Caching and Incrementalisation in the Java Query Language

Darren Willis, David J. Pearce and James Noble
Computer Science, Victoria University of Wellington, NZ
{darren,djp,kjx}@mcs.vuw.ac.nz

Abstract

Many contemporary object-oriented programming languages support first-class queries or comprehensions. These language extensions make it easier for programmers to write queries, but are generally implemented no more efficiently than the code using collections, iterators, and loops that they replace. Crucially, whenever a query is re-executed, it is recomputed from scratch. We describe a general approach to optimising queries over mutable objects: query results are cached, and those caches are incrementally maintained whenever the collections and objects underlying those queries are updated. We hope that the performance benefits of our optimisations may encourage more general adoption of first-class queries by object-oriented programmers.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.1.5 [*Programming Techniques*]: Objected-Oriented Programming

General Terms Languages, Performance, Design

Keywords Querying, Incrementalization, Java

1. Introduction

First-class constructs for querying lists, sets, collections, and databases are making their way from research systems into practical object-oriented languages. Language-level queries have their roots in the *set comprehensions* and *list comprehensions* originating in the 1960s in SETL [36], and named in the 1970s in NPL [5]. Meanwhile, object-oriented languages have made do with external iterators (in Java/C++) and internal iterators (Smalltalk) to query collection libraries [7]. More recent languages such as Python, LINQ for C♯ [28] and Cω [2] have included queries as first-class language constructs.

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.

Explicit, first-class query constructs have several advantages over expressing queries implicitly using collection APIs and external/internal iterators. Explicit queries can be more compact and readable than open-coding queries using other APIs. Queries (especially in LINQ) can be made polymorphic across different collection implementations, using the same query syntax for XML and relational data as well as objects. By making programmers' intentions to query collections explicit in their code, first-class query constructs enable numerous automatic optimisations, often drawing on database techniques [41].

The problem addressed in this paper is that existing language-based query constructs don't really take mutable state into account. List and set comprehensions were introduced in functional languages, where neither the collections being queried nor the data contained in those collections were mutable; thus, the questions of how queries should best perform when their underlying data was updated does not arise. Relational databases support updates to their underlying tables, but don't support object identity and generally assume relatively few updates to small parts of large, external file structures. In contrast, collections of objects are routinely mutated in object-oriented programming languages: objects are added to and removed from collections between queries as part of the general run of a program. Objects themselves are also mutable: an object's fields can be assigned to new values by any code with permission to write to those fields. Changing the collections underlying a query, or altering the objects within those collections can certainly change the set of objects that result from the query. This means that exactly the same query code, executed at two different instants during a program's run, may produce very different results.

In this paper, we present a general technique for caching and incrementalising queries in object-oriented languages in the presence of mutable state. In previous work [41] we have presented JQL, the Java Query Language, which provides explicit queries for Java and uses a range of techniques — primarily join ordering — to optimise queries. Unfortunately, the previous version of JQL has no memory for the queries it calculates: every query is re-evaluated from scratch. In this paper we detail an extension to JQL that optimises results across multiple queries in two important

ways. First, we cache query results: if the same query is resubmitted, the cached result is reused. Second, we incrementally update these caches to take account of changes in objects and collections between queries. This is crucial because Java is an imperative, object-oriented language: the objects and collections used in queries can be updated by the rest of the program in between query executions. Incremental updating means that JQL can rely on its caches, and can avoid re-executing queries from scratch, even when their underlying objects have been updated. Of course, such caching costs time (the incremental updates) and space (to store): so our extended caching JQL includes mechanisms to decide when and where to cache queries.

This paper makes the following contributions:

- We present a general approach to optimising repeated first-class queries in object-oriented programs, by caching results and updating the caches as the programs run.
- We detail an implementation of our approach for the Java Query Language, using AspectJ to maintain and incrementalise cached query results.
- We present experimental data demonstrating that our implementation performs efficiently.
- We describe an inspection study of the use of loops in programs which suggests that many loops could be converted to queries and would then benefit from our approach.

