Queryll: Java Database Queries Through Bytecode Rewriting

Ming-Yee Iu and Willy Zwaenepoel

School of Computer and Communication Sciences,
EPFL, Lausanne, Switzerland

Abstract. When interfacing Java with other systems such as databases, programmers must often program in special interface languages like SQL. Code written in these languages often needs to be embedded in strings where they cannot be error-checked at compile-time, or the Java compiler needs to be altered to directly recognize code written in these languages. We have taken a different approach to adding database query facilities to Java. Bytecode rewriting allows us to add query facilities to Java whose correctness can be checked at compile-time but which don't require any changes to the Java language, Java compilers, Java VMs, or IDEs. Like traditional object-relational mapping tools, we provide Java libraries for accessing individual database entries as objects and navigating among them. To express a query though, a programmer simply writes code that takes a Collection representing the entire contents of a database, iterates over each entry like they would with a normal Collection, and choose the entries of interest. The query is fully valid Java code that, if executed, will read through an entire database and copy entries into Java objects where they will be inspected. Executing queries in this way is obviously inefficient, but we have a special bytecode rewriting tool that can decompile Java class files, identify queries in the bytecode, and rewrite the code to use SQL instead. The rewritten bytecode can then be run using any standard Java VM. Since queries use standard Java set manipulation syntax, Java programmers do not need to learn any new syntax. Our system is able to handle complex queries that make use of all the basic relational operations and exhibits performance comparable to that of hand-written SQL.

1 Introduction

Queryll is a middleware system that uses bytecode rewriting to allow programmers to interface Java with other systems without needing to use an intermediary language. Currently, Queryll is focused on interfacing Java with SQL databases by providing database query facilities to Java. With Queryll, programmers can encode database queries using standard Java syntax for working with collections. No special compiler or IDE is needed. The queries are also semantically correct in that if they are executed as written, they will connect to a database, iterate through all the entries in the database, and find the desired entries (though executing queries in this way is obviously inefficient). When the compiled Java

M. van Steen and M. Henning (Eds.): Middleware 2006, LNCS 4290, pp. 201–218, 2006.
© IFIP International Federation for Information Processing 2006

bytecode is fed into the Queryll bytecode rewriter, the queries in the bytecode stream are identified, and they are replaced with code that executes equivalent SQL queries instead. The bytecode rewriting acts, in fact, like a type of code optimization in which whole algorithms are replaced with more efficient substitutes.

Unlike Queryll, other middleware systems use special programming languages to interface Java to other systems. Databases, graphics cards, and symbolic computation engines all require the use custom languages to access their features. This approach can be very cumbersome. Not only does the Java programmer need to learn another programming language, but mismatches between the underlying models of these other languages and Java mean that programmers often have to write extra code for translating concepts between the two models. Since the Java compiler does not recognize the syntax of any of these other languages, their code has to be embedded in strings where they cannot be statically error-checked. Often parameters must be marshaled into special data structures before they can be passed to and from these other systems. Ultimately, these annoyances distract programmers from larger algorithmic and architectural issues.

One solution to these problems is to create hybrid programming languages that mix other languages with Java. For example, SQLJ [1] is a hybrid of Java and SQL. In SQLJ code, SQL queries can be intermixed with Java, and the queries can make reference to Java variables. Although hybrid languages do allow for static error-checking and do eliminate the need for data marshaling, they require special compilers and IDE changes. The approach also falls apart when multiple interface languages are merged with Java, resulting in a hybrid language with a complex tangle of additional language constructs.

Ideally, it should be possible to interact with databases and other systems using regular Java code. That way, programmers would not need to learn a new language but only a few new API calls to interface with a system. Programmers would not need special compilers, nor would they have to deal with issues such as data marshaling or embedding code inside strings. Unfortunately, it is impractical to use Java in this way. The primary language construct that these other interface languages have but Java lacks is a facility for inspecting and modifying one's own code. Queries written in a query language need to be understood and manipulated by a query optimizer to be executed on a database. A fragment shader program needs to be compiled into instructions that can be run on a graphics card. Java does support reflection, but it does not have APIs for understanding and manipulating code.

Unlike other database middleware layers for Java, Queryll provides a pure Java interface to databases that allows programmers to describe complex queries without resorting to another programming language. As such, programmers do not need to learn a new language, and the standard Java compiler can catch many potential errors at compile-time. There is also no unnatural split in the middleware API where a simplified API is available for performing basic queries and a more extensive API is needed for more complex queries. Queryll is able to achieve this behaviour because it is designed as a bytecode rewriting tool. As such, it does not make any changes to the Java language, meaning that a

standard Java compiler and IDE can be used by programmers. The rewritten bytecode can also be run on a standard Java VM. Using bytecode rewriting for extending Java does not force programmers to change any of their existing tools, and it can be used to interface Java with multiple systems without adding new complexity to the Java language.

2 Related Work

There are many middleware languages and tools for interfacing Java with various databases. These different tools provide differing levels of abstraction and differing levels of integration with Java.

