JReq: Database Queries in Imperative Languages

Ming-Yee Iu[1], Emmanuel Cecchet[2], and Willy Zwaenepoel[1]

[1] EPFL, Lausanne, Switzerland
[2] University of Massachusetts Amherst, Amherst, Massachusetts

Abstract. Instead of writing SQL queries directly, programmers often prefer writing all their code in a general purpose programming language like Java and having their programs be automatically rewritten to use database queries. Traditional tools such as object-relational mapping tools are able to automatically translate simple navigational queries written in object-oriented code to SQL. More recently, techniques for translating object-oriented code written in declarative or functional styles into SQL have been developed. For code written in an imperative style though, current techniques are still limited to basic queries. JReq is a system that is able to identify complex query operations like aggregation and nesting in imperative code and translate them into efficient SQL queries. The SQL code generated by JReq exhibits performance comparable with hand-written SQL code.

1 Introduction

Because of the widespread use of databases by computer programs, language designers have often sought to find natural and elegant ways for programmers to write database queries in general purpose programming languages. Although techniques have been developed to integrate database query support into functional languages, for imperative languages such as Java, current techniques are not yet able to handle complex database queries involving aggregation and nesting. Support for aggregation is important because it allows a program to calculate totals and averages across a large dataset without needing to transfer the entire dataset out of a database. Similarly, support for nesting one query inside another significantly increases the expressiveness of queries, allowing a program to group and filter data at the database instead of transferring the data to the program for processing.

We have developed an approach for allowing programmers to write complex database queries inside the imperative language Java. Queries can be written using the normal imperative Java style for working with large datasets—programmers use loops to iterate over the dataset. The queries are valid Java code, so no changes are needed to the Java language to support these complex queries. To run these queries efficiently on common databases, the queries are translated into SQL using an algorithm based on symbolic execution. We have implemented these algorithms in a system called JReq.

These are the main technical contributions of this work: a) We demonstrate how complex queries can be written in Java code using loops and iterators. We call this programming style the JReq Query Syntax (JQS) b) We describe an algorithm that can robustly translate complex imperative queries involving aggregation and nesting into SQL c) We have implemented this algorithm in JReq and evaluated its performance.

2 Background

Currently, the most common interface for accessing database queries from Java is to use a low-level API like JDBC. With JDBC, queries are entirely separated from Java. They are written in the domain-specific language SQL, they are stored in strings (which must be compiled and error-checked at runtime), and programmers must manually marshal data into and out of queries (Fig. 12).

Object-oriented databases [11] and object-relational mapping tools like Hibernate, Ruby on Rails, or EJB3 provide a higher-level object-oriented API for accessing databases. Although these tools provide support for updates, error-handling, and transactions, their support for queries is limited. Traditional object-oriented operations such as navigational queries are well-supported, but relational-style queries that filter or manipulate datasets must still be encoded in strings and data must still be manually marshaled into and out of queries. Figure 1 shows an example of such a query written using the Java Persistence API [5].

```
List l = em.createQuery("SELECT a FROM Account a "
  + "WHERE 2 * a.balance < a.creditLimit AND a.country = :country")
  .setParameter("country", "Switzerland")
  .getResultList();
```

Fig. 1. A sample query written in the Java Persistence Query Language (JPQL)

In imperative languages like Java, the normal style for filtering and manipulating large datasets is for a programmer to use loops to iterate over the dataset. As a result, researchers have tried to develop systems that allow programmers to write database queries in imperative languages using such a syntax. We have previously developed a system called Queryll [8] that was able to translate simple queries written in an imperative form to SQL. The system made use of fairly ad hoc algorithms that could not be scaled to support more complex queries involving nesting or aggregation. Wiedermann, Ibrahim, and Cook [19,20] have also successfully translated queries written in an imperative style into SQL. They use abstract interpretation and attribute grammars to translate queries written in Java into database queries. Their work focuses on gathering the objects and fields traversed by program code into a single query (similar to the optimisations performed by Katz and Wong [9]) and is also able to recognise simple filtering constraints. Their approach lacks a mechanism for inferring loop invariants and hence cannot handle queries involving aggregation or complex nesting since these operations span multiple loop iterations.

An alternate approach for supporting complex database queries in imperative languages is to incorporate declarative and functional language features into the languages. Kleisli [21] demonstrated that it was possible to translate queries written in a functional language into SQL. Microsoft was able to add query support to object-oriented languages by extending them with declarative and functional extensions in a feature called Language INtegrated Query (LINQ) [15]. LINQ adds a declarative syntax to .Net languages by allowing programmers to specify SQL-style SELECT..FROM..WHERE queries from within these languages. This syntax is then internally converted to a functional style in the form of lambda expressions, which is then translated to SQL at runtime. Unfortunately, adding similar query support to an imperative programming language like Java without adding specific syntax support for declarative or functional programming results in extremely verbose queries [4].

The difficulty of translating imperative program code to a declarative query language can potentially be avoided entirely by translating imperative program code to an imperative query language. The research of Liewen and DeWitt [10] or of Guravannavar and Sudarshan [7] demonstrate dataflow analysis techniques that could be used for such a system. Following such an approach would be impractical though because all common query languages are specifically designed to be declarative because declarative query languages allow for more optimisation possibilities.

3 JReq Query Syntax

The JReq system allows programmers to write queries using normal Java code. JReq is not able to translate arbitrary Java code into database queries, but queries written in a certain style. We call the subset of Java code that can be translated by JReq into SQL code the JReq Query Syntax (JQS). Although this style does impose limitations on how code must be written, it is designed to be as unrestrictive as possible.

3.1 General Approach and Syntax Examples

Databases are used to store large amounts of structured data, and the most common coding convention used for examining large amounts of data in Java is to iterate over collections. As such, JReq uses this syntax for expressing its queries. JQS queries are generally composed of Java code that iterates over a collection of objects from a database, finds the ones of interest, and adds these objects to a new collection (Fig. 2). For each table of the database, a method exists that returns all the data from that table, and a special collection class called a QueryList is provided that has extra methods to support database operations like set operations and sorting.

```
QueryList<String> results = new QueryList<String>();
for (Account a: db.allAccounts())
   if (a.getCountry().equals("UK"))
      results.add(a.getName());
```

Fig. 2. A more natural Java query syntax

JQS is designed to be extremely lenient in what it accepts as queries. For simple queries composed of a single loop, arbitrary control-flow is allowed inside the loop as long as there are no premature loop exits nor nested loops (nested loops are allowed if they follow certain restrictions), arbitrary creation and modification of variables are allowed as long as they are scoped to the loop, and methods from a long list of safe methods can be called. At most one value can be added to the result-set per loop iteration, and the result-set can only contain numbers, strings, entities, or tuples. Since JReq translates its queries into SQL, the restrictions for more complex queries, such as how queries can be nested or how variables should be scoped, are essentially the same as those of SQL.

