

Speeding up Database Applications with Pyxis

Alvin Cheung
Samuel Madden
MIT CSAIL

{akcheung, madden}@csail.mit.edu

Owen Arden
Andrew C. Myers
Dept of Computer Science
Cornell University

{owen, andru}@cs.cornell.edu

db.csail.mit.edu/pyxis

ABSTRACT

We propose to demonstrate Pyxis, a system that optimizes database applications by pushing computation to the database server. Our system applies program analysis techniques to the application source code to determine pieces of application logic that should be moved to the database server to improve performance. This frees the developer from the need to understand the intricacies of database operations or learn a new programming language for stored procedures. In addition, by dynamically monitoring resource utilization on the database server, Pyxis can migrate computation between application and database in response to workload changes. Our previous experiments have shown that Pyxis can decrease latency up to $3\times$ for transactional applications, and improve throughput up to $1.7\times$ when compared to a standard implementation using embedded SQL statements in application logic. We will demonstrate these capabilities via a visualization of real-time performance as well as an interactive code partitioning tool we have developed.

Categories and Subject Descriptors

H.2.4 [Database Management]: Systems

General Terms

database applications; performance; program optimization

1. INTRODUCTION

Applications that store persistent data in databases are prevalent. Typically, such applications are written in a general-purpose programming language like Java or Python, with SQL statement embedded in the source code. The application is hosted on an application server, while the database runs on a separate database server. For security and performance reasons, the two servers are usually physically separate and dedicated machines residing in the same data center in order to decrease communication latency. Meanwhile, multiple applications are hosted on the same application

server, and the database processes requests from multiple applications.

For transactional applications (such as e-commerce websites) that make many accesses for small amounts of data from the database while processing an incoming request, one common way to improve application efficiency is to reduce the number of data accesses using *stored procedures*, as each data access incurs a network roundtrip between the application and database server. The idea of stored procedures is to group application logic between consecutive data accesses into a new function and install that into the database. Moving computation between the data accesses from the application server to the database server reduces the number of roundtrips between the two servers and thus improves efficiency. Unfortunately, there are a number of pitfalls with this approach:

- Instead of general-purpose languages, stored procedures are often expressed in a database-specific programming language such as PL/SQL. Consequently, developers need to learn a new programming language before rewriting portions of their applications into stored procedures. In addition, developers now need to maintain two different repositories for their applications, one for the stored procedures and another for the application logic.
- Identifying what application logic to convert into stored procedures is not always easy. First, it is not obvious which SQL statements result in the most data being transferred from the database, and this makes identifying stored procedure candidates tricky. Second, transforming code into stored procedures may require passing application state to the database, and passing it back to the application if it is mutated. Doing this correctly can incur substantial amount of code and endless debugging sessions.
- Since server utilization can change over time, moving computation to the database server does not always improve performance. For instance, if the database server is already under heavy load from serving other requests, pushing more computation to it can increase latency and hurt performance. Unfortunately, since server load is difficult to predict, the developer can only assume—and therefore, provision—the application to be operating under a certain load, or alternatively maintain multiple versions of the same application, each one tailored for a different server utilization. Both of these solutions are far from ideal.

To alleviate these challenges, we have developed a system called Pyxis (for a detailed description of the Pyxis system, see our VLDB

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD'13, June 22–27, 2013, New York, New York, USA.
Copyright 2013 ACM 978-1-4503-2037-5/13/06 ...\$15.00.

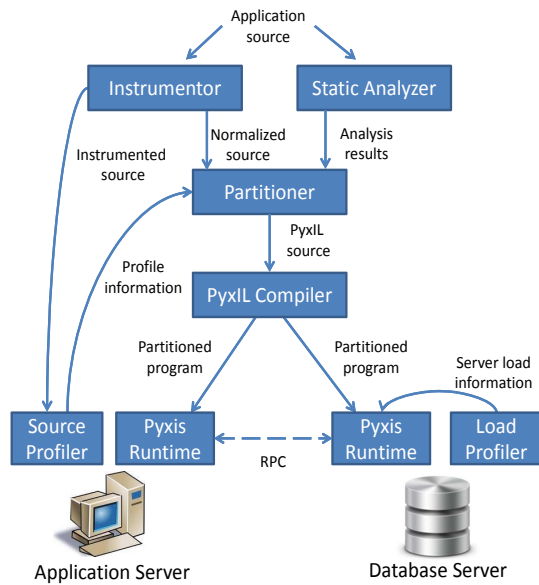


Figure 1: Pyxis system design

2012 paper [1]). Given a Java application with embedded SQL statements for data accesses, Pyxis analyzes the source code and transforms it into a new application that references a generated set of stored procedures. The identified stored procedures are chosen by profiling the application to determine the amount of data transferred by each database access. Because the load on the database may vary over time, Pyxis generates several partitionings based on different assumptions about database server load, and dynamically chooses between them at runtime. All of this is performed without any developer intervention or changes to the source code. In the following, we describe the architecture of Pyxis followed by some experimental results, and then discuss our detailed demonstration plan of Pyxis at SIGMOD.