2.1 JDBC and SQLJ

The standard database middleware layer for Java is JDBC [2]. With JDBC, queries are described using SQL and are stored in strings. Programmers then pass these strings to the JDBC API, which executes the queries on a database. Although JDBC provides some helper methods to help with data marshaling, programmers must still manually pack parameters into queries and then manually read out and interpret individual fields from the query results.

As described earlier, SQLJ is a language that combines SQL with Java. Because of this integration, both the SQL and Java code can be checked for errors at compile time, programmers can reference Java variables from within SQL, and programmers can reference SQL results from within Java. Typically, a precompiler is used to compile SQLJ into Java code that uses JDBC.

2.2 ORM Tools

Both JDBC and SQLJ are tightly bound to the SQL table-oriented view of data, which is inconsistent with Java's object-oriented model. Object-Relational Mapping (ORM) tools such as Hibernate [3] or EJB [4] allow programmers to specify a mapping from SQL tables to an object representation. The ORM tool then generates code that allows programmers to manipulate these objects in Java and have these changes be persisted automatically to the corresponding SQL tables. Although these objects do hide data marshaling issues and allow programmers to execute simple queries with just a simple method call, they cannot be used for more complex queries. For complex queries, ORM tools typically supply a special object query language such as HQL or EJBQL (Fig. 1). Like JDBC, queries in these languages are encoded in strings, and programmers must manually encode parameters into their queries.

2.3 LINQ

In C# 3.0, Microsoft has added a feature called Language Integrated Query (LINQ) [5]. This feature allows programmers to inline queries with their C#

```
List l = em.createQuery("SELECT c FROM Customer c WHERE c.id = :id")
     .setParameter(":id", 2500)
     .getResultList();
```

Fig. 1. A sample EJBQL query

code. Unlike the approach used by Queryll, extensive changes to the C# compiler and language were made in order to support LINQ. Notably, C# now supports lambda expressions, and C# compiles lambda expressions into two forms: executable code and a data structure representation that can be inspected at runtime. The new language constructs in C# only provide support for queries. They cannot be used to interface C# with systems such as graphics cards, for example.

2.4 Bytecode Rewriting and Decompiling

All Java compilers compile Java programs into a machine independent intermediate representation known as bytecode. This bytecode is stored in files called classfiles. Java programs are distributed as classfiles which can be executed using a Java VM. Bytecode rewriting is a well-known Java technique for modifying the behaviour of compiled Java code. A typical example would be J-Orchestra [6] which can alter Java objects so that they can be invoked remotely without requiring changes to the original code. Many aspect-oriented programming tools also make use of bytecode rewriting to support dynamic aspect weaving [7]. And some ORM tools already make use of bytecode rewriting to transparently add persistence code to ordinary Java objects to enable those objects to be stored in databases. These uses of bytecode rewriting are limited to only modifying surface features of code such as intercepting method calls; however, some tools such as the automatic parallelization program javab [8] perform more detailed code analysis. One can consider classfile decompilation [9], where bytecode is converted to Java source files, to be an extreme form of bytecode rewriting. There are several Java decompilation tools, and Queryll borrows some of their techniques for its work.

3 Queries with Queryll

As mentioned earlier, Queryll is able to take database queries written in regular Java and rewrite the queries to use SQL. Clearly though, the Queryll bytecode rewriter is not able to convert arbitrary Java to SQL.

Queryll's query syntax is designed to conform with standard Java patterns for working with collections, resulting in a syntax that feels "natural" and consistent with existing Java code. It is also designed to have the properties of being executable and semantically correct. This means that if the query is compiled with a standard Java compiler and run on a standard Java VM, the code will not only execute but will return the correct query result as well. Although the

Queryll bytecode rewriter detects query code and rewrites them to use SQL, even if no rewriting occurs, the query code is perfectly functional. Admittedly, without rewriting, the query will be horribly inefficient since it will download the entire database and iterate through each row; nonetheless, the code will behave correctly. By forcing query code to be executable and semantically correct, we ensure that queries are expressed in sufficient detail that the standard Java compiler can verify much of the correctness of the query using its existing static type checking. These properties also preclude a syntax that introduces new domain-specific constructs to the Java language.

Queryll uses an object-relational mapping to allow database entities to be represented and manipulated as objects within Java. Queryll queries are expressed using iterations over collections of these objects. The current Queryll query syntax supports selection, projection, and join operations, thereby making Queryll functionally equivalent to basic relational algebra. Unfortunately, Queryll does not yet support aggregation operations or nested queries, meaning that it is not currently able to handle the extended query operations needed to express arbitrary SQL queries. Queryll does have support for SQL ordering and limit operations though.

3.1 Queryll ORM

Because SQL tables are a foreign concept to the object-oriented model of Java, they need to be translated into some sort of representation that can be manipulated by Java code. Queryll uses a custom light-weight ORM tool to map tables to classes. Like with other ORM tools, programmers must describe how table rows should map to objects, how table fields should be mapped into object fields, and the various relationships between tables. They are essentially defining an object representation of a database and defining how to convert between the SQL representation and the object representation.