While our design and implementation are based on the Java Query Language (JQL), we expect our techniques will apply to queries and set comprehensions in any programming language.

2. Queries, Caching and Incrementalisation

To illustrate our approach, we present an example based on a real-world application called *Robocode* [31]. This simple Java game pits user-created, simulated robots against each other in a 2D arena. The game has a serious side as it has been used to develop and teach ideas from Artificial Intelligence [13, 16, 32]. A Robocode Battle object maintains a private list of robots, with an accessor method that returns all robots in the battle:

```
class Battle {
  private List<Robot> robots;
  ...
  public List<Robot> getRobots() {
    return robots;
  }
}
```

During each turn of the game, robots scan their field-of-view within the battle arena to locate other Robots which they can attack:

```
class Robot {
  public int state = STATE_ACTIVE;
  public boolean isDead() {return state == STATE_DEAD;}
  public void die() { state = STATE_DEAD; }
  ...
  private void scan() {  // Scan field-of-view to find robots
    for(Robot r : battle.getRobots()) {
      if(r!=null && r!=this && !r.isDead() && r.intersects(...)){
        ....
}}}}
```

In JQL (and similarly for other languages with first-class queries, such as LINQ), we would rewrite the above into a for-loop query:

```
for(Robot r : battle.getRobots() | r!=null && r!=this &&
      !r.isDead() && r.intersects(...)) {
  ...
}
```

This extended for-loop statement executes its body for each element matching the given conditions (i.e. those after the '|') in the collection returned by battle.getRobots(). This sub-collection is called the query's *result set*. No order of iteration is implied by the for-loop query; furthermore, while side-effecting statements can (currently) be used within query conditions, their use has undefined behaviour. In particular, the programmer is responsible for ensuring method calls used inside query conditions are free from side-effects.

The advantage of the for-loop query is that the conditions are made explicit and, hence, it is more declarative in nature. Our previous work with JQL demonstrated how database optimisations can be applied to such queries to give greatly improved performance, especially for those involving multiple collections [41]. Writing a JQL or LINQ query, or a set or list comprehension, is also generally simpler than writing equivalent code using explicit loops and branches — this is why languages are now adopting constructs to support queries and comprehensions directly.

The existing JQL optimisation techniques detailed in [41] only go so far with methods like scan(), however. The main problem with scan() is that it is called in the inner loop of Robocode, run once for every Robot at every simulation step. While we can optimise each individual query, existing language-based query systems like JQL or LINQ treat every query execution as a separate event: they do not use the previous execution of a query to assist in subsequent executions of the query.

2.1 Query Caching

To optimise programs like Robocode, programmers focus on methods like scan() that are called repeatedly. A common and effective approach is to cache intermediate results which, in this case, are the sub-collection(s) being frequently traversed. For example, the programmer might know that, on average, there are a large number of dead robots. To avoid repetitively and needlessly iterating over many dead robots

in scan(), the programmer might maintain a cache — a list of just the "alive" robots — as follows:

```
class Battle {
  private List<Robot> robots;        // master list of all robots
  private List<Robot> aliveRobots; // cached list of alive robots
  ...
  public List<Robot> getRobots() { return robots; }
  public List<Robot> getAliveRobots() {
    return aliveRobots;
}}
```

Then, each robot can scan the list of alive robots, without needing to check whether each is alive or dead:

```
class Robot {
  ...
  private void scan() {
    for(Robot r : battle.getAliveRobots()) {
      if(r!=null && r!=this && r.intersects(...)) { ... }
}}}
```

Here, aliveRobots is a sub-collection of robots containing only those where !isDead() holds. Thus, the for-loop in scan() no longer needlessly iterates over dead robots. Since (after the game has been running a while) more robots are typically dead than alive, this reduces the time taken for the loop at the cost of extra memory (the cache).

