Native Queries for Persistent Objects
A Design White Paper

William R. Cook
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-0233, U.S.A.
wcook@cs.utexas.edu

Carl Rosenberger
db4objects Inc.
1900 South Norfolk Street
San Mateo, CA, 94403, U.S.A.
carl@db4o.com

August 23, 2005

Abstract

Most persistence architectures for Java and .NET provide interfaces to execute queries written in an architecture-specific query language. These interfaces are *string based*: queries are defined in strings that are passed to the persistence engine for interpretation. String-based query interfaces have significant negative impact on programmer productivity. The queries are not accessible to development environment features like compile-time type checking, auto-completion, and refactoring. Programmers must work in two languages: the implementation language and the query language. This paper introduces *Native Queries*, a concise and type-safe way to express queries directly as Java and C# methods. We describe the design of Native Queries and provide an overview of implementation and optimization issues. The paper also includes a discussion of the advantages and disadvantages of the current design of Native Queries.

1 Introduction

While today's object databases and object-relational mappers do a great job in making object persistence feel native to the developer, queries look foreign in an object-oriented program. They are expressed using either simple strings, or object graphs with strings interspersed. Let's take a look at a few examples.

For all examples in this paper we will assume the following class:

```
// Java
public class Student {
```

```
  private String name;
  private int age;
  public String getName(){
    return name;
  }
  public int getAge(){
    return age;
  }
}

// C#
public class Student {
  private string name;
  private int age;
  public string Name {
    get { return name; }
  }
  public int Age {
    get{ return age; }
  }
}
```

How would queries look for "all Student objects where the Student is younger than 20" using some of the existing object querying languages or APIs?

OQL [8, 1]

```
String oql =
  "select * from student in AllStudents where student.age < 20";
OQLQuery query = new OQLQuery(oql);
Object students = query.execute();
```

JDOQL [7, 9]

```
Query query =
  persistenceManager.newQuery(Student.class, "age < 20");
Collection students = (Collection)query.execute();
```

db4o SODA, using C# [4]

```
Query query = database.Query();
query.Constrain(typeof(Student));
query.Descend("age").Constrain(20).Smaller();
IList students = query.Execute();
```

All of the above approaches share a common set of problems:

- Modern integrated development environments (IDEs) do not check embedded strings for semantic and syntactic errors. In all the queries above, both the field `age` and the value `20` are expected to be numeric, but no IDE or compiler will check that this is actually correct. If the developer mistyped the query code – changing the name or type of the field `age`, for example – all of the above queries would break at runtime, without a single notice at compile time.

- Since modern IDEs will not automatically refactor field names that appear in strings, refactorings will cause class model and query strings to get out of sync. Suppose the field name `age` in the class Student is changed to `_age` because of a corporate decision on standard coding conventions. Now all existing queries for `age` would be broken, and would have to be fixed by hand.

- Modern agile development techniques encourage constant refactoring to maintain a clean and up-to-date class model that accurately represents an evolving domain model. If query code is difficult to maintain, it will delay decisions to refactor and inevitably lead to low-quality source code.

- All listed queries operate against the private implementation of the Student class

  ```
  student.age
  ```

  instead of using it's public interface

  ```
  student.getAge()  /  student.Age
  ```

  and thereby they break object-oriented encapsulation rules, disobeying the object-oriented principle that interface and implementation should be decoupled.

- Developers are constantly required to switch contexts between implementation language and query language. Queries can not use code that already exists in the implementation language.

- There is no explicit support for creating reusable query components. A complex query can be built by concatenating query strings, but none of the reusability features of the programming language (method calls, polymorphism, overriding) are available to make this process manageable. Passing a parameter to a string-based query is also awkward and error-prone.

- Embedded strings can be subject to injection attacks.

