Using Program Analysis to Improve Database Applications

Alvin Cheung
Samuel Madden Armando Solar-Lezama
MIT CSAIL
{akcheung, madden, asolar}@csail.mit.edu

Owen Arden
Andrew C. Myers
Department of Computer Science
Cornell University
{owen, andru}@cs.cornell.edu

Abstract

Applications that interact with database management systems (DBMSs) are ubiquitous. Such database applications are usually hosted on an application server and perform many small accesses over the network to a DBMS hosted on the 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 relatively little work that optimizes database applications by considering these specialized systems in combination and looking for optimization opportunities that span across them.

In this article, we highlight three projects that optimize database applications by looking at both *the programming system and the DBMS in a holistic manner. By carefully revisiting the interface between the DBMS and the application, and by applying a mix of declarative database optimization and modern program analysis techniques, we show that a speedup of multiple orders of magnitude is possible in real-world applications.*

1 Introduction

From online shopping websites to banking applications, we interact with applications that store persistent data in DBMSs every day. Typically, such applications are written using a general-purpose, imperative language such as Java or Python, with embedded data access logic expressed declaratively in SQL. The application is usually hosted on an application server that is physically separated from (although in close proximity to) the server running the DBMS (we refer to the latter as the database server). During execution, the application issues queries to the DBMS to retrieve or manipulate persistent data.

While this separation between the application and the database helps application development, it often results in applications that lack the desired performance. For example, to achieve good performance, both the compiler and query optimizer optimize parts of the program, but they do not share information, so programmers must manually determine 1) whether a piece of computation should be executed as a query by the DBMS or as general-purpose code in the application server; 2) where computation should take place, as DBMSs are capable of

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

```
List<User> getRoleUser () {
  List<User> listUsers = new ArrayList<User>();
  List<User> users = ... /* Database query */
  List<Role> roles = ... /* Database query */
  for (int i = 0; i < users.size(); i++) {
    for (int j = 0; j < roles.size(); j++) {
      if (users.get(i).roleId ==
          roles.get(j).roleId) {
        User userok = users.get(i);
        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; }
```

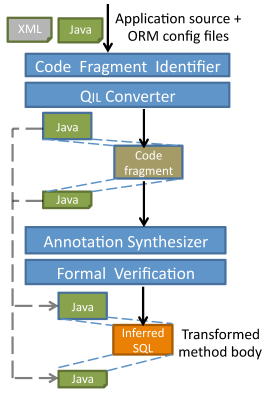


Figure 1: (a) Real-world code example that implements join in Java (left); (b) QBS converted version of code example (middle); (c) QBS architecture (right)

executing general-purpose code using stored procedures, and application servers can similarly execute relational operations on persistent data after fetching them from the DBMS; and 3) how to restructure their code to reduce the number of interactions with the DBMS, as each interaction introduces a network round trip and hence increases application latency.

While handcrafting such optimizations can yield order-of-magnitude performance improvements, these manual optimizations are done at the cost of code complexity and readability. Worse, these optimizations can be brittle. Relatively small changes in control flow or workload pattern can have drastic consequences and cause the optimizations to hurt performance, and slight changes to the data table schema can break manually optimized code. 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.

Applying recent advances in program analysis and synthesis, we have shown in prior work [10] that we can achieve such optimizations automatically without incurring substantial burden on the developers. In particular, 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 demonstrated order-of-magnitude speedups in real-world database applications. Not only that, we are able to achieve such application improvement while allowing developers to use the same high-level programming model to develop such applications. In this article, we describe three recent projects, in which each project holistically optimizes both the runtime system and the DBMS: QBS, a tool that transforms imperative code fragments into declarative queries to be executed by the DBMS; PYXIS, a system that automatically partitions program logic across multiple servers for optimal performance, and SLOTH, a tool that reduces application latency by eliminating unnecessary network round trips between the application and DBMS servers. In the following we describe each of the projects and highlight representative experimental results.

2 Determining How to Execute

As an example of how database application performance can be substantially improved by analyzing the runtime system and the DBMS together as a whole, consider the code fragment shown in Fig. 1(a). The fragment is adopted from a real-world Java web application that uses the Hibernate Object-Relational Mapping (ORM) library to access persistent data.

