DBridge: Translating Imperative Code to SQL

K. Venkatesh Emani
Karthik Ramachandra[†]
Tejas Deshpande[*]
S. Sudarshan

Indian Institute of Technology, Bombay
{venkateshek, sudarsha}@cse.iitb.ac.in, {tejasdeshpande111, karthik.s.ramachandra}@gmail.com

ABSTRACT

Application programs that access data located remotely (such as in a database) often perform poorly due to multiple network round trips and transfer of unused data. This situation is exacerbated in applications that use object-relational mapping (ORM) frameworks such as Hibernate, as developers tend to express complex query logic using imperative code, resulting in poor performance.

DBridge is a system for optimizing data access in database applications by using static program analysis and program transformations. Recently, we incorporated a new suite of optimization techniques into DBridge. These techniques optimize database application programs by identifying relational operations expressed in imperative code, and translating them into SQL. In this demonstration, we showcase these techniques using a plugin for the IntelliJ IDEA Java IDE as the front end. We show the performance gains achieved by employing our system on real world applications that use JDBC or Hibernate.

1. INTRODUCTION

Database applications are written using a mix of declarative SQL queries, and imperative code written in languages such as Java. Developers of database applications express parts of relational logic in imperative code. In applications that use object-relational mapping (ORM) frameworks such as Hibernate, this is a particularly frequent occurrence, since developers working on objects find it easier to write complex query logic using imperative code, rather than writing queries. However, this practice can be inefficient because of multiple network round trips due to iterative invocation of queries, and transfer of unused data.

As a step towards addressing this issue, in our recent paper [3], we proposed techniques for automatically rewriting database applications by identifying various relational operations performed in imperative code, and translating them into equivalent SQL queries. Our techniques can detect operations in imperative code such as conditional execution, nested loops, and collection of results into an aggregate variable, and translate them into SQL queries making use of selections, joins, projections, and aggregations, while preserving the original program semantics. We refer to this approach as *equivalent SQL translation*.

Equivalent SQL translation pushes computation to the data location, and enables the query planner to exploit techniques such as join algorithms, indexes, materialized views etc. that could not be used on the original queries. Consequently, application performance is improved, as the number of database round trips, the query execution time, and the amount of data transferred are reduced. Results from [3] show significant performance improvements, up to an order of magnitude in some cases. Compared to other systems with similar goals such as [2], equivalent SQL translation using our system (a) is faster and cheaper, and (b) supports SQL translation of operations (such as some implementations of `group by`) that other systems are unable to translate.

In this demonstration, we will showcase equivalent SQL translation for database applications written in Java that use JDBC or Hibernate to access the database. The front end to our system is a plugin in the IntelliJ IDEA development environment (IntelliJ IDEA[1] is a popular IDE similar to Eclipse and Netbeans). The plugin provides a simple graphical user interface (GUI) to enable the user to configure and use our system. We discuss the features of our plugin in Section 3.

Our equivalent SQL translation system is part of the DBridge[2] [1] system developed at IIT Bombay, which optimizes data access in database applications by leveraging static program analysis and program transformations. DBridge performs multiple optimizations in imperative programs with embedded SQL queries. These optimizations, described in [4], include (a) batching, i.e., rewriting programs to replace multiple calls to a parameterized query by a batched call to a correspondingly rewritten query, (b) prefetching query results at the earliest possible location in the program while avoiding wasteful prefetches, (c) asynchronous query submission to enable overlap of query and program execution, and (d) hybrid optimizations that combine the above three approaches. These optimizations provide significant performance improvements; we refer the reader to [4] for more details. A previous demonstration of DBridge [1] showcased

[*]Current affiliation: Microsoft IDC
[†]Current affiliation: Microsoft Gray Systems Lab

SIGMOD'17, May 14-19, 2017, Chicago, IL, USA
 ISBN 978-1-4503-4197-4/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3035918.3058747

