StatusQuo: Making Familiar Abstractions Perform Using Program Analysis

Alvin Cheung
Samuel Madden   Armando Solar-Lezama
MIT CSAIL

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

db.csail.mit.edu/statusquo

ABSTRACT

Modern web applications rely on databases for persistent storage, but the strong separation between the application and the database layer makes it difficult to satisfy end-to-end goals, such as performance, reliability, and security. In this paper, we describe StatusQuo, a system that aims to address this problem by using program analysis and program synthesis to enable the seamless movement of functionality between the database and application layers. It does so by taking the functionality that was originally written as imperative code and translating it into relational queries that execute in the database. In addition, it makes runtime decisions about the optimal placement of computation, in order to reduce data movement between the database and application server. In this paper, we describe promising empirical results from the two key technologies that make up StatusQuo, and highlight some open research problems that will need to be addressed in order to achieve the end-to-end vision.

1. INTRODUCTION

The relational database is the *de facto* persistent storage mechanism for web applications. Because of the way that most web applications are developed, there is a separation between application logic—typically written in a general purpose, imperative language such as Java or Python—and data access logic—typically expressed declaratively in SQL and embedded into the application. The separation can be physical as well, as the application and database logic are often run 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. 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 reorganize their code to move computation and data across the application/database boundary. In that respect, database programmers define stored procedures that implement imperative logic inside the database to reduce communication between the database and application servers, or rewrite imperative logic into SQL queries to allow the query optimizer to efficiently join or filter data sets. Handcrafting such optimizations can yield order-of-magnitude performance improvements, but at the cost of code complexity and readability. Worse, these optimizations can be brittle. Relatively small changes in control flow or data sizes could have unpredictable consequences and cause the optimizations to hurt performance. 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.

Reliability and security are also affected by the logical separation between the database and application. For example, the co-existence 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" arising from the logical and physical separation between the two. This was a goal of the object-oriented database (OODB) movement [5, 13], which has seen a resurgence in recent years due to the widely used object-relational mapping (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. ORM libraries aim to provide a unified, often imperative, programming model where the developer can express both the application logic and data accesses, much as object-oriented databases provide constructs beyond standard SQL to allow developers to easily move data between the imperative application world and the declarative database domain. ORM frameworks simplify database application development by hiding the separation between the application and the database, but in many cases make performance problems worse. For instance, 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 contrast to ORMs and OODBs, we are developing a new programming system, StatusQuo[1], which takes a different approach. We explicitly avoid forcing developers to use any one programming style. Instead, we allow them to use the programming style—imperative, declarative, or mixed—that most naturally and con-

6th Biennial Conference on Innovative Data Systems Research (CIDR 13)
January 6-9, 2013, Asilomar, California, USA.

[1]Because we preserve it.

cisely 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.

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 for program analysis and transformation which ensure that programs continue to function correctly even as they are substantially transformed. Our current StatusQuo prototype works for applications written in Java that access the database with embedded SQL via the JDBC interface or the Hibernate ORM framework. However, we believe our approach should be applicable to other web application programming environments, including Django, Rails, and even higher-level environments such as Hadoop.

Key features of StatusQuo include:

- A novel program analysis that identifies blocks of imperative logic that can be translated into relational expressions in SQL. This transformation moves application logic closer to the data and allows the database query optimizer to choose the best implementation for the queries.

- A fine-grained program partitioning method to move imperative logic into the database in the form of stored procedures. This partitioning is generated to minimize data movement within the system, and can be changed adaptively at runtime based on server load.

The rest of this paper describes StatusQuo in more detail. We begin with an overview of the architecture of StatusQuo in Sec. 2. Our methods for automatically extracting SQL queries from Java are sketched in Sec. 3. Sec. 4 discusses our implementation of automatic program partitioning. Some future extensions are suggested in Sec. 5, and related work is covered in Sec. 6. We conclude in Sec. 7.