2. SYSTEM OVERVIEW

In this section we describe the architecture of Pyxis and how Pyxis determines the amount of computation to be pushed to the database server. Fig. 1 shows the overall design of the system. Given the source code of a Java application with embedded SQL commands, Pyxis first benchmarks the application to gauge its workload, then makes use of the collected information along with results from static code analysis to determine an optimal partitioning of the application between the two servers. After that, the partitioned code is compiled and given to the Pyxis runtimes on the servers for execution. In the following, we describe each of these steps in detail.

2.1 Initial Benchmarking and Code Analysis

Given the application source code, Pyxis first instruments, compiles, and deploys the application as-is on the servers in order to collect workload information. The source code is automatically instrumented to collect information such as the number of times that each statement is executed, the network latency between the application and database servers, etc. Such information is needed in order to determine an optimal partitioning of the application source code. Pyxis assumes that there is an underlying pattern in the workload. Should the workload change, the developer can simply ask

the system to benchmark again and generate a new partition of the application.

Meanwhile, the static analyzer component of Pyxis constructs a partition graph of the source code. The partition graph is a type of program dependence graph (PDG) [2]. A PDG is similar to a control flow graph, with nodes representing statements and edges connecting nodes that have control dependencies between them. In addition to control flow edges, however, the graph also records information about definitions and uses of program variables throughout the code. Constructing the partition graph requires having points-to information of the program variables. That information is currently obtained using an object-sensitive analysis [3]. After the partition graph is constructed, it is augmented with data collected during initial benchmarking: each node labeled with the number of times it was executed, and each variable annotated with its size in terms of bytes, as that represents the amount of data that needs to be transferred between the two servers should the partitioned program have a data dependency on that variable.

2.2 Partitioning Source Code

Given the partition graph, the partitioner devises a linear program that will determine the optimal partitioning of the source code when solved. The linear program is formulated by first defining an indicator variable n_i for each node and e_i for each edge in the graph. n_i is set to 1 if the program statement that corresponds to the node is to be executed on the database server (and 0 means that it is to be executed on the application server), and e_i is assigned to 1 if the nodes connected by the edge are assigned on different servers. The objective of the linear program is to minimize the overall latency of the application due to data and control transfers between the servers, i.e. $\sum e_i * w_{e_i}$, where w_{e_i} is a weight representing the latency cost of satisfying a dependency remotely. These weights are set using the collected benchmarking data. To avoid pushing too much computation to the database server, the linear program is subject to an overall budget constraint of $\sum n_i * w_{n_i} < c$, where w_{n_i} is the number of times that statement i was executed during benchmarking, and c is the amount of CPU resources that is available on the database server. Solving the program gives us an assignment of each statement to a server, and Pyxis automatically generates partitions of the program under different values of c in order to accommodate different actual server loads. At the end of the process, the partitioner generates a PyxIL (Pyxis Intermediate Language) program, which has the same syntax as the original Java source, except that each statement is annotated to indicate which server it is intended to be executed on.

2.3 Executing Partitioned Code

After the partitioning process, the PyxIL compiler takes in the generated program to produce two Java programs to be executed on the runtime components residing on the two servers. The two compiled programs operate on a distributed heap, and program control migrates between the two during execution. By communicating via the runtime components on the servers to pass control and data, the two Java programs implement the semantics of the original application.

Compilation of PyxIL programs is mostly straight-forward, except that for ease of maintaining transfer of control between the two servers during execution, the input source is compiled to continuation passing style [4]. In addition, the compiler also inserts appropriate method calls in the two compiled programs to keep the distributed heap in sync. For instance, if the partitioner determines that program statements s_1 and s_2 are to be executed on the two servers respectively, and that s_2 uses a variable v that is defined by

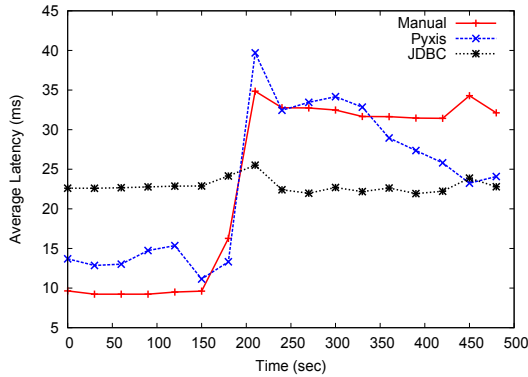


Figure 2: Dynamic switching experiment results

s_1 , then the compiler will insert a method call to forward the value of v before the transferring control to the remote servers.

