Safe Query Objects: Statically Typed Objects as Remotely Executable Queries

William R. Cook
Department of Computer Sciences
University of Texas at Austin
wcook@cs.utexas.edu

Siddhartha Rai
Department of Computer Sciences
University of Texas at Austin

ABSTRACT

Developers of data-intensive applications are increasingly using persistence frameworks such as EJB, Hibernate and JDO to access relational data. These frameworks support both transparent persistence for individual objects and explicit queries to efficiently search large collections of objects. While transparent persistence is statically typed, explicit queries do not support static checking of types or syntax because queries are manipulated as strings and interpreted at runtime. This paper presents *Safe Query Objects*, a technique for representing queries as statically typed objects while still supporting remote execution by a database server. Safe query objects use object-relational mapping and reflective metaprogramming to translate query classes into traditional database queries. The model supports complex queries with joins, parameters, existentials, and dynamic criteria. A prototype implementation for JDO provides a type-safe interface to the full query functionality in the JDO 1.0 standard.

Categories and Subject Descriptors

D.2.12 [**Software Engineering**]: Interoperability; D.3.0 [**Programming Languages**]: General; H.2.3 [**Database Management**]: Languages

General Terms

Languages, Design, Reliability

1. INTRODUCTION

Developers of data-intensive applications are increasingly using persistence frameworks such as EJB [17], Hibernate [6] and JDO [21] to access relational data. These frameworks support transparent persistence for individual objects and explicit queries to efficiently search large collections of objects. Transparent persistence is statically typed, but query execution does not support static checking of types or syntax. The query interfaces are similar to older database interfaces, ODBC [24] and JDBC [12], but use object-based query languages instead of SQL.

The fundamental problem with existing query interfaces is that they force significant parts of program behavior to be encoded inside strings or other runtime data structures. The queries are disconnected from the rest of the program, and the linkage between the program and queries, by passing parameters and decoding results, is awkward and error-prone. For *dynamic queries*, which include a set of criteria that is determined at runtime, the query strings themselves must be constructed dynamically. In addition, programmers must learn two languages that provide overlapping functionality yet subtly different semantics.

These problems are illustrated by the following program fragment [11], which motivated the use of static analysis to perform syntax and type checking on programs that use JDBC:

```
ResultSet getPerishablePrices(String lowerBound) {
  String query = "SELECT '$' || (RETAIL/100) "
               + "FROM INVENTORY WHERE ";
  if (lowerBound != null) {
    query += "WHOLESALE > " + lowerBound + " AND ";
  }
  query += "TYPE IN (" + getTypeCodes() + ");";
  return statement.executeQuery(query);
}
String getTypeCodes() {
  return "SELECT TYPECODE, TYPEDESC FROM TYPES "
       + "WHERE NAME = 'fish' OR NAME = 'meat'";
}
```

To resolve these problems, we present *safe query objects*, an alternative technique for specifying type-safe queries. Safe queries combine object-relational mapping with object-oriented languages to specify queries like the one above in a more understandable and maintainable form:

```
class PerishablePrices
{
  Integer lowerBound;
  boolean filter(Inventory item) {
    return (lowerBound == null
         || item.Wholesale > lowerBound.intValue())
        && (item.Type.Name.equals("fish")
         || item.Type.Name.equals("meat"));
  }
}
```


ICSE'05, May 15–21, 2005, St. Louis, Missouri, USA.

```
interface javax.jdo.PersistenceManager {
    Object getObjectById(Object id);
    javax.jdo.Query newQuery(Class class);
    // methods for transactions not listed
}

interface javax.jdo.Query {
    void setFilter(String filter);
    void setOrdering(String ordering);

    void declareImports(String imports);
    void declareParameters(String params);
    void declareVariables(String vars);

    Object execute();
    Object execute(Object arg1);
    Object executeWithMap(Map parameters);
    // bookkeeping methods not listed
}
```

Figure 1: Selections from the JDO API

A key aspect of safe queries is a database-friendly evaluation strategy: the `PerishablePrices` class can be translated into code that generates and executes query strings using standard database interfaces such as JDBC [12] or JDO [21]. The generated code is similar to the first version given above, but is safe because it is automatically generated from a type-checked class. The translation requires reflective access to the code of the class, but it is performed at compile time or load time, so there is no runtime overhead. A few differences between the safe query and the original version are discussed in Section 5.

Safe query objects support filtering, sorting, relationships (joins), parameters, existential quantification, and dynamic queries. A prototype of safe query objects for JDO provides a statically typed interface to the dynamically typed functionality in the JDO 1.0 specification [21]. The prototype has been successfully applied to existing programs that use JDO and a range of queries from a large enterprise application.