2 Design Goals

What if we could simply express the same query in plain Java or C# [1]?

```
// Java
student.getAge() < 20

// C#
student.Age < 20
```

The developer could write queries without having to think about a custom query language or API. The IDE could actively help to reduce typos. Queries would be fully type safe and accessible to the refactoring features of the IDE. Queries could also be prototyped, tested, and run against plain collections in memory without a database backend.

At first sight, this approach seems unsuitable as a database query mechanism. Naively executing Java/C# code against the complete extent of all stored objects of a class would incur a huge performance penalty, because all candidate objects would have to be instantiated from the database. A solution to this problem has been published in the paper on "Safe Query Objects" by Cook and Rai[3]: The source code or the byte code of the Java/C# query expression can be analyzed and optimized by translating it to the underlying persistence system's query language or API (SQL[6], OQL[1], JDOQL[9], EJBQL[11], SODA[10], etc.), and thereby take advantage of indexes and other optimizations of a database engine. In this paper we refine the original idea of safe query objects to provide a more concise and natural definition of native queries. We will also examine integrating queries into Java and .NET by leveraging more recent features of those language environments, including anonymous classes and delegates.

Therefore, our goals for native queries are:

100% native Queries should be completely expressed in the implementation language (Java or C#), and they should fully obey all language semantics.

100% object-oriented Queries should be runnable in the language itself, to allow unoptimized execution against plain collections without custom preprocessing.

100% type-safe Queries should be fully accessible to modern IDE features like syntax checking, type checking, refactoring, etc.

[1] C# or any other managed language with similar capabilities

optimizable It should be possible to translate a native query to a persistence architecture's query language or API for performance optimization. This could be done at compile time or at load time by source code or bytecode analysis and translation.

3 Defining the Native Query API

How should native queries look ? To produce a minimal design, we will evolve a simple query by adding each design attribute, one at a time. We will use Java and C# (.NET 2.0) as the implementation languages.

Let's begin with the class that we designed at the beginning of this paper. Furthermore, we'll assume that we want to query for "all students that are younger than 20 where the name contains an 'f' ".

1. The main query expression is easily written in the programming languages:

```
// Java
student.getAge() < 20 && student.getName().contains("f")

// C#
student.Age < 20 && student.Name.Contains("f")
```

2. We need some way to pass a Student object to the expression, as well as a way to pass the result back to the query processor. We can do this by defining a student parameter and by returning the result of our expression as a boolean value:

```
// pseudo-Java
(Student student){
  return student.getAge() < 20
      && student.getName().contains("f");
}

// pseudo-C#
(Student student){
  return student.Age < 20
      && student.Name.Contains("f");
}
```

3. Now we have to wrap the above partial construct into an object that is valid in our programming languages. That will allow us to pass it

to the database engine, a collection, or any other query processor. In .NET 2.0, we can simply use a delegate. In Java, we need a named method, as well as an object of some class to put around the method. This requires, of course, that we choose a name for the method as well as a name for the class. We decided to follow the example that .NET 2.0 sets for collection filtering. Consequently, the class name is "Predicate" and the method name is "match".

```
// Java
new Predicate(){
  public boolean match(Student student){
    return student.getAge() < 20
        && student.getName().contains("f");
  }
}

// C#
delegate(Student student){
  return student.Age < 20
      && student.Name.Contains("f");
}
```

4. For .NET 2.0, we are done designing the simplest possible query interface. The above is a valid object. For Java, our querying conventions should be standardized by designing an abstract base class for queries: the Predicate class.

```
// Java
public abstract class Predicate <ExtentType> {
    public <ExtentType> Predicate (){}
    public abstract boolean match (ExtentType candidate);
}
```

We still have to alter our Java query object slightly by adding the extent type to comply with the generics contract.

```
new Predicate <Student> () {
  public boolean match(Student student){
    return student.getAge() < 20
        && student.getName().contains("f");
  }
}
```

5. Although the above is conceptually complete, we would like to finish the derivation of the API by providing a full example. Specifically, we want to show what a query against a database would look like, so we can compare it against the string-based examples given in the introduction.

```
// Java
List <Student> students = database.query <Student> (
  new Predicate <Student> () {
    public boolean match(Student student){
      return student.getAge() < 20
          && student.getName().contains("f");
    }
  });


// C#
IList <Student> students = database.Query <Student> (
  delegate(Student student){
    return student.Age < 20
        && student.Name.Contains("f");
  });
```

The above example completes the core idea. We have refined Cook/Rai's concept of safe queries[3] by leveraging anonymous classes in Java and delegates in .NET. The result is a more concise and straightforward description of queries.

