

# Efficient Object Querying for Java

Darren Willis, David J. Pearce and James Noble

Computer Science, Victoria University of Wellington, NZ,  
{darren,djp,kjx}@mcs.vuw.ac.nz

**Abstract.** Modern programming languages have little or no support for querying objects and collections. Programmers are forced to hand code such queries using nested loops, which is both cumbersome and inefficient. We demonstrate that first-class queries over objects and collections improve program readability, provide good performance and are applicable to a large number of common programming problems. We have developed a prototype extension to Java which tracks all objects in a program using AspectJ and allows first-class queries over them in the program. Our experimental findings indicate that such queries can be significantly faster than common programming idioms and within reach of hand optimised queries.

## 1 Introduction

No object stands alone. The value and meaning of objects arise from their relationships with other objects. These relationships can be made explicit in programs through the use of pointers, collection objects, algebraic data types or other relationship constructs (e.g. [5, 27–29]). This variety suggests that programming languages provide good support for relationships. We believe this is not entirely true — many relationships are *implicit* and, as such, are not amenable to standard relationship constructs. This problem arises from the great variety of ways in which relationships can manifest themselves. Consider a collection of student objects. Students can be related by name, or student ID — that is, distinct objects in our collection can have the same name or ID; or, they might be related by age bracket or street address. In short, the abundance of such implicit relationships is endless.

Most programming languages provide little support for such arbitrary and often dynamic relationships. Consider the problem of *querying* our collection to find all students with a given name. The simplest solution is to traverse the collection on demand to find matches. If the query is executed frequently, we can improve upon this by employing a hash map from names to students to get fast lookup. This is really a *view* of the original collection optimised for our query. Now, when students are added or removed from the main collection or have their names changed, the hash map must be updated accordingly.

The two design choices outlined above (traversal versus hash map) present a conundrum for the programmer: which should he/she choose? The answer, of

course, depends upon the ratio of queries to updates — something the programmer is unlikely to know beforehand (indeed, even if it is known, it is likely to change). Modern OO languages compound this problem further by making it difficult to move from one design to the other. For example, consider moving from using the traversal approach to using a hash map view. The easiest way to ensure both the collection and the hash map are always consistent is to encapsulate them together, so that changes to the collection can be intercepted. This also means providing a method which returns the set of all students with a given name by exploiting the hash map’s fast lookup. The problem is that we must now replace the manual traversal code — which may be scattered throughout the program — with calls to this new method and this is a non-trivial task.

In this paper, we present an extension to Java that supports efficient querying of program objects. We allow queries to range over collection objects and also the set of all instantiated objects. In this way, manual code for querying implicit relationships can be replaced by simple query statements. This allows our query evaluator to optimise their execution, leading to greater efficiency. In doing this, we build upon a wealth of existing knowledge on query optimisation from the database community. Our focus differs somewhat, however, as we are interested in the value of querying as a programming construct in its own right. As such, we are not concerned with issues of persistence, atomic transactions, roll-back and I/O efficient algorithms upon which the database community expends much effort. Rather, our “database” is an object-oriented program that fits entirely within RAM.

This paper makes the following contributions:

- We develop an elegant extension to the Java language which permits object querying over the set of all program objects.
- We present experimental data looking at the performance of our query evaluator, compared with good and bad hand-coded implementations. The results demonstrate that our system is competitive with an optimised hand-coded implementation.
- We present a technique which uses AspectJ to efficiently track object extent sets.
- We present experimental data looking at the performance of our object tracking system on the SPECjvm98 benchmark suite. The results demonstrate that our approach is practical.

## 2 Object Querying

Figure 1 shows the almost generic diagram of students attending courses. Versions of this diagram are found in many publications on relationships [5, 6, 11, 31]. Many students attend many courses; Courses have a course code, a title

string and a teacher; students have IDs and (reluctantly, at least at our university's registry) names. A difficulty with this decomposition is representing students who are also teachers. One solution is to have separate `Student` and `Teacher` objects, which are related by name. The following code can then be used to identify students who are teachers:

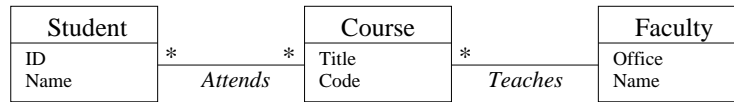
```
List<Tuple2<Faculty,Student>> matches = new ArrayList<..>();
for(Faculty f : allFaculty) {
    for(Student s : allStudents) {
        if(s.name.equals(f.name)) {
            matches.add(new Tuple2<Faculty,Student>(f,s));
        }
    }
}
```

In database terms, this code is performing a *join* on the name field for the `allFaculty` and `allStudent` collections. The code is cumbersome and can be replaced with the following *object query*, which is more succinct and, potentially, more efficient:

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

This gives exactly the same set of results as the loop code. The `selectAll` primitive returns a list of tuples containing all possible instantiations of the *domain variables* (i.e. those declared before the colon) where the *query expression* holds (i.e. that after the colon). The domain variables determine the set of objects which the query ranges over: they can be initialised from a collection (as above); or, left uninitialised to range over the entire *extent set* (i.e. the set of all instantiated objects) of their type. Queries can define as many domain variables as necessary and can make use of the usual array of expression constructs found in Java. One difference from normal Java expressions is that boolean operators, such as `&&` and `||`, do not imply an order of execution for their operands. This allows flexibility in the order they are evaluated, potentially leading to greater efficiency.

As well as its simplicity, there are other advantages to using this query in place of the loop code. The query evaluator can apply well-known optimisations which the programmer might have missed. By leaving the decision of which optimisation to apply until runtime, it can make a more informed decision based upon dynamic properties of the data itself (such as the relative size of input sets) — something that is, at best, difficult for a programmer to do. A good example, which applies in this case, is the so-called *hash-join* (see e.g. [26]). The idea is to avoid enumerating all of `allFaculty`  $\times$  `allStudents` when there are few matches. A hash-map is constructed from the largest of the two collections which maps the value being joined upon (in this case `name`) back to its objects. This still requires  $O(sf)$  time in the worst-case, where  $s = |\text{allStudents}|$  and  $f = |\text{allFaculty}|$ , but in practice is likely to be linear in the number of matches (contrasting with a nested loop which *always* takes  $O(sf)$  time).



```

class Student { String name; int ID; ... }
class Faculty { String name; String office; ... }

class Course {
  String code, title;
  Faculty teacher;
  HashSet<Student> roster;
  void enrol(Student s) { roster.add(s); }
  void withdraw(Student s) { roster.remove(s); }
}
  
```

**Fig. 1.** A simple UML diagram, describing students that attend courses and teachers that teach them and a typical implementation. Protection specifiers on fields and accessor methods are omitted for simplicity.

```

// selectAll(Faculty f=allFaculty, Student s=allStudents
//           : f.name.equals(s.name));

HashMap<String,List<Faculty>> tmp = new HashMap<...>();
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<...>();
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));
    }
  }
}
}
  
```

**Fig. 2.** Illustrating a hash-join implementation of the simple query (shown at the top) from Section 2. The code first creates a map from `Faculty` names to `Faculty` objects. Then, it iterates over the `allStudents` collection and looks up those `Faculty` members with the same name.