The prototype highlights some limitations of JDO 1.0 and Java. For example, JDO allows filters to access related objects (via joins in where clauses) but does not support the return of multiple related objects (via joins in select clauses). In addition, Java does not support the semantics used for computations involving NULL values in the database. Although the current JDO-based prototype inherits these limitations, the concept of safe query objects can be applied to other languages and support more powerful query languages.

2. BACKGROUND

JDO provides access to persistent objects through an instance of the `PersistenceManager` interface shown in Figure 1. Individual objects can be loaded explicitly using `getObjectById`, or implicitly when traversing references to related objects. Query execution is provided by the `Query` interface. The `newQuery` method creates a query to return objects of the given *candidate class*. Candidate classes contain fields and relationships that are mapped to corresponding tables in a relational database. We will use a simple

```
class Employee
{
  String     name;
  float      salary;
  Department department;
  Employee   manager;
}

class Department
{
  String               name;
  Collection<Employee> employees;
}
```

Figure 2: Example classes

database of employees and departments, whose schema is represented by the Java 1.5 classes in Figure 2. Given these definitions, it is easy to write a JDO query to find all employees whose salary is greater than their manager's salary.

```
Collection<Employee> execute(
        javax.jdo.PersistenceManager pm)
{
  javax.jdo.Query payCheck =
      pm.newQuery(Employee.class);
  payCheck.setFilter("salary > maneger.salary");
  Object result = payCheck.execute();
  return (Collection<Employee>) result;
}
```

Filter conditions are written in the JDOQL language, whose grammar is based on Java expressions. However, the semantics of JDOQL and Java are different: JDOQL allows more automatic conversions between types; overloads comparisons like `<`, `==`, and `>` for boxed values including dates, integers, and strings; and interprets the `contains` method in a non-standard way to support existential quantification. JDO uses joins to implement navigation through object-valued fields, as in `manager.salary`, but it ignores the Java access control restrictions on class members. In this paper all members are assumed to be public.

This program is syntactically correct Java code, but there is a problem in the embedded code: `manager` is misspelled as `maneger`. The problem will not be discovered until runtime.

Although the static return type of the `execute` method is `Object`, at runtime the return value is a read-only `Collection` containing instances of the candidate class. Although the current JDO specification does not make use of generic types in Java 1.5, the backward compatibility of generics allows JDO to work with generic classes. Generic types are used throughout this paper to increase precision of static typing for queries.

The example above illustrates problems with filtering. The other methods in the `Query` interface have similar problems. What they have in common is that they are all forms of metaprogramming: their arguments are textual representations of programs, which are manipulated and executed by a JDO implementation. The following sections present a statically typed solution for each `Query` method, leading to a completely type-safe interface to the full query functionality of JDO.

3. SAFE QUERY OBJECTS

In its simplest form, a safe query object is just an object containing a boolean method that can be used to filter a collection of candidate objects.

```
class PayCheck extends SafeQuery<Employee>
               instantiates RemoteQueryJDO
{
  boolean filter(Employee emp) {
    return emp.salary > emp.manager.salary;
  }
}
```

A safe query is a subclass of `SafeQuery<T>`, a default implementation of the `ISafeQuery<T>` interface in Figure 3. The generic type parameter `Employee` is the candidate class of the query. Because this filter is normal Java code, the compiler checks syntax and types and it will produce an error when it finds the misspelled `manager` reference in the filter method.

The key to safe query objects is that type-checked class definitions are translated into code to call standard database interfaces like JDO. This new code is added to the query class to override the `execute` method in the `SafeQuery` base class. The translation could be performed on the classes during compilation, on byte-codes after compilation, or during loading. Our prototype uses OpenJava [22], which follows the first approach.

The query translation is encapsulated in the `RemoteQueryJDO` *metaclass*, which is applied to the safe query using the OpenJava `instantiates` keyword. OpenJava runs the metaclass at compile time, supplying the definition of the `PayCheck` class as input. When applied to `PayCheck`, the `RemoteQueryJDO` metaclass examines the user-defined `filter` method and generates the corresponding `execute` method given in the previous section. Although the JDO code is unsafe and difficult to maintain by hand, it is safe and easy to use when automatically generated from a safe query object.

The execute method provides a statically typed interface to the query.

```
javax.jdo.PersistenceManager pm;
PayCheck query = new PayCheck();
Collection<Employee> r = query.execute(pm);
```

The current version of OpenJava does not support generics, so generic type erasing [5] has been done manually in the prototype implementation. The details of the translation are given in Section 4.