Adding all required elements of the API in a step-by-step fashion has allowed us to find the most natural and efficient way of expressing queries in Java and C#. Additional features, like parameterized and dynamic queries, can be included in native queries using a similar approach.

We have overcome the shortcomings of existing string-based query languages and provided an approach that promises improved productivity, robustness, maintainability and performance.

4 Specification Details

A final and thorough specification of native queries will only be possible after practical experience. Therefore, this section is speculative. We would like to point out where we see choices and issues with the native query approach and how they might be resolved.

4.1 Optimization

Regarding the API alone, native queries are not new. Without optimizations, we have merely provided "the simplest concept possible to run all instances of a class against a method that returns a boolean value". Such interfaces are well-known: Smalltalk 80 includes methods to select items from a collection based on a predicate [5, 2].

Optimization is the key new component of native queries. Users should be able to write native query expressions and the database should execute them with performance on par with the string-based queries that we described in the introduction to this paper.

Although the core concept of native queries is simple, the work needed to provide a performant solution is non-trivial. Code written in a query expression must be analyzed and converted to an equivalent database query format. It is not necessary for *all* code in a native query to be translated. If the optimizer cannot handle some or all code in a query expression, there always is the fallback to instantiate the actual objects and to run the query expression code, or part of it, with real objects after the query has returned intermediate values. Because this may be slow, it will be helpful to provide developers with feedback at development time. This feedback might include how the optimizer "understands" query expressions, and some description of the underlying optimization plan created for the expressions. This will help developers to adjust their development style to the syntax that is optimized best and will enable developers to provide feedback about desirable improved optimizations.

How will optimization actually work? At compile or load time, an enhancer (a separate application, or a "plug-in" to the compiler or loader) will inspect all native query expressions in source code or byte code, and will generate additional code in the most performant format the database engine supplies. At runtime, this substituted code will be executed instead of the former query expression. This mechanism will be transparent to the developer after she adds the optimizer to her compilation- or build-process or both.

Our peers have expressed doubts that satisfactory optimization is possible. Because both the native query format and the native database format are well defined, and because the development of an optimizer can be an ongoing task, we are very optimistic that excellent results are achievable. of an optimizer is ongoing evidence that excellent results are achievable?] The first results that Cook/Rai [3] produced with a mapping to JDO implementations are very encouraging. db4objects already shows a first preview of db4o with

unoptimized native queries today[4] and plans to ship a production ready version 5.0 with optimized native queries in November 2005.

4.2 Restrictions

Ideally, any code should be allowed in a query expression. In practice, restrictions are required to guarantee a stable environment, and to place an upper limit on resource consumption. We recommend:

variables Variable declarations should be legal in query expressions.

object creation Temporary objects are essential for complex queries so their creation should also be supported in query expressions.

static calls Static calls are part of the concept of OO languages so they should be legal.

faceless Query expressions are intended to be fast. They should not interact with the GUI.

threads Query expressions will likely be triggered in large numbers. Therefore, they should not be allowed to create threads.

security restrictions Since query expressions may actually be executed with real objects on the server, there need to be restrictions for what they are allowed to do there. It would be reasonable to allow and disallow method execution and object creation in certain namespaces/packages.

read only No modifications of persistent objects should be allowed within running query code. This limitation guarantees repeatable results and keeps transactional concerns out of the specification.

timeouts To allow for a limit to the use of resources, a database engine may choose to timeout long running query code. Timeout configuration does not have to be part of the native query specification, but it should be recommended to implementors.

memory limitation Memory limitations can be treated like timeouts. A configurable upper memory limit per query expression is a recommended feature for implementors.

undefined actions Unless explicitly not permitted by the specification, all constructs should be allowed.

