if such a partition requested by the runtime exists in the partition cache. If it is not, the controller contacts the code transformer module in the STATUSQUO compiler to generate a new partition. Making use of the data collected by the monitor component, the controller decides if the collected data show significant deviations from which the code partitions were generated. If so, it clears the partition cache, generates new partitions, and forwards the partitions to the STATUSQUO runtime as described. This adaptive optimization process continues until the application finishes execution.

5.3 Summary

In this chapter I described STATUSQUO, a novel end-to-end framework for hosting database applications. Unlike traditional application hosting frameworks, STATUSQUO makes use of various program analysis, program synthesis, and program monitoring techniques described in this thesis to continuously optimize database applications. As a result, STATUSQUO frees developers from restricting themselves to express application logic only to the formalism provided by the application hosting framework. Furthermore, rather than requiring developers to anticipate potential changes to the hosting environment and manually devise changes to applications to ensure that they are performant, STATUSQUO is designed to make the database server and the hosting platform aware of application semantics. In doing so, STATUSQUO enables automatic transformation, deployment, and adaptive maintenance of database applications for optimal performance.

Chapter 6

Future Work

In the earlier chapters, I described a number of projects that improve the performance of database applications. The various techniques used in these projects open up new areas of research and further opportunities in applying programming languages techniques in understanding database applications. In this chapter I discuss a few such opportunities.

6.1 Flexible Data Placement

The techniques discussed in Ch. 2 and Ch. 3 allow data and computation to be pushed from the application server into the database server, either as queries or as imperative code. This can help move computation closer to the data it uses. However, with some workloads, the best performance may be obtained by moving code and data in the other direction. For example, if the database server is under heavy load, it may be better to move computation to the application server. Data needed by this computation can be forwarded by the database server and cached at the application server instead.

Although web caches such as Memcached [12] have similar goals, they are generic object stores that require manual customization to implement application-specific caching policies. By contrast, program analysis and transformation techniques like those used in QBS will make it possible to automatically and adaptively migrate database operations and associated data to the application server, exploiting application semantics to implement an application-specific cache in which cached data is automatically synchronized during program execution. Techniques from data-shipping distributed object stores such as Thor [74] and Fabric [75] are likely to be useful, but challenges remain. First, work is needed on partitioning declarative queries and on optimizing the part of such partitioned queries that is moved to the application server. Since declarative queries do not have side effects, the partitioning problem, at least, is easier than with imperative code. Second, since application and database servers have different recovery models, it is difficult to move computation automatically while seamlessly preserving consistency guarantees. However, I believe that program analysis can be used to automatically identify exception-triggered imperative recovery code and to integrate its actions with transactional rollback mechanisms.

6.2 Support Multiple Application Servers and Back Ends

Applications that access multiple types of back-end database servers (such as NoSQL and graph stores) already exist and seem likely to become more common with the diversity of available data storage platforms and an increasing demand for data integration. Additionally, applications can be supported by multiple communicating application servers. While PYXIS has thus far been in the context of a single application server connecting to a single back-end DBMS, the general approach appears to extend to these more complex systems, facilitating optimization of programs across many diverse systems. Programming such systems in the higher-level way described in Ch. 5 poses interesting problems in how to express and constrain concurrency both among the application servers and among the back-end databases.

6.3 Information Security

To protect the confidentiality and integrity of persistent information, databases are equipped with access controls that mediate how applications may read and write information. This functionality can be useful when different applications share the same database as a way of exposing information in a limited way to applications that are not fully trusted to enforce security, or simply as a way to implement the principle of least privilege.

Many applications also rely on mechanisms to restrict access to data to various users or groups (for example, §17 of [8], [86]). The mismatch between these mechanisms and database access controls already makes security enforcement awkward and error-prone. Since PYXIS views database applications as a unified system, there is an opportunity to offer a clear and consistent security model.

The challenge is how to state and enforce information security policies in a system in which code and data placement is not fixed and even adapts dynamically. If code is moved into the DBMS, its accesses must be checked as if they were made by the application. Dynamically generated code and queries, commonly used by modern web applications, may need to be restricted even further. Moving code and data in the other direction, as discussed in Sec. 6.1, is even more problematic. Data must not be moved to an application server that is not trusted to enforce its confidentiality. Conversely, updates caused by transactions partitioned between the application and database may need to be controlled to prevent an untrustworthy application server from corrupting written data.

