DBridge: A Program Rewrite Tool for Set-Oriented Query Execution

Mahendra Chavan #, Ravindra Guravannavar *, Karthik Ramachandra #, S. Sudarshan #

#*Indian Institute of Technology, Bombay*
{mahcha, karthiksr, sudarsha}@cse.iitb.ac.in

**Indian Institute of Technology, Hyderabad*
ravig@iith.ac.in

Abstract— **We present DBridge, a novel static analysis and program transformation tool to optimize database access. Traditionally, rewrite of queries and programs are done independently, by the database query optimzier and the language compiler respectively, leaving out many optimization opportunities. Our tool aims to bridge this gap by performing holistic transformations, which include both program and query rewrite.**

Many applications invoke database queries multiple times with different parameter values. Such query invocations made using imperative loops are often the cause of poor performance due to random I/O and round trip delays. In practice, such performance issues are addressed by manually rewriting the application to make it set oriented. Such manual rewriting of programs is often time consuming and error prone. Guravannavar et. al.[1] propose program analysis and transformation techniques for automatically rewriting an application to make it set oriented. DBridge implements these program transformation techniques for Java programs that use JDBC to access database.

In this demonstration, we showcase the holistic program/query transformations that DBridge can perform, over a variety of scenarios taken from real-world applications. We then walk through the design of DBridge, which uses the SOOT optimization framework for static analysis. Finally, we demonstrate the performance gains achieved through the transformations.

I. Introduction

Database applications perform queries and updates from within procedural code that encodes business logic. Such applications use a mix of procedural constructs and SQL, and can run either inside the database system, as stored procedures, or outside the database system, as external programs. In such applications, iterative execution of parameterized queries is often the main cause of poor performance as it leads to random I/O and network round-trip delays.

The importance of set oriented execution is well known in the context of nested subqueries. Query decorrelation [2], [3], [4] addresses the problem of iterative execution of nested subqueries, by rewriting them using set operations such as joins. Thus, decorrelation enables set oriented plans with reduced random I/O. However, decorrelation techniques are not directly applicable to imperative program loops. Guravannavar et. al.[1] propose a program transformation approach for rewriting applications to use batched parameter bindings, thereby enabling set oriented execution of database queries. The key idea proposed by Guravannavar et. al.[1] is to replace repeated invocations of a query with a single call to the *batched form* or set oriented form of the query, which is often far more efficient than iterative invocation of the original query. To this end, Guravannavar et. al.[1] propose a set of program transformation rules which can be used to automatically rewrite a given program loop containing query invocation statements. The transformation rules make use of information about inter-statement data dependencies gathered from static analysis of the program. The proposed program transformation rules are powerful enough to rewrite a large class of loops involving complex control flow and arbitrary level of nesting.

DBridge is a program transformation tool based on the techniques presented in [1]. DBridge works on Java applications that use JDBC API [5] to access database. The tool performs static analysis of the input program and identifies opportunities for replacing iterative database access with set oriented access; it then rewrites the application code and the embedded queries together for set oriented processing. DBridge is designed to be a source-to-source transformation tool, and to this end, it ensures readability and maintainability of the transformed code. The tool is thus best suited for integration into an application development environment (IDE). DBridge can also be used as a preprocessing step inside a language compiler, thus making the compiler "database access aware".

II. Overview of DBridge

As an illustration of the kind of transformations DBridge can perform, consider the Java program snippet shown in Example 1.

```
Connection con = DriverManager.getConnection(url);
PreparedStatement pstmt = con.prepare(
    "SELECT count(partkey) FROM part WHERE category=?");
while(category != -1) {
    pstmt.setInt(1, category);
    ResultSet rs = pstmt.executeQuery();
    if (rs.next()) {
        partCount = rs.getInt(0);
        total += partCount;
    }
    category = getParent(category);
}
```

Example 1: A Program Snippet with JDBC Calls

```
Connection con = DriverManager.getConnection(dbrUrl);
PreparedStatement pstmt = con.prepare(
  "SELECT count(partkey) FROM part WHERE category=?");
while(category != -1) {
    pstmt.setInt(1, category);
    pstmt.addBatch();
    category = getParent(category);
}
pstmt.executeBatch();
while (pstmt.getMoreResults()) {
    ResultSet rs = pstmt.getResultSet();
    if (rs.next()) {
        partCount = rs.getInt(0);
        total += partCount;
    }
}
```

Example 2: JDBC Example after Transformation

The program computes the total number of parts in a given category and all its parent categories. Note the repeated execution of a parameterized aggregate query inside the *while* loop. The program snippet after transformation by DBridge is shown in Example 2. The transformed code is a result of the application of several transformation rules. We give an overview of some of the important transformations with the help of the example.