One interesting property of the JQS syntax for queries is that the code can be executed directly, and executing the code will produce the correct query result. Of course, since one might be iterating over the entire contents of a database in such a query, executing the code directly might be unreasonably slow. To run the query efficiently, the query must eventually be rewritten in a database query language like SQL instead. This rewriting essentially acts as an optional optimisation on the existing code. Since no changes to the Java language are made, all the code can compile in a normal Java compiler, and the compiler will be able to type-check the query statically. No verbose, type-unsafe data marshaling into and out of the query is used in JQS.

In JQS, queries can be nested, values can be aggregated, and results can be filtered in more complex ways. JQS also supports navigational queries where an object may have references to various related objects. For example, to find the customers with a total balance in their accounts of over one million, one could first iterate over all customers. For each customer, one could then use a navigational query to iterate over his or her accounts and sum up the balance.

```
QueryList results = new QueryList();
for (Customer c: db.allCustomer()) {
   double sum = 0;
   for (Account a: c.getAccounts())
      sum += a.getBalance();
   if (sum > 1000000) results.add(c);
}
```

Intermediate results can be stored in local variables and results can be put into groups. In the example below, a map is used to track (key, value) pairs of the number of students in each department. In the query, local variables are freely used.

```
QueryMap<String, Integer> students =
      new QueryMap<String, Integer>(0);
for (Student s: db.allStudent()) {
   String dept = s.getDepartment();
   int count = students.get(dept) + 1;
   students.put(dept, count);
}
```

Although Java does not have a succinct syntax for creating new database entities, programmers can use tuple objects to store multiple result values from a query (these tuples are of fixed size, so query result can still be mapped from flat relations and do not require nested relations). Results can also be stored in sets instead of lists in order to query for unique elements only, such as in the example below where only unique teacher names (stored in a tuple) are kept.

```
QuerySet teachers = new QuerySet();
for (Student s: db.allStudent()) {
   teachers.add(new Pair(
         s.getTeacher().getFirstName(),
         s.getTeacher().getLastName()));
}
```

In order to handle sorting and limiting the size of result sets, the collection classes used in JQS queries have extra methods for sorting and limiting. The JQS sorting syntax is similar to Java syntax for sorting in its use of a separate comparison object. In the query below, a list of supervisors is sorted by name and all but the first 20 entries are discarded.

```
QuerySet<Supervisor> supervisors = new QuerySet<Supervisor>();
for (Student s: db.allStudent())
   supervisors.add(s.getSupervisor());
supervisors
   .sortedByStringAscending(new StringSorter<Supervisor>() {
       public String value(Supervisor s) {return s.getName();}})
   .firstN(20);
```

For certain database operations that have no Java equivalent (such as SQL regular expressions or date arithmetic), utility methods are provided that support this functionality.

4 Translating JQS Using JReq

In the introduction, it was mentioned that imperative Java code must be translated into a declarative form in order to be executed efficiently on a database. This section explains this translation process using the query from Fig. 2 as an example.

Since JQS queries are written using actual Java code, the JReq system cannot be implemented as a simple Java library. JReq must be able to inspect and

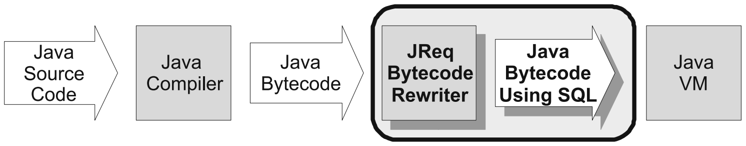


Fig. 3. JReq inserts itself in the middle of the Java toolchain and does not require changes to existing tools

modify Java code in order to identify queries and translate them to SQL. A simple Java library cannot do that. One of our goals for JReq, though, is for it to be non-intrusive and for it to be easily adopted or removed from a development process like a normal library. To do this, the JReq system is implemented as a bytecode rewriter that is able to take a compiled program outputted by the Java compiler and then transform the bytecode to use SQL. It can be added to the toolchain as an independent module, with no changes needed to existing IDEs, compilers, virtual machines, or other such tools (Fig. 3). Although our current implementation has JReq acting as an independent code transformation tool, JReq can also be implemented as a post-processing stage of a compiler, as a classloader that modifies code at runtime, or as part of a virtual machine.

The translation algorithm in JReq is divided into a number of stages. It first preprocesses the bytecode to make the bytecode easier to manipulate. The code is then broken up into loops, and each loop is transformed using symbolic execution into a new representation that preserves the semantics of the original code but removes many secondary features of the code, such as variations in instruction ordering, convoluted interactions between different instructions, or unusual control flow, thereby making it easier to identify queries in the code. This final representation is tree-structured, so bottom-up parsing is used to match the code with general query structures, from which the final SQL queries can then be generated.

4.1 Preprocessing

Although JReq inputs and outputs Java bytecode, its internal processing is not based on bytecode. Java bytecode is difficult to process because of its large instruction set and the need to keep track of the state of the operand stack. To avoid this problem, JReq uses the SOOT framework [18] from Sable to convert Java bytecode into a representation known as Jimple, a three-address code version of Java bytecode. In Jimple, there is no operand stack, only local variables, meaning that JReq can use one consistent abstraction for working with values and that JReq can rearrange instruction sequences without having to worry about stack consistency. Figure 4 shows the code of the query from Fig. 2 after conversion to Jimple form.

```
                $accounts = $db.allAccounts()
                $iter = $accounts.iterator()
                goto loopCondition
loopBody:       $next = $iter.next()
                $a = (Account) $next
                $country = $a.getCountry()
                $cmp0 = $country.equals("UK")
                if $cmp0==0 goto loopCondition
loopAdd:        $name = a$.getName()
                $results.add($name)
loopCondition:  $cmp1 = $iter.hasNext()
                if $cmp1!=0 goto loopBody
exit:
```

Fig. 4. Jimple code of a query

4.2 Transformation of Loops

Since all JQS queries are expressed as loops iterating over collections, JReq needs to add some structure to the control-flow graph of the code. It breaks down the control flow graph into nested strongly-connected components (i.e. loops), and from there, it transforms and analyses each component in turn. Since there is no useful mapping from individual instructions to SQL queries, the analysis operates on entire loops. Conceptually, JReq calculates the postconditions of executing all of the instructions of the loop and then tries to find SQL queries that, when executed, produce the same set of postconditions. If it can find such a match, JReq can replace the original code with the SQL query. Since the result of executing the original series of instructions from the original code gives the same result as executing the query, the translation is safe. Unfortunately, because of the difficulty of generating useful loop invariants for loops [3], JReq is not able to calculate postconditions for a loop directly.