2. SYSTEM DESIGN

Figure 1 shows the design of StatusQuo, which consists of the code transformer and runtime modules. Given the application source code, StatusQuo first automatically instruments it and executes the application on the servers for a short period of time (not shown in the figure) to collect a workload profile that captures information such as execution frequency for methods and loops. After that, the code is passed to the SQL extractor component, which uses the QBS (Query By Synthesis) algorithm to convert specific blocks of imperative code into declarative form. 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 the conversion process, the source is given to the partitioner. The partitioner uses the workload data to split the source

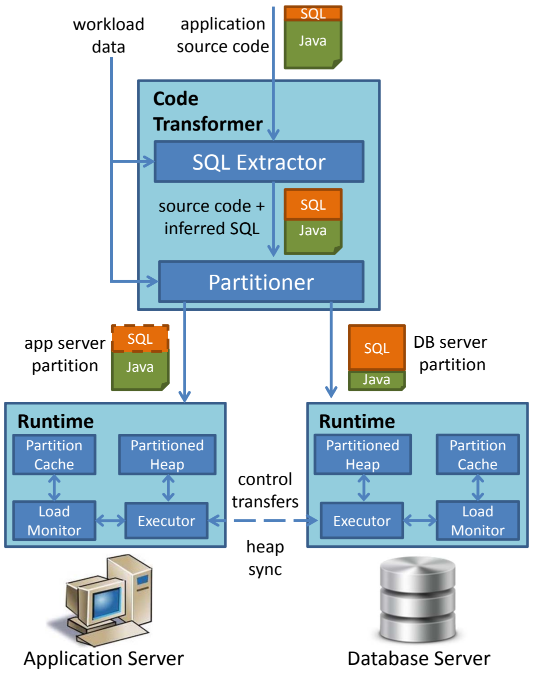


Figure 1: StatusQuo system design

code into two programs: one to be executed on the database server as stored procedures, and another to be executed on the application server. The goal of partitioning is to reduce the amount of data transferred and the number of round trips between the two servers. For declarative code fragments, the partitioner may decide to execute on the database as SQL queries or convert them 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. 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 initially generates several different partitions under a range of load conditions, with the partitions differing in the amount of program logic allocated to each server.

The initial partitions are then sent to the runtime module on each server to execute. In the runtime, the controller component selects a partition from the set of available partitions based on the current server load. During program execution, the executors periodically communicate with each other to transfer the control flow of the program (called *control transfers*) and exchange heap data. Meanwhile, the monitoring component measures the load on the servers. Should any of the servers exceed or otherwise deviate from the workload that the current partition was designed for (e.g., caused by other applications that are hosted on the same machine or changes in workload), the executor informs the controller to dynamically switch to another partition that is better suited to the new load. The controller checks to see if such a partition exists in the partition cache, and if not, it contacts the code transformer module to generate a new partition. If the workload has changed significantly, the code transformer may collect new workload information before generating a new partition.