Statement Reordering: DBridge applies a set of transformation rules to reorder the statements within the loop body, so as to permit set oriented execution. In Example 2, note that the statement invoking *getParent*, is moved up ahead of the query execution. Statement reordering is performed taking inter-statement data dependencies into account. DBridge can also introduce temporary variables to break certain inter-statement data dependencies that otherwise prohibit set oriented execution; see [1], [6] for details.

Loop Splitting: This is the key transformation, described in [1] to enable set oriented execution. In Example 2, the loop in the original program is split into two parts. The first loop in the transformed program generates all the parameter bindings. Next, a rewritten form of the query is executed to obtain results for all the parameter bindings together. Then, the second loop executes statements that depend on the query results. The loop splitting transformation has certain preconditions for its applicability [1], and may require prior application of statement reordering so that the preconditions are met.

In general, the body of a loop may contain more than one parameterized query. Repeated application of the loop splitting transformation allows batching of any number of queries that lie inside the loop.

Query Rewrite: After collecting all the parameter bindings, the program calls the *executeBatch* method, which internally transforms the query statement into a set oriented form, which is often more efficient. For example, the scalar aggregate query in the example would be transformed into the following query, where *pb* is a temporary table in which the parameter bindings are materialized.

```
SELECT pb.category, le.c1
FROM pbatch pb,
     OUTER APPLY (SELECT count(partkey) as c1
                  FROM part
                  WHERE category=pb.category) le;
```

The rewritten query uses the OUTER APPLY construct of Microsoft SQL Server but can instead be written using a left outer join combined with the LATERAL construct of SQL:99. Most widely used database systems can unnest such a query into a form that uses joins or outer joins [4]. For example, the unnested form of the above query could be:

```
SELECT pb.category, count(partkey)
FROM pbatch pb LEFT OUTER JOIN part p
         ON pb.category=p.category
GROUP BY pb.category;
```

Such a rewriting enables the use of efficient set oriented query processing algorithms such as hash or merge join.

In Example 2, note the use of a different JDBC URL (*dbrUrl*) in the transformed program. DBridge wraps JDBC API in order to perform query rewrite as described above. The query rewrite is performed within DBrdge's implementation of the *executeBatch* method.

Rewrite of Conditional Blocks: DBridge can deal with conditional control transfer statements (*if-then-else*), and query execution statements inside conditional blocks. DBridge also handles *order-sensitive* operations within the loop correctly. Order-sensitive operations are operations whose order of execution is important for the correctness of the program. We illustrate these two features using Example 3. The program, after transformation, is shown in Example 4.

```
while(category != -1) {
    if(isActive(category)) {
        pstmt.bind(1, category);
        rs = pstmt.executeQuery();
        rs.next();
        partCount = rs.getInt(0);
        total += partCount;
        print(category, partCount);
    }
    category = getParent(category);
}
```

Example 3: Queries inside Conditional Blocks

DBridge first transforms conditional blocks into a sequence of guarded statements by introducing a boolean variable to remember the branching decision. It then applies the loop splitting transformation and replaces repeated execution of the query with a single invocation of its set oriented form. Finally, the sequence of guarded statements are merged back to have conditional blocks, as can be seen in the second loop of Example 4.

Queries, being side-effect free, can be executed in any order of the parameter bindings. However, the loop can contain other

```
LoopContextTable ctx;
while(category != -1) {
    boolean f = isActive(category);
    if(f) {
        pstmt.bind(1, category);
        pstmt.addBatch();
    }
    tempCat = category;
    category = getParent(category);
    ctx.addRecord(new Record(loopKey, f, tempCat));
}
pstmt.executeBatch();
// Now, read all the results and augment ctx with a new
// column partCount to hold the part count for each category.
ctx.mergeResults(pstmt);
for (Record r: ctx) { // We assume ctx to be an ordered table
    if(r.f) {
        total += r.partCount;
        print(r.tempCat, r.partCount);
    }
}
```

Example 4: Transformation of Example 3

order-sensitive operations, which must be executed in the same order as in the original program. To ensure this, the loop splitting transformation of DBridge maintains the loop context in an ordered table, and iterates over the loop context records in the order in which they are produced (see Example 4).