[1]https://www.jetbrains.com/idea/
[2]http://www.cse.iitb.ac.in/infolab/dbridge/

```
1 Set<Project> getUnfinishedProjects() {
2   Set<Project> unfinishedP = new HashSet<Project>();
3   List<Project> projects  =  projectDao.
                              getAllProjects();
    /* getAllProjects() internally uses Hibernate APIs
    to fetch all rows of Project table */
4   for  (Project  project  :  projects)
5     if  (!(project.getIsFinished()))
6       unfinishedP.add(project);

7   return  unfinishedP; }
```

Figure 1: Code performing filter inside application

batching. In this demonstration, we will focus on equivalent SQL translation (described in [3]), and use the opportunity to showcase other additions to DBridge since [1]. In the remainder of this paper, we will use the term DBridge to refer specifically to the equivalent SQL translation component of DBridge, unless explicitly specified otherwise.

2. OVERVIEW

In this section, we first discuss the capabilities of our system through examples, and then discuss the underlying techniques and plugin architecture, briefly.

2.1 Examples of SQL Translation

Our system is able to rewrite complex real world programs containing a mix of one or more of the following relational operations implemented in imperative code.

Selections: DBridge identifies filters on result attributes expressed using conditional imperative constructs such as `if`, and pushes them into the query as a selection. For example, consider the code fragment shown in Figure 1, which is adapted from an open source application [6]. It computes the list of unfinished projects by fetching all tuples and filtering them inside the application. DBridge extracts the following SQL query from the above program:

```
"select * from Project where isFinished <> 1"
```

The condition `!(project.getIsFinished())` from line 5 in Figure 1 is translated as the `where` clause condition in the extracted query.

Projections: DBridge generates SQL queries from an intermediate representation (refer Section 2.2). In the process, query result attributes that are fetched in the original code but not used subsequently, are eliminated.

Joins: Joins are implemented in imperative code using nested loops over query results. DBridge identifies and translates such joins to SQL. However, in applications using ORM frameworks, joins may also be implicitly specified. This is done by specifying associations (one of `many-to-many`, `one-to-many`, or `many-to-one`) between attributes of mapped classes. For example, consider the program shown in the left pane of Figure 2, which is a screenshot from one of the executions of our system. This function fetches all `RoleDefinition` objects mapped to a particular `Guidance` object. A `one-to-many` join association has been specified through ORM mapping, so accessing the `roleDefinitions` attribute in a `Guidance` object results in an implicit join. The right pane of Figure 2 shows how DBridge has identified this implicit join and translated it into an appropriate SQL query. The above example can be modified to iterate over a set of all `Guidances`, in which case, the SQL query generated by DBridge would include a join with the set of `Guidances`, which would be significantly more efficient than the original program.

Figure 2 has a method `saveOrUpdate` (line 22), which performs a database update. This illustrates another capability of DBridge. Although DBridge currently does not translate database updates to SQL, DBridge can translate reads to SQL as long as the interleaved updates do not introduce any write-read dependencies on the reads, and the reads do not span transaction boundaries.

Aggregations: DBridge identifies aggregation inside cursor loops, and replaces it with a query using an SQL aggregate function, if such a function is available. In some cases, if there is no equivalent aggregate SQL function, it may be possible to use a custom aggregation function. In other cases, translation into SQL fails. DBridge handles failures gracefully, as discussed in Section 3. DBridge is able to translate a common implementation of `group by` using nested loops, where the inner loop computes aggregation for each value of the outer loop.

Other operations: DBridge can identify scalar computations performed on attributes of query results in imperative code, and push these computations into SQL when possible. DBridge can detect and translate some other operations that can be expressed using SQL such as checking for existence of a tuple using a loop over query results, combining nested scalar queries inside a cursor loop (commonly encountered when data is organized as a star schema), etc. We refer the reader to Appendix B of [3] for more details. Similar to database query optimizer rules, more transformation rules can be added to DBridge easily, to enable inference of other relational operations from imperative code.

Note that in this paper, we show translations from imperative code to SQL. However, some users may prefer translation into HQL (Hibernate Query Language), as it has the advantage of being independent of the database engine being used. DBridge is able to generate HQL queries as well.