We have implemented a prototype of the profiler, the code transformer that converts imperative code into SQL expressions, the partitioner, and the runtime modules of StatusQuo, which we describe

```
List<User> getRoleUser () {
  List<User> listUsers = new ArrayList<User>();
  List<User> users = ... /* Hibernate query */
  List<Role> roles = ... /* Hibernate 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 2: (a) Real-world code example that implements join in Java (top); (b) converted version using QBS (bottom)

in the next two sections. We plan to extend the system with the ability to generate new partitions during execution time as described earlier.

3. CONVERTING JAVA TO SQL

As mentioned in Sec. 1, current frameworks do not allow cross-system optimizations between the application and database, and developers need to commit at development time as to how and where computation should take place. For example, the code shown in Fig. 2(a) is abridged from a real-world open source application developed using Hibernate. The nested loops implement a join between two lists of persistent data. Had the operation been expressed in SQL, it would have been executed in the database, where the query optimizer could have chosen the most efficient way to compute the join, but the fact that it was expressed as imperative code forces it to run on the application server and puts the operation outside the reach of the query optimizer. StatusQuo uses a technology called QBS to extract relational queries from code like that in Fig. 2(a). This allows functionality to be pushed to the database and optimized by the query optimizer even if it was originally written as imperative code.

The QBS algorithm [9] is part of the SQL extractor component in StatusQuo. Given an imperative code fragment that uses persistent data, the algorithm applies program analysis and synthesis technologies to try to extract a relational specification. For instance, using the algorithm, the translator converts the code from Fig. 2(a) into the SQL expression shown in Fig. 2(b). The challenge in translating Java code fragments into SQL expressions is to bridge the semantic divide between imperative and declarative code, as discussed next.

3.1 QBS at a Glance

The first step in bridging the gap between the imperative code and SQL is to isolate imperative code fragments that manipulate lists of persistent data and contain no other side-effects. The code identification is achieved by a preprocessing step that uses standard program analysis techniques. The next step is to define a notation that can represent SQL queries, but is also rich enough to describe the intermediate values computed by the imperative code. Our notation, called QIL, is similar to relational algebra, but uses ordered lists rather than multisets. Using lists is necessary because ORM

$$\mathsf{listUsers} = \pi(\mathsf{sort}(\sigma(\bowtie(\mathsf{users}, \mathsf{roles}, \mathsf{True}), f_\sigma), l), f_\pi)$$

where:

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

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

Figure 3: (a) Specification found by QBS (top); (b) potential QIL expressions for `listUsers` from Fig. 2(a) (bottom).

frameworks (and JDBC) expose persistent data using an ordered list interface, so it is important to be able to reason about the order of the rows in the computed relations in addition to their contents.

QBS uses a new approach based on software synthesis to infer a QIL specification for the given imperative code. To illustrate the idea, consider the code fragment from Fig. 2(a). The central observation behind the algorithm is that if someone were to annotate this code with a QIL description of the contents of `listUsers` like the one in Fig. 3(a), together with some QIL descriptions of the intermediate values computed after every iteration of the loops, it would be possible to use Hoare-style reasoning [17], a classical verification technique for program specifications, to prove that the annotations are indeed correct, and that the expression in Fig. 3(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. 3(a) into SQL is a purely syntactic process.

A naïve strategy for finding the QIL annotations is to search over all possible QIL expressions and check them for validity using Hoare-style reasoning. QBS improves upon this strategy by following a two-step process. In the first step, the algorithm 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, the algorithm 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 aggregation, for instance) as a possible operator involved in the expression, as shown in Fig. 3(b).

In the second step, the algorithm performs the search symbolically; this means that instead of trying different expressions one by one, the algorithm 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 the possible ways to complete the template is the basis for a great deal of recent research in software synthesis. Our system uses this technology through an off-the-shelf system called Sketch [28], which automatically performs the symbolic search and produces the desired annotations.

3.2 Experimental Results