In the prototype, the actual SQL query sent to the database depends on the JDO implementation. However, the following query is an example of what might be generated. The `Employee` table is joined to itself through the `ManagerID` attribute. The automatically generated labels `E1` and `E2` represent the candidate employee and manager, respectively.

```
SELECT E1.*
FROM Employee E1 INNER JOIN Employee E2
        ON E1.ManagerID = E2.ID
WHERE E1.salary > E2.salary
```

To enable remote execution in a database server, the filter method in a safe query class must be free of *side-effects*: it must not modify any state or call any methods that modify state. An additional restriction is that the filter method must not contain any iterative constructs, as certain classes of iterative statements may not have a closed form boolean representation needed by database queries.

```
interface ISafeQuery<T>
{
  boolean       filter(T item);
  Sort          order(T item);

  Collection<T>
    execute(javax.jdo.PersistenceManager pm);

  <R> boolean exists(Collection<R> items,
                     ISafeQuery<R> query);
}
```

Figure 3: SafeQuery interface

Queries may also be executed *locally* within the Java VM, by iterating through a collection to select the items for which the filter returns true. Potential differences between Java execution semantics and remote SQL execution are discussed in Section 3.5.

3.1 Sorting Results

Queries often specify *sort order* for the set of query results. Relational query languages define sort order with a list of sort expressions that are annotated to indicate whether to sort in ascending or descending order. A similar approach is used in JDO, as shown in Figure 4.

To specify sorting, a safe query must associate a list of sortable values with each object in the result set. This is done by adding an `order` method that takes a candidate element and returns a linked list of sortable values, as shown in Figure 5.

Each sort object contains a value, a flag indicating ascending or descending order, and an optional secondary `Sort` value.

```
class Sort implements Comparable<Sort>
{
  enum Direction ASCENDING, DESCENDING;
  Sort(Comparable v, Direction dir)...
  Sort(Comparable v, Direction dir, Sort next)...
  // Rest of class definition...
}
```

It is possible to write an `order` method that is not well-behaved. For example, the method might return lists with different lengths or containing different types of Comparable values. The OpenJava metaclass rejects such methods and signal a compile-time error.

3.2 Parameterized Queries

Parameterized queries are needed when a query's behavior depends upon one or more input values. For example, the parameterized query in Figure 6 finds all employees whose salary is greater than a limit. The parameter is used in the string passed to `setFilter`. It is declared by passing a string containing a fragment of Java syntax to `declareParameters`. It is bound to a value when the query is executed; the `execute` method supports variable number of arguments of type `Object`. Although JDO parameters resemble Java parameters, they are awkward to specify and

```
Collection sortEmployees()
{
  javax.jdo.Query q = pm.newQuery(Employee.class);
  q.setOrdering("department.name ascending,"
              + "salary descending");
  return (Collection) q.execute();
}
```

Figure 4: Sorting with JDO

```
class SortQuery instantiates RemoteQueryJDO
                extends SafeQuery<Employee>
{
  Sort order(Employee emp) {
    return new Sort(emp.department.name,
                    Sort.Direction.ASCENDING,
           new Sort(emp.salary,
                    Sort.Direction.DESCENDING));
  }
}
```

Figure 5: Safe query with sorting

lack the basic static checks provided by Java. For example, if the call to `declareParameters` is omitted or the wrong type of value is passed to `execute`, the problem will be undetected until runtime.

For a safe query, shown in Figure 7, query parameters are defined as standard Java function parameters. They are declared as arguments to the query constructor and stored in member variables of the query object. Thus they are accessible in both the `filter` and `order` methods.

The resulting code is idiomatic object-oriented programming and could be maintained by any programmer. Note that the parameters were not added to the `filter` method itself because that would require the filter method to have a different type in each parameterized query.

```
SafeQuery<Employee> q = new SalaryLimit(50000.00);
Collection<Employee> result = q.execute(pm);
```

After typechecking, the safe query class is processed to generate the declaration strings and parameter arguments needed by JDO to execute the query remotely in the database server. The resulting code is the same as the unsafe JDO code at the start of this section.

3.3 Dynamic Queries

Dynamic queries involve filters, parameters, or sort orders that are constructed at runtime. They are used when different filter criteria must be combined to form a complete filter. For example, if a search form in a user interface allows a set of optional search criteria to be specified, the filters that result from different combinations of criteria will be different. A similar situation arises when optional criteria must be conditionally included in a query. Dynamic queries also arise in implementing fine-grained authorization rules that apply to individually to each user [20].