A common problem underlies all these difficulties: access control mechanisms are fundamentally not compositional. They do not permit code to be moved and decomposed freely while enforcing the same security guarantees. However, *information flow* controls do offer a compositional way to state and enforce information security requirements, and recent work has explored compositional information flow controls on persistent data in a distributed system [75, 41, 35]. In particular, the approach taken in the Fabric system might be applicable to PYXIS. Fabric supports secure computation over persistent data at both application server and persistent stores; it checks whether the (manual) partitioning of code and data protects confidentiality and integrity. (Fabric does not, however, support declarative queries or automatic partitioning.)

6.4 Rethink the software stack

Recently, custom data storage engines and architectures such as column stores and flash memory have outperformed traditional DBMSs running on spinning disks in specific domains. The cross-layer techniques discussed in this thesis can be leveraged to automatically design the entire software stack from the ground up, creating application-specific storage engines. For instance, query optimization focuses on finding the query plan with the minimal execution time. This is currently done by a combination of dynamic programming and rewrite heuristics, e.g., merging queries with nested select statements into a single flattened query. Such heuristics allow the DBMS to consider multiple selection predicates simultaneously, possibly enabling further optimization opportunities such as executing the most selective predicate first. Unfortunately, it is very difficult for developers to come up with such heuristics manually, as they need to ensure that the heuristics reduce query execution cost while preserving semantics. We can automatically synthesize such rewrite rules in an application-specific manner. One idea is to take the queries issued by the application, provide a few basic relational query equivalences and a cost model to the synthesizer (in terms of CPU and I/O cost for each type of query operator like selection and join), and ask the synthesizer to find new equivalences that reduce query execution time. Such approach has been used in recent work [73]. We can be even more ambitious, however, and ask the synthesizer to derive the equivalences from definitions of relational algebra operators, using techniques similar to those used in QBS.

We can also optimize across the DBMS and the architecture layer to search for the storage format that the DBMS should use (e.g., row or column major, and spinning disks or flash memory). We first define an "I/O algebra" that can be used to describe different storage formats and architectures, and then use synthesis to discover the equivalences among them. We can then search for the optimal disk layout, i.e., a realization of I/O algebra expressions on disks, that minimizes execution time on a given workload. Unlike prior work on automatic disk layout, we are not limited to storing all data using a uniform format–we can implement a storage system that implements I/O algebra expressions and adapts to changes during application execution.

6.5 Contextual query processing

Having accurate workload information is essential to efficient query processing. As many queries are now programmatically generated by applications rather than entered free-form by users, analyzing the application will give us insights about the issued queries. For example, as shown in Sec. 4.2, many database applications frequently issue many queries during execution (e.g., as a page is rendered). Such programmatically-issued queries have highly deterministic structure, for instance query parameters are derived from program variables. Analyzing the application source allows us to create *application contexts*. Each context represents a program path in the code and describes the queries that are issued. It also includes the application code that manipulates the query results, the relationship among the query parameters, the frequencies of each query (which can be approximated from the code or query logs), and the actions that trigger each query such as a button click.

Application contexts are useful in many scenarios that I outline below. First, the DBMS can use application contexts to combine queries and prefetch a subset of the results. Moreover, the correlations among the query parameters allow the DBMS to improve the generated query plans. For instance, if the context indicates that the same program variable is passed into two different queries as parameters, then once the value of the program variable is known, the DBMS can estimate the selectivity of both queries accurately. In addition, the DBMS can make use of context information to determine when to evict query results from the buffer pool when they are no longer needed.

6.5.1 Infer functional dependencies

Besides optimizing read queries, application contexts can be used to learn functional dependencies. For instance, after a user puts in a city name in a form, the application issues a query to fetch the given city's current population from the DBMS. After that, the application computes a prediction of the city's future population, shows it to the user for validation, and stores the (city name, predicted population) pair in another table. There is an obvious correlation between the current and predicted populations stored in the DBMS. Discovering such correlation from the logs is difficult as it involves application code, and the log might include queries issued by multiple concurrent users. We can instead use the application context for this purpose. For instance, in this case the context will inform us that predicted populations are derived from another table that stores the current values.

In general, we can use contexts to learn rich functional dependencies of the form $X \rightarrow f(Y)$, where some part of f might be implemented in the application code. Such dependencies are very difficult to infer using statistical techniques, especially those that involve continuous data values, such as converting between different measurement units, and computing interest rate given credit score. One idea is to use program synthesis (as discussed in Ch. 2) to derive a succinct representation of f from the source code. Once f is derived, we can retrieve statistics about X from the DBMS and propagate through f to estimate the distribution of Y.