Nested Loops: A query execution statement can be inside a loop, which is nested within another loop. DBridge works with arbitrary levels of loop nesting. We omit an example in the interest of space, but include it as a case in the demonstration.

III. System Design and Implementation

DBridge is designed to meet the following requirements:

- **Semantics Preservation**: For a program transformation tool like DBridge, ensuring correctness is a *strict requirement*. DBridge ensures that the transformed program is *equivalent* in its functionality to the original program. DBridge's transformations are based on a set of formally defined equivalence rules, whose correctness proofs can be found in [6].

- **Robustness**: In our context, robustness is the ability of the tool to accommodate various kinds of programs, to successfully identify opportunities for set orientation, and to transform them. DBridge can deal with a variety of procedural language constructs, such as variable assignments, conditional control transfer, loops, method calls and field dereferences. Work to support exceptions is in progress. Further, DBridge employs inter-procedural data flow analysis to find data dependencies between program statements caused due to inter-procedural reads/writes. These abilities make DBridge applicable for a large number of real-world programs.

  It is not always possible to transform every query execution statement within a loop into a statement that uses set oriented form of the query. Inter-statement data dependencies may prohibit such a rewrite. The transformation rules and statement reordering algorithm which DBridge uses are powerful enough to transform a large number of the identified opportunities for set orientation. In fact, in [6] we prove the following: the transformation rules, together with our statement reordering algorithm can transform every query execution statement that does not have a cyclic data dependency on itself. Intuitively, DBridge can transform all iterative query execution statements except those such as the ones used for computing transitive closure of a relation.

- **Readability**: Programmers may need to read the transformed code to debug a program, or even to gain confidence in the correctness of the transformed code. Therefore, maintaining readability of the transformed code is very important. We achieve this goal through several measures. For instance, when we rewrite conditional blocks and then split a loop, the resulting code will have many guarded statements. We therefore introduce a pass where such guarded statements are grouped back in each of the two generated loops, so that the resulting code resembles the original code. Further, the transformed program uses standard JDBC calls, and contains very few calls to the DBridge runtime library.

- **Extensibility**: DBridge is designed in a way that provides an elegant framework for introducing new transformation rules or extending existing rules. Each rule is encapsulated as an object, and all the information necessary to apply a rule is provided by the framework via the program dependence graph.

The important phases in the program transformation process are shown in Figure 1. The input Java source file is first converted into an intermediate representation, on which we perform data flow analysis. Using this analysis, we construct a *Dependence Graph*, which is the basic data structure on which our transformation rules rely. The main task of our program transformation tool appears in the *Apply Trans Rules*

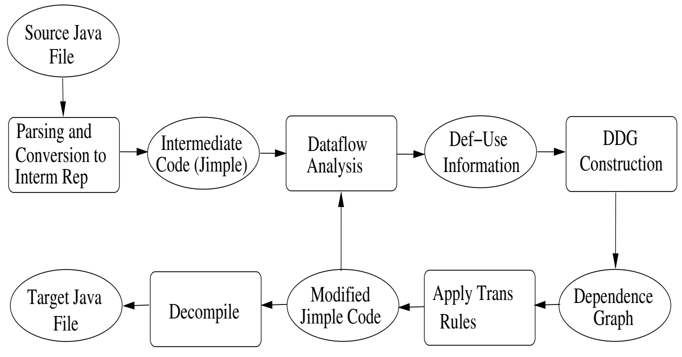

Fig. 1. Program Transformation Phases

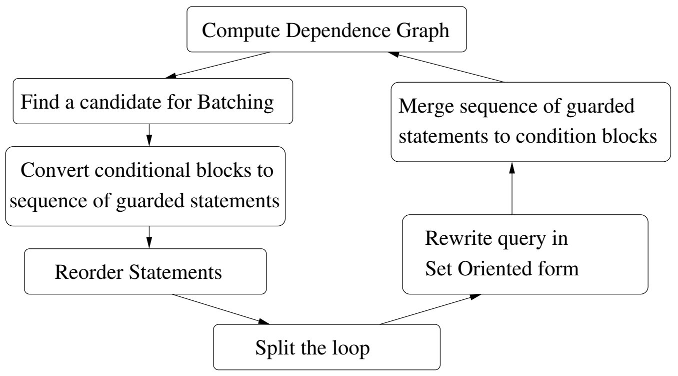
Compute Dependence Graph
Find a candidate for Batching
Convert conditional blocks to sequence of guarded statements
Reorder Statements
Split the loop
Rewrite query in Set Oriented form
Merge sequence of guarded statements to condition blocks

Fig. 2. Iterative application of Transformation rules

phase. The program transformation rules are applied in an iterative manner, updating the dataflow information each time the code changes. The rule application process stops when all (or the user chosen) query execution statements within loops are transformed into their corresponding set oriented forms. Once all the transformations are done, the intermediate representation is converted back to a target Java source file.