So, consider a simple database describing bank clients, each of whom may have multiple bank accounts. This database might be composed of two tables (Fig. 2): Client and Account. Using the Queryll ORM tool, this database can be mapped to the class diagram in Fig. 3.

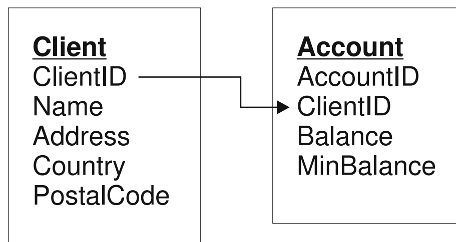

Fig. 2. A simple database

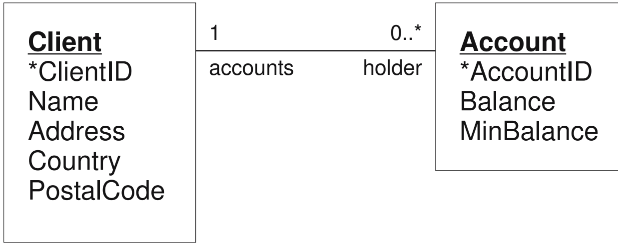

Fig. 3. Class diagram of database entities (* denotes primary keys)

From the mapping, Queryll generates the classes for each entity with accessor methods for fields and special methods for traversing relationships between objects (for example, retrieving a Collection of accounts belonging to a client). These objects act as a cache of database data and are all lazily instantiated. Queryll also creates a special class named EntityManager that is responsible for ensuring that the database data and their in-memory object representations remain consistent. Figure 4 shows how the generated classes may be used. Queryll's approach to object-relational mapping is fairly standard among existing ORM tools.

```
EntityManager em = db.beginTransaction();
Client c = em.findClient(1000);
System.out.println("Client 1000 lives at " + c.getAddress());
System.out.println("Client 1000 has " + c.getAccounts().size()
                   + " accounts");
db.endTransaction(em, true);
// Note: the findClient() method is used here for illustrative
// purposes. In actuality, no such method exists because Queryll
// supports using full queries instead.
```

Fig. 4. ORM tools can generate classes that allow programmers to access database data as objects instead of having to deal with SQL tables

3.2 Simple Queries and Selection

Since the main Java construct for working with large amounts of data is the for-each loop for iterating over arrays and Collections, we built our syntax around that construct. The for-each loop restricts our queries to using Collections to represent database contents. As such, we created a special type of Java Collection called a QuerySet. A QuerySet is a lazily initialized container of database entities. It holds a SQL query, and when any attempt is made to access any of the elements of a QuerySet, the QuerySet will execute the query on a database, and fill itself with the results of the query, and from then on behave like a normal Java Collection.

To write a simple Queryll query then, a programmer takes an existing QuerySet, iterates over each element of the QuerySet to find the elements that she is interested in, and adds these elements to a new QuerySet. All the elements of the original QuerySet must be iterated over (no premature loops exits), and the loop code can have no side-effects beyond adding elements to the new QuerySet. The query syntax is purposely based on adding elements to a new QuerySet as opposed to modifying an existing QuerySet. The elements added to the new QuerySet may be of a different type than the elements in the QuerySet being iterated over, so two different QuerySets are needed for everything to type-check correctly.

Finally, the programmer must also label the methods containing Queryll queries with the @Query annotation. Since bytecode rewriting is an expensive operation, the Queryll bytecode rewriter will only look at the bytecode of @Query methods when converting queries to their SQL equivalents.

Figure 5 shows a simple Queryll query that finds bank clients who come from Canada. Notice that the EntityManager object em has methods for returning a QuerySet of all the Client entities in the database. Since all queries must start with an existing QuerySet, the EntityManager provides the initial QuerySet objects on which queries can be constructed. As mentioned previously, the standard Java type rules impose a certain amount of correctness on the query. The string "country" acts as a parameter in the query, and the Java compiler ensures that this parameter is of the correct type. The Java compiler also ensures that the entity fields being examined during the query actually exist (otherwise the accessor methods would not exist) and that the result of the query is of the expected type.

```
QuerySet<String> canadian = new QuerySet<String>();
String country = "Canada";
for (Client c: em.allClient())
   if (c.getCountry().equals(country))
      canadian.add(c.getName());
```

Fig. 5. A simple query for finding all clients from Canada

The Queryll query syntax is flexible enough to allow programmers to express a wide variety of query operations in a natural way. For example, by simply changing the conditions in which an element is added to a new QuerySet, a programmer is writing a selection operation.