6.5.2 Improve query optimization

Classical query optimization focuses on finding efficient implementations of each operator in the query tree. Given application contexts, we can treat each contextual query as a user defined function (UDF) that includes both queries and application code that processes the results. We can then specialize the optimizer to find efficient implementations of each UDF given the physical design.

This insight raises new research opportunities. Since UDFs are traditionally treated as black boxes in query optimization, it will be interesting to optimize UDFs using classical program optimization techniques [77]. One challenge is to extend such techniques to be I/O cost-aware in addition to the traditional goal of minimizing the number of instructions, and consider implementing DBMS functionalities in hardware functional units such as flash controllers and FPGAs. Addressing these challenges will advance both data management and programming systems research.

6.6 Improve user interaction with databases

While SQL is the de facto language for interacting with relational databases, writing SQL queries is often a tedious task. The inductive program synthesis techniques discussed in Ch. 2 can be used to help non-expert SQL users write queries easily. For instance, we can define a synthesizer-backed visual language similar to Query By Example [116]. Using such language, users "write" queries by providing sample output values in a spreadsheet. Similar to earlier work [39], the system treats the provided values as a partial specification and generates a SQL query that outputs the same (or a superset of) values. When there are multiple such queries, the system will solicit help from the user. Listing all the potential queries and asking the user to choose is difficult given the user's lack of SQL expertise. Instead, the system generates a small "differential database" that returns different outputs for each of the generated queries, and asks the user to choose the output of interest given the differential database. Such a language is innovative in a number of ways. First, it extends tools such as Data Wrangler [71] to generate complex SQL queries for users with no exposure to SQL. Moreover, generating minimal differential databases raises new challenges in synthesis technology as current techniques do not focus on minimality.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 7

Related Work

This thesis presents new techniques to optimize applications that interact with databases. In this chapter I discuss related work that our techniques are based on, along with other approaches to address related problems.

7.1 Transforming Code Representations

While one of the goals of traditional compilers is to convert code representation from the source language to the target language, converting from an imperative language (such as Java) to a declarative one (such as SQL) presents additional challenges as discussed in Sec. 2.1. This has been previously studied as the problem of inferring relational specifications from imperative code [109, 110]. The idea described in the prior work is to compute the set of data access paths that the imperative code traverses using abstract interpretation, and replace the imperative code with SQL queries. The analysis is applicable to recursive function calls, but does not handle code with loop-carried dependencies, or those with join or aggregation. It is also unclear how modeling relational operations as access paths can be extended to handle such cases. In contrast, our tool, QBS, is able to infer both join and aggregation from imperative code. While QBS currently does not handle recursive function calls as compared to prior work, we have not encountered such uses in the applications used in our experiments.

Modeling relational operations. The ability to infer relational specifications from imperative code in QBS relies on using the theory of ordered relations (TOR) to bridge the imperative and relational worlds. There are many prior work in modeling relational operations, for instance using bags [45], sets [69], and nested relational calculus [111]. There are also theoretical models that extend standard relational algebra with order, such as [83, 25]. QBS does not aim to provide a new theoretical model. Instead, one key insight of our work is that TOR is the appropriate abstraction for the query inference, as they are similar to the interfaces provided by the ORM libraries in the imperative code, and it also allows us to design a sound and precise transformation into SQL.

To our knowledge, QBS is the first to address the ordering of records in relational transformations. Record ordering would not be an issue if the source program only operated on orderless data structures such as sets, or only perform operations that preserve the order of records as stored in the DBMS such as selections and projections. Unfortunately, most ORM libraries provide interfaces based on ordered data structures, such as lists and arrays, and most operations alter or produce new ordering based on the input records, such as joins. And imperative implementations of join proves to be common at least in the benchmarks we studied.

Finding invariants to validate transformations. QBS's verification-based approach to finding a translatable postcondition is similar to earlier work [67, 68], although they acknowledge that finding invariants is difficult. Similar work has been done for general-purpose language compilers [102]. Scanning the source code to generate the synthesis template is inspired by the PINS algorithm [101], although QBS does not require user intervention. There has been earlier work on automatically inferring loop invariants, such as using predicate refinement [54] or dynamic detection [52, 59].

Constraint-based synthesis. Constraint-based program synthesis has been an active topic of research in recent years. For instance, there has been work on using synthesis to discover invariants [100]. QBS differs from previous approaches in that we only need to find invariants and postconditions that are *logically strong enough* to validate the transformation, and our predicate language greatly prunes the space of invariants to those needed for common relational operations. Synthesis has also been applied in other domains, such as generating data structures [95], processor instructions [56], and learning queries from examples [39].

