Running head: RESEARCH CRITIQUE

Research Critique of "Efficient Object Querying for Java"

David Moskowitz

Doctoral Student

Nova Southeastern University

Fall 2010

Introduction: The problem of querying Java objects

This document reviews the 2006 paper "Efficient Object Querying for Java" by Willis, Pearce, and Noble, published in Proceedings of the European Conference on Object-Oriented Programming (Willis, Pearce, & Noble, Efficient Object Querying for Java, 2006). The paper, based on the Master Thesis and earlier undergraduate work of one the authors (Willis, The Java Query Language , 2008), introduces a prototype system for querying Java objects: The Java Query Language (JQL). This review discusses the material in the original paper, developments in the field that have occurred since the original publication, and provides suggestions for future directions of research in this area.

Object Querying with JQL

A primary characteristic of programming languages, specifically object oriented languages, is the relationship between objects. This structure is facilitated through object composition, specifically encapsulated links to other objects or collections of objects. While the structural functionality of modern object oriented languages is high, the ability to query this structure is lacking.

JQL specifically addresses searching and joining of Java objects and collections. Join operations are commonly expresses in SQL as

```
SELECT [some fields] from [Table] Inner Join [Another Table]
where [Some Condition]
```

The authors illustrate Java's lack of support for this type of querying with a simple example:

```
Given a domain of Students and Faculty, find all Students who are
also Faculty
```

This query could be written in SQL as

```
SELECT students.* FROM students, Faculty
WHERE students.name = faculty.name
```

A Java implementation would most likely be done using nested loop. The paper gives the following example code for such a join:

```
List<Tuple2<Faculty,Student>> matches = new ArrayList<..>();
for(Faculty f : allFaculty) {
      for(Student s : allStudents) {
            if(s.name.equals(f.name)) {
                  matches.add(new Tuple2<Faculty,Student>(f,s));
}}}
```

While relatively simple, this implementation may not be the most efficient. In most cases, a hash join would be faster. However, the performance of the join depends on the size of each collection as well as the distribution of the name values within. A hash join is also more complicated to code. A code example of a hash join is included in Appendix A. The authors argue that in most cases a programmer will not perform this optimization, or even have the knowledge at coding time to determine if this is in fact the correct approach. By contrast, the JQL version of the above query is the following:

```
List<Tuple2<Faculty,Student>> matches;
matches = selectAll(Faculty f=allFaculty, Student s=allStudents:
                    f.name.equals(s.name));
```

JQL provides a direct analogy to SQL. In this JQL query, a Cartesian Join is performed on all collections listed before the colon and the results are filtered by the criteria specified after the colon. As in an SQL query on a database system, the query describes *what* needs to get done, not *how* it should be implemented. Similar to an SQL-based database system, it is the system, not the programmer, that determines how to optimize and execute the requested operations. By abstracting the object query implementation, JQL can encapsulate the necessary optimization and determine how best to perform the desired operation. The authors do a good job at describing the need and advantage of such a system. Most programmers who frequently code such manual searches would likely agree.

In addition to querying over specify collections, allStudents and allFacutly in the above example, queries can be performed over *all* instances of a class, referred to in the paper as object extents. The above JQL query would be modified as follows to perform this query

```
matches = selectAll(Faculty f, Student s: f.name.equals(s.name));
```

Here, no object instances are specified so the query defaults to all objects of a particular class. As an additional example of the utility of object extent querying, consider the test to very that a Singleton object is actually a Singleton (i.e., there exists only one instance the specific class). This assertion can be written in JQL as

```
assert 1 == selectAll(Singleton x).size();
```

This assertion is the equivalent of a database-level constraint and is generally difficult to achieve in a language such as Java unless access to an object is tightly controlled. JQL provides the mechanism to track objects extents and make a query such as this possible.

JQL Implementation

The JQL implementation comprises three components:

1. JQL Query Evaluator
2. JQL Compiler
3. The runtime system

The JQL Query Evaluator determines the optimal join ordering and selection strategy for each object collection in the query. JQL will generally use a hash join when an equality comparison is requested and a nested join for other cases. These two join strategies comprise an extremely limited subset of the strategies used by typical DBMS. There is no reason that JQL could not be extended to include additional strategies relevant to in-memory operations. Issues specific to disk-based systems, such as memory overflow and disk buffering, would not be an issue. In this respect, JQL is analogous to an in-memory object oriented databases.