3.3 Projection

To support projection operations, Queryll supplies a Pair object that can hold two arbitrary values. Similar to a LISP list which also holds only two values (car and cdr), the Pair object can be used to construct simple data structures during a query, which can then be added to a new QuerySet. This ability to create

new data structures is equivalent to using projection operations to create new columns for database relations or to remove columns from database relations. Projection operations themselves are not directly expressible in Queryll, as doing so would mean that Queryll would have to support the creation of new classes at runtime. The Java syntax for the creation of new classes is quite verbose and cumbersome, and working with a large number of these classes creates headaches for programmers because they would not work well with Java's type system. In fact, to support projection in LINQ, Microsoft had to create a new C# syntax for creating new classes at runtime and change the C# type system. Queryll's use of Pair objects to provide power equivalent to projection is much more consistent with existing Java syntax. Figure 6 shows how a programmer might use Pair objects to hold data about the penalty that should be applied to bank accounts that are below their minimum balance and hence overdrawn.

```
QuerySet<Pair<Account, Double>> overdrawn
      = new QuerySet<Pair<Account, Double>>();
for (Account a: em.allAccount()) {
   if (a.getBalance() < a.getMinBalance()) {
      double penalty = (a.getMinBalance() - a.getBalance()) * 0.001;
      overdrawn.add(new Pair<Account, Double>(a, penalty);
   }
}
```

Fig. 6. Queryll provides Pair objects, which can be used to create data structures for holding calculated values, thus providing power equivalent to projection

3.4 Join

Expressing join operations is quite easy in Queryll. Since the relationship between entities is described during the ORM phase, Queryll generates methods for navigating among objects, and these methods can be used during queries. Some types of joins, such as those where a single table row is joined with multiple rows from another table, are potentially difficult to express in Java, so Queryll provides a few utility methods for handling these cases. Figure 7 shows two different ways that joins can be used to find all the bank accounts belonging to clients in Switzerland.

3.5 Ordering and Limit

Currently, Queryll only has preliminary support for the SQL ordering and limit operations. The syntax for ordering is not yet finalized, but the current syntax requires programmers to create a sorter class that describes which fields of the elements should be used for sorting. This is similar to the existing use of the Comparator object in Java for sorting. Figure 8 shows an example of ordering in Queryll.

```
QuerySet<Pair<Client, Account>>
   swiss1 = new QuerySet<Pair<Client, Account>>(),
   swiss2 = new QuerySet<Pair<Client, Account>>();

for (Account a: em.allAccount())
   if (a.getHolder().getCountry().equals("Switzerland"))
      swiss1.add(new Pair<Client, Account>(a.getHolder(), a));

for (Client c: em.allClient())
   if (c.getCountry().equals("Switzerland"))
      swiss2.addAll(Pair.PairCollection(c, c.getAccounts());
```

Fig. 7. Two different join queries that give the same results

```
QuerySet<Account> top10Accounts = em.allAccount();
top10Accounts = top10Accounts.sortedByDoubleDescending(
      new DoubleSorter<Account>() {
         public double value(Account val) {
            return val.getBalance();
         }
      });
top10Accounts = top10Accounts.firstN(10);
```

Fig. 8. Queryll supports ordering and limit operations as well

4 Implementation

The Queryll system (Fig. 9) is composed of two programs: an ORM tool and a bytecode rewriter. The bytecode rewriter is by far the more complicated of the two.

Suppose the query defined in Fig. 10 is given to the Queryll bytecode rewriter. As mentioned earlier, all methods containing queries should be labelled with a @Query annotation to help Queryll focus its optimizations on the right pieces of code. Queryll finds all such methods and feeds the bytecode of these methods into Sable's Soot [10] framework for conversion into Jimple code, a representation that is easier to analyze. Jimple is a three-address code for Java where all variables are typed (Java objects on the execution stack are usually typeless). Three-address code is useful because it eliminates Java's execution stack, resulting in one less structure that the bytecode rewriter needs to analyze and making it easier to rearrange code without having to worry about whether the state of the stack remains consistent. Queryll does not actually make use of the typing feature of Jimple, meaning that a simpler three-address code framework than Soot could be used if one becomes available. Figure 11 shows the Jimplified version of the compiled bytecode of the previous query. Being a three-address code, most instructions consist of an operation on two variables, the result of which is then assigned into third variable.