Integrated query languages. Integrating application and database query languages into has been an active research area, with projects such as LINQ [79], Kleisli [111], Links [44], JReq [67], the functional language proposed in Cooper [45], Ferry [58], and DBPL [93]. These solutions embed database queries in imperative programs without ORM libraries. Unfortunately, many of them do not support all relational operations, and the syntax of such languages resemble SQL, thus developers still need to learn new programming paradigms and rewrite existing applications.

7.2 Placing Computation Across Servers

There has been prior work on automatically partitioning applications across distributed systems. However, PYXIS is the first system that partitions general database applications between an application server and a database server. PYXIS also contributes new techniques for reducing data transfers between host nodes.

Partitioning general-purpose programs. Program partitioning has been an active research topic for the past decade. Most of these approaches require programs to be decomposed into coarse-grained modules or functions that simplify and make a program's dependencies explicit. Imposing this structure reduces the complexity of the partitioning problem and has been usefully applied to several tasks: constructing pages in web applications with Hilda [113], processing data streams in sensor networks with Wishbone [82], and optimizing COM applications with Coign [65].

Using automatic partitioning to offload computation from handheld devices has featured in many recent projects. Odessa [87] dynamically partitions computation intensive mobile applications but requires programs to be structured as pipelined processing components. Other projects such as MAUI [47] and CloneCloud [43] support more general programs, but only partition at method boundaries. Chroma [30] and Spectra [30] also partition at method boundaries but improve partition selection using code annotations called *tactics* to describe possible partitionings.

Prior work on secure program partitioning such as Swift [42] and Jif/split [114, 115] focuses more on security than performance. For instance, Swift minimizes control transfers but does not try to optimize data transfers.

PYXIS supports a more general programming model compared to these approaches. Our implementation targets Java programs and the JDBC API, but our technique is applicable to other general purpose programming languages. PYXIS does not require special structuring of the program nor does it require developer annotations. Furthermore, by considering program dependencies at a fine-grained level, PYXIS can automatically create code and data partitions that would require program refactoring or design changes to affect in other systems.

Data partitioning and query rewriting. Besides program partitioning, another method to speed up database applications is to rewrite or batch queries embedded in the application. One technique for automatically restructuring database workloads is to issue queries asynchronously, using program analysis to determine when concurrent queries do not depend on each other [34], or using data dependency information to batch queries together [60]. Automatic partitioning has advantages over this approach: it works even in cases where successive queries do depend on each other, and it can reduce data transfers between the hosts.

Data partitioning has been widely studied in the database research community [26, 90], but the techniques are very different. These systems focus on distributed query workloads and only partition data among different database servers, whereas PYXIS partitions both data and program logic between the database server and its client.

Custom language runtimes. Sprint [89] speculatively executes branches of a program to predict future accesses of remote data and reduce overall latency by prefetching. The Sprint system is targeted at read-only workloads and performs no static analysis of the program to predict data dependencies. PYXIS targets a more general set of applications and does not require speculative execution to conservatively send data between hosts before the data is required.

Executable program slicing [31] extracts the program fragment that a particular program point depends on using the a system dependence graph. Like program slicing, PYXIS uses a control and data dependencies to extract semantics-preserving fragments of programs. The code blocks generated by PYXIS, however, are finer-grained fragments of the original program than a program slice, and dependencies are satisfied by automatically inserting explicit synchronization operations.

7.3 Batching queries to reduce network round trips

SLOTH uses extended lazy evaluation to dynamically batch database queries together to reduce network communication between the application and database servers. Lazy evaluation was first introduced for lambda calculus [62]. Lazy evaluation aims to increase the expressiveness of the language by allowing programmers to define infinite data structures. Lazy evaluation is often implemented using thunks in languages that do not readily support it [66, 107]. In contrast, the extended lazy evaluation proposed in this chapter is fundamentally different: rather than increasing language expressiveness, SLOTH uses lazy evaluation to improve application performance by batching queries, and SLOTH is the first system to do so to our knowledge.

Batching query plans and sharing query results are well-known techniques in query optimization [55, 94]. However, they aim to combine queries issued by multiple concurrent clients, whereas SLOTH batches queries that are issued by the same client over time, although we can make use of such techniques to merge SLOTH-generated query batches from multiple clients for further performance improvement. Meanwhile, there has been work on demonstrating application speedups if transactions can be restructured to perform reads prior to writes [103], and work on using static analysis to expose batching opportunities [60, 34]. However, branches in code make static analysis imprecise in identifying queries that can be batched. Because branch outcomes are unknown at compile time, the analysis has to either assume that queries that are guarded by branches are not executed, which reduces batching efficiency; or it has to speculatively assume that branches are executed. Speculation can result in executing more queries than the original application would have and hurt performance. Dynamic dispatch of method calls exhibit the same issue.