Figure 2 illustrates a hash-join implementation of the original loop. We believe it is highly unlikely a programmer would regularly apply this optimisation in practice. This is because it is noticeably more complicated than the nested-loop implementation and requires considerable insight, on behalf of the programmer, to understand the benefits. Even if he/she had appreciated the value of using a hash join, the optimal ordering (i.e. whether to map from names to Faculty or from names to Students) depends upon the relative number of students and faculty (mapping to the smaller is generally the best choice [26]). Of course, a clever programmer could still obtain optimal performance by manually enumerating all possible orderings and including a runtime check to select the best one. But, is this really likely to happen in practice? Certainly, for larger queries, it becomes ridiculously impractical as the number of valid orderings grows exponentially. Using a query in place of a manual implementation allows the query evaluator to perform such optimisations. And, as we will see in Section 4, there is a significant difference between a good hand-coded implementation and a poor one — even for small queries.

## 2.1 Querying Object Extents

While object querying could be limited to collections alone, there is additional benefit in allowing queries to range over the set of all instantiated objects. An interesting example lies in expressing and enforcing *class invariants*. Class invariants are often captured using universal/existential quantifiers over object extents (e.g. [3, 30, 34, 35]). Queries provide a natural approach to checking these invariants.

In a conventional OO language, such as Java, it is possible to express and enforce some class invariants using simple assertions. For example:

```
class BinTree {
    private BinTree left;
    private BinTree right;
    private Object value;

    public BinTree(BinTree l, BinTree r) {
        left = l; right = r;
        assert left != right;
    }
    void setLeftTree(BinTree l) {
        left = l;
        assert left != right;
    }
    ...
}
```

Here, the class invariant `left!=right` is enforced by asserting it after every member function. This allows programmers to identify the exact point during

a program's execution that an incorrect state is reached — thus preventing a *cause-effect gap* [12].

A limitation of this approach is that it cannot easily express more wide-ranging class invariants. The above tries (unsuccessfully) to enforce the invariant that there is no aliasing between trees. In other words, that no `BinTree` object has more than one parent and/or the same subtrees. The simple approach using `assert` can only *partially* express this because it is limited to a particular instance of `BinTree` — there is no way to quantify over all instances. The programmer could rectify this by maintaining a collection of the class's instances in a static class variable. This requires a separate implementation for each class and, once in place, is difficult to disable (e.g. for software release). Another option is to maintain a back-pointer in each `BinTree` object, which points to its parent. Then, before assigning a tree to be a subtree of another, we check whether it already has a parent. Again, this approach suffers from being difficult to disable and requiring non-trivial implementation. Indeed, properly maintaining this parent-child relationship is a tricky task that could easily be implemented incorrectly — potentially leading to a false impression that the class invariant holds.

Using an object query offers a cleaner, more succinct and more manageable solution:

```
assert null == selectA(BinTree a, BinTree b :
                      (a.left == b.left && a != b) ||
                      (a.right == b.right && a != b) ||
                      (a.left == b.right));
```

This uses the `selectA` primitive which returns a matching tuple (if there is one), or `null` (otherwise). Using `selectA` (when applicable) is more efficient than `selectAll` because the query evaluator can stop as soon as the first match is made. Notice that, in the query, `a` and `b` can refer to the same `BinTree` object, hence the need to guard against this in the first two cases.

Other examples of interesting class invariants which can be enforced using object queries include the singleton pattern [13]:

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

and, similarly, the fly-weight pattern [13]:

```
assert null == selectA(Value x, Value y : x.equals(y) && x != y);
```

The above ensures that flyweight objects (in this case `Value` objects) are not duplicated, which is the essence of this pattern.

## 2.2 Dynamic Querying

So far, we have assumed that queries are statically compiled. This means they can be checked for well-formedness at compile time. However, our query evaluator maintains at runtime an Abstract Syntax Tree (AST) representation of the query

for the purposes of query optimisation. An AST can be constructed at runtime by the program and passed directly to the query evaluator. This form of *dynamic query* has the advantage of being more flexible, albeit at the cost of runtime type checking. The syntax of a dynamic query is:

```
List<Object []> selectAll(Query stmt);
```

Since static typing information is not available for dynamic queries, we simply implement the returned tuples as `Object` arrays. The `Query` object encapsulates the domain variables and query expression (a simple AST) making up the query. The following illustrates a simple dynamic query:

```
List<Object []> findEquivInstances(Class C, Object y) {  
    // build selectAll(C x : x.equals(y));  
    Query query = new Query();  
    DomainVar x = query.newDomainVar(C);  
    query.addConjunct(new Equals(x,new ConstRef(y)));  
    // run query  
    return query.selectAll();  
}
```

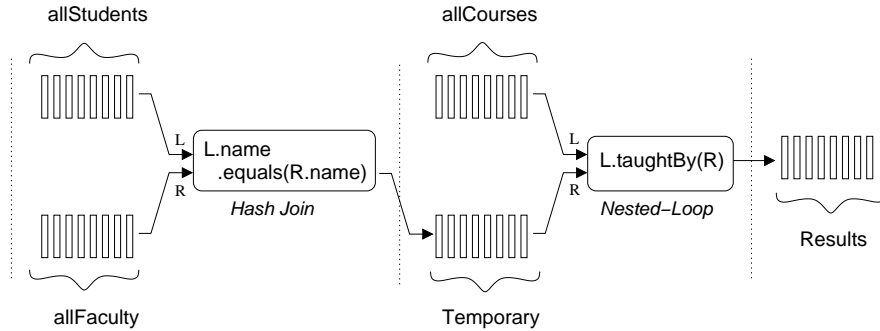
This returns all instances, `x`, of class `c` where `x.equals(y)` holds. This query cannot be expressed statically, since the class `c` is unknown at compile time. Dynamic queries are more flexible: in particular, we can construct queries in direct response to user input.