Explicitly maintaining extra collections has several drawbacks. Caches can be difficult to introduce when the interface of the providing object (i.e. Battle) is fixed (e.g. it's part of a third-party library, and/or the source is not available, etc). Furthermore, the optimisation reduces readability and maintainability as the source becomes more cluttered. Maintaining cached collections is also rather tedious, since they need to be updated whenever the underlying collection or the objects in those collections are updated — whenever a new robot "spawns" into the game, or whenever an alive robot dies. Finally, code to maintain these optimised collections must be written anew for each collection. For example, Robocode's Battle class also maintains a list of Bullets and employs a loop similar to scan() for collision detection. Programmers can introduce a sub-collection to cache live bullets, but only by duplicating much of the code necessary for the sub-collection of live robots.

To address these issues, we have developed an extension to JQL that automatically caches the result set of a query *when this is beneficial*. Specifically, our system caches a query's result set so it can be quickly recalled when that query is evaluated again. Our system employs heuristics to decide when it is beneficial to cache a query's result set. Thus, the advantages of caching important sub-collections are obtained without the disadvantages of performing such optimisations by hand. The pragmatic effect is that programmers can write code using queries directly over underlying collections, but the program performs as if specialised sub-collections were hand-coded to store just the information required for particular queries.

2.2 Cache Incrementalisation

When the source collection(s) of a query are updated (e.g. by adding or removing elements), or an element of a source collection is itself updated, any cached result sets may become invalidated. Traditionally, encapsulation is used to prevent this situation from arising, by requiring all updates are made via a controlled interface. Thus, updates to a collection can be intercepted to ensure any cached result sets are updated appropriately. To illustrate, consider a simple addRobot() method for adding a new robot to the arena, where a cache is being maintained explicitly:

```
class Battle {
  private List<Robot> robots, aliveRobots;
  ...
  public List<Robot> getRobots() { return robots; }
  public List<Robot> getAliveRobots() {
    return aliveRobots;
  }

  public void addRobot(Robot r) {
    robots.add(r);
    if(!r.isDead()) { aliveRobots.add(r); }
  }

  public void robotDied(Robot r) {
    aliveRobots.remove(r);
}}
```

Here we see that when a robot is added via addRobot(), the aliveRobots list is *incrementally updated* to ensure it remains consistent with the robots collection. Likewise, when a robot dies, robotDied() is called to ensure aliveRobots is updated accordingly.

To deal with object updates that may invalidate a cached result set for some query, our system incrementally updates that result set. To do this, we intercept operations which may alter the result set of a given query — that is, where evaluating the same query again, with the same input collections, would yield different results. There are essentially two kinds of operations in programs we must consider: those that add or remove objects to the underlying collections (like the addRobot() method) or those that change the state of objects in those collections (e.g. the die() method on class Robot, or any other assignments to the dead field).

An important issue in this respect is the *query/update ratio* for a particular query. This arises because there is a cost associated with incrementally maintaining a cached result set: when the number of updates affecting a result set is high, compared with how often it is actually used, it becomes uneconomical to cache that result set. To deal with this, our system monitors the query/update ratio and dynamically determines when to begin, and when to stop caching the result set. Queries that are not repeated, or that occur infrequently, are not cached; neither are queries where the underlying data changes often between queries. Where there is only a relatively small change to the data between

each query over that data, then our system can effectively cache and optimise those queries.

2.3 Discussion

We believe our approach frees the programmer from tedious and repetitive optimisations; from the burden of working around fixed interfaces; from the possibility of bugs caused by out-of-synch result sets (e.g. if someone forgets to call robotDied() above); and, finally, that it opens up the door for more sophisticated optimisation strategies. For example, we could employ a *cache replacement policy* that saves memory by discarding infrequently used result sets.

One might argue, however, that our approach will further limit the ability of programmers to make important "performance tweaks". Such arguments have been made before about similar "losses of control" (e.g. assembly versus high-level languages, garbage collection, etc); and yet, in the long run, these have proven successful. Furthermore, JQL must retain the original looping constructs of Java and, hence, the programmer may retake control if absolutely necessary, by writing loops and filters explicitly.