The JQL Compiler translates JQL queries embedded in Java source code to the equivalent Java statements. Any source file containing JQL must be given the .jql extension. The JQL compiler will translate this file into a .java file with Java compatible syntax.

While this step could be included as part of an ANT build process (The Apache ANT Project, 2010), this required translation step is a shortcoming of the system. By including Java incompatible syntax, JQL is more difficult to incorporate into the development process. All JQL source files will fail to compile within a Java IDE. A better approach would be to use Java compatible syntax to specify JQL queries. Queries could be included as one or more String parameters to a method. For example, the above JQL could have been implemented as

```
matches = query.selectAll("Faculty f=allFaculty, Student
s=allStudents","f.name.equals(s.name)");
```

A shortcoming of this proposed approach is that static checking could not be done at compile time. However, an additional compile step could still be performed if needed. A similar class of tools, Object-relational mapping tools , such as Hibernate (Relational Persistence for Java and .NET, 2010), do dynamic query processing almost exclusively. Static syntax and type checking could also be incorporated as IDE plug-in, as is available for Hibernate.

A novel component JQL is it's run time object tracking mechanism. As mentioned above, all java objects are available for querying, a capability not native to the language. This feature is accomplished using AspectJ (The AspectJ Project, 2010), an aspect oriented programming tool for Java. Using AspectJ, each new object creation call is intercepted and a reference to the new object is saved. Using a popular tool such as AspectJ is an advance over other methods used by early approaches (ex. (Raimondas Leucevicius, 1997)) such as custom class loaders or static heap analysis.

This method of object extent tracking can impact run time performance since that *all* new objects created by the application are monitored. Such an approach will lead to unneeded memory usage, since querying is likely to be limited to a small subset of instantiated objects. Since only

references to the objects, not necessarily copies of the objects, need be created, this overhead may be acceptable. However, the system should allow selection as to which objects are monitored, as this is likely known at compile time. A production level implementation would certainly include this capability. The absence of selective object monitoring can be overlooked in a prototype such as JQL, though no mention is made of this as a possible future enhancement.

The authors rightly state that performance of this system is a key to adoption. To this end, they agree that JQL needs to be competitive with the best optimized code and must better than average or poor implemented code.

Several benchmarks are included in the paper. To profile joins, the authors developed comparison programs called HANDOPT and HANDPOOR, to simulate fully optimized non- JQL code and poorly optimized non-JQL code, representative of the range of solutions likely for non-JQL implementations.

The benchmark results presented in the paper are in line with expectations. The query evaluation benchmarks show that JQL marginally slower than the HANDOPT code while significantly better than the HANDPOOR implementation. This is certainly expected, as the benchmark is comparing a nested join with a hash join[1]. Benchmark results of object extent tracking show memory overhead dependent on the object creation profile of the program. This is also expected, as more object creation would lead to additional object tracking. Selective loading and efficient index structures would likely alleviate most memory overhead issues.

Background and Related Work

Most of the related work cited in the paper is in the area of Query Based Debuggers (QBD). Initial research in this field was done by Raimondas Leucevicius (Raimondas Leucevicius, 1997) in

[1] A nested join has time complexity o(MN) while with a hash join has complexity o(M+N) where M and N are the size of the two collections being joined.

the late 1990s. Additional work on QBDs was done by Willis during his undergraduate studies and presented in a paper by Willis, Pierce, and Noble in (Willis, Pearce, & Noble, Querying in Java, 2005).

Query Based Debuggers allow inspection of object graphs for debugging purposes and general application exploration. The 1997 Leucevicius paper describes using a QDB to understand a new application through ad hoc queries on instantiated objects. This example is seen as similar to how an analyst or programmer would use SQL to understand the data and constraints of an unfamiliar database system. QDBs are also proposed as an extension to simple invariant debugger break points (break when an explicit value is reached) to allow conditions to be set on object relationships. This use in debugging is analogous to the singleton example listed above. Instead of an assertion regarding object singularity, a breakpoint would be triggered when the invariant relation constraint is broken. Both approaches have their utility, one at run time and the other at debug time.