We have implemented a prototype of QBS in StatusQuo using the Polyglot compiler framework [26]. The translator currently takes in code fragments written in Java that uses the Hibernate ORM framework, and attempts to convert the input into a SQL expression.

Wilos (project management application)		
operation type	# benchmarks	# translated by QBS
projection	1	0
selection	12	4
join	7	7
aggregation	12	10
total	**32**	**21**

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

Figure 4: QBS experiment results using real-world benchmarks

We tested the ability of QBS to transform real-world code fragments. We scanned through 120k lines of open-source code written in two applications: a project management application [3] with 62k LOC, and a bug tracking system [2] with 61k LOC. Both applications are written in Java using Hibernate. We identified classes that are mapped to persistent storage, and randomly selected code fragments (such as Fig. 2(a)) that use such classes. This resulted in 49 benchmark code fragments in total. We passed each benchmark to the QBS prototype, and Fig. 4 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.

The experiment shows that our prototype 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. 2 as that involve join operations, and the average was around 3 minutes. The largest benchmarks involve around 30 lines of source code, and the majority of the time was spent in the synthesis process. Our current prototype only handles read-only persistent operations, and as a result cannot handle a few benchmarks. In addition, QBS was unable to transform benchmarks that 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 relations should allow QBS to process most of the benchmarks.

4. AUTOMATIC CODE PLACEMENT

In this section, we describe how StatusQuo takes a Java program with embedded relational expressions (some of which may have been generated by the SQL extractor) and splits it into a program that runs as stored procedures on the database server and a modified program that runs on the application server.

As noted in the introduction, the performance of database applications is closely coupled to the way queries are issued to the database. Experts must learn to avoid common pitfalls like fetching excessive data with broad queries or causing network delays by issuing many independent queries. Carefully crafted queries as well as mechanisms like stored procedures help optimize performance, but are difficult to maintain and complicate the application's design. Furthermore, these solutions may not perform well under all workloads and could require refactoring as workloads evolve.

Instead, our goal is to allow developers to write code without having to think about its distributed implementation. Therefore StatusQuo must seamlessly and adaptively move code between the application and database servers. StatusQuo achieves this by converting the original program into a distributed one to be executed

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

Figure 5: An example code fragment

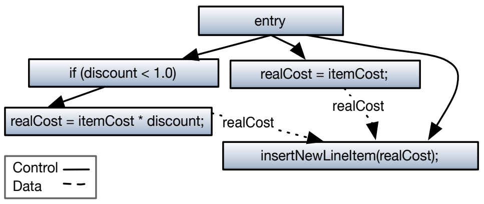


Figure 6: Dependency graph

on the two servers using a distributed heap. This automatic partitioning allows developers to think about database applications at a higher level of abstraction, but it creates three challenges:

Code placement. StatusQuo automatically decides whether each statement in the program should be executed on the app server or the database server. This code placement partitions the original application code into two distributed programs that communicate when control is transferred from one side to the other. Because these control transfers introduce network latency, they must be placed carefully to avoid unnecessary delays.

Data synchronization. The imperative code running on each server should have the same, consistent view of the program heap. For example, if the code that is moved to the database updates a heap location, any code that accesses the modified location later on at the application server must see the update.

Simple strategies for keeping the application and database heaps synchronized incur unacceptable performance penalties. Eagerly synchronizing the entire heap after every update would lead to very high data transfer overhead. Conversely, querying the other server for possible updates to each location only at the time it is used would introduce intolerable communication delays.

Adaptive partitioning. Dynamic metrics like the number of times a statement executes or the size of a data structure are very useful for deciding where to place code. Unfortunately, these properties change over time—even within the same day—meaning that for many workloads, a single optimal placement doesn't exist. For good performance, the partitioning of code in StatusQuo must adapt dynamically to the current workload.

4.1 Approach

StatusQuo employs the approach taken in Pyxis [8], which addresses the above challenges with a combination of static and dynamic program analysis. The system works in the following way. First, the internal dependencies of the application are analyzed using a series of static analyses that leverage an inter-procedural pointer analysis. These dependencies are used to build a *partition graph* that precisely captures the dependencies between statements without introducing unnecessary dependencies.