### 3 Implementation

We have prototyped a system, called the *Java Query Language (JQL)*, which permits queries over object extents and collections in Java. The implementation consists of three main components: a compiler, a query evaluator and a runtime system for tracking all active objects in the program. The latter enables the query evaluator to range over the extent sets of all classes. Our purpose in doing this is twofold: firstly, to assess the performance impact of such a system; secondly, to provide a platform for experimenting with the idea of using queries as first-class language constructs.

#### 3.1 JQL Query Evaluator

The core component of the JQL system is the query evaluator. This is responsible for applying whatever optimisations it can to evaluate queries efficiently. The evaluator is called at runtime with a tree representation of the query (called the *query tree*). The tree itself is either constructed by the JQL Compiler (for static queries) or by the user (for dynamic queries).



**Fig. 3.** Illustrating a query pipeline.

**Evaluation Pipeline.** The JQL evaluator evaluates a query by pushing tuples through a staged pipeline. Each stage, known as a *join* in the language of databases, corresponds to a condition in the query. Only tuples matching a join's condition are allowed to pass through to the next. Those tuples which make it through to the end are added to the result set. Each join accepts two lists of tuples, L(ef) and R(igh)t, and combines them together producing a single list. We enforce the restriction that, for each intermediate join, either both inputs come from the previous stage or one comes directly from an input collection and the other comes from the previous stage. This is known as a *linear processing tree* [19] and it simplifies the query evaluator, although it can lead to inefficiency in some cases.

The way in which a join combines tuples depends upon its type and the operation (e.g. `==`, `<` etc) it represents. JQL currently supports two join types: *nested-loop join* and *hash join*. A nested-loop join is a two-level nested loop which iterates each of  $L \times R$  and checks for a match. A hash join builds a temporary hash table which it uses to check for matches. This provides the best performance, but can be used only when the join operator is `==` or `equals()`. Future implementations may take advantage of B-Trees, for scanning sorted ranges of a collection.

Consider the following simple query for finding all students who teach a course (presumably employed as an RA):

```
r = selectAll(Student s=allStudents, Faculty f=allFaculty,
             Course c=allCourses : s.name.equals(f.name) && c.taughtBy(f));
```

Figure 3 illustrates the corresponding query pipeline. Since the first join represents an `equals()` operation, it is implemented as a hash-join. The second join, however, represents an arbitrary method call and must be implemented as a nested-loop. The tuples which are produced from the first join have the form  $\langle Student, Faculty \rangle$  and are stored in a temporary location before being passed into the second join.

**Join Ordering.** The ordering of joins in the pipeline can dramatically effect the time needed to process the query. The cost of a single join is determined by its input size, (i.e.  $|L| \times |R|$ ) while its *selectivity* affects the input size (hence, cost) of subsequent joins. Selectivity is the ratio of the number of tuples which do not match to the input size<sup>1</sup>. Thus, highly selective joins produce relatively few matches compared with the amount of input. To find a minimal cost ordering, we must search every  $n!$  possible configurations and, in fact, this is known to be an NP-complete problem [18]. A further difficulty is that we cannot know the true selectivity of a given join without first running it. One approach is to use a fixed selectivity heuristic for each operation (e.g. `==` is highly selective, while `!=` is not). Alternatively, we can *sample* the join’s selectivity by testing a small number of tuples and seeing how many are matched before evaluating the whole query [15, 20].

The JQL evaluator supports both approaches for estimating selectivity. For the fixed heuristic approach, the following selectivity values are used: 0.95 for `==` and `equals()`; 0.5 for `<`, `<=`, `>`, `>=` and `compare()`; 0.2 for `!=`; finally, 0.1 for arbitrary methods. The sampling approach passes 10 randomly selected tuples from the input through each join and uses the number of matches as an estimator of selectivity. We find that, even with a sample size of just 10 tuples, surprisingly accurate results can be obtained.

We have implemented several join ordering strategies in an effort to assess their suitability. We now briefly discuss each of these in turn:

- EXHAUSTIVE: This strategy enumerates each possible configuration, selecting that with the lowest cost. To determine the overall cost of a pipeline, we use essentially the same procedure as outlined above.
- MAX\_SELECTIVITY: This strategy orders joins based solely on their selectivity, with the most selective coming first. This simple approach has the advantage of avoiding an exponential search and, although it will not always find the best ordering, we find it generally does well in practice. This is essentially the same strategy as that used in the PTQL system [14].

Many other heuristics have been proposed in the literature (see e.g. [18, 19, 37, 36]) and, in the future, we hope to implement more strategies to help determine which is most suited to this domain.

### 3.2 JQL Compiler

The JQL compiler is a prototype source-to-source translator that replaces all `selectAll` / `selectA` statements with equivalent Java code. When a query statement is encountered the compiler converts the query expression into a sequence

---

<sup>1</sup> We follow Lencevicius [20] with our meaning of selectivity here. While this contrasts with the database literature (where it means the opposite), we feel this is more intuitive.

of Java statements that construct a query tree and pass this to the query evaluator. The value of using a compiler, compared with simply writing dynamic queries (as in Section 2.2), is twofold: firstly, the syntax is neater and more compact; secondly, the compiler can check the query is well-formed, the value of which has been noted elsewhere [4, 9, 10].

The query tree itself is a fairly straightforward Abstract Syntax Tree. For ease of implementation, our prototype requires that queries be expressed in CNF. This way the query can be represented as an array of expressions, where each is ANDed together.

### 3.3 JQL Runtime System

To track the extent sets of all objects, we use AspectJ to intercept and record all calls to `new`. The following example illustrates a simple aspect which does this:

```
aspect MyAspect {
    pointcut newObject() : call(* *.new(..)) && !within(MyAspect);
    after() : newObject() { System.out.println("new called"); }
}
```

This creates a pointcut, `newObject()`, which captures the act of calling `new` on any class except `MyAspect`. This is then associated with advice which executes whenever a join point captured by `newObject()` is triggered (i.e. whenever `new` is called). Here, `after()` indicates the advice executes immediately after the join point triggers. Notice that we must use `!within(MyAspect)` to protect against an infinite loop which could arise if `MyAspect` allocates storage inside the advice, resulting in the advice triggering itself.