JReq instead examines each loop iteration separately. It starts at the entry point to the loop and walks the control flow graph of the loop until it arrives back at the loop entry point or exits the loop. As it walks through the control flow graph, JReq enumerates all possible paths through the loop. The possible paths through the query code from Fig. 4 are listed in Fig. 5. Theoretically, there can be an exponential number of different paths through a loop since each `if` statement can result in a new path. In practise, such an exponential explosion in paths is rare. Our Java query syntax has an interesting property where when an `if` statement appears in the code, one of the branches of the statement usually ends that iteration of the loop, meaning that the number of paths generally grows linearly. The only type of query that seems to lead to an exponential number of paths are ones that try to generate "CASE WHEN...THEN" SQL code, and these types of queries are rarely used. Although we do not believe exponential path explosion to be a problem for JReq, such a situation can be avoided by using techniques developed by the verification community for dealing with similar problems [6].

Type	**Path**
Exiting	loopCondition → exit
Looping	loopCondition → loopBody →↻
Looping	loopCondition → loopBody → loopAdd →↻

Fig. 5. Paths through the loop

```
Path: loopCondition → loopBody → loopAdd →↻
   Preconditions   $iter.hasNext() != 0
                   ((Account)$iter.next()).getCountry().equals("UK") != 0
   Postconditions     $iter.hasNext()
                      $cmp1 = $iter.hasNext()
                   $iter.next()
                      $next = $iter.next()
                      $a = (Account) $iter.next()
                      ((Account)$iter.next()).getCountry()
                      $country = ((Account)$iter.next()).getCountry()
                      ((Account)$iter.next()).getCountry().equals("UK")
                      $cmp0 = ((Account)$iter.next()).getCountry().equals("UK")
                      ((Account)$iter.next()).getName()
                      $name = ((Account)$iter.next()).getName()
                   $results.add(((Account)$iter.next()).getName())
```

Fig. 6. Hoare triple expressing the result of a path (expressions that will be pruned by liveness analysis are indented)

For each path, JReq generates a Hoare triple. A Hoare triple describes the effect of executing a path in terms of the preconditions, code, and postconditions of the path. JReq knows what branches need to be taken for each path to be traversed, and the conditions on these branches form the preconditions for the paths. Symbolic execution is used to calculate the postconditions for each path. Essentially, all variable assignments and method calls become postconditions. The use of symbolic execution means that all preconditions and postconditions are expressed in terms of the values of variables from the start of the loop iteration and that minor changes to the code like simple instruction reordering will not affect the derived postconditions. There are many different styles of symbolic execution, and JReq's use of symbolic execution to calculate Hoare triples is analogous to techniques used in the software verification community, particularly work on translation validation and credible compilation [14,12].

Figure 6 shows the different preconditions and postconditions of the last path from Fig. 5. Not all of the postconditions gathered are significant though, so JReq uses variable liveness information to prune assignments that are not used outside of a loop iteration and uses a list of methods known not to have side-effects to prune safe method calls. Figure 7 shows the final Hoare triples of all paths after pruning.

Basically, JReq has transformed the loop instructions into a new tree representation where the loop is expressed in terms of paths and various precondition

Exiting Path	
Preconditions	\$iter.hasNext() == 0
Postconditions	
Looping Path	
Preconditions	\$iter.hasNext() != 0
	((Account)\$iter.next()).getCountry().equals(”UK”) == 0
Postconditions	\$iter.next()
Looping Path	
Preconditions	\$iter.hasNext() != 0
	((Account)\$iter.next()).getCountry().equals(”UK”) != 0
Postconditions	\$iter.next()
	\$results.add(((Account)\$iter.next()).getName())

Fig. 7. Final Hoare triples generated from Fig. 4 after pruning

and postcondition expressions. The semantics of the original code are preserved in that all the effects of running the original code are encoded as postconditions in the representation, but problems with instruction ordering or tracking instruction side-effects, etc. have been filtered out.

In general, JReq can perform this transformation of loops into a tree representation in a mechanical fashion, but JReq does make some small optimisations to simplify processing in later stages. For example, constructors in Java are methods with no return type. In JReq, constructors are represented as returning the object itself, and JReq reassigns the result of the constructor to the variable on which the constructor was invoked. This change means that JReq does not have to keep track of a separate method invocation postcondition for each constructor used in a loop.

4.3 Query Identification and Generation

Once the code has been transformed into Hoare triple form, traditional translation techniques can be used to identify and generate SQL queries. For example, Fig. 8 shows how one general Hoare triple representation can be translated into a corresponding SQL form. That particular Hoare triple template is sufficient to match all non-nested SELECT...FROM...WHERE queries without aggregation functions. In fact, because the transformation of Java code into Hoare triple form removes much of the syntactic variation between code fragments with identical semantics, a small number of templates is sufficient to handle most queries.

Since the Hoare triple representation is in a nice tree form, our implementation uses bottom-up parsing to classify and translate the tree into SQL. When using bottom-up parsing to match path Hoare triples to a template, one does have to be careful that each path add the same number and same types of data to the result collection (e.g. in Fig. 8, one needs to check that the types of the various $valA_n$ being added to $\$results$ is consistent across the looping paths). One can use a unification algorithm across the different paths of the loop to ensure that these consistency constraints hold.

Exiting Path		
	Preconditions	\$iter.hasNext() == 0
	Postconditions	*exit loop*
Looping Path$_i$		
	Preconditions	\$iter.hasNext() != 0
		...
	Postconditions	\$iter.next()
...etc.		
Looping Path$_n$		
	Preconditions	\$iter.hasNext() != 0
		$\boldsymbol{pred}_n$
	Postconditions	\$iter.next()
		\$results.add($\boldsymbol{valA}_n$, $\boldsymbol{valB}_n$, ...)
...etc.		

```
SELECT
   CASE WHEN pred_1 THEN valA_1
        WHEN pred_2 THEN valA_2
        ...
   END,
   CASE WHEN pred_1 THEN valB_1
        WHEN pred_2 THEN valB_2
        ...
   END,
   ...
FROM ?
WHERE pred_1 OR pred_2 OR ...
```

Fig. 8. Code with a Hoare triple representation matching this template can be translated into a SQL query in a straight-forward way

One further issue complicating query identification and generation is the fact that a full JQS query is actually composed of both a loop portion and some code before and after the loop. For example, the creation of the object holding the result set occurs before the loop, and when a loop uses an iterator object to iterate over a collection, the definition of the collection being iterated over can only be found outside of the loop. To find these non-loop portions of the query, we recursively apply the JReq transformation to the code outside of the loop at a higher level of nesting. Since the JReq transformation breaks down a segment of code into a finite number of paths to which symbolic execution is applied, the loop needs to be treated as a single indivisible "instruction" whose postconditions are the same as the loop's postconditions during this recursion. This recursive application of the JReq transformation is also used for converting nested loops into nested SQL queries. Figure 9 shows the Hoare triples of the loop and non-loop portions of the query from Fig. 2.