Since the different filters have structurally different criteria, rather than simply different parameter values, parameterized queries alone are not sufficient. Dynamic queries are sometimes used instead of parameterized queries, because it is easier to concatenate a parameter text into a query string than to declare and invoke a complete parameterized query. However, this practice is notoriously unsafe and can interfere with reuse of query plans.

```
Collection salaryLimitEmployees(double limit)
{
  javax.jdo.Query q = pm.newQuery(Employee.class);
  q.setFilter("salary > limit");
  q.declareParameters("Double limit");
  Collection r = (Collection)
      q.execute(new Double(limit));
}
```

Figure 6: Passing parameters in JDO

```
class SalaryLimit instantiates RemoteQueryJDO
                  extends SafeQuery<Employee>
{
  double limit;  /* parameter */
  SalaryLimit(double limit) {
    this.limit = limit;
  }

  boolean filter(Employee employee) {
    return employee.salary > limit;
  }
}
```

Figure 7: Parameterized safe query

Dynamic filters are commonly created by concatenating portions of a filter string together to create the complete filter. If optional components have parameters, the parameter list of the overall query will also be dynamic. Figure 8 illustrates the creation of dynamic filters and parameters using JDO. In this example, a user can search for employees by name, salary range, or both. Different filters are constructed to respond to user-selected criteria.

In a safe query, a dynamic filter is simply a normal filter in which some sub-expressions depend only on query parameters, not the database. Figure 9 illustrates this technique for the dynamic query given in Figure 8. Short-circuit evaluation of disjunction allows `new DynQuery("F", null)` to find employees whose name begins with "F", but with no limit on salary. The translation of dynamic queries takes advantage of the inherent *staging* [9] of the `execute` method: the parts of the filter that do not depend on the database are evaluated as Java expressions in the `execute` method, while the rest of the filter is translated to a query string for execution in the database. The generated code is similar to the hand-written version in Figure 8. Details of the translation, including the mechanism for staging operations, are given in Section 4.

```
Collection search(String namePrefix,
                  Double minSalary)
{
  String filter = null;
  String paramDecl = "";
  HashMap paramMap = new HashMap();

  if (namePrefix != null) {
    q.declareParameters("String namePrefix");
    paramMap.put("namePrefix", namePrefix);
    filter = and(filter,
                 "(name.startsWith(namePrefix))");
  }

  if (minSalary != null) {
    q.declareParameters("double minSalary");
    paramMap.put("minSalary", minSalary);
    filter = and(filter, "(salary >= minSalary)");
  }

  javax.jdo.Query q = makeQuery(Employee.class);
  q.setFilter(filter);
  return q.executeWithMap(paramMap);
}

String and(String a, String b) {
  return (a == null) ? b : (a + " && " + b);
}
```

Figure 8: Dynamic query using JDO

```
class DynQuery instantiates RemoteQueryJDO
               extends SafeQuery<Employee>
{
  private String namePrefix; // may be null
  private Double minSalary;  // may be null

  DynQuery(String namePrefix, Double minSalary) {
    this.namePrefix = namePrefix;
    this.minSalary = minSalary;
  }

  boolean filter(Employee item) {
    return (namePrefix == null
              || item.name.startsWith(namePrefix))
        && (minSalary == null
              || item.salary >= minSalary);
  }
}
```

Figure 9: Safe query with dynamic filter

3.4 Existential Quantification

Most query languages support existential quantification to test whether any member of a set meets a condition. The query in Figure 10 finds departments whose names begin with a given prefix and have an employee whose salary is above a given minimum. The query uses the `exists` method defined in the `ISafeQuery` interface (Figure 3). The first parameter defines the extent of quantification, while the second parameter is another safe query. The query reuses the `SalaryLimit` query defined in Section 3.2. The approach is similar to the use of Boolean-valued functions for membership tests in languages with first-class functions: the `detect:ifNone:` method in Smalltalk [7], or the `any` function in Haskell [14].

In JDOQL, existentials are expressed by a nonstandard interpretation of the `contains` method as a binding construct. `C.contains(v)` normally tests if a collection `C` contains the value of the variable `v`, but in JDO the form `C.contains(v) && P` is interpreted to mean $\exists v.P$ where P is a predicate that may mention v. The entire form returns true if there exists a member of `C` for which P is true. The variable v must be listed in the string passed to `declareVariables`.

The query in Figure 11 is a hand-written equivalent to Figure 10. This query illustrates the kind of complexity that programmers face in writing even fairly simple queries. Although we define a parameterized query to find employees whose salary is above a minimum in Section 3.2, reusing this query here would require complex and unsafe string manipulation. As in previous examples, after type checking is complete, the safe query is translated to code that is similar to the hand-written form.

3.5 Null Values

The interpretation of null values is a significant complication in the integration of databases and programming languages. Most relational databases allow columns to be marked as *nullable*, so that null values can be used to indicate missing data. Taking null as an undefined value, databases define most operations involving null as returning null, although logical *and* (*or*) returns false (true) if at least one of its arguments is false (true). Object-oriented programming languages typically allow object references to be null, but primitive types like integer cannot be null. Our goal is to find a common ground in which database and programming language semantics are equivalent.

If nullable fields are avoided, as some recommend [8], then query and Java semantics are the same. To allow primitive data types to be nullable, there are two standard approaches: boxing and data abstraction.