The partition graph extends the well-known dependency graph data structure [15] with cost information that guides the partitioning process. Like a dependence graph, the partition graph has a node for each statement and edges representing data and control dependencies. Figure 5 shows an example code fragment; its corresponding dependency graph is shown in Fig. 6. The data edges in the graph show that the value of the variable `realCost` could be determined by either of the two assignments.

Edges in the partition graph have weights that represent an estimate of the latency that will be incurred if the source and target statements are assigned to different hosts. Nodes also have weights that represent a statement's contribution to CPU load. Frequently executed statements have higher weights. Edge and node weights are set using the metrics collected through dynamic profiling either beforehand (as in Pyxis) or at runtime.

The weighted partition graph functions as a cost model for potential partitions. Given a placement of statements onto network hosts, a partition can be thought of as cutting all the graph edges connecting statements on different hosts. The sum of the weight of the cut edges is then a measure of the expected latency due to network messages. StatusQuo finds an optimal placement for program statements by encoding the partition graph as an integer programming (IP) problem. An IP solver attempts to find a placement that minimizes the sum of weights of the cut edges but does not exceed a budget constraining the increased CPU load on the database, and multiple partitionings can be generated by assuming different amount of CPU resources that are available on the database server.

When a solution is found, StatusQuo generates two programs that implement the original application, one to be run on the application server and one on the database server. 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. 5 is placed on the database 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.

4.2 Runtime Support

The programs that are generated by the StatusQuo partitioner are compiled and then executed by the StatusQuo runtimes on the two servers. During execution, program control is transferred between the two runtimes as needed. As mentioned, the two programs execute on a distributed heap that is maintained by the runtimes. During compilation, the partitioner automatically inserts heap sync instructions in the code to indicate the program variables whose values need to be forwarded to the remote runtime. However, rather than forwarding the values immediately when such heap sync instructions are encountered during execution, the runtime instead records such values and forwards them in a batch when the next control transfer occurs. Batching the heap updates reduces the amount of communication between the two servers and decreases application latency.

StatusQuo adapts to changing workloads using runtime monitoring to select a partition dynamically. As mentioned in Sec. 4.1, several partitions are created by assuming different CPU load budgets on the database server. At runtime, the system monitors load on the database server and selects a partition that best utilizes the database's CPU without overloading it. Our implementation generates a number of partitions ahead of time. A future extension is to generate the partitions dynamically and perform code swapping on the fly.

4.3 Experimental Results

We tested our implementation on common benchmarks such as TPC-C and TPC-W. We created two program partitions—one using a low CPU budget and one using a high CPU budget. We compared the generated programs to a traditional client-side implementation as well as a version with manually created stored procedures.

Our experiments showed that StatusQuo-generated programs are competitive with manually created implementations under a range of performance profiles. Figure 7(a) shows the results of an experiment in which a high-budget program partition in run when the database has extra CPU resources. The high-budget program had similar performance to the manually created stored procedure implementation. Since more code could be placed on the server, latency was low and CPU utilization at the database was higher. The low-budget case is shown in Fig. 7(b). Here, a low-budget partition was run against a database server with a high CPU load. The StatusQuo-generated program is similar to the client-side implementation, where all application logic, except for database queries, is placed on the application server. Latency was higher, but less load was placed on the database, and as a result a higher throughput was achieved as compared with the stored procedure version.

When CPU load on the database server changes dynamically, however, neither the client-side nor the stored-procedure implementation perform adequately. Figure 7(c) shows an experiment where the load on the server starts out low, but increases significantly after three minutes as artificial load is introduced on the database server. Although the manually created stored procedures perform well initially, after the load increases, the database server's CPU becomes saturated and latency increases. In contrast, the StatusQuo runtime monitors the load at the server and switches to the lower-budget partition when the load exceeds a threshold.