**Implementation** To track all program objects in the program we use an Aspect (similar to above) which advises all calls to `new()` with code to record a reference to the new object. This aspect is shown in Figure 4. One exception is the use of “`returning(...)`” which gives access to the object reference returned by the `new` call. We use this aspect to provide a facility similar to the ‘`allInstances`’ message in Smalltalk, without having to produce a custom JVM.

The `TrackingAspect` maintains a map from classes to their `ExtentSets`. An `ExtentSet` (code not shown) holds every object of its class using a weak reference. Weak references do not prevent the garbage collector from reclaiming the object they refer to and, hence, an `ExtentSet` does not prevent its objects from being reclaimed. In addition, an `ExtentSet` has a redirect list which holds the `ExtentSets` of all its class’s subclasses. In the case of an interface, the redirect list refers to the `ExtentSet` of all implementing classes and subinterfaces. The reason for using redirect lists is to avoid storing multiple references for objects whose class either implements some interface(s) or extends another class. Note that only `ExtentSets` which correspond to concrete classes will actually contain object references, as interfaces and abstract classes cannot be instantiated.

```

public aspect TrackingAspect {
    Hashtable<Class,ExtentSet> extentSets = new Hashtable<...>();

    pointcut newObject() : call(*.new(..)) && !within(TrackingAspect.*);

    after() returning(Object o) : newObject() {
        Class C = o.getClass();
        getExtentSet(C).add(o);
    }

    ExtentSet getExtentSet(Class C) {
        // Find extent set for C. If there is none, create one.
        ExtentSet set;
        synchronized(extentSets) {
            set = extentSets.get(C);
            if(set == null) {
                set = new ExtentSet();
                extentSets.put(C, set);
                Class S = C.getSuperClass();
                if(S != null) {
                    getExtentSet(S).link(set);    // Link superclass set
                }
                for(Class I : C.getInterfaces()) {
                    getExtentSet(I).link(set);    // Link interface set
                }
            }
        }
    }
}

```

**Fig. 4.** An aspect for tracking all objects in the program.

An important consideration is the effect of synchronisation. We must synchronise on the `Hashtable` and, hence, we are essentially placing a lock around object creation. In fact, the multi-level nature of the extent set map can help somewhat. This is because new `ExtentSets` will only be added to the outer `Hashtable` infrequently. A more advanced data structure should be able to exploit this and restrict the majority of synchronisation to within individual `ExtentSets`. This way, synchronisation only occurs between object creations of the same class. We have experimented with using `ConcurrentHashMap` for this purpose, although we saw no performance improvements. We hope to investigate this further in the future and expect it likely the two tier structure of the extent sets will obviate most of the synchronisation penalty.

## 4 Performance

We consider that the performance of the JQL system is important in determining whether it could gain widespread use. Ideally, the system should be capable of evaluating queries as fast as the optimal hand-coded loop. This is difficult to achieve in practice due to the additional overheads introduced by the pipeline design, and in the necessary search for good join orders. However, we argue that merely being competitive with the best hand-coded loops is a sufficient indicator of success, since it is highly unlikely a programmer will often write optimal hand-coded loops in large-scale programs.

Therefore, in this section we investigate the performance of the JQL system in a number of ways. Firstly, we compare its performance against hand-coded loops across three simple benchmarks to determine its competitiveness. Secondly, we evaluate the overhead of the object tracking mechanism using the SPECjvm98 benchmarks [1].

In all experiments which follow, the experimental machine was an Intel Pentium IV 2.5GHz, with 1GB RAM running NetBSD v3.99.11. In each case, Sun's Java 1.5.0 (J2SE 5.0) Runtime Environment and Aspect/J version 1.5M3 were used. Timing was performed using the standard `System.currentTimeMillis()` method, which provides millisecond resolution (on NetBSD). The source code for the JQL system and the three query benchmarks used below can be obtained from <http://www.mcs.vuw.ac.nz/~djp/JQL/>.

### 4.1 Study 1 — Query Evaluation

The purpose of this study was to investigate the query evaluator's performance, compared with equivalent hand-coded implementations. We used three queries of different sizes as benchmarks for comparison. Two hand-coded implementations were written for each: HANDOPT and HANDPOOR. The former represents the best implementation we could write. This always used hash-joins when possible and employed the optimal join ordering. The HANDPOOR implementation was the exact opposite, using only nested-loop joins and the worst join ordering possible — a pessimistic but nonetheless possible outcome for novice or distracted programmers. Our purpose with these implementations was to determine the range of performance that could be expected for hand-coded queries. This is interesting for several reasons: firstly, the programmer is unlikely to apply all the optimisations (e.g. hash-joins) that are possible; secondly; the programmer is unlikely to select an optimal join order (indeed, the optimal may vary dynamically as the program executes). The question, then, was how close the JQL evaluator performance was, compared with the HANDOPT implementation.

**Experimental setup.** The three query benchmarks are detailed in Table 1. The queries range over the lists of `Integers` L1, ..., L4 which, for simplicity, were always kept the same size. Let  $n$  be the size of each list. Then, each was generated by initialising with each integer from  $\{1, \dots, n\}$  and randomly shuffling. Note,

Name	Details
OneStage	<pre>selectAll(Integer a=L1, Integer b=L2 : a == b);</pre> <p>This benchmark requires a single pipeline stage. Hence, there is only one possible join ordering. The query can be optimised by using a hash-join rather than a nested loop implementation.</p>
TwoStage	<pre>selectAll(Integer a=L1, Integer b=L2, Integer c=L3           : a == b &amp;&amp; b != c);</pre> <p>This benchmark requires two pipeline stages. The best join ordering has the joins ordered as above (i.e. == being first). The query can be further optimised by using a hash-join rather than a nested loop implementation for the == join.</p>
ThreeStage	<pre>selectAll(Integer a=L1, Integer b=L2, Integer c=L3,           Integer d=L4 : a == b &amp;&amp; b != c &amp;&amp; c &lt; d);</pre> <p>This benchmark requires three pipeline stages. The best join ordering has the joins ordered as above (i.e. == being first). The query is interesting as it makes sense to evaluate <code>b != c</code> before <code>c &lt; d</code>, even though the former has lower selectivity. This query can be optimised using a hash-join as before.</p>