Figure 10 shows some sample operational semantics that illustrate how the example query could be translated to SQL. In the interest of space, these operational semantics do not contain any error-checking and show only how to match the specific query from Fig. 2 (as opposed to the general queries supported by JReq). The query needs to be processed three times using mappings S, F, and W to generate SQL select, from, and where expressions respectively. σ holds information about variables defined outside of a loop. In this example, σ describes the table being iterated over, and Σ describes how to look up fields of this table.

JReq currently generates SQL queries statically by replacing the bytecode for the JQS query with bytecode that uses SQL instead. Static query generation allows JReq to apply more optimisations to its generated SQL output and makes debugging easier because we can examine generated queries without running the program. During this stage, JReq can also optimise the generated SQL queries for specific databases though our prototype currently does not contain such an optimiser. In a previous version of JReq, SQL queries were constructed at

```
Hoaretriples(
  Exit(
    Pre($iter.hasNext() == 0),
    Post()
  ),
  Looping(
    Pre($iter.hasNext() != 0,
        ((Account)$iter.next()).getCountry().equals("UK") == 0),
    Post(Method($iter.next()))
  ),
  Looping(
    Pre($iter.hasNext() != 0,
        ((Account)$iter.next()).getCountry().equals("UK") != 0),
    Post(Method($iter.next()),
         Method($uk.add(((Account)$iter.next()).getName()))))
```

```
PathHoareTriple(
   Pre(),
   Post($results = (new QueryList()).addAll(
            $db.allAccounts().iterator().AddQuery())))
```

Fig. 9. The Hoare triples of the loop and non-loop portion of the query from Fig. 2. The loop Hoare triples are identical to those from Fig. 7, except they have been rewritten so as to emphasise the parsability and tree-like structure of the Hoare triple form.

runtime and evaluated lazily. Although this results in slower queries, it allows the system to support a limited form of inter-procedural query generation. A query can be created in one method, and the query result can later be refined in another method.

During query generation, JReq uses line number debug information from the bytecode to show which lines of the original source files were translated into SQL queries and what they were translated into. IDEs can potentially use this information to highlight which lines of code can be translated by JReq as a programmer types them. Combined with the type error and syntax error feedback given by the Java compiler at compile-time, this feedback helps programmers write correct queries and optimise query performance.

4.4 Implementation Expressiveness and Limitations

The translation algorithm behind JReq is designed to be able to recognise queries with the complexity of SQL92 [1]. In our implementation though, we focused on the subset of operations used in typical SQL database queries. Figure 11 shows a grammar of JQS, the Java code that JReq can translate into SQL. We specify JQS using the grammar of Hoare triples from after the symbolic execution stage of JReq. We used this approach because it is concise and closely describes what

$$\dfrac{\begin{array}{l} a = \texttt{Exit(Pre(\$iter.hasNext()==0), Post())} \\ b = \texttt{Looping(Pre(\$iter.hasNext()!=0, ...),} \\ \qquad\qquad \texttt{Post(Method(\$iter.next())))} \\ c = \texttt{Looping(Pre(\$iter.hasNext()!=0, } d\texttt{),} \\ \qquad\qquad \texttt{Post(Method(\$iter.next()), } e\texttt{))} \\ e = \texttt{Method(}\mathit{resultset}\texttt{.add(}\mathit{child}\texttt{))} \\ S \vdash \langle \mathit{child}, \sigma \rangle \Downarrow \mathit{select} \\ W \vdash \langle d, \sigma \rangle \Downarrow \mathit{where} \end{array}}{\begin{array}{l} S \vdash \langle \texttt{Hoaretriples}(a,b,c), \sigma \rangle \Downarrow \mathit{select} \\ W \vdash \langle \texttt{Hoaretriples}(a,b,c), \sigma \rangle \Downarrow \mathit{where} \end{array}}$$

$$\dfrac{\begin{array}{c} W \vdash \langle \mathit{left}, \sigma \rangle \Downarrow \mathit{where}_l \\ W \vdash \langle \mathit{right}, \sigma \rangle \Downarrow \mathit{where}_r \end{array}}{W \vdash \langle \mathit{left}\texttt{.equals(}\mathit{right}\texttt{)==0}, \sigma \rangle \Downarrow \mathit{where}_l \texttt{<>} \mathit{where}_r}$$

$$\dfrac{\begin{array}{l} W \vdash \langle \mathit{left}, \sigma \rangle \Downarrow \mathit{where}_l \\ W \vdash \langle \mathit{right}, \sigma \rangle \Downarrow \mathit{where}_r \end{array}}{W \vdash \langle \mathit{left}\texttt{.equals(}\mathit{right}\texttt{)!=0}, \sigma \rangle \Downarrow \mathit{where}_l = \mathit{where}_r} \qquad \dfrac{}{\begin{array}{l} S \vdash \langle \texttt{"UK"}, \sigma \rangle \Downarrow \textbf{``UK''} \\ W \vdash \langle \texttt{"UK"}, \sigma \rangle \Downarrow \textbf{``UK''} \end{array}}$$

$$\dfrac{\Sigma \vdash \langle \mathit{child}, \sigma, \textsc{Name} \rangle \Downarrow \mathit{val}}{\begin{array}{l} S \vdash \langle \mathit{child}\texttt{.getName()}, \sigma \rangle \Downarrow \mathit{val} \\ W \vdash \langle \mathit{child}\texttt{.getName()}, \sigma \rangle \Downarrow \mathit{val} \end{array}} \qquad \dfrac{\Sigma \vdash \langle \mathit{child}, \sigma, \textsc{Country} \rangle \Downarrow \mathit{val}}{\begin{array}{l} S \vdash \langle \mathit{child}\texttt{.getCountry()}, \sigma \rangle \Downarrow \mathit{val} \\ W \vdash \langle \mathit{child}\texttt{.getCountry()}, \sigma \rangle \Downarrow \mathit{val} \end{array}}$$

$$\dfrac{}{\begin{array}{l} \Sigma \vdash \langle \texttt{(Account)\$iter.next()}, \sigma, \textsc{Country} \rangle \Downarrow \sigma(\textsc{next})\textbf{.Country} \\ \Sigma \vdash \langle \texttt{(Account)\$iter.next()}, \sigma, \textsc{Name} \rangle \Downarrow \sigma(\textsc{next})\textbf{.Name} \end{array}}$$