The central tenet of this paper then, is that we can provide an extremely flexible interface (first-class queries), whilst at the same time yielding the performance (or close to) of hand-coded implementations. The key to this is a mechanism for caching and incrementalising our queries. We choose JQL as a test-bed for exploring this, but the ideas should apply more generally. For example, Python's list comprehensions, or C♯'s LINQ would be excellent candidates for our approach.

3. Incrementalised Caching for JQL

The *Java Query Language (JQL)* is a prototype extension to Java which introduces first-class queries [41]. Querying is provided through the extended for-loop syntax (as shown in previous sections), and also the list-comprehension syntax. For example, the following query uses list comprehension syntax to *select* elements from two collections:

```
List<StringBuffer> words= new ArrayList<StringBuffer>();
List<Integer> lengths= new ArrayList<Integer>();
words.add(new StringBuffer("Hello"));
words.add(new StringBuffer("Worlds"));
words.add(new StringBuffer("blah"));
lengths.add(4); lengths.add(5);

List<Object[]> r = [StringBuffer x:words, Integer y:lengths |
  x.length() == y && x.length() > 1]
```

This returns all pairs of words and lengths matching the condition[1]. So, for the above example, the query returns {["Hello",5], ["Blah",4]}.

JQL employs optimisations from the database literature which can significantly improve upon the obvious nested-loop implementation of a query. To process a query, the JQL evaluator pushes tuples through a staged pipeline. Each stage, known as *join* in the language of databases, corresponds to a condition in the query. Figure 1 shows the query pipeline for the above query. The ordering of joins in the pipeline, as well as the join type, can greatly affect performance. JQL uses heuristics to select good join orders, and supports several join types, including *hash-join*, *sort-join* and *nested-loop join*. In fact, many more strategies have been proposed which could further improve performance (e.g. [29, 38, 12, 37]).

[1] The JQL notation has changed since our earlier work [41] to bring it closer to Haskell and Python.

3.1 Overview

We have extended JQL with an incrementalised caching system. The cache stores evaluation statistics for each query evaluated in the program, and uses a caching policy to determine when to cache queries. The cache is incrementally updated to ensure results stored in the cache are consistent with the program state.

The incrementalised caching scheme is controlled by the *cache manager*, which keeps statistics for all queries, and determines which queries to cache. Whenever a cached query is evaluated, the cache manager intercepts the evaluation call and supplies its cached results back. When non-cached queries are evaluated, the cache manager may decide to begin caching that query, in which case it builds a cache from its results. The decision of when to cache a query is determined by a programmer-specified *caching policy*.

The cache manager needs to match submitted queries with their cached results. Static program position, for example, is insufficient. For example, consider:

```
List<Student> xs = [ Student s : students | s == jim ];
```

The result set of this query depends upon the actual values of students and jim at the time of execution. Cached results generated from a prior evaluation of this query can only be reused in an evaluation *where the values for these variables match*. Therefore, the cache manager maintains a *cache map* from *concrete queries* (i.e. those whose variables are substituted for their actual values) to result sets. Whenever a query is submitted, the cache manager searches this map to check for any matching results. If results are available they are returned as the result of the query. Here, incremental updating ensures they are correct even if the underlying objects or collections have changed. If the results aren't available, the query is executed and the results returned. During this, the cache manager consults the caching policy to determine whether to cache these results. If so, the results are entered into the cache map and scheduled for incremental updating.

3.2 Incremental Cache Maintenance

Once cached results are in place, it is essential that they accurately reflect the program state. That is, they must be identical to those that would be returned by a non-cached evaluation of the query. Updates to heap objects may render cached results inconsistent, and we must incrementally update them