When primitive values are boxed as objects, null references can represent null values. To support operations on boxed values, the safe query base class contains operators that enable uniform handling of null values. Arithmetic and comparison operators `equal`, `add`, etc. on boxed types return `null` if either of their arguments is `null`. Logical operators implement three-valued logic, where null represents the unknown value. Unfortunately there is no way to overload or rebind the built-in operators in Java. The expression

```
salary > min || name.startsWith("F")
```

must be written as

```
or(greater(salary, min), startsWith(name, "F")).
```

```
class DeptQuery instantiates RemoteQueryJDO
                extends SafeQuery<Department>
{
  double min;
  String deptPrefix;

  DeptQuery(double min, String deptPrefix) {
    this.min = min;
    this.deptPrefix = deptPrefix;
  }

  boolean filter(Department dept) {
    SafeQuery<Employee> sub = new SalaryLimit(min);
    return dept.name.startsWith(deptPrefix)
           && exists(dept.employees, sub);
  }
}
```

Figure 10: Safe query with existential subquery

```
Collection<Employee> execute(
        javax.jdo.PersistenceManager pm )
{
  javax.jdo.Query q = pm.newQuery(Employee.class);

  HashMap paramMap = new HashMap();
  paramMap.put("min", min);
  paramMap.put("namePrefix", namePrefix);
  q.declareParameters("double min");
  q.declareParameters("String namePrefix");

  q.declareVariables("Employee e");
  String filter = "name.startsWith(deptPrefix)"
                + "  && (employees.contains(e)"
                + "       && e.salary > min)");
  q.setFilter(filter);

  Object r = q.executeWithMap(paramMap);
  return (Collection<Employee>) r;
}
```

Figure 11: Existential quantification in JDO

This solution provides consistent semantics between remote and local execution on nullable types.

Navigation is also difficult. JDO specifies that "Navigation through a null-valued field, which would throw `NullPointerException`, is treated as if the filter expression returned false for the evaluation of the current set of variable values."[21] But given this query

```
or(emp.department == dept, emp.salary > min),
```

an employee with a null department would not be included in the result, even if their salary was over the minimum. This behavior is not consistent with SQL semantics, and it is not implemented by at least one JDO vendor [23]. It would be more consistent if a navigation through a null relationship returned null. Assuming this, all navigation through a nullable reference `E` must use the form `(E == null ? null : E.F)`. Unfortunately this pattern cannot be expressed as a reusable abstraction in Java. However, it is easily implemented using metaprogramming; this solution ensures consistency between local and remote execution of a filter, but it does subtly change the semantics of java code appearing in a filter.

Although it is not supported by Java, a better solution would be to define a data abstraction to represent nullable values or references. The appropriate semantics for operations could then be built into these abstractions, rather than implemented as separate functions. C# or C++ support value objects which can be used to define nullable abstractions.

4. IMPLEMENTATION DETAILS

In the prototype implementation of safe query objects, OpenJava is used as a framework for compile-time metaprogramming for analysis and generation of code. The translation system is implemented as a *metaclass* that is executed within the OpenJava system. It examines the query class and generates methods for remote execution. Figure 12 specifies a pattern that identifies the components of the query class used in the derivation. The scope of a repeated element a_i is identified by $\langle \ldots \rangle_i$. The "+" operator is used to denote string concatenation. Figure 13 defines the template for generating the `execute` method, based on the bindings of variables from the pattern.

The bulk of the translation involves converting the filter and sort methods into strings that can be passed to JDO. However, the treatment of subqueries is more complex. To enable separate compilation and reuse of query libraries, subquery strings are concatenated together, with appropriate variable renaming, to build a complete query. The function Φ converts a Java expression e into a Java expression that creates a string representation of e in JDOQL.

$\Phi : Variable \times Set(Variable) \times JavaExpr \rightarrow JavaExpr$
$\Phi(v,\ s,\ x) \rightarrow$ *if* $v{=}x$ *then* `""` *else if* $x \in s$ *then* x *else* `"`x`"`
$\Phi(v,\ s,\ \texttt{e}.f) \rightarrow \Phi(v,\ s,\ \texttt{e})$ `+ ".`f`"`
$\Phi(v,\ s,$ `exists(e, new` $Q(\langle$ `a`$_i$`, `$\rangle_i$`)))` $\rightarrow$
  $\Phi(v,\ s,\ \texttt{e})$ `+ "." +` Q`.exists(q,` $\langle \Phi(v,\ s,\ \texttt{a}_i) \rangle_i$`)`