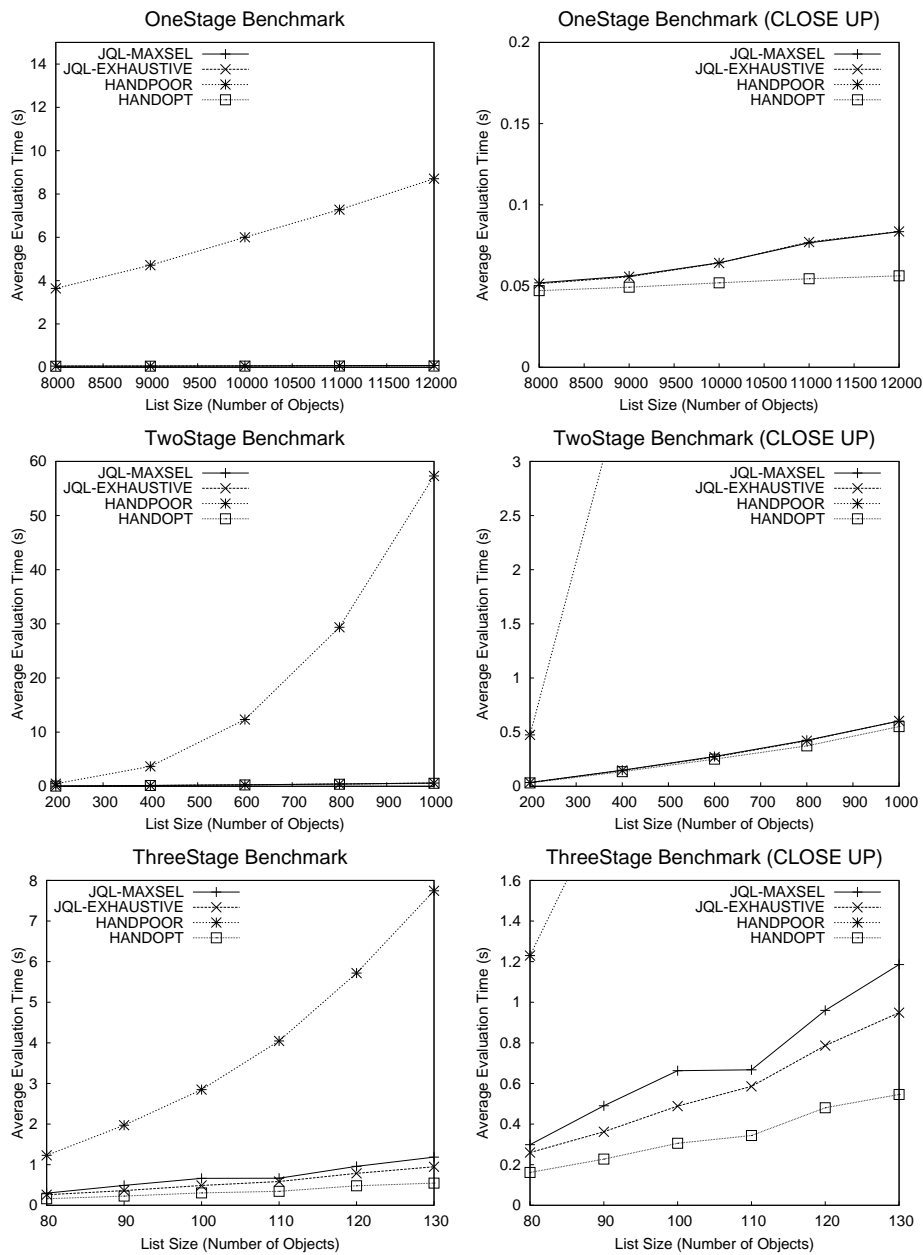
**Table 1.** Details of the three benchmark queries

the worst case time needed to evaluate `StageOne`, `StageTwo` and `StageThree` queries is  $O(n^2)$ ,  $O(n^3)$  and  $O(n^4)$ , respectively.

For each query benchmark, four implementations were tested: the two hand-coded implementations (HANDOPT, HANDPOOR); and, the JQL query evaluator using the MAX\_SELECTIVITY and EXHAUSTIVE join ordering strategies. For all JQL tests, join selectivity was estimated using the fixed heuristic outlined in Section 3.1, but not the sampling approach. The reason for this is simply that, for these queries, the two approaches to estimating selectivity produced very similar results.

Each experiment consisted of measuring the average time taken for an implementation to evaluate one of the query benchmarks for a given list size. The average time was measured over 10 runs, with 5 ramp-up runs being performed beforehand. These parameters were sufficient to generate data with a variation coefficient (i.e. standard deviation over mean) of  $\leq 0.15$  — indicating low variance between runs. Experiments were performed for each query and implementation at different list sizes (i.e.  $n$ ) to gain insight into how performance varied with  $n$ .

**Discussion** The results of the experiments are shown in Figure 5. The main observation is that there is a significant difference between the HANDOPT and HANDPOOR implementations, even for the small OneStage query. Furthermore, while the performance of JQL is always marginally slower (regardless of join ordering strategy) than HANDOPT, it is always much better than HAND-



**Fig. 5.** Experimental results comparing the performance of JQL with different join ordering strategies against the hand-coded implementations. Data for the OneStage, TwoStage and ThreeStage benchmarks are shown at the Top, Middle and Bottom respectively. The charts on the right give a close ups of the three fastest implementations for each benchmark. Different object sizes are plotted to show the general trends.

Benchmark	Size (KB)	# Objects Created	Time (s)	Heap (MB)	Multi- Threaded
_201_compress	17.4	601	6.4	55	N
_202_jess	387.2	$5.3 \times 10^6$	1.6	61	N
_205_raytrace	55.0	$5.0 \times 10^6$	1.6	60	N
_209_db	9.9	$1.6 \times 10^5$	11.5	63	N
_213_javac	548.3	11152	3.9	78	N
_222_mpegaudio	117.4	1084	6.0	26	N
_227_mtrt	56.0	$5.2 \times 10^6$	2.6	64	Y
_228_jack	127.8	$6.9 \times 10^5$	2.3	59	N

**Table 2.** The benchmark suite. Size indicates the amount of bytecode making up the benchmark, excluding harness code and standard libraries. Time and Heap give the execution time and maximum heap usage for one run of the benchmark. # Objects Created gives the total number of objects created by the benchmark during one run.

POOR. We argue then, that the guaranteed performance offered by JQL is very attractive, compared with the range of performance offered by hand-coded implementations — especially as it’s unlikely a programmer will achieve anything close to HANDOPT in practice.