As written, the application first issues two queries using Hibernate to fetch a list of user and role objects from the DBMS. The code then iterates the lists in a nested-loop fashion to select the users of interest and returns the list at the end. Note that the code fragment essentially performs a join between two lists. Had

the programming logic been represented as a SQL query, the DBMS query optimizer would be able to take advantage of the indices and join optimizations in choosing the most efficient way to perform the operation. Unfortunately, executing relational operations in the application forgoes all the optimization machinery that DBMSs provide. Programmers frequently write code like this, perhaps because they lack understanding of DBMS functionality, but more often because DBMS operations are abstracted into libraries, such as ORM libraries. While such libraries greatly ease application development, it is difficult for library developers to anticipate how their libraries are used, and application developers to understand the performance implications of their code. As a result, application performance often suffers.

To improve application performance, we would like application frameworks to automatically convert code from one representation to another (e.g., converting from Java to SQL) while presenting the same high-level programming model to the developer as before. Unfortunately, doing so requires a detailed understanding of the database internals. In addition, converting code representations also requires bridging between the application code that is often written in an imperative language such as Java, and database queries that tend to be expressed using a declarative language such as SQL. We are not aware of any application frameworks that perform such cross-system optimizations.

In this section, we describe QBS (Query By Synthesis) [11], a tool that performs such optimization automatically without any user intervention. Given application source code and database configuration, QBS automatically scans the source code to find code fragments that can potentially be converted into SQL to improve performance. For each code fragment, such as the one shown in Fig. 1(a), QBS transforms it into the code shown in Fig. 1(b), where the nested loop is converted into a SQL join query. QBS improves upon prior work on query extraction [29] by recognizing a wide variety of queries in addition to simple selections. In the following, we describe how QBS works and present representative experiment results.

2.1 Converting Program Representations Automatically

QBS uses program synthesis to bridge the gap between imperative and declarative programming paradigms. Fig. 1(c) shows the overall architecture of QBS. Given the source code of the application and database configuration files (the latter is used to determine the classes of objects that are persistently stored and how such objects are mapped to persistent tables), QBS first performs a pre-processing pass to isolate the code fragments that operate on persistent data and contain no other side-effects, such as the one shown in Fig. 1(a). After that, QBS converts each of the identified code fragments into an intermediate code representation called QIL (QBS Intermediate Language). QIL is similar to relational algebra, but is expressive enough to describe operations on ordered lists (which is the data structure that many ORM libraries and JDBC expose for persistent data manipulation). QIL includes operations such as selecting list elements, concatenating two lists, and joining two lists just like in relational algebra, except that the join operates on ordered lists rather than relations.

The resemblance of QIL to relational algebra allows us to easily convert such QIL expressions to SQL queries. However, we still need to reason about the contents of the variables in the code fragment before converting the fragment into SQL. For instance, in the example shown in Fig. 1(a), we need to formally prove that when the nested loop exits, the contents of ”listUsers” is equivalent to that as a result of joining the ”users” and ”roles” lists. QBS uses Hoare-style program reasoning [17] to come up with such proofs. The idea behind coming up with proofs is that if the developer has labeled the contents of each variable in the code fragment (such as ”listUsers”) in terms of other variables, Hoare-style reasoning provides us a way to formally prove that the annotations are indeed correct, and that the expression in Fig. 2(a) indeed represents the value of ”listUsers” at the end of the execution. The problem, of course, is that no such annotations are present in the code. By making use of this observation, however, we can now frame the problem as a search for annotations that can be proven correct using Hoare-style reasoning. Once the annotations are found, translating an expression like the one in Fig. 2(a) into SQL is a purely syntactic process.

A naive strategy for finding the annotations is to search over all possible QIL expressions and check them

$$\mathsf{listUsers} = \pi(\mathsf{sort}(\sigma(\bowtie(\mathsf{users},\mathsf{roles},\mathsf{True}),f_\sigma),l),f_\pi) \quad \begin{array}{rl} & \text{where:} \\ f_\sigma & := \mathsf{get}(\mathsf{users},\mathrm{i}).\mathsf{roleId} = \mathsf{get}(\mathsf{roles},\mathrm{j}).\mathsf{roleId} \\ f_\pi & := \text{projects all the fields from the User class} \\ l & := [\mathsf{users},\mathsf{roles}] \end{array}$$

$$\mathsf{listUsers} = \left\{ \begin{array}{l} \text{expressions involving operators: } \sigma,\pi,\bowtie,\mathsf{sort} \\ \text{and operands: } \mathsf{users},\mathsf{roles} \end{array} \right\}$$

Figure 2: (a) Specification found by QBS (top); (b) potential QIL expressions for `listUsers` from Fig. 1(a) (bottom).

for validity using Hoare-style reasoning. QBS improves upon this strategy using a two-step process. First, QBS analyzes the structure of the input code fragment to come up with a template for the annotations; for example, if the code involves a nested loop, then the annotations are likely to involve a join operator, so this operator is added to the template. By deriving this template from the structure of the code, QBS significantly reduces the space of candidate expressions that needs to be analyzed. In our example, since objects are inserted into ”listUsers” inside a nested loop involving ”users” and ”roles”, the analysis determines that potential QIL expressions for ”listUsers” would involve those two lists as opposed to others. Furthermore, the ”if” statement inside the loop leads the analysis to add selection (but not projection, for instance) as a possible operator involved in the expression, as shown in Fig. 2(b).