2.2 Underlying Techniques

We now present a brief overview of our techniques for equivalent SQL translation, described in detail in [3].

To detect potential opportunities for rewriting to SQL, DBridge primarily targets *cursor loops*, i.e., loops that iterate over a collection. Our examples in this paper use loops that iterate over collections that can be inferred as direct or indirect results of a database query. These are a special case of cursor loops. Generic cursor loops are handled similarly, after some preprocessing.

Given a source program (such as Figure 1) consisting of imperative code and embedded SQL, DBridge uses information about inter-statement data dependencies to identify program variables (like `unfinishedP`) whose computation inside cursor loops can be replaced by an SQL query. DBridge represents modifications to such a variable inside the loop using an algebraic intermediate representation (IR).

The IR for `unfinishedP` from Figure 1 is shown in Figure 3(a). Our IR is represented as a DAG. Edges are directed from an operator towards its operands, i.e., downwards in Figure 3 (directions have been omitted in the figure, for readability). The `fold` operator is used to represent cursor loops algebraically. The semantics of `fold` operator correspond to

```
/**
 * Get the set of RoleDefinitions for a specific Guidance
 */
@DBridge
public Set<RoleDefinition> getRoleDefinitions (Guidance _guidance) {
    Set<RoleDefinition> tmp = new HashSet<RoleDefinition>();
    guidanceDao.getSessionFactory().getCurrentSession().
            saveOrUpdate(_guidance);
    for (RoleDefinition td : _guidance.getRoleDefinitions()) {
        tmp.add(td);
    }
    return tmp;
}
```

```
/**
 * Get the set of RoleDefinitions for a specific Guidance
 */
@DBridge
public Set<RoleDefinition> getRoleDefinitions (Guidance _guidance) {
    Session session;
    Query query;
    guidanceDao.getSessionFactory().getCurrentSession().
            saveOrUpdate(_guidance);
    session = DBridgeUtils.getSession(guidanceDao);
    query = session.createSQLQuery("select T3.* from guidance " +
            "AS T1, roledefinition AS T3 where " +
            "T1.guidance_id = T3.roledefinition_id" +
            "And T1.guidance_id = :param0");
    query = query.setParameter("param0", _guidance.getId());
    return new HashSet(query.list());
}
```

Figure 2: Screenshot showing rewrite of implicit join in Hibernate (Left – original, Right – rewritten)

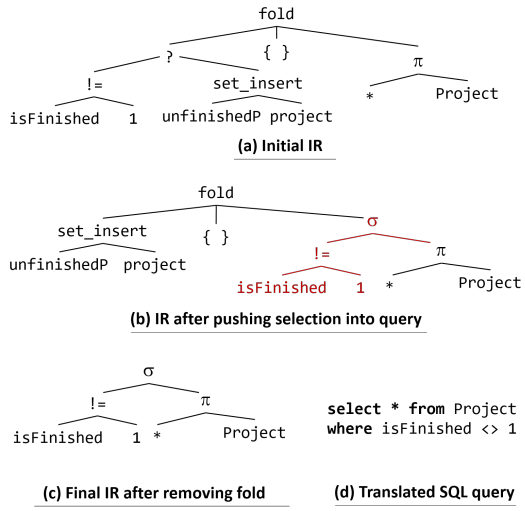

Figure 3: IR for `unfinishedP`

the higher order function `fold` in functional programming. Intuitively, in our IR, the first child of `fold` represents computation inside the loop body, the second child represents the initial value of the variable before entering the loop, and the third child represents the cursor loop query.

The '?' (question mark) operator is used to denote conditional execution. All arithmetic, logical, and extended relational algebra operators (including group by, sorting and duplicate elimination), and operators for important library functions (such as the `set_insert` operator to represent insertion into a set) are available in our IR, to enable representation of real world programs. We believe that this high level understanding of our IR is sufficient to follow this demonstration. We omit details for lack of space, and refer the reader to [3] for a complete discussion of our IR.