The first argument of Φ is the parameter of the filter method, which identifies the object being filtered. This object is implicit in JDOQL, so Φ must strip off uses of this variable from the Java code. The second argument is a set of variables that must be renamed when reusing a query for

```
⟨ import import_m; ⟩_m
class QueryName instantiates RemoteQueryJDO
                  extends SafeQuery<Type>
{
  // all members are parameters:
⟨ ParamType_i param_i; ⟩_i

  boolean filter(Type elem)
  {
    return filter;
  }

  Sort order(Type orderVar)
  {
    return new Sort(order_j,
                    Sort.Direction.dir_j,
                    sort_{j+1} );
  }
}
```

Figure 12: Safe query pattern

```
// constructor for QueryName class
QueryName(⟨ ParamType_i param_i; ⟩_i)
{
⟨ this.param_i = param_i; ⟩_i
}

// remote execution method
Collection<Type> execute(
        javax.jdo.PersistenceManager pm)
{
  javax.jdo.Query q = pm.newQuery(Type.class);
  Map paramMap = new HashMap();
⟨ paramMap.put("param_i", param_i ); ⟩_i
⟨ q.declareParameters("ParamType_i param_i"); ⟩_i
  q.setFilter( Φ(elem, ∅, filter) );
  q.setOrdering(⟨Φ(elem, ∅, order_j) + " dir_j,"⟩_j);
  Object result = q.executeWithMap(paramMap);
  return (Collection<Type>) result;
}

// exists query and JDO declarations
static String exists(javax.jdo.Query q,
                     ⟨ String param_i, ⟩_i)
{
  q.declareVariables("Type uniqueName");
  q.declareImports("⟨ import import_m; ⟩_m");
  return "contains(uniqueName) && "
    + Φ(_, { "uniqueName", ⟨ param_i ⟩_i }, filter );
}
```

Figure 13: Template for generated methods

existential quantification. The third argument is the Java expression syntax being translated.

The first case handles variables. Uses of the filter parameter v are deleted; parameter variables in s are translated to java variables which at runtime contain JDOQL expressions representing the actual parameter for the variable; other variables are converted to literal JDOQL variables. The second case translates java field access to the corresponding JDOQL field access. The last case converts existentials to JDOQL syntax using the helper function `exists`, which returns a new `contains` expression. The exists clause allows a query to be used from within another query. To enable parameters to be passed from one query to anther, the calling query creates JDOQL expressions which are passed as parameters to the `exists` function. Query parameters are passed as strings to `exists`, which constructs a new expression in which the parameters have been replaced by the actual parameters from the calling query. Because these parameters are JDOQL expressions, not actual user-defined values, substitution is always safe.

Here is the translation of the `SalaryLimit` class:

```
static String exists(javax.jdo.query q,
                     String limit)
{
  q.declareVariables("Employee v32");
  return "contains(v32) && v32.salary > " + limit;
}
```

The variable `limit` is a query parameter which must be replaced before the exists clause can be embedded into a larger query. `limit` is bound to the text of an expression that defines the limit in the context of the calling query; it is not the actual numeric limit value.

The following expression, which uses `SalaryLimit`,

```
exists( d.employees, new SalaryLimit(min))
```

is translated by Φ to

```
"employees." + SalaryLimit.exists(q, "min")
```

The final query string sent to JDO is:

```
"employees.contains(v32) && v32.salary > min"
```

Dynamic queries are translated by separating the evaluation of filters into stages. *Local* sub-expressions, which only involve parameters, are identified and inlined into the Java code that constructs the remote query. *Remote* sub-expressions are converted to query strings. When a local and a remote expression are involved in a conjunction or disjunction, evaluation of the local expression creates two code paths which generate different query strings. For example, if e_1 is local and e_2 is remote, then e_1 `||` e_2 generates:

```
if (e1)
  filter = "true";
else
  filter = "e2";
```

The branches may have different parameter sets, and additional local sub-expressions in e_2 can create nested alternatives.

5. EVALUATION

We used safe query objects to create type-safe versions of JDO sample applications and problems posted in the literature. We also evaluated how well safe query objects handle the query patterns in a large open-source enterprise application. Most of the queries in the applications that we studied are parameterized. Many of these queries also use dynamic criteria and sort orders. We found that our model of safe query objects can handle common patterns of query construction. Our evaluations also brought to light some limitations of our model in handling dynamic data-driven query construction.

In many programs the query strings passed to JDO are static, literal strings. Almost all queries in the JDO sample programs are of this form. These programs are easily converted to use safe query objects. The resulting code is simpler, clearer, and statically type-safe. The performance is unchanged because the safe query version is translated at compile time into code equivalent to the original unsafe version.

Programs that create queries dynamically are a more interesting case. The `getPerishablePrices` query mentioned in the introduction is an example. Previous work [11] used static analysis to check the syntax and types of dynamic queries. One advantage of this approach is that it can check existing programs. Using safe queries requires rewriting code, although the resulting program is clearer and easier to maintain.