After that, QBS performs the search symbolically; instead of trying different expressions one by one, QBS defines a set of equations whose solution will lead to the correct expression. The technology for taking a template and performing a symbolic search over all possible ways to complete the template is the basis for a great deal of recent research in software synthesis. QBS uses this technology through an off-the-shelf system called Sketch [28], which automatically performs the symbolic search and produces the desired annotations. After producing the annotations, QBS uses them to translate part of the code fragment into SQL queries.

2.2 Experimental Results

We implemented a prototype of QBS using the Polyglot compiler framework [21]. Our prototype takes in Java source code that uses the Hibernate ORM library to interact with DBMSs, identifies code fragments that can be converted into SQL expressions, and attempts to convert them into SQL using the algorithm described above.

We tested the ability of QBS to transform real-world code fragments. We used QBS to compile 120k lines of open-source code written in two applications: a project management application Wilos [4] with 62k LOC, and a bug tracking system itracker [1] with 61k LOC. Both applications are written in Java using Hibernate. QBS first identified the classes that are persistently stored, and scanned through the application for code fragments that use such classes, such as that in Fig. 1(a). This resulted in 49 benchmark code fragments in total. QBS then attempted to rewrite parts of each benchmark into SQL. Fig. 3 shows the number of fragments that QBS was able to convert into relational equivalents. We broadly categorize the code fragments according to the type of relational operation that is performed. While many fragments involve multiple relational operations (such as a join followed by a projection), we only give one label to each fragment in order not to double count.

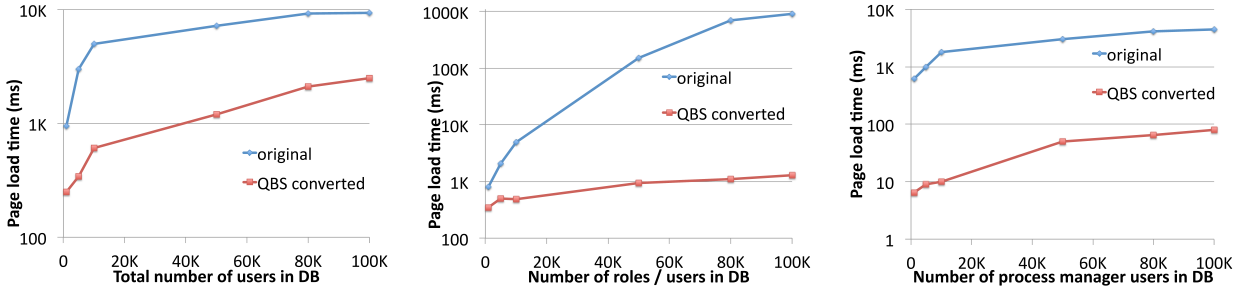
The result shows that QBS is able to recognize and transform a variety of relational operations, including selections, joins, and aggregations such as finding max and min values, sorting, and counting. The slowest transformation of any individual benchmark took 5 minutes to complete, such as the one shown in Fig. 1(a) as it involves join operations, and the average was around 3 minutes. The largest benchmarks involve around 200 lines of source code, and the majority of the time was spent in the annotation search process.

Our QBS prototype currently only handles read-only persistent operations, and as a result cannot handle code that updates to inserts persistent data. In addition, there were a few benchmarks involving read-only operations

Wilos (project management application – 62k LOC)		
operation type	# benchmarks	# translated by QBS
projection	2	2
selection	13	10
join	7	7
aggregation	11	10
total	**33**	**29**

itracker (bug tracking system – 61k LOC)		
operation type	# benchmarks	# translated by QBS
projection	3	2
selection	3	2
join	1	1
aggregation	9	7
total	**16**	**12**

Figure 3: QBS experiment results from real-world benchmarks conversions



(a) Selection with 50% selectivity (b) Join code fragment from Fig. 1(a) (c) Aggregation code fragment

Figure 4: Webpage load times comparison of representative code fragments

that QBS were not able to transform. Some of these benchmarks use type information in program logic, such as storing polymorphic records in the database and performing different operations based on the type of records retrieved. Including type information in ordered lists should allow QBS to process most of the benchmarks. Meanwhile, some benchmarks re-implement list operations that are provided by the JDK (such as sorting or computing the max value) in a custom manner. Extending QIL expressions to include such operations will allow QBS to convert such benchmarks.