Transformations on the IR enable relational operations in the loop body to be identified and pushed into the relational algebra query. This can be seen in Figure 3(b), where computation inside the query increases as the selection logic is pushed into the query. Translating the transformed IR into SQL, and rewriting the program to use the extracted query gives the target program.

Note that DBridge is able to identify relevant code in the nested function call on line 3 of Figure 1, and represent it in the IR, along with code from the caller location. In general, DBridge can handle arbitrary levels of function call nesting without recursion.

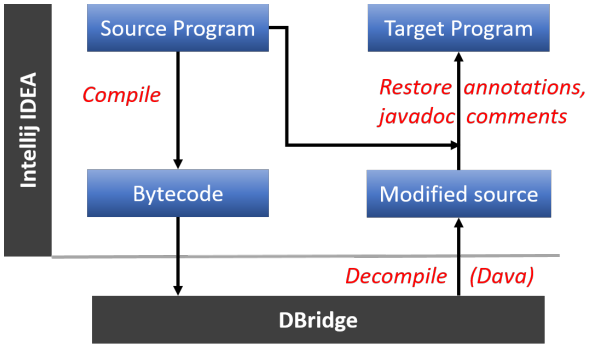

Figure 4: Plugin Architecture

2.3 Plugin Implementation

Figure 4 describes the architecture of our plugin. Our plugin is built on top of DBridge and IntelliJ IDEA. The input source program is compiled and the bytecode is passed to DBridge. DBridge leverages the Soot Java optimization framework [5], which translates the bytecode into *Jimple* – a convenient intermediate representation for program transformations. We use the Dava decompiler provided by Soot to convert transformed Jimple code back to Java.

However, comments and annotations from the original program are not preserved by Dava, and `for` loops are translated as `while` loops. So, we perform post processing of the decompiled file (using APIs provided by IntelliJ IDEA) to restore javadoc comments, annotations, and the original `for` loops, when those parts of the code were not modified by DBridge. Currently, if some part of the code is modified, we replace the function in the original program containing that code with the corresponding function in the rewritten program. We are working on extending our implementation to replace only the modified lines, instead of an entire function.

3. DEMONSTRATION

In this section, we describe our demonstration setup. Our demonstrations will showcase various features of our plugin for rewriting Java programs to use SQL, as well as the performance benefits of the transformed code.