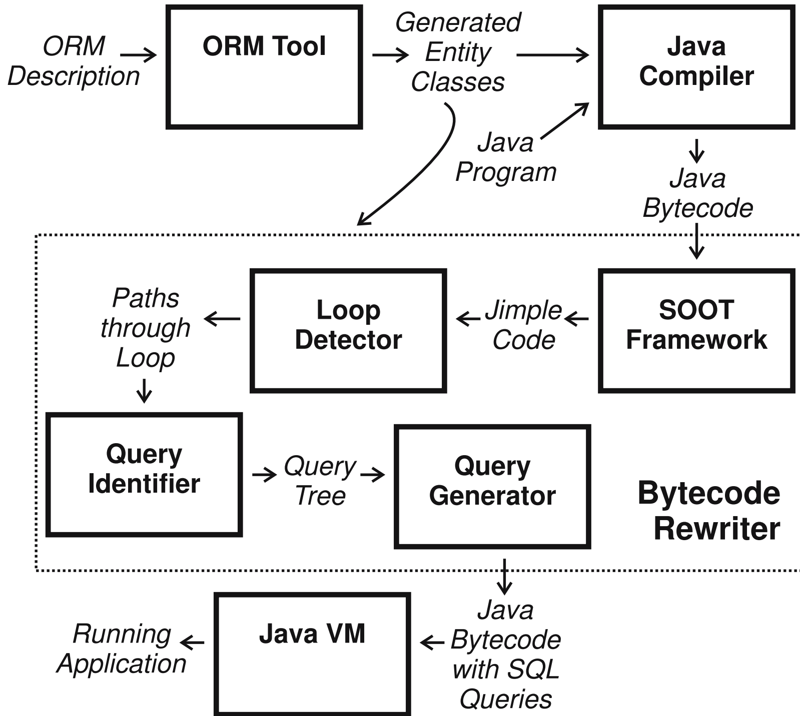

Fig. 9. Queryll system design

```
for (Office of: em.allOffice()) {
   if (of.getName().equals("Seattle"))
      westcoast.add(of);
   else if (of.getName().equals("LA"))
      westcoast.add(of);
}
```

Fig. 10. A simple query that can be analyzed by the Queryll bytecode rewriter

The next stage of the analysis then involves identifying loops within the code. Although loops are easy to identify in Java source code, compiled Java code uses only GOTO statements to describe its control flow. There are generally two approaches for extracting loops from program code that uses GOTOs for control flow. One approach is called GOTO-elimination [11] where code transformations are applied to individual GOTO statements to convert them into looping structures. Instead, Queryll uses the alternate approach where the control flow graph is analyzed as a whole and restructured to make use of loops [12]. This latter approach is used because it provides a deeper understanding of the loop structure than the former approach. Loops are defined as being strongly connected

```
          1:  $r12 = r1.<EntityManager: Set allOffice()>();
          2:  r6 = $r12.<Set: Iterator iterator()>();
          3:  goto label3;

label1:   4:  $r13 = r6.<Iterator: Object next()>();
          5:  r14 = (Office) $r13;
          6:  $r15 = r14.<Office: String getName()>();
          7:  $z3 = $r15.<String: boolean equals(Object)>("Seattle");
          8:  if $z3 == 0 goto label2;

          9:  r11.<Set: boolean add(Object)>(r14);
         10:  goto label3;

label2:  11:  $r16 = r14.<Office: String getName()>();
         12:  $z5 = $r16.<String: boolean equals(Object)>("LA");
         13:  if $z5 == 0 goto label3;

         14:  r11.<Set: boolean add(Object)>(r14);

label3:  15:  $z7 = r6.<Iterator: boolean hasNext()>();
         16:  if $z7 != 0 goto label1;
```

Fig. 11. When a query is rewritten to be in a Jimple representation, it is easier to analyze and manipulate

components in the control flow graph that have a single entry point. Queryll further restricts its definition of loops to require that all exits from the strongly connected component exit to the same instruction. Standard graph algorithms can be used to find pieces of code that satisfy these requirements and label that code as being a loop.

Since Queryll queries are all composed of a for-each loop over a QuerySet, the Queryll bytecode rewriter must be able to determine whether a loop is a for-each loop or not. A for-each loop that iterates over a Java Collection compiles down to code that creates an Iterator object from the Collection, and then continually advances the iterator until there are no more objects left to iterate over (see instructions 2, 4, 15, and 16 in Fig. 11). Queryll tries to identify this pattern in loops by looking for iterators being incremented within loops. Queryll also checks other properties of the loop such as whether each loop instruction has no side effects except for adding elements to a Collection or incrementing the iterator. If that is the case, the loop is labelled as a candidate for being a query, the Collection being iterated over is labelled as the source collection, and the Collection to which elements are added is labelled as the destination collection.

It then becomes necessary to interpret what sort of query is being performed by the loop. Since the loop might contain many variables and branching instructions, it can be difficult to understand what is going on. On the other hand, analyzing straight-line code is much easier because it is easy to calculate both the values of variables at any point in the code and dependencies between any

instructions. To take advantage of that fact, Queryll breaks loops down into straight paths to do its analysis. It does this by examining every control flow path through a loop that results in a new element being added to the destination collection. The instructions that form a path are then treated as a straight-line piece of code. Table 1 shows the two paths that Queryll finds when examining the code in Fig. 11.