We also compared the load times for web pages containing the benchmarks before and after QBS conversion. The results from representative code fragments are shown in Fig. 4, with Fig. 4(b) showing the results from the code example from Fig. 1(a). In all cases, performance improved in part because the application only needs to fetch the query results from the DBMS. Moreover, as in the case of the code example in Fig. 1(a), expressing the program logic as a SQL query allows the DBMS to execute the join using efficient join algorithms, as opposed to executing the join in a nested loop fashion as in the original code, which leads to further speedup. The aggregate code fragments illustrate similar behavior.

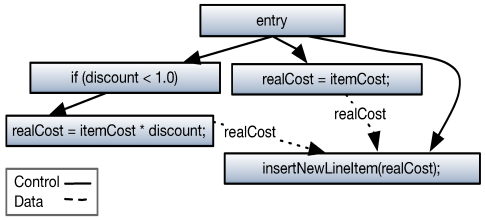
3 Improving Locality With Program Partitioning

In addition to deciding how application logic should be expressed, determining *where* it should be executed is another important factor that affects application performance. *Automatic program partitioning* is a technique for splitting a program into communicating processes that implement the original program in a semantics-preserving way. While prior applications of automatic program partitioning include optimizing distributed applications [18, 20, 31, 13, 14, 5], parallelizing sequential programs [25, 19], and implementing security policies and principles [32, 12, 6, 30], we are not aware of prior work that applies program partitioning to improve database application performance. Partitioning database applications is challenging as the optimal solution often depends on the current server workload. To that end, we developed PYXIS [8], a tool that automatically partitions a database application between the application and database servers, and adaptively changes how the application is split based on the amount of resources available on the database server.

PYXIS consists of two components: a compiler and a runtime Java library. The PYXIS compiler assigns the statements and fields in Java classes to either an application server or a database server. We call the code

```
realCost = itemCost;
if (discount < 1.0)
    realCost = itemCost * discount;
insertNewLineItem(realCost);
```

(a) An example code fragment



(b) Dependency graph

Figure 5: Illustration of the data structures used in program partitioning

and data assigned to a particular host a *partition* and the set of partitions a *partitioning*. Placing code that accesses the DBMS on the database server results in lower latency since it requires fewer network round trips. However, running more code on the database server places additional computational load on the server, which could negatively impact latency and reduce throughput. PYXIS manages this tradeoff by finding partitionings that minimize overall workload latency within a specified budget. This budget limits the impact on server load.

Since the optimal partitioning depends on the current workload, the PYXIS runtime adapts to changing workload characteristics by switching between partitionings dynamically. Given a set of partitionings with varying budgets, PYXIS alleviates load on the database server by switching to a partitioning with a lower budget that places more code at the application server.

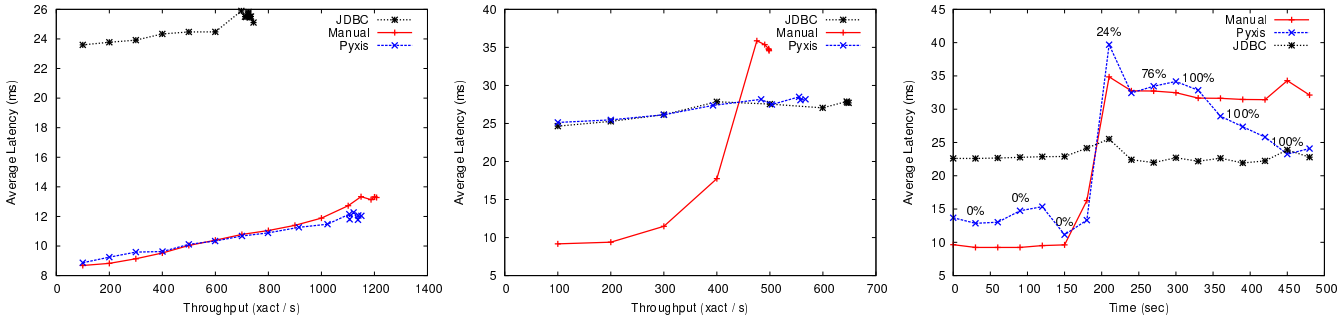
3.1 Analyzing Dependencies in Database Applications