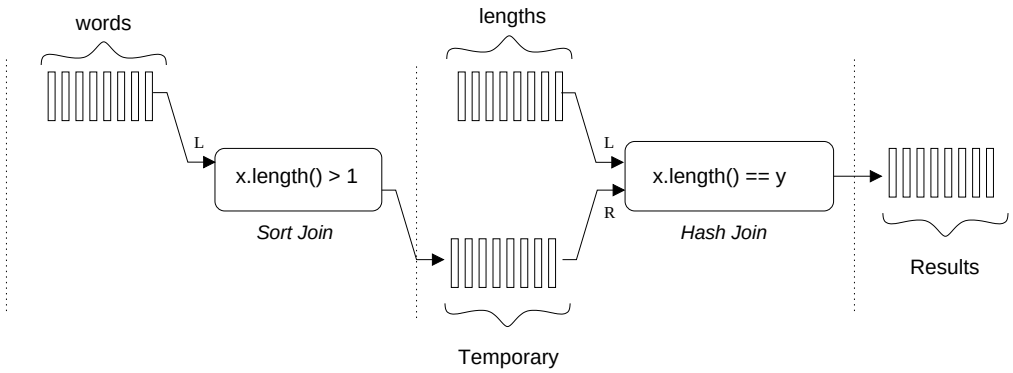

Figure 1. A query pipeline

by adding/removing tuples as necessary. To capture this, we introduce the concept of *query dependencies*. The intuition is that a (concrete) query is dependent upon some object *if changes to the state of that object can affect the results of the query*. For example, consider:

```
xs = [ Student s : students | s.status().code == ENROLLED ];
```

Here, the collection object students, the Student objects it contains and, for each, the objects returned by status() are all dependencies of this query. This is because adding/removing a student from students, or changing the status code of a student in students can affect its results.

When an object in the system changes state, we must determine which (if any) cached queries depend upon it. For each, we check whether any tuples must be added/removed from its result set. For additions/removals from source collections, this is easy enough, whilst for other updates it's more challenging.

Object Addition/Removal. The results returned from a query may change if an object is added to one of its source collections. While this may increase the size of the result set, it cannot decrease it because the evaluation of each tuple is independent of others.

Updating a cached result set after an object addition requires generating the new tuples (if any), and adding them to the result set. Generating these tuples is simple, and uses the existing query pipeline. The cache manager evaluates a simplified version of the query, with the new object substituted for its query variable. This requires time proportional to the product of the sizes of the source collections, excluding that which was updated. For example, in Figure 1, adding a new word to words means first checking it's length is > 1, and then comparing this against all elements of lengths.

The strategy for dealing with object removal from a source collection is similar, but not identical — the result set may decrease in size, but cannot increase. To remove tuples which are now invalid, we search the cached result set deleting any involving the removed object.

Object Updates. Cache consistency can also be affected when an object changes state, since a query may now fail a condition it previously passed, and vice versa. The strategy is, again, to run affected objects through a simplified query pipeline to avoid re-evaluating entirely from scratch. The issue is that we must determine which objects in the source collections are actually affected.

When the updated object is actually a member of one or more source collections, the process is similar to before: we first remove all tuples from the result set involving the object, and then evaluate a simplified query pipeline with it substituted for the relevant query variable(s). Any tuples produced are then added back to the result set. For example, updating the length of a word from words in Figure 1 can potentially invalidate any resulting tuple involving that word; at the same time, it can also result in more tuples being added to the results. Our strategy ensures the right tuples are present at the end by essentially "starting over" with respect to this object.

When the updated object is only indirectly accessible through a member of a source collection(s), things are more complex. We must decide which objects in the source collections reach the updated object, and then "start over" with respect to them using a simplified query pipeline. Tracking the necessary dependency information to determine this set is non-trivial and, in Section 4.1, we detail how our prototype implementation conservatively approximates this.

3.3 Caching Policy

Cached results need incremental updating whenever operations affecting them occur. Such operations incur non-trivial overheads when they cause cached results to be incrementally updated. The more frequently a cached result set is incrementally updated, the greater the cost of keeping it. Likewise, the more frequently its originating query is evaluated, the greater the *benefit* from keeping it. However, queries whose result sets aren't cached don't incur these costs. Hence, re-evaluating a query from scratch each time may be cheaper when the query/update ratio is low.

Caching policy dictates when a query warrants caching, and when it doesn't. An intelligent caching policy is critical to obtaining the best performance possible. In the absence of updates which can invalidate the cache, there's no overhead from incrementalisation, and so caching is always worthwhile[2]. In the presence of updates, however, we must balance the cost of incrementalising cached results versus the benefit from keeping them.