Many other concepts in the JQL paper are introduced in the 1997 Leucevicius work, such as nested loop versus hash join algorithms, join ordering and heuristics, object extent loading, and performance benchmarking. Willis' work on QDBs is an extension of the work from Leucevicius to Java and AspectJ. JQL is an extension of the Java based QDB to Java application level querying and was mentioned in Willis' QDB paper (Willis, Pearce, & Noble, Querying in Java, 2005) as a point of future work. While not currently in mainstream usage, QDBs are still an area of research (Czyz & Jayaraman, 2007).

The paper does make mention of what is currently the most established tool in the area of object querying, Microsoft's LINQ (Box & Hejlsberg, 2007). LINQ, or Language-Integrated Query, is a framework for the Microsoft .NET platform. Using LINQ, the same syntax can be used to query program objects, XML documents, SQL databases, or other data (via third party providers). At the time of the JQL paper, LINQ was just recently announced (Fernandez, 2005) and had not yet

been incorporated into .NET. Since that time, LINQ has become the standard for data access on the Microsoft platform. The LINQ framework is seen by some as a significant advantage of .NET over Java (Worthington, 2008). An example of a LINQ query is provided in Appendix B.

Surprisingly, no mention was made in the paper of Java-based tools for querying relational databases, such as Object-relational mapping tools. One of the more popular of these, Hibernate (Relational Persistence for Java and .NET, 2010), was released in 2001, and was quite popular by 2004. Hibernate provides many of the features of JQL, but only for persistent objects (objects and collections that will eventually be written to a database). Though not directly related, it is possible that the approach taken by Hibernate and other such tools could be extended to in memory data structures.

Significance of the Work

The JQL paper introduced an innovative and potentially seminal solution to the lack of object querying capabilities in Java and other programming languages. Using established research in Query Based Debuggers, the authors extend this earlier work to potentially wider usage through integration into the application layer and potentially the Java language itself. The prototype developed illustrated the potential of such a system and approach. As such, the paper was certainly worthy of publication.

However, the paper has not been seen to influence subsequent developments in this problem area. The same limited object querying capabilities mentioned in the paper still exist in Java to this day. Other languages, specifically Microsoft's' .NET , have made more progress towards incorporating these features. Subsequent work in the field, including LINQ and other LINQ implementations for other language, do not reference or appear influenced by JQL.

Future Research

As the authors state in their conclusion, widespread acceptance of a tool such as JQL would require integration into the Java language itself. As Java is a language governed by a large corporation (Oracle) and a strict change control process (the Java Community Process), such an integration would certainly take years to achieve. A more realistic goal is to promote JQL as a third party library, but one more compatible with Java syntax. An approach like this enabled tools such as Hibernate and Spring to influence later Java technologies such as EJB3 and CDI. After it's success and widespread usage as a add on library, discussion of incorporation into Java itself could be undertaken.

Further work can also be done, not in enhancing JQL's query capabilities, but in integrating it with an in-memory databases (IMDB). JQL functionality could be seen as loading the desired object collections into an IMDB and then executing the desired query. The query could be written in pure SQL and sent to the IMDB without an additional parsing step. Likely IMDB candidates are the open source HQSLQ (The HSQL Development Group, 2010) or JavaDB (Oracle, 2010) databases, both of which are full featured SQL compatible DBMS systems. Issues would likely arise, such the overhead involved in loading the IMDB and maintaining synchronization between IMDB and source objects. An approach like the AspectJ interceptor used by JQL would be feasible in this situation as well. This approach could realize the goals set forth in JQL without implementing work already done in the mature field of database query processing.

Conclusion

This review of the paper "Efficient Object Querying for Java" has shown how the Java Query Language developed from earlier research on Query Based Debuggers toward the goal of enabling high level querying of Java objects. The ideas presented in the paper are found to be sound and address a widespread need, but have not gained mainstream use or acceptance. Additional

recommendations for future work, particularly towards incorporating an in-memory databases to enhance querying capabilities, were proposed.

References

Box, D., & Hejlsberg, A. (2007, February). *LINQ: .NET Language-Integrated Query.* Retrieved November 13, 2010, from MSDN: http://msdn.microsoft.com/library/bb308959.aspx

Czyz, J. K., & Jayaraman, B. (2007). Declarative and Visual Debugging in Eclipse. *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology Exchange*, (pp. 31-35).