At the core of the PYXIS compiler is a program partitioner. Program partitioning is done using a static interprocedural dependency analysis based on an object-sensitive pointer analysis [27]. The analysis constructs a dependency graph in which each statement of the program is a node and each edge represents a possible dependency between statements. PYXIS distinguishes two kinds of dependencies: control and data. Control dependency edges connect statements to branch points in control flow such as ”if” statements, method calls, or exception sites. Data dependencies connect uses of heap data with their possible definitions. Fig. 5a shows some example code and Fig. 5b its corresponding dependency graph. The statement at line 3 is *control dependent* on the ”if” condition at line 2. The statement at line 4 is *data-dependent* on the two statements that make assignments to ”realCost” at line 1 and 3.

The PYXIS compiler transforms the input program into two separate programs that share a distributed heap. A novel aspect of PYXIS is that the runtime is designed to allow the partitioner to take advantage of the static analysis results to generate optimized code. To this end, all synchronization of the distributed heap is performed explicitly by custom code generated for each application. When executing a block of instructions, all updates to the distributed heap are batched until control is transferred to the remote node. At this point, updates that may be observed by the remote runtime are transferred and execution continues at the remote node. By only transferring the observed updates, PYXIS significantly reduces communication overhead for heap synchronization. The static analysis conducted by the partitioner is conservative, ensuring that all heap locations accessed are guaranteed to be up to date, even though other parts of the heap may contain stale data.

3.2 Partitioning the input program

We now discuss the mechanics involved in partitioning the input program. First, PYXIS gathers profile data such as code coverage and data sizes from an instrumented version of the application under a representative workload. This data is used to weight the dependency graph. We call this weighted dependency graph a *partition graph*. Each statement node in the graph is annotated with how many times it was executed in the workload. Each edge in the graph is annotated with an estimated cost for transferring control or data between the application and



(a) High-budget latency vs throughput (b) Low-budget latency vs throughput (c) Latency over time with switching

Figure 6: TPC-C experiment results on a 16-core database server

database server should the two associated statements be placed separately.

The weights in the partition graph encode the cost model of the PYXIS runtime. Since updates are batched, the weights of data dependencies are relatively cheap compared to control dependencies. Node weights capture the relative load incurred server for different statements—more frequently executed statements typically result in higher associated load.

Next, we determine how to optimally cut the partition graph. A *cut* in the partition graph is an assignment of statements to either the database or the application server. The best cut should minimize the weights of the cut edges while obeying certain constraints. Among these constraints are the budget constraint: the sum of node weights assigned to the database server cannot exceed the amount of resources available on the server. The problem of finding the best cut is translated to a binary integer programming problem and sent to a solver. Once the solver finds a solution, the partitioner outputs the two components that make up the partitioned program. One component runs at the application server and one at the database. Note that all partitionings are generated offline by setting different values for the resource available on the database server.

The partitioned graph is then compiled to code. Statements are grouped into execution blocks that share the same placement and control dependencies. The exit points in each execution block specify a local or remote block where the computation proceeds. For instance, say the call to ”insertNewLineItem” in Fig. 5a is placed on the database server while the other statements remain at the application server. Then, when control is transferred from the application server to the database, the current value of ”realCost” will be transferred as well, ensuring that the call’s parameter is up to date.

3.3 Experimental Results

We built a prototype of PYXIS using Polyglot [21] and the Accrue analysis framework [3]. We have evaluated our implementation on TPC-C and TPC-W. We used PYXIS to create two partitionings—one using a low CPU budget and one using a high CPU budget. We compared the PYXIS generated programs to a traditional client-side implementation that executes all application logic on the application server and uses JDBC to execute queries (referred to as **JDBC**), as well as a version that encodes most application logic in stored procedures that are executed on the database server (referred to as **Manual**).

Our experiments show that PYXIS automatically generated partitionings that were competitive with Manual in performance. In Fig. 6a, a high-budget partition was evaluated in which the database server has extra CPU resources. In this case, the PYXIS-generated program performed similarly to Manual. This is because much of the program code—especially methods with multiple SQL queries—could be placed on the database server, and latency was low. In Fig. 6b, a low-budget partition was evaluated in a context where the database server has limited CPU resources. Here, the PYXIS-generated program performed like JDBC, where all application statements other than database queries are placed at the application server. Latency was higher than Manual at

low throughput, but because less computational load was placed on the database, the PYXIS-generated program was able to achieve a higher throughput.