Applying Transformation rules

The program transformation rules presented in [6] can be applied repeatedly to refine a given program. Applying a rule to a program involves substituting a program fragment that matches the antecedent (LHS) of the rule with the program fragment instantiated by the consequent (RHS) of the rule. Some rules facilitate the application of other rules and together achieve the goal of batching a desired statement w.r.t. a loop.

In DBridge, the transformation rules are applied iteratively as shown in Figure 2. A query execution statement present within a loop body is considered as a *candidate for batching*, and for each such candidate, the rules are applied. First, conditional blocks if any, are converted into a sequence of guarded statements as explained through Example 3. Then the statement reordering algorithm is run, which enables loop splitting. The query is then rewritten into a set oriented form. Finally, the control structure of the program is restored by merging back guarded statements into conditional blocks, leading to a program with a set oriented query execution statement which lies outside the loop body.

Choice of technologies

Java with JDBC is one of the most widely used platform to build database applications. The JDBC API is the standard mode of connecting to a database from a Java program. Object relational mapping tools such as *Hibernate* provide a higher level of abstraction than JDBC, but internally use JDBC itself. So, by handling JDBC based programs, we cover a large class of database applications.

The availability of Soot [7], a powerful static analysis tool for Java, has also been a compelling reason to choose Java for developing this tool. Soot provides a convenient intermediate representation called Jimple, and also performs data flow analysis which enables us to compute inter-statement data dependencies.

IV. DEMONSTRATIONS

We demonstrate the working of DBridge using a number of programs, some of them taken from real-world applications, which faced performance problems. These applications include an employee stock option management system and an end-of-day processing system in a government organization. We also show the impact of using DBridge on publicly available benchmark applications [8], which model real world systems such as an auction system modeled after *ebay.com*, a bulletin board system modeled after *slashdot.org*. These programs highlight the applicability of DBridge, and also demonstrate the resulting gains in performance.

Along with programs such as Example 1, we demonstrate the transformation of (a) programs with nested loops, (b) programs with conditional blocks (Example 3), (c) programs that perform a series of INSERTs into ones that performs BULK INSERTs.

We have tested DBridge on a variety of Java programs in order to evaluate (a) its applicability for real world applications and (b) the performance gains that could be achieved by using such a tool. We have observed that, by the use of the sound techniques of reordering and other rules, DBridge can exploit most of the potential opportunities for batching, even though they are not explicit in the program. For the above mentioned benchmark applications, DBridge was able to transform a significant number of the opportunities. Our evaluation, which has been conducted on multiple database systems, shows performance gains achieved by the transformed programs to the extent of about 70% in several cases. The details of the evaluation are available in [6].

V. CONCLUSION

We present DBridge, a tool that combines program and query transformations to make query execution set oriented. We highlight challenges in building such a tool, and briefly outline how we addressed them. Using complex real-world programs we demonstrate the transformation capabilities of DBridge, and the potential performance gains due to such transformations.

REFERENCES

[1] R. Guravannavar and S. Sudarshan, “Rewriting Procedures for Batched Bindings,” in *Intl. Conf. on Very Large Databases*, 2008.

[2] W. Kim, “On Optimizing an SQL-like Nested Query,” in *ACM Trans. on Database Systems, Vol 7, No.3*, 1982.

[3] P. Seshadri, H. Pirahesh, and T. C. Leung, “Complex Query Decorrelation,” in *Intl. Conf. on Data Engineering*, 1996.

[4] M. Elhemali, C. A. Galindo-Legaria, T. Grabs, and M. M. Joshi, “Execution Strategies for SQL Subqueries,” in *ACM SIGMOD*, 2007.

[5] “Java Database Connectivity (JDBC) API http://java.sun.com/products/jdbc/overview.html.”

[6] R. Guravannavar, “Optimization and evaluation of nested queries and procedures,” Ph.D. dissertation, IIT Bombay, 2009.

[7] “Soot: A Java Optimization Framework.” [Online]. Available: http://www.sable.mcgill.ca/soot

[8] “ObjectWeb Consortium-JMOB (Java middleware open benchmarking).” [Online]. Available: http://jmob.ow2.org/