Table 1. There are two paths through the loop that lead to new elements being added to the destination collection

Path 1	Path 2
15: $z7 = r6.hasNext()	15: $z7 = r6.hasNext()
16: if $z7 != 0 goto label1	16: if $z7 != 0 goto label1
4: $r13 = r6.next()	4: $r13 = r6.next()
5: r14 = (Office) $r13	5: r14 = (Office) $r13
6: $r15 = r14.getName()	6: $r15 = r14.getName()
7: $z3 = $r15.equals("Seattle")	7: $z3 = $r15.equals("Seattle")
8: if $z3 == 0 goto label2 (branch not taken)	8: if $z3 == 0 goto label2 (branch taken)
9: r11.add(r14)	11: $r16 = r14.getName()
	12: $z5 = $r16.equals("LA")
	13: if $z5 == 0 goto label3 (branch not taken)
	14: r11.add(r14)

For each path, Queryll determines what the values of local variables need to be for the path to be followed. So, essentially, for each branch instruction in the path, Queryll will make a note of what values a variable must take for the branch to be taken or not. These restrictions on the variables will be ANDed together to form an expression describing the conditions that need to hold for the path to be followed. These variables are likely only local variables holding intermediate calculations that do not directly refer to any concrete object fields though. To map these variables onto database entries, the bytecode rewriter starts at the last instruction in the path and goes over each instruction in the path backward. Since Jimple is a type of three-value code, most instructions are of the form where a binary operation on two variables is assigned to another variable. If the variable being assigned to is part of the expression representing the path, the right-hand side of the instruction (made up of the binary operation on two variables) is substituted for the left-hand variable in the path expression. When the first instruction of the path is reached, the resulting expression should be made up of operations acting on constants, outside variables, or entries from the source collection. For example, if Queryll was trying to construct an expression to describe the second path of Table 1, it would go through the steps shown in Table 2. Because Java bytecode instructions for conditional GOTOs can only work with conditions involving integers, when the above procedure is used on code that works with non-integers, the resulting expression contains redundant

comparisons. So in Table 2, the expression for the path compares Office.Name with “Seattle”, resulting in an integer, and then compares this integer with 0. These extra comparisons can confuse some SQL implementations, so Queryll always performs a simplification step on the final expression to remove them.

Table 2. For a given path, Queryll can construct an expression that describes when the path is executed

Instruction	Expression
Initial	$z3 = 0 AND $z5 != 0
14: r11.add(r14)	
13: if $z5 == 0 goto label3	
12: $z5 = $r16.equals(”LA”)	$z3 = 0 AND ($r16 = ”LA”) != 0
11: $r16 = r14.getName()	$z3 = 0 AND (r14.Name = ”LA”) != 0
8: if $z3 == 0 goto label2	
7: $z3 = $r15.equals(”Seattle”)	($r15 = ”Seattle”) = 0 AND (r14.Name = ”LA”) != 0
6: $r15 = r14.getName()	(r14.Name = ”Seattle”) = 0 AND (r14.Name = ”LA”) != 0
5: r14 = (Office) $r13	(((Office)$r14).Name = ”Seattle”) = 0 AND (((Office)$r13).Name = ”LA”) != 0
4: $r13 = r6.next()	(((Office)entry).Name = ”Seattle”) = 0 AND (((Office)entry).Name = ”LA”) != 0
16: if $z7 != 0 goto label1	
15: $z7 = r6.hasNext()	
Simplification	(((Office)entry).Name != ”Seattle”) AND (((Office)entry).Name = ”LA”)

Each path found by Queryll represents a different way in which a new entry can be added to the destination collection. So to construct a description of which elements of the source collection should appear in the destination collection, Queryll takes the expressions representing each path and ORs them together (Fig. 12). This giant expression can then be put into the WHERE clause of a SELECT..FROM..WHERE statement to create a SQL query. Similar techniques are used to create SQL queries that calculate new columns or which join together multiple tables.

```
SELECT ...
  FROM Office AS A
 WHERE (((A).Name  != "Seattle") AND ((A).Name = "LA"))
       OR ((A).Name  = "Seattle")
```

Fig. 12. Queryll ORs together the expressions for each path through the for-each loop to construct the WHERE clause of a SQL query

5 Benchmarks

The SQL query code generated by Queryll tends to be a little more verbose than hand-written SQL. Queryll also imposes some additional overhead at run-time because it uses various abstractions to allow it to construct SQL queries programmatically. These factors do negatively affect the performance of Queryll queries. Adopters of middleware must always deal with the trade-off between increased programmer productivity versus system performance, but ideally the overhead of Queryll should be tolerably low if not negligible.

We have built a microbenchmark based on TPC-W [13]. TPC-W is a benchmark suite that models the behaviour of database-driven websites. We have taken the Rice University implementation of TPC-W [14], which uses JDBC SQL queries, as a benchmark base. The full TPC-W benchmark makes use of application servers and web clients browsing through the website. Instead, we have taken a select number of queries from the benchmark and evaluated the throughput of these queries using JDBC and Queryll.