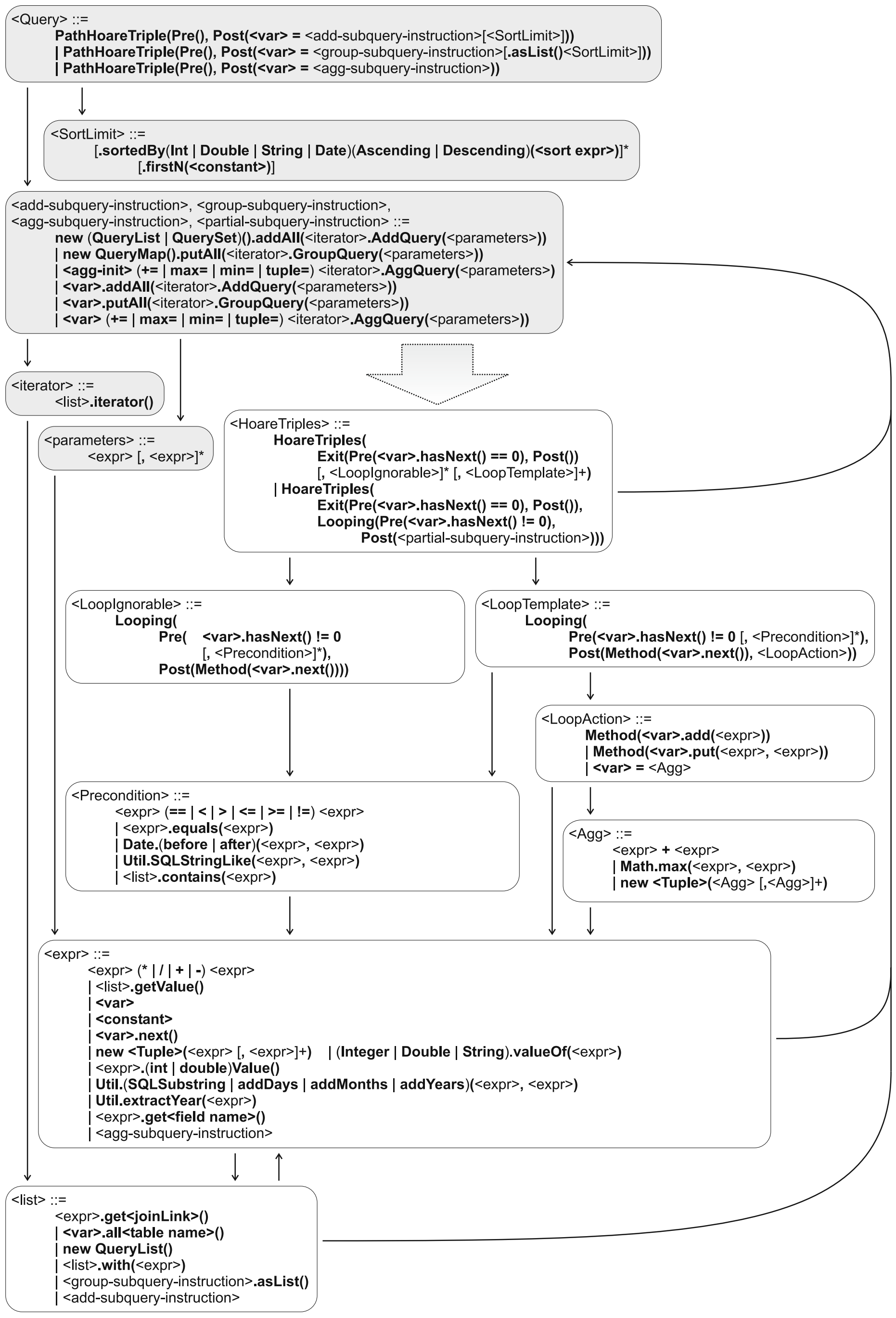
$$\dfrac{}{F \vdash \langle \texttt{\$db.allAccounts().iterator()}, \sigma \rangle \Downarrow \textbf{Account}}$$

$$\dfrac{\begin{array}{l} S \vdash \langle \texttt{HoareTriples(...)}, \sigma[\textsc{next} := \mathbf{A}] \rangle \Downarrow \mathit{select} \\ W \vdash \langle \texttt{HoareTriples(...)}, \sigma[\textsc{next} := \mathbf{A}] \rangle \Downarrow \mathit{where} \\ F \vdash \langle \mathit{iterator}, \sigma \rangle \Downarrow \mathit{from} \end{array}}{\begin{array}{l} \langle \mathit{resultset}\texttt{.addAll(}\mathit{iterator}\texttt{.AddQuery())}, \sigma \rangle \Downarrow \\ \qquad \textbf{SELECT } \mathit{select} \textbf{ FROM } \mathit{from} \textbf{ AS A WHERE } \mathit{where} \end{array}}$$

Fig. 10. Sample operational semantics for translating Fig. 9 to SQL

queries will be accepted. We have found that specifying JQS using a traditional grammar directly describing a Java subset to be too imprecise or too narrow to be useful. Because JReq uses symbolic execution, for each query, any Java code variant with the same semantic meaning will be recognised by JReq as being the same query. This large number of variants cannot be captured using a direct specification of a Java grammar subset.