Fernandez, D. (2005, September 13). *Announced at PDC: The LINQ Project.* Retrieved November 13, 2010, from Dan Fernandez's Blog: http://blogs.msdn.com/b/danielfe/archive/2005/09/13/464904.aspx

Meijer, E., Beckman, B., & Bierman, G. (2006). LINQ: reconciling object, relations and XML in the .NET framework. *Proceedings of the 2006 ACM SIGMOD international conference on Management of data* , 706 - 706.

Microsoft. (2010). *How to: Perform Inner Joins (C# Programming Guide).* Retrieved November 13, 2011, from MSDN: http://msdn.microsoft.com/en-us/library/bb397941.aspx

Oracle. (2010). *JavaDB.* Retrieved November 13, 2010, from Oracle.com: http://www.oracle.com/technetwork/java/javadb/overview/index.html

Raimondas Leucevicius, U. H. (1997). Query-Based Debugging of Object-Oriented Programs. *OOPSLA '97 Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, (pp. 304-317). Atlanta.

Relational Persistence for Java and .NET. (2010). Retrieved November 15, 2010, from Hibernate Web Site: http://www.hibernate.org/

The Apache ANT Project. (2010). Retrieved November 14, 2010, from The Apache ANT Project: http://ant.apache.org/

The AspectJ Project. (2010). Retrieved November 13, 2010, from Eclipse.org: http://www.eclipse.org/aspectj/

The HSQL Development Group. (2010). *HSQLDB.* Retrieved November 13, 2010, from hsqldb.org: http://hsqldb.org/

Willis, D. (2005). *Squirrel : A Query Based Debugger For Java.* Victoria University of Wellington.

Willis, D. (2008). *The Java Query Language* . Victoria University of Wellington.

Willis, D., Pearce, D. J., & Noble, J. (2008). Retrieved November 13, 2010, from JQL: The Java Query Language: http://homepages.mcs.vuw.ac.nz/~djp/JQL/index.html

Willis, D., Pearce, D. J., & Noble, J. (2006). Efficient Object Querying for Java. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* , 28-49.

Willis, D., Pearce, D. J., & Noble, J. (2005). Querying in Java. *Workshop on Multi-paradigm Programming in Object Oriented Languages(OOPSLA).*

Worthington, D. (2008, January 29). *Does .NET With LINQ Beat Java?* Retrieved November 13, 2010, from SD Times: http://www.sdtimes.com/content/article.aspx?ArticleID=31643

Appendix A - Hash Join Code

```
HashMap<String,List<Faculty>> tmp=
                        new HashMap<String,List<Faculty>> ();
for(Faculty f : allFaculty) {
      List<Faculty> fs = tmp.get(f.name);
      if(fs == null) {
            fs = new ArrayList<Faculty>();
            tmp.put(f.name,fs);
      }
      fs.add(f);
}
List<Tuple2<Faculty,Student>> matches=
                  new ArrayList<Tuple2<Faculty,Student>>();
for(Student s : allStudents) {
      List<Faculty> fs = tmp.get(s.name);
      if(fs != null) {
            for(Faculty f : fs) {
                  matches.add(new Tuple2<Faculty,Student>(f,s));
            }
      }
}
```

Appendix B- LINQ Query

The following C# query is taken from the LINQ online reference material (Microsoft, 2010).

```
//Given the following Person and Pet objects
Person magnus = new Person {FirstName = "Magnus", LastName = "Hedlund" };
Person terry = new Person {FirstName = "Terry", LastName = "Adams" };
Person charlotte = new Person {FirstName = "Charlotte", LastName = "Weiss" };
Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };
Person rui = new Person { FirstName = "Rui", LastName = "Raposo" };

List<Person> people = new List<Person> {
      magnus, terry, charlotte, arlene, rui };

Pet barley = new Pet { Name = "Barley", Owner = terry };
Pet boots = new Pet { Name = "Boots", Owner = terry };
Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
Pet bluemoon = new Pet { Name = "Blue Moon", Owner = rui };
Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

List<Pet> pets = new List<Pet> {
      barley, boots, whiskers, bluemoon, daisy };

//return a list of Pets and their Owners
var query = from person in people
            join pet in pets on person equals pet.Owner
           select new {
                  OwnerName = person.FirstName, PetName = pet.Name };
```

Similar syntax would be used for SQL database data sources.