Of the queries in the Rice TPC-W implementation, all the queries involving updates were removed. Queryll uses an approach to persistence that is standard among other ORM tools whereby programmers load table rows into objects, programmers manipulate the fields of the objects, and the ORM tool will write the objects' data back to individual table rows before a transaction completes. Since this technique is already quite pervasive, evaluating update performance does not provide any new insight into the behaviour of Queryll. Queries making use of temporary tables, GROUP BY, aggregation functions, and LIKE were also removed as Queryll does not support these features yet. Of the remaining queries, many were similar (e.g. reading individual fields from a row in a table), so we have taken a representative sample of these for the microbenchmark. Table 3 lists the queries included in the microbenchmark. Each query is given a name, each query is described briefly, and the hand-written SQL used in the Rice TPC-W implementation of each query is shown.

We created a 600 MB database in PostgreSQL 8.1.3 [15] by populating the database using these parameters: the number of items was set to 10000 and the number of EBs was set to 100. During a run of the benchmark, each query was run 100 times using random valid parameters to warm the database cache, and then a measurement was taken of the time needed to execute the query 2000 times using random valid parameters. Each configuration was benchmarked at least 30 times, with only the last 20 runs included in the final measurement averages. This was needed to remove the effect of Java dynamic compilation from the measurements and to further warm the database cache. The database and the query code were both run on the same machine, a 2.5 GHz Pentium IV Celeron Windows machine, with 1 GB of RAM (though the benchmark harness was run using Java's default maximum heap size of 64 MB).

The results of the benchmark are shown in Table 4. Hand-written SQL queries are generally faster than the queries generated by Queryll except in the doSubjectSearch query. Most of the time differences can be explained by miscellaneous overhead in the generated Java query code or small differences in query execution

Table 3. Queries used in the benchmark

getName
Find a specific row in a table using its primary key
SELECT c_fname, c_lname
FROM customer
WHERE c_id = ?

getCustomer
Find a specific row in a table and then join it to two other tables
SELECT ...
FROM customer, address, country
WHERE customer.c_addr_id = address.addr_id AND address.addr_co_id = country.co_id AND customer.c_uname = ?

doSubjectSearch
Find all entries in a table with a field set to a certain value, join these entries to another table, sort them, and take the first 50
SELECT i.i_id, i.i_title, a.a_fname, a.a_lname
FROM item i, author a
WHERE i.i_subject = ? AND i.i_a_id = a.a_id ORDER BY i.i_title (LIMIT 0,50)

getRelated
Find an entry in a table using its primary key, then follow its five references to other entries in the same table
SELECT J.i_id,J.i_thumbnail
FROM item I, item J
WHERE (I.i_related1 = J.i_id or I.i_related2 = J.i_id or I.i_related3 = J.i_id or I.i_related4 = J.i_id or I.i_related5 = J.i_id) and I.i_id = ?

at the database. For example, the generated code for the getName query (Table 5) is essentially the same as the hand-written code, but the generated code sends a commit command to the database separately from its query, reads columns out from ResultSets by referring to columns by name instead of by index number, stores results in intermediate data structures, and has other additional overhead. When the hand-written JDBC code was modified to include some of the same inefficiencies, its running time shot up dramatically to almost match the time taken by the Queryll queries. This behaviour suggests that even though the time difference between hand-written queries and Queryll queries are large in percentage terms, in absolute terms the difference is quite small. Given sloppily hand-written JDBC code or highly optimized generated Queryll code, the time differences could be easily reversed.

In fact, the generated code for the doSubjectSearch query was consistently faster than the hand-written code for the query despite the extra overhead in the generated code. This fact suggests something unusual with the SQL queries, but the generated SQL query was essentially the same as the hand-written query, except that the ordering of the columns was different and each column was given

Table 4. Benchmark results

Query	Queryll Time (ms)	Queryll Std Dev	Hand-Written SQL Time (ms)	Hand-Written SQL Std Dev	Difference (ms)
getName	3360	12.3	2053	19.3	1307
with extra processing			3030	18.9	330
getCustomer	7716	141.2	5163	69.1	2552
doSubjectSearch	21450	329.5	22384	25.3	-934
with modified query			20378	18.1	1072
doGetRelated	8124	16.8	3262	10.1	4862

Table 5. SQL queries generated by Queryll

getName
```
SELECT (A.C_FNAME) AS COL0, (A.C_LNAME) AS COL1
  FROM Customer AS A
 WHERE ( ( ((A.C_ID) = ?) ) )
```

getCustomer
```
SELECT ...
  FROM Customer AS A, Address AS B, Country AS C
 WHERE ( ( ((A.C_UNAME) = ?) ) ) AND A.C_ADDR_ID = B.ADDR_ID AND
       B.ADDR_CO_ID = C.CO_ID
```

doSubjectSearch
```
SELECT (A.I_TITLE) AS COL1, (B.A_FNAME) AS COL2, (B.A_LNAME) AS
       COL3, (A.I_ID) AS COL0
  FROM Item AS A, Author AS B
 WHERE ( ( ((A.I_SUBJECT) = ?) ) ) AND A.I_A_ID = B.A_ID ORDER BY
       (A.I_TITLE)
```

doGetRelated
```
SELECT ...
  FROM Item AS A, Item AS B, Item AS C, Item AS D, Item AS E, Item AS F
 WHERE ( ( ((A.I_ID) = ?) ) ) AND A.I_RELATED1 = B.I_ID AND
       A.I_RELATED2 = C.I_ID AND A.I_RELATED3 = D.I_ID AND
       A.I_RELATED4 = E.I_ID AND A.I_RELATED5 = F.I_ID
```

a column alias. When we changed the hand-written query to match the generated one, its running time became better than that of the generated queries. We can only assume that the ordering of the columns somehow caused the database to execute the automatically generated queries in a slightly more optimal way than the hand-generated one.