Obtaining a perfect caching policy is impossible since it requires future knowledge of the query/update ratio. Therefore, any solution must employ heuristics to determine when it is beneficial to cache results. We have currently two simple policies as part of our incremental caching scheme. These are:

- **Always On**. This policy begins caching a query on the first evaluation and never stops (unless the cached result set is forceably evicted due to memory constraints).
- **Query/Update Ratio**. This policy records how often a query is evaluated and also the number of update operations that could affect the query. Caching the result set of a query begins when the query/update ratio reaches a certain threshold (we use a default of 0.25 for this) and ceases when it drops below.

A subtle aspect of the Query/Update Ratio policy is that we must maintain execution information on queries *even when their result sets aren't being cached*. Otherwise, we cannot tell when the query/update ratio for a query crosses our threshold and becomes beneficial to cache. Therefore, we maintain a record for each concrete query encountered. This incurs additional overhead, especially as the number of concrete queries is, in principle, unbounded. To deal with this, the cache manager must garbage collect records for queries which become inactive.

4. Evaluation

Query caching can yield a large performance benefit in programs. However, caching introduces overhead — the cache must be built and kept consistent with changes in program state. We have developed a prototype implementation of our incrementalised caching system for JQL and we now discuss this, and present the results of two experiments investigating performance: the first examines the performance of Robocode with and without incrementalised caching; the second investigates the trade-off of incrementally maintaining a cached result set, versus always recomputing it from scratch. Finally, we report on a study into the number of loops which can be turned into cachable queries across a small corpus of Java applications.

[2] Subject to the constraints of finite memory resources. For example, if caching a query exhausts physical memory and causes paging, performance may be adversely affected. In principle, a predetermined cache size — much like the JVM heap limit — could be used to control the cache size.

In all experiments which follow, the experimental machine was an Intel Pentium IV 3.2GHz, with 1.5GB RAM running NetBSD v3.99.11, Sun's Java 1.5.0 Runtime Environment and Aspect/J version 1.5.4. Timing was performed using the standard System.currentTimeMillis() method, which provides millisecond resolution (on NetBSD). The source code for JQL and the incrementalised caching prototype can be obtained from `http://www.mcs.vuw.ac.nz/~djp/JQL/`.

4.1 Prototype Implementation

We have developed a prototype implementation of our incrementalised caching system for JQL built using Aspect/J. While this supports the main features of our system, it is not optimal for several reasons. In particular, the use of Aspect/J imposes large memory overheads, whilst its ability to track dependency information is somewhat limited (indeed, to do this effectively may require VM support). The choice to use AspectJ here is purely for convenience, since it eliminates the need to write a custom bytecode manipulation mechanism; however, replacing it with such a mechanism would likely yield performance improvements. Nevertheless, we believe the results obtained using our prototype demonstrate the potential that incrementalised query caching can offer. We now discuss its salient features:

Cache Map. For efficiency, the cache manager implements the cache map (recall this maps concrete queries to their results) using a hash map. For this to work, a hash code must be computed for a concrete query which is not affected by state changes in objects referred to by query variables. For example, in an evaluation of the following query:

```
List<Student> xs = [ Student s : students | s == jim ];
```

the variables students and jim refer to particular objects. If the query is re-evaluated with the same values for those variables, it should map to the same result set. Furthermore, if, in the meantime, say the object referred to by jim is updated, the (concrete) query must still map to the same result set (which must be incrementally updated to ensure consistency).

Source Collections. To determine when an addition/removal from a source collection occurs, we use AspectJ to instrument collection operations like Collection.add(Object). When such operations occur, the cache manager first checks to ensure the addition/removal proceeded correctly (i.e., that the call returned **true**) and, if so, updates the cache using a simplified query pipeline. We assume user-supplied collections adhere to the Collection interface. For example, a class implementing Collection could break our incrementalisation with an add() method that returned **true**, but actually did nothing. Implementations that adhere to the Collection interface, such as the standard Collection implementations, will function correctly.