The ThreeStage benchmark is the most complex of those studied and highlights a difference in performance between the MAX\_SELECTIVITY and EXHAUSTIVE join ordering strategies used by JQL. This difference arises because the MAX\_SELECTIVITY heuristic does not obtain an optimal join ordering for this benchmark, while the EXHAUSTIVE strategy does. In general, it seems that the EXHAUSTIVE strategy is the more attractive. Indeed, for queries that have relatively few joins, it is. It is also important to remember that it uses an exponential search algorithm and, hence, for large queries this will certainly require a prohibitive amount of time. In general, we believe that further work investigating other possible join ordering heuristics from the database community would be valuable.

## 4.2 Study 2 — Object Extents Tracking

The purpose of this study was to investigate the performance impact of the JQL object tracking system. This is necessary to permit querying over the object extent sets. However, each object creation causes, at least, both a hashtable lookup and an insertion. Keeping an extra reference of every live object also causes memory overhead. We have used the SPECjvm98 benchmark suite to test the memory and execution overhead incurred.

**Experimental setup.** The benchmarks used for these experiments are described in Table 2. To measure execution time, we averaged time taken by each benchmark with and without the tracker enabled. These timings were taken over 10 runs with a 5-run rampup period. This was sufficient to generate data with

a variation coefficient of  $\leq 0.1$ , again indicating very low variance between runs. For memory overhead measurements, we monitored the resource usage of the JVM process using `top` while running the benchmark and recorded the peak measurement. For these experiments the JVM was run with a 512MB heap.

**Discussion** The relative slowdowns for each benchmark’s execution time are shown in Figure 6. The memory overhead incurred for each benchmark is shown in Figure 7.

Both memory overhead and execution overhead are directly influenced by the number of objects created. Execution overhead is linked to how much time is spent creating objects, rather than operating on them. `mpegaudio` and `compress`, for example, create relatively few objects and spend most of their runtime working with those objects. Hence, they show relatively minor slowdown. `jess` and `raytrace`, on the other hand, create millions of objects in the space of a few seconds. This leads to a fairly significant slowdown.

Memory overhead is also influenced by relative size of objects created. A benchmark like `raytrace` creates hundreds of thousands of small `Point` objects, likely smaller than the `WeakReference` objects necessary to track them. This means the relative overhead of the tracker for each object is very large. `db`, on the other hand, creates a hundred thousand-odd `String` instances, which outweigh the `WeakReferences`. Compared to these bulky `Strings`, the tracker’s overhead is minor.

We consider these benchmarks show the object tracker is likely to be practical for many Java programs. Certainly, using the system in development would offer significant benefits, for example, if class invariants we used properly. We feel performance data for using the object tracker with various ‘real-world’ Java applications would be valuable, and plan to investigate this in the future.

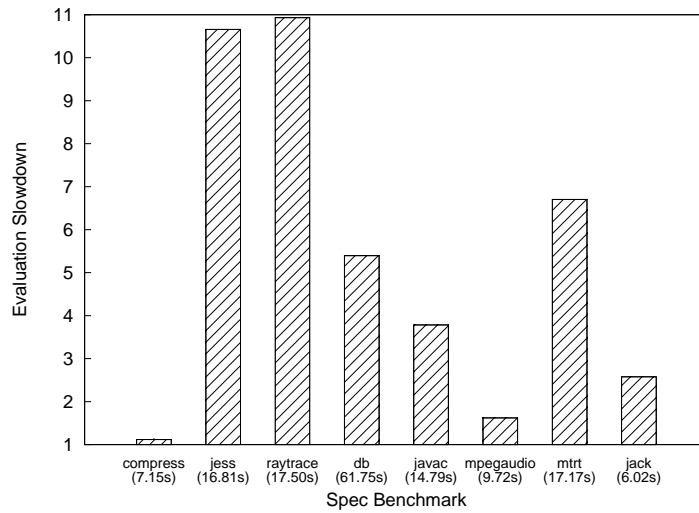
## 5 Discussion

In this section, we discuss various issues with the JQL system and explore interesting future directions.

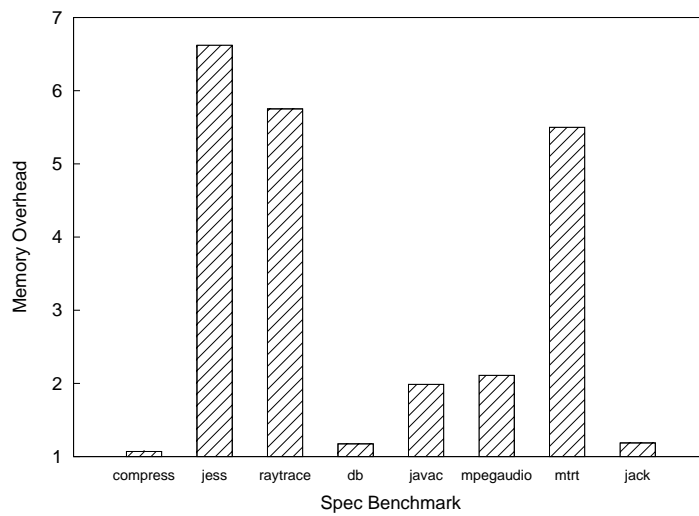
### 5.1 Side Effecting Queries

One of the issues we have not addressed is that of side-effecting queries. That is, queries using a method which mutates the objects it is called upon. While this is possible in principle, we believe it would be difficult to use in a sensible way. This is because the query evaluator is free to optimise queries as it sees fit — there is no evaluation order which can be relied upon. Indeed, there is no guarantee upon what combinations of the domain variable’s objects such a method will be called. For example:

```
r = selectAll(Object x, Object y : x.equals(y) && x.fn(y));
```



**Fig. 6.** Slowdown factors for SPECjvm98 benchmark programs executed with object tracking enabled. The execution time with tracking enabled is shown below each benchmark; the untracked execution times are given in Table 2. Slowdown factors are computed as the division of the tracked time over the untracked time.



**Fig. 7.** Memory overhead factors for SPECjvm98 benchmark programs executed with object tracking enabled. Memory overhead is computed as the division of the tracked memory usage over the untracked usage.

Even for this simple query, it is difficult to predict upon which objects `fn()` will be called. The most likely scenario is that `x.equals(y)` will be the first join in the pipeline — in which case `x.fn(y)` is only called on those `x` and `y` which are `equals()`. However, if the JQL’s sampling strategy for estimating selectivity is used, then `fn()` could be placed before `equals()` in the pipeline. In this case, `x.fn(y)` is called for all possible `x` and `y` combinations.