Safe query objects directly support a particular style of dynamic queries - queries where the inclusion of optional criteria is decided by locally evaluated conditionals:

```
if ( <java condition> )
  sql.append(" AND <SQL condition>");
```

Some dynamic query constructions cannot be directly expressed with safe query objects, but may be expressible after sufficient redesign.

The key question is whether safe queries can handle the kinds of dynamic queries used in real programs. The Compiere ERP application [13] provides a sophisticated range of queries for evaluation. With 314K commented lines of code in 1059 java source files, there are 871 places where SQL "select" queries are defined. Most involve joins, complex conditions, and parameters. There are over 200 cases where criteria are dynamically added to a query condition. Most of these fit the pattern of conditional criteria or sorting that is supported by safe query objects. The remainder are discussed below. In a few cases Compiere creates SQL "select" clauses dynamically. To translate these, a safe query must be written for each dynamic variation.

Compiere also uses *data-driven query construction*, where a generic algorithm builds a query based on a data structure. For the ten `Info` classes in the `compiere.apps.search` package, the data structure is static, so data-driven queries can be rewritten as safe queries.

The class `MQuery` builds queries based on field lists stored in the database itself. Although some of this dynamic machinery may be accidental (and dangerous) rather than essential, the core of this pattern is support for highly configurable user interfaces. Such explicit reflective programming, in which the system manipulates a dynamic representation of its own user interface, is difficult to capture in a static query. Not all applications provide this feature, although Compiere uses it extensively. More work is needed to determine if building a query that loads just the data needed in highly configurable user interfaces can be described in a way that supports static typing.

One potential issue with safe queries is a loss of code locality, because queries must be specified in their own class. In most cases we reviewed, query execution was already encapsulated in a method, so placing the query in an inner class provides a similar degree of locality.

We also note a few differences between the original JDBC version of the `getPerishablePrices` and the safe query version. In the JDBC version, the query formats the output by prepending "$" to the result of dividing the retail price by 100. Safe queries currently can only return objects, so the formatting of the result must be performed in the application server after the query results are returned.

Though we have gained significant insights by studying Compiere and other examples in literature, more work will be needed to extract and evaluate a complete set of query patterns from real-world applications. Our experience also identified some limitations in JDO. Although joins can be used for filters, the result of a JDO query is always a set of objects of a single type – no related objects can be returned at the same time. We are currently developing a version of safe query objects for JDBC to overcome these limitations.

6. RELATED WORK

The full integration of persistence and programming languages has been the focus of intense research over the last 20 years [4, 3]. This effort has demonstrated the importance of *orthogonal persistence*, *static type systems*, and *reflection*, and significantly influenced the design of modern database interfaces like EJB, Hibernate, and JDO. Research languages included specialized syntax for query operations, which could be compiled to SQL for remote execution against a relational database [18]. However, this line of research did not provide guidance on how to express type-safe queries in an existing programming language. As a result, existing languages have adopted the model of *call level interfaces* [24] in which queries are represented as strings or runtime data structures. These interfaces do not support traditional static type checking.

Query languages may also be embedded into programming languages [2]. These interfaces do not support dynamic queries, so they are not able to handle the full range of queries needed in complete applications. They also require programmers to learn two languages.

HaskellDB [16] is a domain-specific language for relational queries embedded in Haskell. The language includes operators to scan and join tables, test conditions and define query results. Special operators must be used in place of the normal functions for comparison of values and Boolean connectives. Expressions in the embedded query language are statically typed at compile time. The current version of HaskellDB does not support existential quantification or sorting. Parameterized queries are easily defined because HaskellDB is embedded cleanly within the general-purpose Haskell language. However, the parameters are embedded in dynamic query strings, unlike the JDBC parameter passing mechanisms. HaskellDB differs from the approach described here in that it uses Haskell as a meta language to define a strongly typed embedded language that generates SQL queries when executed, while we use standard Java classes

to represent queries and reflection to convert these classes into JDO queries at compile time.

The Xen language [19] adapts the approach of HaskellDB for an object-oriented language. Xen is still under development and its capabilities for accessing relational databases have not been published yet.

Linguistic reflection is another form of metaprogramming that can be applied to database interfaces. It has been used to generate code for relational joins [15], although the usability of this approach as a general programming model has not been demonstrated. Linguistic reflection can also be used to compile query strings into class definitions [1]. This approach is the inverse of the model proposed here: dynamic query strings are compiled at runtime into classes that resemble safe queries, while we convert statically typed query classes into dynamic query strings for remote execution. As a result, this approach provides neither compile-time type checking nor the ability to leverage external databases.

SchemeQL [25] is a SQL library for Scheme. It provides an interface for creating and composing queries as data structures, much like Hibernate-QL. In keeping with the dynamic typing of the Scheme language, SchemeQL validates queries at runtime and provides no static checks.