Dependency Tracking. To determine when an object update occurs, we use Aspect/J to intercept field writes. Ideally we would only intercept assignments to fields of objects upon which cached queries depend. In practice, we must intercept all field writes and then check whether any cached queries depend upon them. Since this would add a prohibitive overhead, we require the programmer to annotate fields with @Cachable to indicate which fields to monitor. For safety, queries involving fields which are not annotated @Cachable cannot be cached, as we can't guarantee cache consistency. If a programmer knows a particular field is used in a time-sensitive query, we assume they will annotate it accordingly.

Determining whether an updated object may invalidate a cached query's result set requires knowing the dependencies of that query. This is non-trivial and necessitates maintaining a dependency tree for that query. For simplicity, we only consider updates to objects held directly in source collections. Thus, for safety, queries involving multi-level indirections cannot be cached.

The use of method calls in a query represents a similar problem in determining when an object update may invalidate cached results. For example, if a get() method call is used to access a field within a query, rather than a direct access, the cache manager needs to know which field updates may invalidate the cache. Rather than developing a full implementation for this, we instead support just the most common case — namely, accessors. We again require the programmer to annotate accessors with @Cachable and, within the annotation, to list those fields which are read. This ensures our system knows which fields to monitor. The accessor cannot, however, further dereference fields of its object (since this results in the multi-level indirection issue from before) and cannot call other methods which have side-effects or depend on fields not listed. For safety, our system will not cache queries containing other kinds of method call. A program analysis could be used to statically determine such dependencies for our accessors; likewise, full dependency tracking would allow general method calls to be used (although we would still require a notion of purity to prevent side-effects). While we are investigating static and dynamic techniques for resolving these issues, the contribution of this paper is in the design and evaluation of the incrementalised caching system, not object-update tracking.

4.2 Study 1: Robocode

This study was an investigation into the benefit obtainable from incrementalised query caching on a real-world benchmark, namely Robocode [31]. Here, software "robots" fight each other in an arena. The robots can move around, swivel their turret to adjust their field-of-view and fire bullets. There are several tunable parameters to a game, including the number of robots and size of the arena.

We profiled every loop in the Robocode application and found only six were heavily executed. Of these, four could be easily converted into JQL extended for-loop queries. They all had the following form:

```
for(Robot r : battle.getRobots()) {
  if(r!=null && r!=this && !r.isDead() && r.intersects(...)) {
    ....
}}
```

We translated the four loops into an extended for-loop, such as:

```
for(Robot r : battle.getRobots() | r!=null && !r.isDead()) {
  if(r!=this && r.intersects(...)) {
    ....
}}
```

An important observation here is that we explicitly chose not to include the condition r!=**this** in the query. To understand why, it's important to consider that each robot executes this query during a turn of the game. Thus, including r!=**this** means its result set differs by exactly one element for each robot, with **this** robot omitted in each case. This causes many near identical result sets to be cached, with each requiring an incremental update for each change to the robots collection. In contrast, omitting the r!=**this** comparison ensures the result set is the same for each robot and, hence, that only one cached result set is required, substantially lowering the cost.

Experimental Setup. For this experiment, we measured the time for a one round game to complete, whilst varying the number of robots and the arena size. A single ramp up run was used, followed by five proper runs from which the average time was taken. These parameters were sufficient to generate data with a variant coefficient ≤ 0.2 indicating low variance between runs.

Discussion. Figure 2 presents the results of the Robocode experiments. In the figure, "no caching" corresponds to the base JQL system with caching disabled; observe that, in this situation, *the Aspect/J weaver is not used to weave any classes* — weaving only occurs when caching is enabled.

The main observation from Figure 2 is that, on the two larger arenas, the effectiveness of using incrementalised query caching becomes apparent as the number of robots increases. In the largest arena (size 4096x4096), with a large number of robots, we find a speedup of around one third. This is quite a significant result, especially as this speedup includes the cost of detecting changes, incrementally updating caches, and the underlying AspectJ dynamic weaver. The reason for this is that the length of the game and, most importantly, the amount of time a robot will be in the dead state increases with the number of robots. Thus, the caching scheme becomes effective when there are more robots, since there will be many dead robots that it avoids scanning, unlike the uncached implementation.