We make the simplifying assumption that all methods used as part of a query (such as `fn()` above) have no side effects. This applies to the use of `equals()` and `compare()` as well as all other method calls. Since JQL cannot enforce this requirement (at least, not without complex static analysis), it falls upon the programmer to ensure all methods used are in fact side-effect free.

## 5.2 Incrementalisation

An interesting observation is that, to improve performance, the results of a query can be cached and reused later. This approach, often called incrementalisation [7, 24], requires that the cached results are updated in line with changes to the objects and collections they derive from.

Determining whether some change to an object/collection affects a cached query result it is not without difficulty. We could imagine intercepting all field reads/writes (perhaps using AspectJ’s field read/write pointcuts) and checking whether it affects any cached query results. While this might be expensive if there are a large number of cached queries, it could be lucrative if the cache is restricted to a small number of frequently called queries. Alternatively, cached queries could be restricted to those involving only `final` field values. This way, we are guaranteed objects involved in a query cannot change, thereby simplifying the problem.

While incrementalisation can improve performance, its real power lies in an ability to improve program structure by eliminating many uses of collections. Consider the problem of recording which `Students` are enrolled in which `Courses`. In Figure 1, this is achieved by storing a collection of `Students` in each `Course` object. With incrementalisation we can imagine taking another route. Instead of adding a `Student` to a `Course`’s collection, we construct a new `Enroll` object which has fields referencing the `Student` and `Course` in question. To access `Students` enrolled in a particular `Course`, we simply query for all `Enroll` objects with a matching `Course` field. Incrementalisation ensures that, if the query is executed frequently, the cached query result — which is equivalent to the collection in `Course` that was replaced — is maintained. This provides more flexibility than the original design as, for example, it allows us to efficiently traverse the relationship in either direction with ease. With the original design, we are limited to a single direction of traversal, unless we add a collection to `Student` that holds enrolled `Courses`. In fact, this has many similarities with the approach taken to implementing relationships in Rel/J [5].

Incrementalisation is not currently implemented in JQL, although we hope to explore this direction in the future.

## 6 Related Work

An important work in this area is that of Lencevicius *et al.*, who developed a series of *Query-Based Debuggers* [20–23] to address the *cause-effect gap* [12]. The effect of a bug (erroneous output, crash, etc) often occur some time after the statement causing it was executed, making it hard to identify the real culprit. Lencevicius *et al.* observed that typical debuggers provide only limited support for this in the form of breakpoints that trigger when simple invariants are broken. They extended this by allowing queries on the object graph to trigger breakpoints — thereby providing a mechanism for identifying when complex invariants are broken. Their query-based debuggers re-evaluate the query whenever a program event occurs that could change the query’s result. Thus, a breakpoint triggers whenever one or more objects match the query. To reduce the performance impact of this, they employed a number of query optimisations (such as operator selectivity and join ordering).

Several other systems have used querying to aid debugging. The Fox [32, 33] operates on heap dumps of the program generated using Sun’s Heap Analysis Tool (HAT), allowing queries to range over the set of objects involved in a given snapshot. The main focus of this was on checking that certain ownership constraints were being properly maintained by a given program. The Program Trace Query Language (PTQL) permits relational queries over program traces with a specific focus on the relationship between program events [14]. PTQL allows the user to query over relations representing various aspects of program execution, such as the set of all method invocations or object creations. The query evaluator in PTQL supports nested-loop joins (but not hash-joins as we do) and performs join ordering using something very similar to our MAX\_SELECTIVITY heuristic. The Program Query Language (PQL) is a similar system which allows the programmer to express queries capturing erroneous behaviour over the program trace [25]. A key difference from other systems is that static analysis was used in an effort to answer some queries without needing to run the program. As a fallback, queries which could not be resolved statically are compiled into the program’s bytecode and checked at runtime.

Hobatr and Malloy [16, 17] present a query-based debugger for C++ that uses the OpenC++ Meta-Object Protocol [8] and the Object Constraint Language (OCL) [38]. This system consists of a frontend for compiling OCL queries to C++, and a backend that uses OpenC++ to generate the instrumentation code necessary for evaluating the queries. In some sense this is similar to our approach, except that we use JQL to specify queries and AspectJ to add the necessary instrumentation for resolving them. Like the others, their focus is primarily on catching violations of class invariants and pre/post conditions, rather than as a general purpose language extension. Unfortunately, no details are given regarding what (if any) query optimisations are employed and/or how program performance is affected by the added instrumentation code.

More recently, the Language INtegrated Query (LINQ) project has added querying capabilities to the C# language. In many ways, this is similar to JQL and, while LINQ does not support querying over object extent sets, its queries

can be used to directly access databases. At this stage, little is known about the query evaluator employed in LINQ and the query optimisations it performs. We hope that our work will motivate studies of this and it will be interesting to see how the LINQ query evaluator performs in practice. The C $\omega$  language [4] preceded LINQ and they have much in common as regards queries.

One feature of LINQ is the introduction of lambda expressions to C $\sharp$ . Lambda expressions can be used in place of iterators for manipulating / filtering collections [2]. In this way, they offer a form of querying where the lambda expression represents the query expression. However, this is more simplistic than the approach we have taken as, by permitting queries over multiple collections, we can exploit a number of important optimisations. Lambda expressions offer no help in this regard as, to apply such optimisations, we must be able to break apart and manipulate the query expression to find operators that support efficient joins and to determine good join orderings.

Another related work is that of Liu *et al.* [24], who regard all programs as a series of queries and updates. They use static program analysis to determine which queries can be incrementalised to permit efficient evaluation. To do this, they employ a cost model to determine which queries are expensive to compute and, hence, which should be incrementalised. This incrementalisation can be thought of as creating a view which represents the query results and automatically updating when the underlying data is changed. This optimisation could be implemented in JQL (albeit in a dynamic, rather than static setting) and we wish to explore this in the future.

Cook and Rai [10] describe how building queries out of objects (rather than using e.g. SQL strings) can ensure typesafety and prevent spoofing attacks. While these safe query objects have generally been used to generate database queries, they could also act as a front-end to our query system. Similarly, the object extents and query optimisations we describe in this paper could be applied in the context of a safe query object system.

Finally, there are a number of APIs available for Java (e.g. SQLJ, JSQL, etc.) which provide access to SQL databases. These are quite different from the approach we have presented in this work, as they do not support querying over collections and/or object extents. Furthermore, they do not perform any query optimisations, instead relying on the database back end to do this.