In contrast, SLOTH uses lazy evaluation as a mechanism to expose query batching opportunities during application execution. Lazy evaluation allows us to avoid deciding on the batches during compile time. Instead, query batches are constructed dynamically when the application executes. Because of that, batches can be constructed across both branches and procedure calls, and our experiments show reduction in execution time that would not be achievable using pure static analysis.

As discussed in Sec. 4.1, data prefetching is another means to reduce database round trips.

Prefetching has been studied theoretically [97] and implemented in open source systems [1, 3, 17], although they all require programmer annotations to indicate what and when to prefetch. Finally, there is work on moving application code to execute in the database as stored procedures to reduce the number of round trips (such as PYXIS), which is similar to our goals. In comparison, SLOTH does not require program state to be distributed.

Chapter 8

Conclusion

In this thesis I described the challenges associated with developing database applications, and argued that the narrow query interface between the application and the DBMS is insufficient for the needs of modern database applications. While such a narrow API allows developers to cleanly separate their applications into the database and server components, the hard separation between the application and the database makes development difficult and often results in applications that lack the desired performance.

Through a series of projects, I show that by leveraging program analysis, program synthesis, and automated verification techniques, we can alleviate many of the problems that plague database application developers. In Ch. 2, we investigated the problem of how a piece of computation from the database application should be implemented. For instance, whether it should be implemented as imperative code to be executed on the application server, or as declarative queries to be executed by the DBMS. In response, we devised new techniques that can automatically convert code representations from one to another in a tool called QBS, which automatically infers relational specifications from imperative code such that the code can be converted into SQL queries. We have evaluated QBS using 75 code fragments automatically extracted from real-world database applications, and showed that QBS can improve application performance asymptotically by orders of magnitude.

Once it is decided which part of the application will be executed on the application and database servers, the two servers will inevitably need to communicate with each other in order to forward queries and result sets, and such communication slows down the application. In Ch. 3, I described algorithms for automatic code partitioning between multiple servers to reduce the amount of network communication between the application and database servers, and implemented such algorithms in PYXIS. Using TPC-C and TPC-W as benchmarks, PYXIS achieved up to 3$\times$ reduction in latency and 1.7$\times$ improvement in throughput as compared to a non-partitioned implementation.

Finally, in Ch. 4, I discussed SLOTH, which proposes another approach to reduce communication overhead through the use of extended lazy evaluation and query buffering during application execution. Using real-world applications, we demonstrated SLOTH's ability to reduce application execution time by up to 3$\times$ as compared to the original implementation.

All of these projects illustrate the power of how understanding different layers of the software stack can lead to co-optimization of both the runtime system and the DBMS at the same time. I believe that the techniques presented are complementary to each other. As such, in Ch. 5 I proposed STATUSQUO, a framework for compiling, hosting, and executing database applications that makes use of the techniques discussed in this thesis. Furthermore, in Ch. 6 I described future directions in programming systems and database research that are enabled by the techniques discussed in this thesis. In sum, the techniques discussed in this thesis represents initial steps towards helping programmers develop database applications, and the lessons learned will be applicable to other application domains as well.

Bibliography

[1] Apache Cayenne ORM documentation. `http://cayenne.apache.org/docs/3.0/prefetching.html`.

[2] Django web framework. `http://www.djangoproject.com`.

[3] Hibernate fetching strategies. `http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html/ch20.html`.

[4] Hibernate Performance. `http://stackoverflow.com/questions/5155718/hibernate-performance`.

[5] itracker Issue Management System. `http://itracker.sourceforge.net/index.html`.

[6] JBoss Hibernate. `http://www.hibernate.org`.

[7] JSR 152: JavaServer Pages 2.0 specification. `http://jcp.org/en/jsr/detail?id=152`.

[8] JSR 220: Enterprise JavaBeans 3.0 specification (persistence). `http://jcp.org/en/jsr/detail?id=220`.

[9] JSR 221: JDBC 4.0 API Specification. `http://jcp.org/en/jsr/detail?id=221`.

[10] JSR 317: Java Persistence API. `http://jcp.org/en/jsr/detail?id=317`.

[11] Marissa Mayer at Web 2.0. `http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html`.

[12] Memcached. `http://memcached.org`.