In the figure, the white boxes refer to grammar rules used for classifying loops. The grey boxes are used for combining loops with context from outside of the loop. There are four primary templates for classifying a loop: one for adding



```
<Query> ::=
        PathHoareTriple(Pre(), Post(<var> = <add-subquery-instruction>[<SortLimit>]))
        | PathHoareTriple(Pre(), Post(<var> = <group-subquery-instruction>[.asList()<SortLimit>]))
        | PathHoareTriple(Pre(), Post(<var> = <agg-subquery-instruction>))

<SortLimit> ::=
        [.sortedBy(Int | Double | String | Date)(Ascending | Descending)(<sort expr>)]*
                [.firstN(<constant>)]

<add-subquery-instruction>, <group-subquery-instruction>,
<agg-subquery-instruction>, <partial-subquery-instruction> ::=
        new (QueryList | QuerySet)().addAll(<iterator>.AddQuery(<parameters>))
        | new QueryMap().putAll(<iterator>.GroupQuery(<parameters>))
        | <agg-init> (+= | max= | min= | tuple=) <iterator>.AggQuery(<parameters>)
        | <var>.addAll(<iterator>.AddQuery(<parameters>))
        | <var>.putAll(<iterator>.GroupQuery(<parameters>))
        | <var> (+= | max= | min= | tuple=) <iterator>.AggQuery(<parameters>))

<iterator> ::=
        <list>.iterator()

<parameters> ::=
        <expr> [, <expr>]*

<HoareTriples> ::=
        HoareTriples(
                Exit(Pre(<var>.hasNext() == 0), Post())
                [, <LoopIgnorable>]* [, <LoopTemplate>]+)
        | HoareTriples(
                Exit(Pre(<var>.hasNext() == 0), Post()),
                Looping(Pre(<var>.hasNext() != 0),
                        Post(<partial-subquery-instruction>)))

<LoopIgnorable> ::=
        Looping(
                Pre(    <var>.hasNext() != 0
                        [, <Precondition>]*),
                Post(Method(<var>.next())))

<LoopTemplate> ::=
        Looping(
                Pre(<var>.hasNext() != 0 [, <Precondition>]*),
                Post(Method(<var>.next()), <LoopAction>))

<LoopAction> ::=
        Method(<var>.add(<expr>))
        | Method(<var>.put(<expr>, <expr>))
        | <var> = <Agg>

<Precondition> ::=
        <expr> (== | < | > | <= | >= | !=) <expr>
        | <expr>.equals(<expr>)
        | Date.(before | after)(<expr>, <expr>)
        | Util.SQLStringLike(<expr>, <expr>)
        | <list>.contains(<expr>)

<Agg> ::=
        <expr> + <expr>
        | Math.max(<expr>, <expr>)
        | new <Tuple>(<Agg> [,<Agg>]+)

<expr> ::=
        <expr> (* | / | + | -) <expr>
        | <list>.getValue()
        | <var>
        | <constant>
        | <var>.next()
        | new <Tuple>(<expr> [, <expr>]+)    | (Integer | Double | String).valueOf(<expr>)
        | <expr>.(int | double)Value()
        | Util.(SQLSubstring | addDays | addMonths | addYears)(<expr>, <expr>)
        | Util.extractYear(<expr>)
        | <expr>.get<field name>()
        | <agg-subquery-instruction>

<list> ::=
        <expr>.get<joinLink>()
        | <var>.all<table name>()
        | new QueryList()
        | <list>.with(<expr>)
        | <group-subquery-instruction>.asList()
        | <add-subquery-instruction>
```

Fig. 11. JQS grammar

elements to a collection, one for adding elements to a map, one for aggregating values, and another for nested loops resulting in a join. Most SQL operations can be expressed using the functionality described by this grammar.

Some SQL functionality that is not currently supported by JQS include set operations, intervals, and internationalisation because the queries we were working with did not require this functionality. We also chose not to support NULL and related operators in this iteration of JQS. Because Java does not support three-value logic or operator overloading, we would have to add special objects and methods to emulate the behaviour of NULL, resulting in a verbose and complicated design. Operations related to NULL values such as OUTER JOINs are not supported as well.

JQS also currently offers only basic support for update operations since it focuses only on the query aspects of SQL. SQL's more advanced data manipulation operations are rarely used and not too powerful, so it would be fairly straight-forward to extend JQS to support these operations. Most of these operations are simply composed of a normal query followed by some sort of INSERT, DELETE, or UPDATE involving the result set of the query.

In the end, our JReq system comprises approximately 20 thousand lines of Java and XSLT code. Although JReq translations can be applied to an entire codebase, we use annotations to direct JReq into applying its transformations only to specific methods known to contain queries. Additionally, we had some planned features that we never implemented because we did not encounter any situations during our research that required them: we did not implement handling of non-local variables, we did not implement type-checking or unification to check for errors in queries, and we did not implement pointer aliasing support.

5 Evaluation

5.1 TPC-W

To evaluate the behaviour of JReq, we tested the ability for our system to handle the database queries used in the TPC-W benchmark [16]. TPC-W emulates the behaviour of database-driven websites by recreating a website for an online bookstore.

We started with the Rice implementation of TPC-W [2], which uses JDBC to access its database. For each query in the TPC-W benchmark, we wrote an equivalent query using JQS and manually verified that the resulting queries were semantically equivalent to the originals. We could then compare the performance of each query when using the original JDBC and when using the JReq system. Our JReq prototype does not provide support for database updates, so we did not test any queries involving updates. Since this experiment is intended to examine the queries generated by JReq as compared to hand-written SQL, we also disabled some of the extra features of JReq such as transaction and persistence lifecycle management.

We created a 600 MB database in PostgreSQL 8.3.0 [13] by populating the database with the number of items set to 10000. We did not run the complete TPC-W benchmark, which tests the complete system performance of web servers, application servers, and database servers. Instead, we focused on measuring the performance of individual queries instead. For each query, we first executed the query 200 times with random valid parameters to warm the database cache, then we measured the time needed to execute the query 3000 times with random valid parameters, and finally we garbage collected the system. Because of the poor performance of the getBestSellers query, we only executed it for 50 times to warm the cache and measured the performance of executing the query only 250 times. We first took the JQS version of the queries, measured the performance of each query consecutively, and repeated the benchmark 50 times. We took the average of only the last 10 runs to avoid the overhead of Java dynamic compilation. We then repeated this experiment using the original JDBC implementation instead of JQS. 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. The benchmark harness was run using Sun's 1.5.0 Update 12 JVM. JReq required approximately 7 seconds to translate our 12 JQS queries into SQL.

The performance of each of the queries is shown in Table 1. In all cases, JReq is faster than hand-written SQL. These results are a little curious because one usually expects hand-written code to be faster than machine-generated code. If we look at the one query in Fig. 12 that shows the code of the original hand-written JDBC code and compares it to the comparable JQS query and the JDBC generated from that query, we can see that the original JDBC code is essentially the same as the JDBC generated by JReq. In particular, the SQL queries are structurally the same though the JReq-generated version is more verbose. What

Table 1. The average execution time, standard deviation, and difference from hand-written JDBC/SQL (all in milliseconds) of the TPC-W benchmark are shown in this table with the column JReq NoOpt referring to JReq with runtime optimisations disabled. One can see that JReq offers better performance than the hand-written SQL queries.