In general, the availability of CPU resources may change as the workload evolves. Consequently, a particular partitioning may not perform adequately across all workload profiles. Figure 6c shows the results of an experiment that evaluates the ability of PYXIS to adapt to changing resource availability. In this experiment, the load on the database server started out small, but was artificially increased after three minutes. The Manual implementation performed well initially, but its latency exceeded that of the JDBC implementation as the amount of CPU resources on the database server decreased. PYXIS, on the other hand, first chose to serve the incoming requests using a partitioning similar to Manual. But as CPU resources decrease, the PYXIS runtime monitors the load at the server and switches to a partitioning similar to JDBC instead when the load exceeds a threshold. Thus, PYXIS obtains the minimal latencies of both implementations at all times.

A single application may be subjected to several different performance profiles. By examining the DBMS and application server together, we are able to generate program partitionings specialized for each profile using PYXIS. In sum, PYXIS enhances the performance and scalability of database applications without forcing developers to make workload-specific decisions about where code should be executed or data should be stored. It would be interesting to use PYXIS to partition larger programs, and the same technique to other application domains beyond database applications.

4 Batching DBMS Interactions

PYXIS shows that reducing round trips between the application and database servers can improve application performance significantly. *Query batching* is another common technique to reduce such round trips. For instance, many ORM libraries allow developers annotate their code to prefetch extra persistent objects before such objects are needed by the program. Unfortunately, deciding when and what objects to prefetch is difficult, especially when developers do not know how the results that are returned by their methods will be used.

There is prior research that uses static program analysis to extract queries that will be executed unconditionally by a single client in the application [16, 23, 22]. The extracted queries are executed asynchronously in a single round trip when all query parameters are computed [7]. There has also been work on multi-query optimization [15, 26, 24] that focuses on sharing query plans or reordering transactions among multiple queries. However, such approaches aim to combine queries issued by multiple concurrent clients at the database rather than in the application.

Rather than prefetching of query results, where its effect is often limited by the imprecision in static analysis, instead we aim to *delay* queries as long as possible, with the goal to batch the delayed queries so that they can be issued in a single round trip to the DBMS. In this section we describe SLOTH [9], a new system we built to achieve this goal. Instead of using static analysis, SLOTH creates query batches by using *extended lazy evaluation*. 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 printed on the console), 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. In the following we describe SLOTH in detail and highlight some experiment results using real-world database applications.

4.1 Extending Lazy Evaluation for Query Batching

SLOTH builds upon traditional lazy evaluation for query batching. Lazy evaluation, as pioneered by functional languages, aims to delay computation until its results are needed. As mentioned, in SLOTH we extend lazy

```
1  ModelAndView handleRequest(...) {
2  Map model = new HashMap<String, Object>();
3  Object o = request.getAttribute("patientId");
4  if (o != null) {
5    Integer patientId = (Integer) o;
6    if (!model.containsKey("patient")) {
7      if (hasPrivilege(VIEW_PATIENTS)) {
8        Patient p = getPatientService().getPatient(patientId);
9        model.put("patient", p);
10       ...
11       model.put("patientEncounters",
12         getEncounterService().getEncountersByPatient(p));
13       ...
14       List visits = getVisitService().getVisitsByPatient(p);
15       CollectionUtils.filter(visits, ...);
16       model.put("patientVisits", visits);
17       model.put("activeVisits", getVisitService().
18         getActiveVisitsByPatient(p));
19   ...
20   return new ModelAndView(portletPath, "model", model);
21 }
```

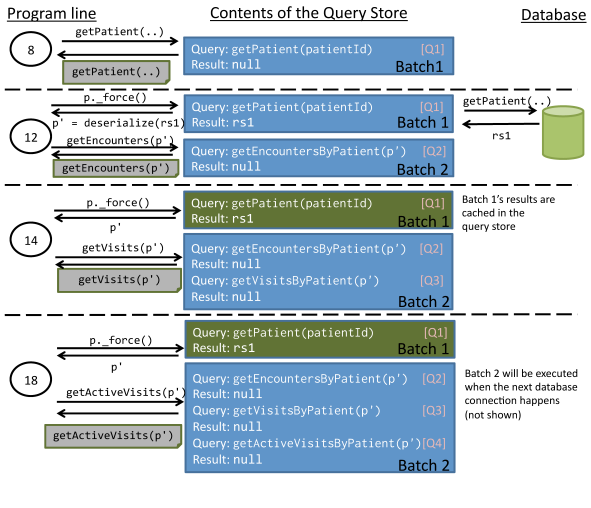


Figure 7: (a) Code fragment abridged from OpenMRS (left); (b) How SLOTH executes the code fragment (right)

evaluation where computation is deferred, except that when queries are encountered they are batched in a query store. When any of the query results are needed, all batched queries are executed in a single interaction with the DBMS. Using extended lazy evaluation allows us to batch queries across conditional statements and even method boundaries without any extra work for the developer, as queries that are accumulated in the query store are guaranteed to be executed by the application. Moreover, the batched queries are *precise* in that they are those that were issued by the application.