[13] Network latency under Hibernate/c3p0/MySQL. `http://stackoverflow.com/questions/3623188/network-latency-under-hibernate-c3p0-mysql`.

[14] ODBC Programmer's Reference. `http://msdn.microsoft.com/en-us/library/windows/desktop/ms714177(v=vs.85).aspx`.

[15] OpenMRS medical record system. `http://www.openmrs.org`.

[16] PhoCusWright/Akamai Study on Travel Site Performance. `http://uk.akamai.com/html/about/press/releases/2010/press_061410.html`.

[17] PHP query prefetch strategies. `http://php.net/manual/en/function.oci-set-prefetch.php`.

[18] PL/Java. `http://pgfoundry.org/projects/pljava`.

[19] Round trip / network latency issue with the query generated by hibernate. `http://stackoverflow.com/questions/13789901/round-trip-network-latency-issue-with-the-query-generated-by-hibernate`.

[20] Ruby on rails. `http://rubyonrails.org`.

[21] The Accrue Analysis Framework. `http://people.seas.harvard.edu/~chong/accrue.html`.

[22] TPC-C and TPC-W reference implementations. `http://sourceforge.net/apps/mediawiki/osdldbt`.

[23] What is the n+1 selects issue? `http://stackoverflow.com/questions/97197/what-is-the-n1-selects-issue`.

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

[25] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[26] Sanjay Agrawal, Vivek R. Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 359–370, 2004.

[27] Frances E. Allen. Control flow analysis. *SIGPLAN Notices*, 5(7):1–19, July 1970.

[28] John R. Allen and Ken Kennedy. Automatic loop interchange. In *Proc. International Conference on Compiler Construction (CC)*, pages 233–246, 1984.

[29] Malcom Atkinson, François Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik. The object-oriented database system manifesto. In *Proc. International Conference on Deductive Object Oriented Databases*, December 1989.

[30] Rajesh Krishna Balan, Mahadev Satyanarayanan, SoYoung Park, and Tadashi Okoshi. Tactics-based remote execution for mobile computing. In *Proc. International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 273–286, 2003.

[31] D. Binkley. Precise executable interprocedural slices. *ACM Letters on Programming Languages and Systems*, 2(1-4):31–45, 1993.

[32] Andreas Blass and Yuri Gurevich. Inadequacy of computable loop invariants. *ACM Transactions on Computer Logic (TOCL)*, 2(1):1–11, 2001.

[33] George Candea, Neoklis Polyzotis, and Radek Vingralek. A scalable, predictable join operator for highly concurrent data warehouses. *Proceedings of the VLDB Endowment (PVLDB)*, 2(1):277–288, 2009.

[34] Mahendra Chavan, Ravindra Guravannavar, Karthik Ramachandra, and S. Sudarshan. Program transformations for asynchronous query submission. In *Proc. International Conference on Data Engineering (ICDE)*, pages 375–386, April 2011.

[35] Winnie Cheng, Dan R. K. Ports, David Schultz, Victoria Popic, Aaron Blankstein, James Cowling, Dorothy Curtis, Liuba Shrira, and Barbara Liskov. Abstractions for usable information flow control in Aeolus. In *Proc. USENIX Annual Technical Conference*, June 2012.

[36] Alvin Cheung, Owen Arden, Samuel Madden, and Andrew C. Myers. Automatic partitioning of database applications. *Proceedings of the VLDB Endowment (PVLDB)*, 5, 2011.

[37] Alvin Cheung, Owen Arden, Samuel Madden, Armando Solar-Lezama, and Andrew C. Myers. StatusQuo: Making familiar abstractions perform using program analysis. In *Proc. Conference on Innovative Data Systems Research (CIDR)*, 2013.

[38] Alvin Cheung, Samuel Madden, and Armando Solar-Lezama. Sloth: Being lazy is a virtue (when issuing database queries). In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 931–942, 2014.

[39] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Using program synthesis for social recommendations. In *Proc. ACM International Conference on Information and Knowledge Management (CIKM)*, pages 1732–1736, 2012.

[40] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. In *Proc. SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 3–14, 2013.

[41] Adam Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[42] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *Proc. ACM Symposium on Operating System Principles (SOSP)*, pages 31–44, October 2007.

[43] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. CloneCloud: elastic execution between mobile device and cloud. In *Proc. European Conference on Computer Systems (EuroSys)*, pages 301–314, 2011.

[44] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *Formal Methods for Components and Objects*, pages 266–296, 2007.