In general, many different performance profiles may exist for an application. By automatically creating program partitions tuned for each profile, StatusQuo enhances the performance and scalability of database applications without increasing development complexity. More details about our experimental setup can be found in [8].

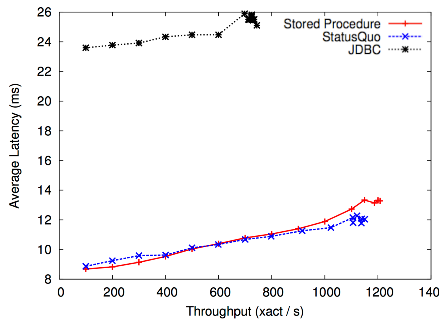
5. FUTURE WORK

We have shown how program analysis and synthesis techniques can be used to migrate from imperative code to declarative queries, and move code from the application to the database. However, more can be done to seamlessly integrate computation and data across application and database servers.

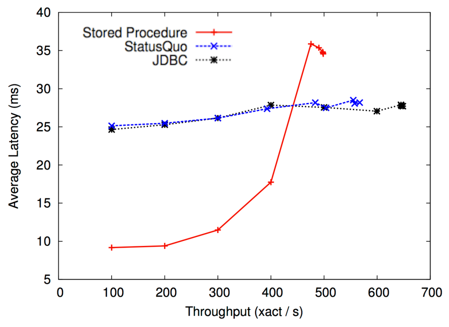
5.1 Flexible Data Placement

The techniques discussed thus far allow data and computation to be pushed from the app 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 app server.

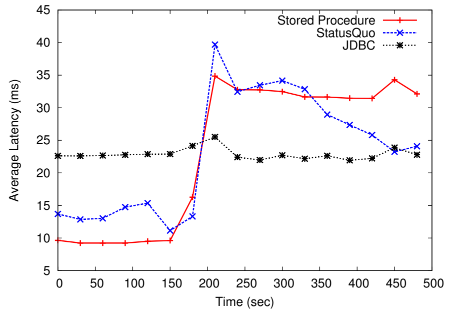
Although web caches such as Memcached have similar goals, they are generic object stores that require manual customization to implement application-specific caching policies. By contrast, program analysis and transformation techniques like those described in previous sections should make it possible 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 [21] and Fabric [22] 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 app server. Since declarative queries do not have side effects, the partitioning problem, at least, is easier than with imperative code. Second, since app and database servers have different recovery models, it is difficult to move computation automatically while seamlessly preserving con-



(a) High-budget latency vs throughput



(b) Low-budget latency vs throughput



(c) Latency over time with dynamic switching

Figure 7: TPC-C experiment results on 16-core database server

sistency guarantees. However, we believe that program analysis can be used to automatically identify exception-triggered imperative recovery code and to integrate its actions with transactional rollback mechanisms.

5.2 Multiple App Servers and Back Ends

Applications that access multiple back-end database servers 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 app servers. While the work on StatusQuo has thus far been in the context of a single app server connecting to a single back-end database, 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 that we describe here poses interesting problems in how to express and constrain concurrency both among the app servers and among the back-end databases.

5.3 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, [1, 27]). The mismatch between these mechanisms and database access controls already makes security enforcement awkward and error-prone. Since StatusQuo 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 database server, 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. 5.1, 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 [22, 10, 7]. In particular, the approach taken in the Fabric system might be applicable to StatusQuo. 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. RELATED WORK

ORM libraries are not the only means for database application development. Database applications are traditionally developed using interfaces such as JDBC that are provided by the application language runtime. In recent years, there has been work on query integrated languages such as LINQ [24], along with systems that introduce new programming constructs into application programming languages for accessing persistent data, such as JReq [20] and Links [12]. Unfortunately, these systems still require developers to learn new programming models for accessing persistent data. StatusQuo is the first proposal we know of that presents a holistic vision from initial application development to final deployment and maintenance, and proposes to do so without requiring the developers to learn new programming models or deploy their applications on custom database or runtime systems.

QBS [9] is inspired by earlier work on query extraction [29]. The idea in the earlier work is to first infer the persistent data access paths in the code and record any branch conditions incurred along the way, and then convert the data access paths into SQL queries with the branch conditions as selection predicates. It is not clear how this approach can be extended to handle joins and aggregates, as joins require combining multiple data access paths and aggregates often involve loop-carried dependencies [4]. In the latter case the value of the aggregate is modified inside a loop, and query inference will need to reason about the relationship between modified value and the lists that contain persistent data. QBS substantially extends previous work and demonstrates applicability to real-world applications by being able to process joins and aggregates, and StatusQuo makes use of the QBS algorithm as a building block of the end-to-end system.