The doGetRelated query is the only query that is significantly slower when using generated queries instead of hand-written ones. This likely results from the fact that the generated query is quite different from the original query. While the original query joins the Item table to itself once, the generated query joins the Item table to itself five times—one for each reference to another Item row.

This happens because Queryll does not currently support arbitrary cross joins between tables. Instead, the Queryll query is written exactly as it is described in Table 3. When Queryll analyzes the query, it sees one Item entity with five separate fields referring to five other Item entities, and it rewrites each reference to be a separate join operation.

Overall, the results show that in most cases, using generated queries instead of hand-written queries should not cause major performance problems. The use of generated queries does impose some overhead on the application (as opposed to the database) because it creates more intermediate data structures and uses more abstractions. Of course, even hand-written JDBC calls can suffer from similar overhead if programmers aren't careful. And much of this overhead can be reduced by improving the automatic code generation of Queryll.

6 Conclusion

Queryll is a middleware layer that allows Java programmers to access databases without having to resort to a separate interface language. The query syntax is consistent with existing Java syntax for searching Java collections. Unlike other database middleware, the Queryll API can handle both simple and complex queries. And database queries written using Queryll generally have comparable performance to hand-written queries even though Queryll provides a much higher level of abstraction.

7 Future Work

Although Queryll currently supports basic relational algebra, it would be useful to add aggregation and nested query support to Queryll to allow it to handle the extended algebra behind SQL. The existing code could also be made more robust through the addition of more error-checking. Additionally, it would be useful to formalize Queryll's query syntax and to rigorously define how it is converted to SQL. One difficult aspect of this is that since Queryll operates on Java bytecode, the query syntax needs to be defined in terms of bytecode. But this query syntax must then be backward translated to the regular Java that programmers would write.

Overall our success in using bytecode rewriting to add query support to Java makes us hopeful that the approach will also work well for integrating other interface facilities into Java. We would like to expand Queryll into a general bytecode rewriting framework that would allow programmers to plug-in various behaviours appropriate for different interfacing middleware.

References

1. Eisenberg, A., Melton, J.: SQLJ part 0, now known as SQL/OLB (object-language bindings). SIGMOD Rec. **27**(4) (1998) 94–100
2. Sun Microsystems: JDBC technology. (`http://java.sun.com/products/jdbc/`)
3. JBoss: Hibernate. (`http://www.hibernate.org/`)

4. Sun Microsystems: Enterprise JavaBeans technology. (http://java.sun.com/products/ejb/)
5. Microsoft: The LINQ project. (http://msdn.microsoft.com/netframework/future/linq/)
6. Tilevich, E., Smaragdakis, Y.: Portable and efficient distributed threads for Java. In: Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware, New York, NY, USA, Springer-Verlag New York, Inc. (2004) 478–492
7. Pawlak, R., Seinturier, L., Duchien, L., Florin, G.: Jac: A flexible solution for aspect-oriented programming in Java. In: REFLECTION '01. Volume 2192 of LNCS., London, UK, Springer-Verlag (2001) 1–24
8. Bik, A.J., Gannon, D.B.: Javab—a prototype bytecode parallelization tool. Technical Report TR489, Indiana University (1997)
9. Miecznikowski, J., Hendren, L.: Decompiling Java bytecode: Problems, traps and pitfalls. In: CC 2002, Springer-Verlag (2002) 111–127
10. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot - a Java bytecode optimization framework. In: CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research, IBM Press (1999) 13
11. Erosa, A.M., Hendren, L.J.: Taming control flow: A structured approach to eliminating GOTO statements. In: ICCL. (1994)
12. Peterson, W.W., Kasami, T., Tokura, N.: On the capabilities of while, repeat, and exit statements. Commun. ACM **16**(8) (1973) 503–512
13. Transaction Processing Performance Council (TPC): TPC Benchmark W (Web Commerce) Specification Version 1.8. Transaction Processing Performance Council (2002)
14. Amza, C., Cecchet, E., Chanda, A., Elnikety, S., Cox, A., Gil, R., Marguerite, J., Rajamani, K., Zwaenepoel, W.: Bottleneck characterization of dynamic web site benchmarks. Technical Report TR02-389, Rice University (2002)
15. PostgreSQL Global Development Group: PostgreSQL. (http://www.postgresql.org/)