[45] Ezra Cooper. The script-writer's dream: How to write great sql in your own language, and be sure it will succeed. In *Proc. International Symposium on Database Programming Languages (DBPL)*, pages 36–51, 2009.

[46] George Copeland and David Maier. Making smalltalk a database system. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 316–325, 1984.

[47] Eduardo Cuervo, Aruna Balasubramanian, Dae ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. MAUI: making smartphones last longer with code offload. In *Proc. International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 49–62, 2010.

[48] Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[49] Hugh Darwen and C. J. Date. The third manifesto. *ACM SIGMOD Record*, 24(1), 1995.

[50] David Déharbe, Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. Exploiting

symmetry in SMT problems. In *Proc. International Conference on Automated Deduction*, pages 222–236, 2011.

[51] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[52] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proc. International Conference on Software Engineering (ICSE)*, pages 213–224, 1999.

[53] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, July 1987.

[54] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, pages 191–202, 2002.

[55] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. SharedDB: Killing one thousand queries with one stone. *Proceedings of the VLDB Endowment (PVLDB)*, 5(6):526–537, 2012.

[56] Patrice Godefroid and Ankur Taly. Automated synthesis of symbolic instruction encodings from I/O samples. In *Proc. SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 441–452, 2012.

[57] David Gries. *The Science of Programming*. Springer, 1987.

[58] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. FERRY: database-supported program execution. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1063–1066, 2009.

[59] Ashutosh Gupta and Andrey Rybalchenko. Invgen: An efficient invariant generator. In *Proc. International Conference on Computer-Aided Verification (CAV)*, pages 634–640, 2009.

[60] Ravindra Guravannavar and S. Sudarshan. Rewriting procedures for batched bindings. *Proceedings of the VLDB Endowment (PVLDB)*, 1(1):1107–1123, 2008.

[61] Gurobi Optimizer. `http://www.gurobi.com`.

[62] Peter Henderson and James H. Morris, Jr. A lazy evaluator. In *Proc. ACM Symposium on*

Principles of Programming Languages (POPL), pages 95–103, 1976.

[63] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[64] Susan Horwitz, Thomas W. Reps, and David Binkley. Interprocedural slicing using dependence graphs. In *Proc. SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 35–46, 1988.

[65] Galen C. Hunt and Michael L. Scott. The Coign automatic distributed partitioning system. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 187–200, 1999.

[66] Peter Zilahy Ingerman. Thunks: a way of compiling procedure statements with some comments on procedure declarations. *Communications of the ACM*, 4(1):55–58, 1961.

[67] Ming-Yee Iu, Emmanuel Cecchet, and Willy Zwaenepoel. Jreq: Database queries in imperative languages. In *Proc. International Conference on Compiler Construction (CC)*, pages 84–103, 2010.

[68] Ming-Yee Iu and Willy Zwaenepoel. HadoopToSQL: a mapreduce query optimizer. In *Proc. European Conference on Computer Systems (EuroSys)*, pages 251–264, 2010.

[69] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[70] Simon L. Peyton Jones and André L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1-3):3–47, 1998.

[71] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler: Interactive visual specification of data transformation scripts. In *Proc. SIGCHI Conference on Human Factors in Computing Systems (CHI)*, pages 3363–3372, 2011.

[72] Gary A. Kildall. A unified approach to global program optimization. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, pages 194–206, 1973.

[73] Yannis Klonatos, Andres Nötzli, Andrej Spielmann, Christoph Koch, and Victor Kuncak. Automatic synthesis of out-of-core algorithms. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 133–144, 2013.

[74] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day, and L. Shrira. Safe and efficient sharing of persistent objects in thor. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 318–329, 1996.

[75] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: a platform for secure distributed computation and storage. In *Proc. ACM Symposium on Operating System Principles (SOSP)*, pages 321–334, 2009.

[76] lpsolve. `http://sourceforge.net/projects/lpsolve`.

[77] Henry Massalin. Superoptimizer: A look at the smallest program. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 122–126, 1987.

[78] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[79] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling objects, relations and XML in the .NET framework. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 706–706, 2006.

[80] Microsoft. z3 Theorem Prover. `http://research.microsoft.com/en-us/um/redmond/projects/z3`.

[81] A. Milanova, A. Rountev, and B.G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(1):1–41, 2005.

[82] Ryan Newton, Sivan Toledo, Lewis Girod, Hari Balakrishnan, and Samuel Madden. Wishbone: Profile-based partitioning for sensornet applications. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 395–408, April 2009.