4.3 Exceptions

It seems desirable that processing should continue after any Exception occurs in query expressions. A query expression that throws an uncaught exception should be treated as if it returned false. There should be a mechanism for developers to discover and track exceptions. We recommend that implementors support both exception callback mechanisms and exception logging.

4.4 Sorting

The sort order of returned objects might also be defined using native code. An exact definition goes beyond the scope of this paper but a simple example illustrates what this might look like. Using a Java Comparator:

```
// Java
List <Student> students = database.query <Student> (
  new Predicate <Student> () {
    public boolean match(Student student){
      return student.getAge() < 20 && student.getName().contains("f");
    }
});
Collections.sort(students, new Comparator <Student>(){
    public int compare(Student student1, Student student2) {
        return student1.getAge() - student2.getAge();
    }
});
```

The above code should be runnable both with and without an optimization processor. Querying and sorting could be optimized to be executed as one step on the database server, using the sorting functionality of the database engine.

5 Conclusion

There are powerful reasons for considering native queries as a mainstream standard. As we have shown, they overcome the shortcomings of string-based APIs. The full potential of native queries will be explored with their use in practice. They have already been demonstrated to provide high value in these areas:

Power Standard object-oriented programming techniques are available for querying.

Productivity Native queries enjoy the benefits of advanced development tools, including static typing, refactoring, and auto-completion.

Standard What SQL has never managed to achieve because of the diversity of SQL dialects may be achievable for native queries: Because the standard is well defined by programming language specifications, native queries can provide 100% compatibility across different database implementations.

Efficiency Native queries can be automatically compiled to traditional query languages or APIs so that they can leverage existing high-performance database engines.

Simplicity As shown, the API for native queries is only one class with one method. Hence, native queries are easy to learn, and a standardization body will find them easy to define. They could be submitted as a JSR to the Java Community Process.

Acknowledgments

We would like to thank Johan Strandler for his posting to a thread at TheServerSide that brought the two authors together, Patrick Römer for getting us started with first drafts of this paper, Rodrigo B. de Oliveira for contributing the delegate syntax for .NET, Klaus Wuestefeld for suggesting the term "Native Queries", Roberto Zicari, Rick Grehan and Dave Orme for proofreading drafts of this paper and all of the above for always being great peers to review ideas.

References

[1] R. G. G. Cattell, D. K. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez, editors. *The Object Data Standard ODMG 3.0*. Morgan Kaufmann, January 2000.

[2] W. Cook. Interfaces and specifications for the Smalltalk collection classes. In *OOPSLA*, 1992.

[3] W. R. Cook and S. Rai. Safe query objects: statically typed objects as remotely executable queries. In G.-C. Roman, W. G. Griswold, and B. Nuseibeh, editors, *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 97–106. ACM, 2005.

[4] db4objects web site. `http://www.db4o.com`.

[5] A. Goldberg and D. Robson. *Smalltalk-80: the Language and Its Implementation*. Addison-Wesley, 1983.

[6] ISO/IEC. Information technology - database languages - SQL - part 3: Call-level interface (SQL/CLI). Technical Report 9075-3:2003, ISO/IEC, 2003.

[7] JDO web site. `http://java.sun.com/products/jdo`.

[8] ODMG web site. `http://www.odmg.org`.

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

[10] SODA - Simple Object Database Access. . `http://sourceforge.net/projects/sodaquery` .

[11] Sun Microsystems. *Enterprise Java Beans Specification, version 2.1*. 2002. `http://java.sun.com/j2ee/docs.html`.