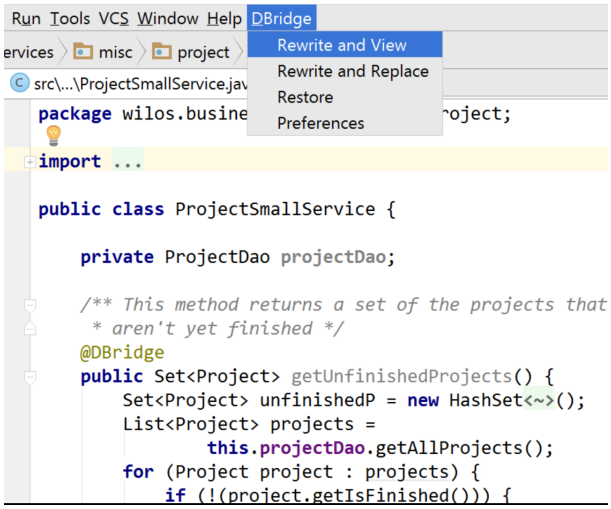

Figure 5: DBridge Rewrite Menu

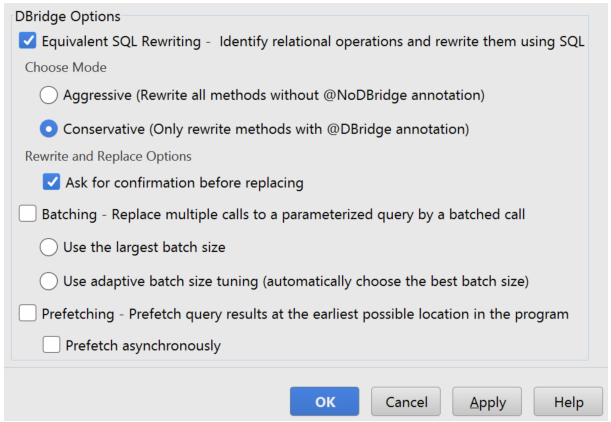

Figure 6: DBridge Preferences

The DBridge SQL translation plugin can be installed in the IntelliJ IDEA Java IDE, as a third party plugin. This adds a new main menu item in IntelliJ titled "DBridge". This is shown in Figure 5. The plugin identifies the currently active file from the IDE editor to be rewritten. Users can choose to rewrite the program, and do one of the following. (a) Compare the rewritten file with the original. This invokes IntelliJ's diff-viewer, a screenshot of which is shown in Figure 2. This option is provided by the "Rewrite and View" sub-menu item. (b) Directly replace the original file. This option is provided by the "Rewrite and Replace" sub-menu item. The "Restore" option allows the user to restore a file to its state before rewriting. "Rewrite and Replace" and "Restore" together enable the user to easily switch between the rewritten program and the original program, to facilitate testing and performance measurements.

Finally, the "Preferences" sub-menu opens a dialog to configure DBridge. This dialog is shown in Figure 6. For equivalent SQL translation, users can select one of two modes: *Aggressive* and *Conservative*. In the Aggressive mode, the plugin will attempt to rewrite all methods unless excluded using the annotation `@NoDBridge`. In the Conservative mode, only methods annotated with `@DBridge` will be considered for rewriting. In Figure 2, we ran the plugin in Conservative mode, so `getRoleDefinitions()` was annotated with `@DBridge` for rewriting. Note that DBridge may attempt to rewrite some parts of the code but fail, if the required preconditions are not met. In this case, that part of the code remains unchanged. Other parts of the code can still be attempted for rewrite as usual.

The dialog in Figure 6 also allows users to enable other optimizations in DBridge (discussed in Section 1), and configure them. The various optimizations in DBridge use source to source transformations and preserve program semantics, so multiple optimizations can be applied one after another on a single program.

Our demonstrations will use a number of Java programs from real world applications that use Hibernate and JDBC adapted from open source applications. Our experimental evaluation [3] shows that rewriting with SQL using DBridge is fast, and can provide significant performance improvements. For instance, for the program shown in Figure 1, using 20% selectivity and 100000 rows, the rewritten program ran 2.3x faster, and transferred 2.5x lesser data. (We omit further details on evaluation due to lack of space, and refer the reader to [3].) We will allow users to rewrite parts of example applications, and run the original and rewritten applications, to compare the execution times and amount of data transferred. We also invite the audience to play with DBridge SQL translation plugin using their own programs.

4. CONCLUSION

In this demonstration, we showcased (a) the DBridge system for rewriting database applications programs to use SQL, as a plugin for a popular Java IDE, (b) performance gains due to such rewriting. We plan to release our plugin, and believe that it will help generate efficient implementations with minimal developer effort.

We are developing a framework that considers alternative rewrites of a program, and chooses the optimal rewrite in a cost based manner. We also plan to extend our system to other languages such as R and JavaScript ORMs, in the future.

Acknowledgments: This work is partially supported by a Ph.D. fellowship from TCS.

5. REFERENCES

[1] M. Chavan, R. Guravannavar, K. Ramachandra, and S. Sudarshan. DBridge: A program rewrite tool for set-oriented query execution (demo). In *ICDE*, 2011.

[2] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing database-backed applications with query synthesis. PLDI, 2013.

[3] K. V. Emani, K. Ramachandra, S. Bhattacharya, and S. Sudarshan. Extracting Equivalent SQL from Imperative Code in Database Applications. SIGMOD, 2016.

[4] K. Ramachandra, M. Chavan, R. Guravannavar, and S. Sudarshan. Program transformations for asynchronous and batched query submission. *TKDE '15*, 27(2):531–544.

[5] Soot: A Java Optimization Framework http://www.sable.mcgill.ca/soot.

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