[83] Wilfred Ng. An extension of the relational data model to incorporate ordered domains. *ACM Transactions on Database Systems (TODS)*, 26(3):344–383, September 2001.

[84] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *Proc. International Conference on Compiler Construction*

(CC), pages 138–152, 2003.

[85] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *Proc. International Conference on Compiler Construction (CC)*, pages 138–152, April 2003.

[86] OASIS Web Services Security Technical Community. Web Services Security v1.1.1. `http://docs.oasis-open.org/wss-m/wss/v1.1.1/os/wss-KerberosTokenProfile-v1.1.1-os.html`.

[87] Moo-Ryong Ra, Anmol Sheth, Lily B. Mummert, Padmanabhan Pillai, David Wetherall, and Ramesh Govindan. Odessa: enabling interactive perception applications on mobile devices. In *Proc. International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 43–56, 2011.

[88] Karthik Ramachandra and S. Sudarshan. Holistic optimization by prefetching query results. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 133–144, 2012.

[89] A. Raman, G. Yorsh, M. Vechev, and E. Yahav. Sprint: speculative prefetching of remote data. In *Proc. ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 259–274, 2011.

[90] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy M. Lohman. Automating physical database design in a parallel database. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 558–569, 2002.

[91] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, pages 105–118, 1999.

[92] V. Sarkar. Automatic partitioning of a program dependence graph into parallel tasks. *IBM Journal of Research and Development*, 35(5.6):779–804, September 1991.

[93] Joachim W. Schmidt and Florian Matthes. The DBPL project: Advances in modular database programming. *Information Systems*, 19(2):121–140, 1994.

[94] Timos K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*

(TODS), 13(1):23–52, 1988.

[95] Rishabh Singh and Armando Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *Proc. ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 289–299, 2011.

[96] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: Understanding object-sensitivity. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, pages 17–30, 2011.

[97] Alan Jay Smith. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems (TODS)*, 3(3):223–247, 1978.

[98] The SML/NJ Fellowship. *Standard ML of New Jersey*. http://www.smlnj.org/.

[99] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 404–415, 2006.

[100] Saurabh Srivastava and Sumit Gulwani. Program verification using templates over predicate abstraction. In *Proc. SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 223–234, 2009.

[101] Saurabh Srivastava, Sumit Gulwani, Swarat Chaudhuri, and Jeffrey S. Foster. Path-based inductive synthesis for program inversion. In *Proc. SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 492–503, 2011.

[102] Zachary Tatlock and Sorin Lerner. Bringing extensibility to verified compilers. In *Proc. SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 111–121, 2010.

[103] Alexander Thomson and Daniel J. Abadi. The case for determinism in database systems. *Proceedings of the VLDB Endowment (PVLDB)*, 3(1-2):70–80, 2010.

[104] Emina Torlak and Daniel Jackson. Kodkod: a relational model finder. In *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 632–647, 2007.

[105] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. In *Proc. SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 87–97, 2009.

[106] R.J. Vanderbei. *Linear programming: foundations and extensions*, volume 114 of *International series in operations research & management science*. Springer Verlag, 2008.

[107] Alessandro Warth. LazyJ: Seamless lazy evaluation in java. *Proc. FOOL/WOOD Workshop*, 2007.

[108] John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. In *Proc. ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 187–206, 1999.

[109] Ben Wiedermann and William R. Cook. Extracting queries by static analysis of transparent persistence. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, pages 199–210, 2007.

[110] Ben Wiedermann, Ali Ibrahim, and William R. Cook. Interprocedural query extraction for transparent persistence. In *Proc. ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 19–36, 2008.

[111] Limsoon Wong. Kleisli, a functional query system. *Journal of Functional Programming*, 10(1):19–56, 2000.

[112] Yichen Xie and Alex Aiken. Scalable error detection using boolean satisfiability. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, pages 351–363, 2005.

[113] Fan Yang, Nitin Gupta, Nicholas Gerner, Xin Qi, Alan Demers, Johannes Gehrke, and Jayavel Shanmugasundaram. A unified platform for data driven web applications with automatic client-server partitioning. In *Proc. International World Wide Web Conference (WWW)*, pages 341–350, 2007.

[114] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems (TOCS)*, 20(3):283–328, August 2002.

[115] Lantian Zheng, Stephen Chong, Andrew C. Myers, and Steve Zdancewic. Using replication

and partitioning to build secure distributed systems. In *Proc. IEEE Symposium on Security and Privacy*, pages 236–250, May 2003.

[116] Moshé M. Zloof. Query-by-example: The invocation and definition of tables and forms. In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 1–24, 1975.