	JDBC		JReq NoOpt			JReq		
Query	Time	σ	Time	σ	Δ	Time	σ	Δ
getName	3592	112	3633	24	1%	2241	15	(38%)
getCustomer	8424	79	8944	57	6%	3939	24	(53%)
doGetMostRecentOrder	29108	731	88831	644	205%	8009	57	(72%)
getBook	6392	30	7347	55	15%	3491	27	(45%)
doAuthorSearch	10216	24	10414	559	2%	7306	46	(28%)
doSubjectSearch	16999	128	16898	86	(1%)	13667	120	(20%)
getIDandPassword	3706	33	3820	41	3%	2375	25	(36%)
doGetBestSellers	4472	50	4455	51	(0%)	3936	39	(12%)
doTitleSearch	27302	203	26979	418	(1%)	23985	61	(12%)
doGetNewProducts	23111	68	24447	128	6%	21086	70	(9%)
doGetRelated	6162	52	7731	92	25%	2690	34	(56%)
getUserName	3506	57	3569	13	2%	2214	11	(37%)

Original hand-written JDBC query

```
PreparedStatement getUserName = con.prepareStatement(
   "SELECT c_uname FROM customer WHERE c_id = ?");
getUserName.setInt(1, C_ID);
ResultSet rs=getUserName.executeQuery();
if (!rs.next()) throw new Exception();
u_name = rs.getString("c_uname");
rs.close(); stmt.close();
```

Comparable JQS query

```
EntityManager em = db.begin();
DBSet<String> matches = new QueryList<String>();
for (DBCustomer c: em.allDBCustomer())
   if (c.getCustomerId()==C_ID) matches.add(c.getUserName());
u_name = matches.get();
db.end(em, true);
```

JDBC generated by JReq

```
PreparedStatement stmt = null; ResultSet rs = null;
try { stmt = stmtCache.poll();
      if (stmt == null) stmt = em.db.con.prepareStatement(
          "SELECT (A.C_UNAME) AS COL0 "
        + "FROM Customer AS A WHERE (((A.C_ID)=?))");
      stmt.setInt(1, param0);
      rs = stmt.executeQuery();
      QueryList toReturn = new QueryList();
      while(rs.next()) { Object value = rs.getString(1);
                         toReturn.bulkAdd(value); }
      return toReturn;
} catch (SQLException e) { ... } finally {
   if (rs != null) try { rs.close(); } catch...
   stmtCache.add(stmt); }
```

Fig. 12. Comparison of JDBC vs. JReq on the getUserName query

makes the JReq version faster though is that JReq is able to take advantage of small runtime optimisations that are cumbersome to implement when writing JDBC by hand. For example, all JDBC drivers allow programmers to parse SQL queries into an intermediate form. Whenever the same SQL query is executed but with different parameters, programmers can supply the intermediate form of the query to the SQL driver instead of the original SQL query text, thereby allowing the SQL driver to skip repeatedly reparsing and reanalysing the same SQL query text. Taking advantage of this optimisation in hand-written JDBC code is cumbersome because the program must be structured in a certain way and a certain amount of bookkeeping is involved, but this is all automated by JReq.

Table 1 also shows the performance of code generated by JReq if these runtime optimisations are disabled (denoted as JReq NoOpt). Of the 12 queries, the performance of JReq and hand-written JDBC is identical for six of them. For the six queries where JReq is slower, four are caused by poorly formulated queries that fetched more data than the original queries (for example, they fetch entire entities whereas the original queries only fetched most of the fields of the entity). Two other queries are slower because JReq generates queries that are more verbose than the original queries thereby requiring more work from the SQL parser.

Overall though, all the queries from the TPC-W benchmark, a benchmark that emulates the behaviour of real application, can be expressed in JQS, and JReq can successfully translate these JQS queries into SQL. JReq generates SQL queries that are structurally similar to the original hand-written queries for all of the queries. Although the machine-generation of SQL queries may result in queries that are more verbose and less efficient than hand-written SQL queries, by taking advantage of various optimisations that a normal programmer may find cumbersome to implement, JReq can potentially exceed the performance of hand-written SQL.

5.2 TPC-H

Although TPC-W does capture the style of queries used in database-driven websites, these types of queries make little use of more advanced query functionality such as nested queries. To evaluate JReq's ability to handle more difficult queries, we have run some benchmarks involving TPC-H [17]. The TPC-H benchmark tests a database's ability to handle decision support workloads. This workload is characterised by fairly long and difficult ad hoc queries that access large amounts of data. The purpose of this experiment is to verify that the expressiveness of the JQS query syntax and JReq's algorithms for generating SQL queries are sufficient to handle long and complex database queries.

We extracted the 22 SQL queries and parameter generator from the TPC-H benchmark and modified them to run under JDBC in Java. We chose to use MySQL 5.0.51 for the database instead of PostgreSQL in this experiment in order to demonstrate JReq's ability to work with different backends. For this benchmark, we used a 2.5 GHz Pentium IV Celeron machine with 1 GB of RAM running Fedora Linux 9, and Sun JDK 1.5.0 Update 16.

We then rewrote the queries using JQS syntax. All 22 of the queries could be expressed using JQS syntax except for query 13, which used a LEFT OUTER JOIN, which we chose not to support in this version of JQS, as we described in Sect. 4.4. To verify that the JQS queries were indeed semantically equivalent to the original queries, we manually compared the query results between JDBC and JReq when run on a small TPC-H database using a scale factor of 0.01, and the results matched. This shows the expressiveness of the JQS syntax in that 21 of the 22 queries from TPC-H can be expressed in the JQS syntax and be correctly translated into working SQL code. JReq required approximately 33 seconds to translate our 21 JQS queries into SQL.

We then generated a TPC-H database using a scale factor of 1, resulting in a database about 1GB in size. We executed each of the 21 JQS queries from

Table 2. TPC-H benchmark results showing average time, standard deviation, and time difference (all results in seconds)

	JDBC		JReq				JDBC		JReq		
Query	Time	σ	Time	σ	Δ	Query	Time	σ	Time	σ	Δ
q1	73.5	0.4	71.9	3.4	(2%)	q12	23.4	0.5	29.7	0.2	27%
q2	145.4	2.2	146.0	1.9	0%	q14	491.7	8.9	500.8	10.1	2%
q3	37.9	0.6	38.6	0.9	2%	q15	24.9	0.7	24.8	0.6	(0%)
q4	23.0	0.5	23.8	0.2	3%	q16	21.3	0.6	> 1 hr	0.2	-
q5	209.1	4.2	206.1	3.2	(1%)	q17	2.1	0.2	11.0	3.6	429%
q6	15.2	0.3	15.8	0.3	4%	q18	> 1 hr	0.0	349.3	4.0	-
q7	79.1	0.5	83.1	1.6	5%	q19	2.8	0.1	18.1	0.4	540%
q8	48.8	1.7	51.0	1.9	4%	q20	69.4	4.3	508.4	11.4	633%
q9	682.0	97.4	690.2	97.9	1%	q21	245.5	3.2	517.0	7.1	111%
q10	47.1	1.0	47.2	0.5	0%	q22	1.1	0.0	1.6	0.0	43%
q11	41.7	0.6	41.9	0.7	1%						

TPC-H in turn using random query parameters, with a garbage collection cycle run in-between each query. We then executed the corresponding JDBC queries using the same parameters. This was repeated six times, with the last five runs kept for the final results. Queries that ran longer than one hour were cancelled. Table 2 summarises the results of the benchmarks.