The output is then compiled using a standard Java compiler and the results are sent to the runtime components for deployment. The Pyxis runtime is simply a Java program running on a standard JVM on the two servers. The runtime on the database server, however, includes a monitor component that periodically samples the amount of resources that are available on the database server. As mentioned in Sec. 2.2, the partitioner generates multiple versions of the application assuming different CPU loads on the database server. Using the real-time information from the monitor component, the runtime dynamically switches among the different pre-generated partitions based on the current server load in order to achieve optimal application performance.

2.4 Experiments

We have previously measured the resulting programs generated by Pyxis under different scenarios [1]. As an example, we implemented the TPC-C benchmark in Java using embedded JDBC SQL statements hosted on an application server that issues queries to the database server with of 16 CPU cores. There is a 0.2ms roundtrip network latency between the two servers. We executed a workload consisting of 20 warehouses and 20 clients issuing new order transactions simultaneously with a combined rate of 500 transactions per second. We chose a workload of new order transactions as that one is the most complex in terms of lines of code and involves the most number of queries. We measured the average latency of processing transactions over a 10-minute period.

To measure the effectiveness of Pyxis in improving application performance, we first asked the system to generate a partition of the application assuming that the application has full access to all 16 cores of the database server. After that, we asked the system to generate another partition for the case when only half of the cores available. During the experiment, we first allow the application to use all cores that are available on the database server, then about three minutes into the experiment, we reduce the number of cores available to approximately half. Fig. 2 shows the measured latency over the 10 minute period. For comparison purposes, we include two other reference implementations of the same benchmark: JDBC, which is the original version of the application with embedded JDBC SQL statements; and Manual, which is a custom “stored-procedurized” version of the benchmark where most of the application logic is manually rewritten to execute on the database server rather than the application server.

The results illustrate the ability of Pyxis to achieve optimal ap-

plication performance. They demonstrate Pyxis runtime’s ability to choose the optimal partition given the current server utilization. When there is sufficient amount of CPU resources available on the database server in the beginning of the experiment, the Pyxis runtime chooses a partition that resembles the Manual implementation, where most of the application logic is executed on the database server. On the other hand, when only limited resources are available, the runtime automatically switches to process incoming transactions using a partition where most of the application logic on the application server instead. In comparison, the JDBC implementation maintains high latencies throughout the experiment due to the excess number of network roundtrips incurred during transaction processing. Meanwhile, the Manual implementation incurs low latency in the beginning when resources are plentiful, but latency increases as the database becomes heavily utilized. In contrast, Pyxis is able to capture the “best of both worlds” by first choosing a partition that resembles the Manual implementation when resources are available, and then switching to a JDBC-like implementation when resources are limited. We have performed similar experiments using the TPC-W benchmark with similar results.

3. DEMONSTRATION SCENARIO

We have implemented a prototype of the Pyxis system. The system currently takes in source code of applications written in Java that uses JDBC to connect to the database server for queries. We plan to demonstrate the end-to-end ability of Pyxis to automatically achieve the optimal application performance under different server load conditions. From the application source code, the audience will be able to see Pyxis in action by viewing how the system partitions the source code under different server load conditions, and how the runtime component dynamically chooses among the pre-generated partition to serve incoming requests. To that end, the demonstration will consist of two components:

Code visualization. We will provide a tool to visualize the application source code as a result of code partitioning. As shown in Fig. 3, given a CPU resource budget and benchmarking data (as shown in the left panel of Fig. 3), the tool will send the source code to the Pyxis partitioner to generate a partition between the application and database servers. The tool will then show the results of the partition by highlighting the original application source code based on where each statement is allocated to.

Live system demonstration. The second component of our demonstration will consist of a running OLTP-style application (such as TPC-C or TPC-W) that has been deployed using Pyxis on the application and database servers. We will pre-generate a set of partitions using Pyxis with different CPU budgets, and allow the Pyxis runtime to choose which partition to serve the incoming requests.

For comparison purposes, we will also provide reference implementations of the same application deployed as-is, and another implementation where most of the application logic is moved to the database server, similar to the Manual implementation in Sec. 2.4. Each version of the application will be running on a separate set of application and database servers.