Toolkits like the Google Web Toolkit (GWT) [16], Volta [23], and Fabric [22] aim to make manual program partitioning easier by abstracting away details of distributed computation like network communication and asynchronous execution. These techniques are distinct from automatic program partitioning; they still require the programmer to specify code placements explicitly.

Pyxis is the first system that automatically partitions database applications written in general-purpose languages between application and database servers. Other systems such as Hilda [30], Wishbone [25], DryadLINQ [19], and MapReduce [14] implement relational operator placement optimizations across multiple servers, but require developers to express program logic in custom programming models and do not consider the amount of data transferred between servers.

Program partitioning has been studied previously in the programming languages community, in systems such as Swift [11], Chrome [6], and Coign [18]. Prior systems focus on different aspects such as offloading computation to back-end servers or security, and do not consider network latency between servers or data transfer size. Furthermore, Pyxis partitions source code in a fine-grained manner, whereas most prior work partitions code at method and class boundaries.

7. CONCLUSIONS

StatusQuo is a novel end-to-end solution that aims to ease database application development. We described two key technologies of the system, QBS and Pyxis, that enable automatic transformation, deployment, and adaptive maintenance of database applications for optimal performance. We demonstrated the feasibility of StatusQuo on different real-world applications, and outlined some future challenges in extending the system, such as optimizing for multiple back ends and ensuring end-to-end system security.

8. ACKNOWLEDGMENTS

Owen Arden is supported by the Department of Defense through the National Defense Science & Engineering Graduate Fellowship Program. This research is supported by ONR grant N000140910652 and NSF grant 0964409.

9. REFERENCES

[1] Enterprise Java Beans Security Specification, §17 of JSR 220: EJB Core Contracts and Requirements.

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

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

[4] J. R. Allen and K. Kennedy. Automatic loop interchange. In *Proc. SIGPLAN Symp. on compiler construction*, pages 233–246, 1984.

[5] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proc. International Conference on Deductive Object Oriented Databases*, December 1989.

[6] R. K. Balan, M. Satyanarayanan, S. Park, and T. Okoshi. Tactics-based remote execution for mobile computing. In *Proc. MobiSys*, pages 273–286, 2003.

[7] W. Cheng, D. R. K. Ports, D. Schultz, V. Popic, A. Blankstein, J. Cowling, D. Curtis, L. Shrira, and B. Liskov. Abstractions for usable information flow control in Aeolus. In *Proc. USENIX ATC*, June 2012.

[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, A. Solar-Lezama, and S. Madden. Inferring SQL queries using program synthesis. arXiv:1208.2013 [cs.PL].

[10] A. Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *Proc. OSDI*, Oct. 2010.

[11] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proc. SOSP*, pages 31–44, Oct. 2007.

[12] 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.

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

[14] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. OSDI*, pages 137–150, 2004.

[15] 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.

[16] Google Web Toolkit. `http://code.google.com/webtoolkit/`.

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

[18] G. C. Hunt and M. L. Scott. The Coign automatic distributed partitioning system. In *Proc. OSDI*, pages 187–200, 1999.

[19] M. Isard and Y. Yu. Distributed data-parallel computing using a high-level programming language. In *Proc. SIGMOD*, pages 987–994, 2009.

[20] M.-Y. Iu, E. Cecchet, and W. Zwaenepoel. JReq: Database queries in imperative languages. In *Proc. CC*, pages 84–103, 2010.

[21] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, and L. Shrira. Safe and efficient sharing of persistent objects in Thor. In *Proc. SIGMOD*, pages 318–329, June 1996.

[22] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: A platform for secure distributed computation and storage. In *Proc. SOSP*, pages 321–334, 2009.

[23] D. Manolescu, B. Beckman, and B. Livshits. Volta: Developing distributed applications by recompiling. *IEEE Softw.*, 25(5):53–59, Sept. 2008.

[24] E. Meijer, B. Beckman, and G. Bierman. Linq: Reconciling objects, relations and XML in the .NET framework. In *Proc. SIGMOD*, pages 706–706, 2006.

[25] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden. Wishbone: Profile-based partitioning for sensornet applications. In *Proc. NSDI*, pages 395–408, Apr. 2009.

[26] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: an extensible compiler framework for Java. In *Proc. CC*, pages 138–152, 2003.

[27] OASIS Web Services Security TC. Web Services Security v1.1.1.

[28] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodík, V. A. Saraswat, and S. A. Seshia. Sketching stencils. In *Proc. PLDI*, pages 167–178, 2007.

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

[30] F. Yang, J. Shanmugasundaram, M. Riedewald, and J. Gehrke. Hilda: A high-level language for data-driven web applications. In *Proc. ICDE*, pages 32–43, 2006.