Unlike TPC-W, the queries in TPC-H take several seconds each to execute, so runtime optimisations do not significantly affect the results. Since almost all the execution time occurs at the database and since the SQL generated from the JQS queries are semantically equivalent to the original SQL queries, differences in execution time are mostly caused by the inability of the database's query optimiser to find optimal execution plans. In order to execute the complex queries in TPC-H efficiently, query optimisers must be able to recognise certain patterns in a query and restructure them into more optimal forms. The particular SQL generated by JReq uses a SQL subset that may match different optimisation patterns in database query optimisers than hand-written SQL code. For example, the original SQL for query 16 evaluates a COUNT(DISTINCT) operation inside of GROUP BY. This is written in JQS using an equivalent triply nested query, but MySQL is not able to optimise the query correctly, and running the triply nested query directly results in extremely poor performance. On the other hand, in query 18, JReq's use of deeply nested queries instead of a more specific SQL operation (in this case, GROUP BY...HAVING) fits a pattern that MySQL is able to execute efficiently, unlike the original hand-written SQL. Because of the sensitivity of MySQL's query optimiser to the structure of SQL queries, it will be important in the future for JReq to provide more flexibility to programmers in adjusting the final SQL generated by JReq.

Overall, 21 of the 22 queries from TPC-H could be successfully expressed using the JQS syntax and translated into SQL. Only one query, which used a LEFT OUTER JOIN, could not be handled because JQS and JReq do not currently support the operation yet. For most of the queries, the JQS queries executed with similar performance to the original queries. Where there are differences in execution time, most of these differences can be eliminated by either improving the MySQL query optimiser, adding special rules to the SQL generator to generate patterns that are better handled by MySQL, or extending the syntax of JQS to allow programmers to more directly specify those specific SQL keywords that are better handled by MySQL.

6 Conclusions

The JReq system translates database queries written in the imperative language Java into SQL. Unlike other systems, the algorithms underlying JReq are able to analyse code written in imperative programming languages and recognise complex query constructs like aggregation and nesting. In developing JReq, we have created a syntax for database queries that can be written entirely with normal Java code, we have designed an algorithm based on symbolic execution to automatically translate these queries into SQL, and we have implemented a research prototype of our system that shows competitive performance to hand-written SQL.

We envision JReq as a useful complement to other techniques for translating imperative code into SQL. For common queries, existing techniques often provide greater syntax flexibility than JReq, but for the most complex queries, programmers can use JReq instead of having to resort to domain-specific languages like SQL. As a result, all queries will end up being written in Java, which can be understood by all the programmers working on the codebase.

References

1. American National Standards Institute: American National Standard for Information Systems—Database Language—SQL: ANSI INCITS 135-1992 (R1998). American National Standards Institute (1992)
2. 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. Tech. Rep. TR02-389, Rice University (February 2002)
3. Bradley, A.R., Manna, Z.: The Calculus of Computation: Decision Procedures with Applications to Verification. Springer, New York (2007)
4. Cook, W.R., Rai, S.: Safe query objects: statically typed objects as remotely executable queries. In: ICSE 2005: Proceedings of the 27th international conference on Software engineering, pp. 97–106. ACM, New York (2005)
5. DeMichiel, L., Keith, M.: JSR 220: Enterprise JavaBeans 3.0, `http://www.jcp.org/en/jsr/detail?id=220`
6. Flanagan, C., Saxe, J.B.: Avoiding exponential explosion: generating compact verification conditions. In: POPL 2001, pp. 193–205. ACM, New York (2001)
7. Guravannavar, R., Sudarshan, S.: Rewriting procedures for batched bindings. Proc. VLDB Endow. 1(1), 1107–1123 (2008)
8. Iu, M.Y., Zwaenepoel, W.: Queryll: Java database queries through bytecode rewriting. In: van Steen, M., Henning, M. (eds.) Middleware 2006. LNCS, vol. 4290, pp. 201–218. Springer, Heidelberg (2006)

9. Katz, R.H., Wong, E.: Decompiling CODASYL DML into relational queries. ACM Trans. Database Syst. 7(1), 1–23 (1982)
10. Lieuwen, D.F., DeWitt, D.J.: Optimizing loops in database programming languages. In: DBPL3: Proceedings of the third international workshop on Database programming languages: bulk types & persistent data, pp. 287–305. Morgan Kaufmann, San Francisco (1992)
11. Maier, D., Stein, J., Otis, A., Purdy, A.: Development of an object-oriented DBMS. In: OOPLSA 1986, pp. 472–482. ACM Press, New York (1986)
12. Necula, G.C.: Translation validation for an optimizing compiler. In: PLDI 2000, pp. 83–94. ACM, New York (2000)
13. PostgreSQL Global Development Group: PostgreSQL, `http://www.postgresql.org/`
14. Rinard, M.C.: Credible compilation. Tech. Rep. MIT/LCS/TR-776, Cambridge, MA, USA (1999)
15. Torgersen, M.: Language INtegrated Query: unified querying across data sources and programming languages. In: OOPSLA 2006, pp. 736–737. ACM Press, New York (2006)
16. Transaction Processing Performance Council: TPC Benchmark W (Web Commerce) Specification Version 1.8. Transaction Processing Performance Council (2002)
17. Transaction Processing Performance Council: TPC Benchmark H (Decision Support) Standard Specification Version 2.8.0. Transaction Processing Performance Council (2008)
18. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot - a Java bytecode optimization framework. In: CASCON 1999: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research, p. 13. IBM Press (1999)
19. Wiedermann, B., Cook, W.R.: Extracting queries by static analysis of transparent persistence. In: POPL 2007, pp. 199–210. ACM Press, New York (2007)
20. Wiedermann, B., Ibrahim, A., Cook, W.R.: Interprocedural query extraction for transparent persistence. In: OOPSLA 2008, pp. 19–36. ACM, New York (2008)
21. Wong, L.: Kleisli, a functional query system. J. Funct. Program. 10(1), 19–56 (2000)