To monitor the performance of the three versions of the application, we will demonstrate a monitoring tool (a screenshot is shown in Fig. 4). The tool will plot the average latencies in serving the transactions that finished within the last few minutes. The tool is connected to the database server and can control the amount of CPU resources that are available dynamically. For the Pyxis deployed version of the application, the tool will also show the percentage of transactions that are served using the different pre-

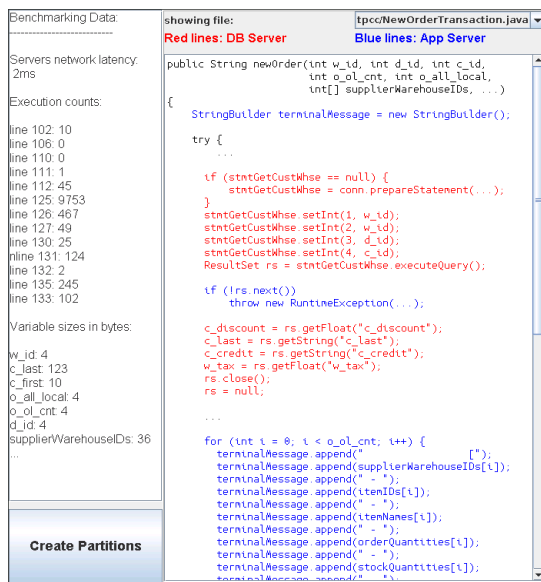


Figure 3: Screenshot of the code visualization tool

generated partitions of the source code, along with the monitoring statistics collected by the runtime component to decide which code partition to use for new incoming requests.

3.1 Storyline

Using the tools above, we plan to walk through the audience in the following manner:

Introduction. We will first explain to the audience our system using a set of slides and poster. The goal is for the audience to understand the challenges in developing efficient database applications, the Pyxis system design, and how our system can help in this regard.

Code partitioning demonstration. Next, we will demonstrate the partitioner component of Pyxis by showing code partitioning in action. We will provide the source code of a transactional database application and the benchmarking data that are previously collected. The audience will be able to see the partition that is generated by the partitioner component using the code visualization tool. In addition, the audience will be able to interact with the tool by changing the amount of CPU resources that are available on the database server, and asking the partitioner to generate new partitions.

Runtime system demonstration. Finally, we will demonstrate our live application deployment. Using the monitoring tool, the audience will get a sense of how the Pyxis runtime component is able to make the optimal decision in deciding the partition to use when an incoming request arrives. To demonstrate the dynamic switching of the runtime component, the audience can interact with the tool by changing the amount of CPU resources that are available on the database server. The audience will see that the Pyxis deployed version of the application is able to achieve the lowest latencies across different CPU utilization conditions, whereas the reference implementations perform optimally only under one specific load conditions.

3.2 Take-away Message

Using the demonstration, we would like to convey the following ideas to the audience:

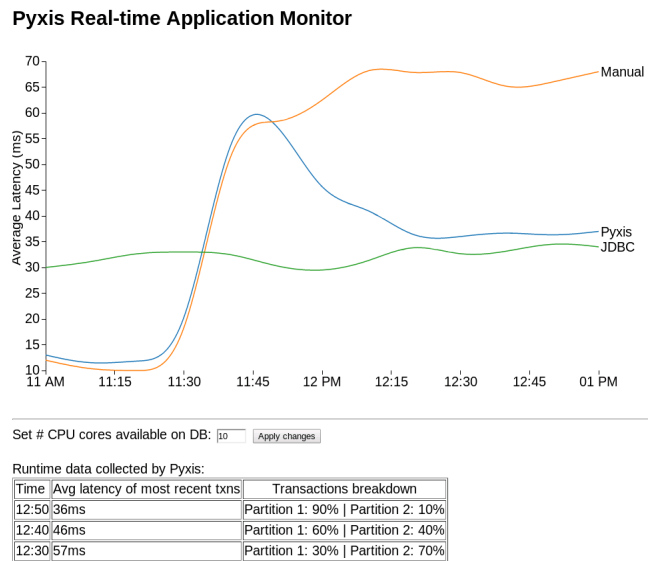


Figure 4: Screenshot of the application monitoring tool

- Our Pyxis prototype is effective in improving database application performance. The system is able to automatically determine the best way to deploy a database application to minimize the latency in processing transactions.
- Our system is able to perform optimization in a completely automatic manner, without any user intervention, changes to the source code, or need to deploy any custom language runtimes or database systems.
- The idea of combining static program analysis and dynamic runtime monitoring in improving application performance, and how similar ideas can be applied in increasing programmers' productivity and security of database applications.

4. ACKNOWLEDGMENTS

The authors are grateful for the support of Intel Corporation, NSF grants 1065219 and 0964409, and ONR grant N00014-13-1-0089. Owen Arden is supported by the Department of Defense through the National Defense Science & Engineering Graduate Fellowship Program.

5. REFERENCES

- [1] A. Cheung, O. Arden, S. Madden, and A. C. Myers. Automatic partitioning of database applications. *PVLDB*, 5(11):1471–1482, 2012.
- [2] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.
- [3] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. *ACM SIGPLAN Notices*, 46(1):17–30, 2011.
- [4] The SML/NJ Fellowship. *Standard ML of New Jersey*. <http://www.smlnj.org/>.