## 7 Conclusion

In this paper, we have presented a language extension for Java (called JQL) which permits queries over object collections and extent sets. We have motivated this as a way of improving both program readability and flexibility and also in providing a stronger guarantee of performance. The latter arises from the query evaluator's ability to perform complex optimisations — many of which the programmer is unlikely to do in practice. Through an experimental study we have demonstrated there is a large difference in performance between a poor hand-coded query and a good one. Furthermore, our prototype implementation

performs very well compared with optimised hand-coded implementations of several queries. We have also demonstrated that the cost of tracking all objects in the program is practical. We would expect that, with direct support for querying from the JVM, the performance overhead of this would improve significantly.

The complete source for our prototype implementation is available for download from <http://www.mcs.vuw.ac.nz/~djp/JQL/> and we hope that it will motivate further study of object querying as a first-class language construct.

## Acknowledgements

The authors would like to thank Stuart Marshall for some insightful comments on an earlier draft of this paper. This work is supported by the University Research Fund of Victoria University of Wellington, and the Royal Society of New Zealand Marsden Fund.

## References

1. The Standard Performance Corporation. SPEC JVM98 benchmarks, <http://www.spec.org/osg/jvm98>, 1998.
2. H. G. Baker. Iterators: Signs of weakness in object-oriented languages. *ACM OOPS Messenger*, 4(3), July 1993.
3. M. Barnett, R. DeLine, M. Fahndrich, K. Rustan, M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.
4. G. Bierman, E. Meijer, and W. Schulte. The essence of data access in *cw*. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 287–311. Springer-Verlag, 2005.
5. G. Bierman and A. Wren. First-class relationships in an object-oriented language. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 262–282. Springer-Verlag, 2005.
6. G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.
7. S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 577–589. Morgan Kaufmann Publishers Inc., 1991.
8. S. Chiba. A metaobject protocol for C++. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOP-SLA)*, pages 285–299. ACM Press, 1995.
9. W. R. Cook and A. H. Ibrahim. Programming languages & databases: What’s the problem? Technical report, Department of Computer Sciences, The University of Texas at Austin, 2005.
10. W. R. Cook and S. Rai. Safe query objects: Statically typed objects as remotely executable queries. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 97–106. IEEE Computer Society Press, 2005.
11. D. F. D’Souza and A. C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.

12. M. Eisenstadt. My hairiest bug war stories. *Communications of the ACM*, 40(4):30–37, 1997.
13. E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
14. S. Goldsmith, R. O’Callahan, and A. Aiken. Relational queries over program traces. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 385–402. ACM Press, 2005.
15. P. J. Haas, J. F. Naughton, and A. N. Swami. On the relative cost of sampling for join selectivity estimation. In *Proceedings of the thirteenth ACM symposium on Principles of Database Systems (PODS)*, pages 14–24. ACM Press, 1994.
16. C. Hobatr and B. A. Malloy. The design of an OCL query-based debugger for C++. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pages 658–662. ACM Press, 2001.
17. C. Hobatr and B. A. Malloy. Using OCL-queries for debugging C++. In *Proceedings of the IEEE International Conference on Software Engineering (ICSE)*, pages 839–840. IEEE Computer Society Press, 2001.
18. T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Transactions on Database Systems.*, 9(3):482–502, 1984.
19. R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *Proceedings of the ACM Conference on Very Large Data Bases (VLDB)*, pages 128–137. Morgan Kaufmann Publishers Inc., 1986.
20. R. Lencevicius. *Query-Based Debugging*. PhD thesis, University of California, Santa Barbara, 1999. TR-1999-27.
21. R. Lencevicius. On-the-fly query-based debugging with examples. In *Proceedings of the Workshop on Automated and Algorithmic Debugging (AADEBUG)*, 2000.
22. R. Lencevicius, U. Hölzle, and A. K. Singh. Query-based debugging of object-oriented programs. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 304–317. ACM Press, 1997.
23. R. Lencevicius, U. Hölzle, and A. K. Singh. Dynamic query-based debugging. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1628 of *Lecture Notes in Computer Science*, pages 135–160. Springer-Verlag, 1999.
24. Y. A. Liu, S. D. Stoller, M. Gorbovitski, T. Rothamel, and Y. E. Liu. Incrementalization across object abstraction. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 473–486. ACM Press, 2005.
25. M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 365–383. ACM Press, 2005.
26. P. Mishra and M. H. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1):63–113, 1992.
27. J. Noble. Basic relationship patterns. In N. Harrison, B. Foote, and H. Rohnert, editors, *Pattern Languages of Program Design 4*, chapter 6, pages 73–94. Addison-Wesley, 2000.
28. J. Noble and J. Grundy. Explicit relationships in object-oriented development. In *Proceedings of the conference on Technology of Object-Oriented Languages and Systems (TOOLS)*. Prentice-Hall, 1995.

29. D. J. Pearce and J. Noble. Relationship aspects. In *Proceedings of the ACM conference on Aspect-Oriented Software Development (AOSD)*, pages 75–86. ACM Press, 2005.
30. C. Pierik, D. Clarke, and F. de Boer. Creational invariants. In *Proceedings of the Workshop on Formal Techniques for Java-like Programs (FTfJP)*, pages 78–85, 2004.
31. R. Pooley and P. Stevens. *Using UML: Software Engineering with Objects and Components*. Addison-Wesley, 1999.
32. A. Potanin, J. Noble, and R. Biddle. Checking ownership and confinement. *Concurrency and Computation: Practice and Experience*, 16(7):671–687, 2004.
33. A. Potanin, J. Noble, and R. Biddle. Snapshot query-based debugging. In *Proceedings of the IEEE Australian Software Engineering Conference (ASWEC)*, pages 251–261. IEEE Computer Society Press, 2004.
34. K. Rustan, M. Leino, and P. Müller. Object invariants in dynamic contexts. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004.
35. K. Rustan, M. Leino, and P. Müller. Modular verification of static class invariants. In *Proceedings of the Formal Methods Conference (FM)*, volume 3582 of *Lecture Notes in Computer Science*, pages 26–42, 2005.
36. M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal*, 6(3):191–208, 1997.
37. A. N. Swami and B. R. Iyer. A polynomial time algorithm for optimizing join queries. In *Proceedings of the International Conference on Data Engineering*, pages 345–354, Washington, DC, USA, 1993. IEEE Computer Society.
38. J. Warmer and A. Kleppe. *The Object Constraint Language: precise modeling with UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.