To understand extended lazy evaluation in action, consider the code fragment in Fig. 7(a), which is abridged from a real-world Java web application [2] that uses Spring as the web framework and the Hibernate ORM library to manage persistent data.

The application is structured using the Model-View-Control design 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 to view the dashboard for a particular patient. 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 (code not shown).

As written, the code fragment can issue up to four queries; the queries are issued by calls of the form ”getXXX” 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 18 to fetch various data associated with the patient, and adds the 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. Thus, the developer could have collected 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 unfortunately would also impose an unacceptable burden on the developer.

Using SLOTH, the code fragment is compiled to execute lazily, where the evaluation of a statement does not cause it 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. As mentioned, when executing statements

that issue queries, the queries themselves are batched in the query store in addition to a thunk being returned. Meanwhile, 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 or committing a DBMS transaction). 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.

This is illustrated in Fig. 7(b). 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 the figure). Then, 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 the code fragment 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, SLOTH 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 18 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. Using SLOTH, the number of DBMS round trips is reduced from four to one during execution of the code fragment.

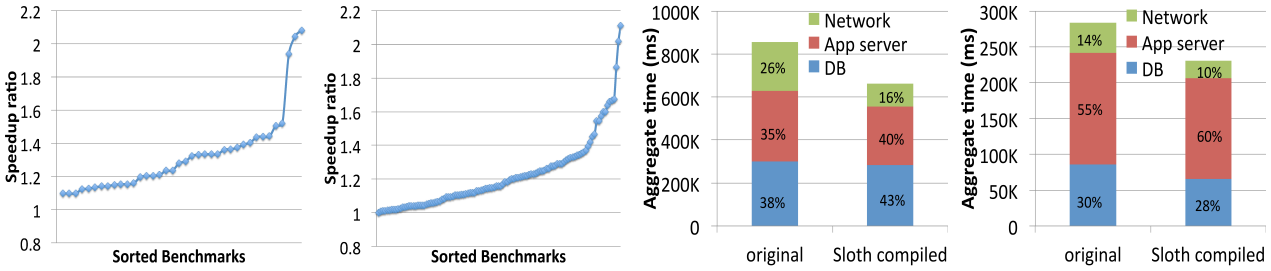
4.2 Evaluating SLOTH

We have built a prototype of SLOTH. The protoype takes in Java source code and compiles it to be executed using extended lazy evaluation. SLOTH also comes with a runtime library that includes the implementation of the query store, along with a custom JDBC driver that allows multiple queries to be issued to the database in a single round trip, and extended versions of the web application framework, ORM library, and application server to process thunks (we currently provided extensions to the Spring application framework, the Hibernate ORM library, and the Tomcat application server.) Note that among the changes to existing infrastructure, only the custom JDBC driver and extended ORM library are essential. The other extensions are included in order to increase batching opportunities.

We evaluated our prototype using two real-world applications: itracker (the same as used in evaluating QBS), and OpenMRS, with a total of 226k lines of Java code. Both applications already make extensive use of prefetching annotations provided by Hibernate. We created benchmarks from the two applications by 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 server. There is a 0.5ms round trip delay between the two machines.

For the experiments, we compiled the two applications using SLOTH, and measured the end-to-end page load times for the original and the SLOTH-compiled benchmarks. Fig. 8(a) and (b) show the load time ratios between the SLOTH-compiled and original benchmarks.

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. There were cases where the total number of queries decreased in the SLOTH-compiled version as they were issued during model creation but were not needed by the view. Most of the



(a) itracker benchmarks load time ratios (b) OpenMRS benchmarks load time ratios (c) itracker benchmarks timing breakdown (d) OpenMRS benchmarks timing breakdown

Figure 8: SLOTH experiment results

speedups, however, resulted from batching multiple queries and reducing round trips to the DBMS (up to 68 queries were batched in one of the benchmarks). To quantify the latter, we tabulated the aggregate amount of time spent in each processing step across all the benchmarks. The results are shown in Fig. 8(c) and (d), which show that the time spent in network traffic was significantly decreased in the SLOTH-compiled benchmarks.