Gould, Su and Devanbu [11] and Deline and Fahndrich [10] use static analysis to type check programs with embedded SQL. They track the construction of strings containing SQL so that they can be checked against the SQL grammar and typing rules. Their analysis does not currently cover query parameters or result types. In addition, it can generate false positives when separate compilation is used. Type checking embedded SQL is a pragmatic solution, given that many existing program can benefit from more static analysis. However, the programming model based on string manipulation is still complex, and programmers must work with two different languages at the same time. Safe query objects provide an alternative that simplifies access to relational development, provides a single uniform programming model, and scales well to separate compilation.

7. CONCLUSIONS

Safe query objects are a new technique for expressing type-safe queries that support remote execution in a database server. Queries are defined using object-oriented classes and methods, which are translated into code that invokes standard database APIs. We evaluated the design by converting existing programs and examples in the literature. We found that the majority of queries were static. A significant number of queries were dynamic or used data-driven generation based on static tables. Finally, we identified cases where queries were dynamic *and* data driven, with the query definition itself being loaded from the database. Any program using the first three techniques can be converted to use safe query objects, while the last case is still an open problem.

Acknowledgments

Thanks to Kevin Loo for initial development, and Shriram Krishnamurthi, Dan Miranker, Jim Browne, Mark Grechanik, Dave Thomas, Ralph Johnson, Martin Gannholm, Nini Silk, Joe Yoder and the anonymous reviewers for their input.

8. REFERENCES

[1] S. Alagic and J. Solorzano. Java and OQL: A reflective solution for the impedance mismatch. *L'OBJET*, 6(2), 2000.

[2] ANSI/INCITS. Database languages - SQLJ - part 1: SQL routines using the Java programming language. Technical Report 331.1-1999, ANSI/INCITS, 1999.

[3] M. Atkinson and R. Welland. *Fully Integrated Data Environments: Persistence programming languages, object stores, and programming environments*. Springer, 2000.

[4] M. P. Atkinson and O. P. Buneman. Types and persistence in database programming languages. *ACM Comput. Surv.*, 19(2):105–170, 1987.

[5] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 183–200, 1998.

[6] D. Cengija. Hibernate your data. *onJava.com*, 2004.

[7] W. Cook. Interfaces and specifications for the Smalltalk collection classes. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, 1992.

[8] H. Darwen and C. J. Date. The third manifesto. *SIGMOD Record*, 24(1):39–49, 1995.

[9] R. Davies and F. Pfenning. A modal analysis of staged computation. *J. ACM*, 48(3):555–604, 2001.

[10] R. DeLine and M. Fahndrich. Typestates for objects, 2004.

[11] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proc. 26th International Conference on Software Engineering (ICSE)*. IEEE Press, 2004.

[12] G. Hamilton and R. Cattell. JDBC™: A Java SQL API. Sun Microsystems, 1997.

[13] J. Janke. Compiere project overview. *compiere.com*, 2004.

[14] S. P. Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

[15] G. Kirby, R. Morrison, and D. Stemple. Linguistic reflection in Java. *Software–Practice and Experience*, 28(10):1045–1077, 1998.

[16] D. Leijen and E. Meijer. Domain specific embedded compilers. In *Proceedings of the 2nd conference on Domain-specific languages*, pages 109–122. ACM Press, 1999.

[17] V. Matena and M. Hapner. *Enterprise Java Beans Specification 1.0*. Sun Microsystems, 1998.

[18] F. Matthes, A. Rudloff, J. Schmidt, and K. Subieta. A gateway from DBPL to Ingres. In W. Litwin and T. Risch, editors, *Applications of Databases, First International Conference, ADB-94*, volume 819, pages 365–380. Springer-Verlag, 1994.

[19] E. Meijer and W. Schulte. Programming with rectangles, triangles, and circles. In *Proceedings of Conference on XML*, 2003.

[20] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on*

Management of data, pages 551–562. ACM Press, 2004.

[21] C. Russell. *Java Data Objects (JDO) Specification JSR-12*. Sun Microsystems, 2003.

[22] M. Tatsubori, S. Chiba, K. Itano, and M.-O. Killijian. OpenJava: A class-based macro system for Java. In *OORaSE'99 Workshop on Object-Oriented Reflection and Software Engineering*, pages 117–133. ACM, 1999.

[23] D. Tinker. High performance JDO. *JDOGenie.com*, 2003.

[24] M. Venkatrao and M. Pizzo. SQL/CLI – a new binding style for SQL. *SIGMOD Record*, 24(4):72–77, 1995.

[25] N. Welsh, F. Solsona, and I. Glover. SchemeUnit and SchemeQL: Two little languages. In *Third Workshop on Scheme and Functional Programming*, 2002.