Overall, the experiments show SLOTH improves application performance significantly despite considerable developer efforts spent in annotating objects for prefetching. It would be interesting to combine SLOTH with other prefetching techniques. For instance, SLOTH currently pauses the application in order to execute the batched queries. Alternatively, such batched queries can be executed asynchronously [7] once they are identified in order to avoid pausing the application. It would also be interesting to investigate query rewrite techniques (such as query chaining [23]) for the batched queries rather than issuing them in parallel at the database as in the current SLOTH prototype.

5 Conclusion

In this article, we described three projects that improve the performance of database applications. We showed that by co-optimizing the runtime system and the DBMS, we can achieve a speedup of multiple orders of magnitude in application performance. We believe that the techniques presented are complementary to each other. In future work, we plan to investigate combining these techniques and measuring the overall performance improvement using real-world applications.

The idea of co-optimization across different software systems opens up new opportunities in both programming systems and database research. For instance, it would be interesting to apply program partitioning to other application domains that involve multiple systems, such as partitioning code and data among web clients, application servers, and the DBMS. In addition, techniques used in QBS can also be used to build new optimizing compilers: for instance, converting code to make use of specialized routines such as MapReduce or machine learning libraries.

References

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

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

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

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

[5] R. K. Balan, M. Satyanarayanan, S. Park, and T. Okoshi. Tactics-based remote execution for mobile computing. In *Proc. International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 273–286, 2003.

[6] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proc. USENIX Security Symposium*, pages 57–72, 2004.

[7] M. Chavan, R. Guravannavar, K. Ramachandra, and S. Sudarshan. Program transformations for asynchronous query submission. In *Proc. IEEE International Conference on Data Engineering (ICDE)*, pages 375–386, 2011.

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

[9] A. Cheung, S. Madden, and A. Solar-Lezama. Sloth: Being lazy is a virtue (when issuing database queries). In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2014.

[10] A. Cheung, S. Madden, A. Solar-Lezama, O. Arden, and A. C. Myers. StatusQuo: Making familiar abstractions perform using program analysis. In *Conference on Innovative Data Systems Research (CIDR)*, Jan. 2013.

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

[12] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Building secure web applications with automatic partitioning. *Comm. of the ACM*, (2), 2009.

[13] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: elastic execution between mobile device and cloud. In *Proc. EuroSys*, pages 301–314, 2011.

[14] E. Cuervo, A. Balasubramanian, D. ki Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: making smartphones last longer with code offload. In *Proc. International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 49–62, 2010.

[15] G. Giannikis, G. Alonso, and D. Kossmann. SharedDB: Killing one thousand queries with one stone. *PVLDB*, 5(6):526–537, 2012.

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

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

[18] G. C. Hunt and M. L. Scott. The Coign automatic distributed partitioning system. In *USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 187–200, 1999.

[19] M. G. Nanda, S. Chandra, and V. Sarkar. Decentralizing execution of composite web services. In *Proc. ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 170–187, 2004.

[20] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden. Wishbone: Profile-based partitioning for sensornet applications. In *USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, pages 395–408, 2009.

[21] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: an extensible compiler framework for Java. In *Proc. International Conference on Compiler Construction (CC)*, pages 138–152, 2003.

[22] K. Ramachandra and R. Guravannavar. Database-aware optimizations via static analysis. *IEEE Data Engineering Bulletin*, 37(1), Mar. 2014.

[23] K. 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.

[24] S. Roy, L. Kot, and C. Koch. Quantum databases. In *Conference on Innovative Data Systems Research (CIDR)*, 2013.

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

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

[27] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. In *ACM SIGPLAN Notices*, volume 46, pages 17–30. ACM, 2011.

[28] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodík, V. A. Saraswat, and S. A. Seshia. Sketching stencils. In *Proc. SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 167–178, 2007.

[29] B. Wiedermann, A. Ibrahim, and W. 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.

[30] Y. Wu, J. Sun, Y. Liu, and J. S. Dong. Automatically partition software into least privilege components using dynamic data dependency analysis. In *Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 323–333, 2013.

[31] F. Yang, J. Shanmugasundaram, M. Riedewald, and J. Gehrke. Hilda: A high-level language for data-driven web applications. In *Proc. IEEE International Conference on Data Engineering (ICDE)*, pages 32–43, 2006.

[32] L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic. Using replication and partitioning to build secure distributed systems. In *Proc. IEEE Symposium on Security and Privacy*, pages 236–250, May 2003.