For a small arena (size 1024x1024) or few robots, little advantage is seen from incrementalised query caching. The reason for this is that, in a smaller arena containing the same number of robots, those robots will be pushed more closely

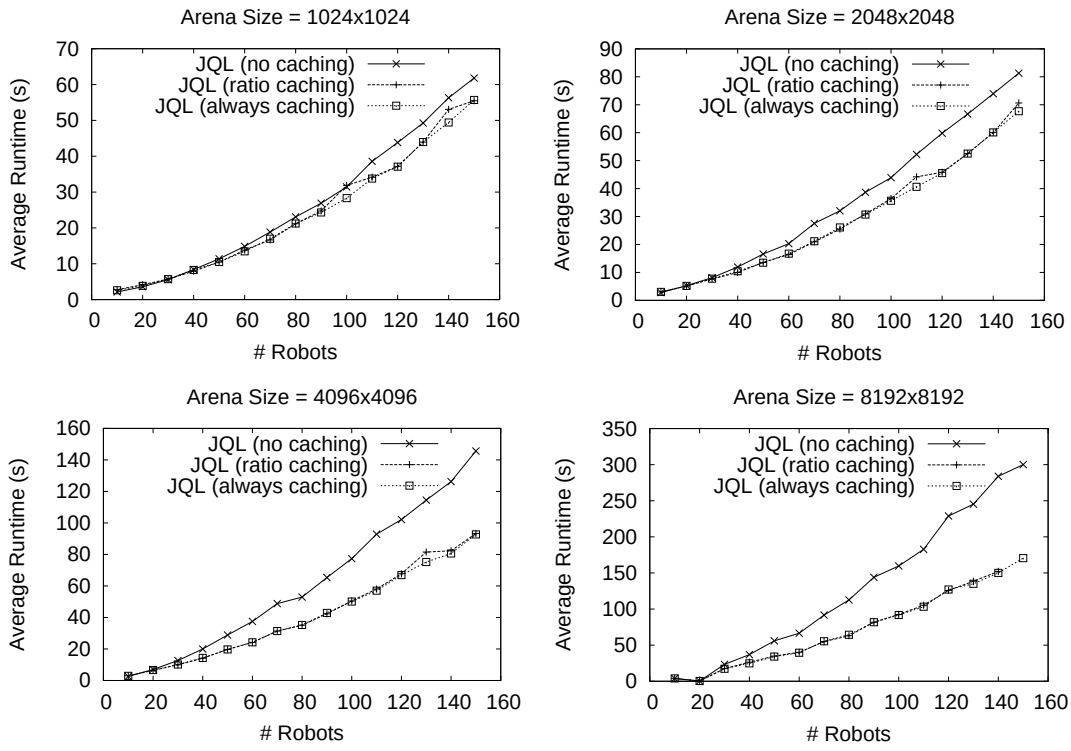

Figure 2. Experimental results comparing the performance of the Robocode application with and without our incrementalised caching scheme.

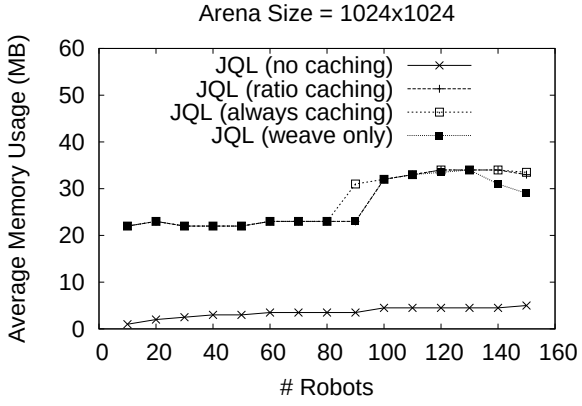

Figure 3. Experimental results evaluating the memory consumption of the Robocode application with and without our incrementalised caching scheme. Observe that "weave only" is used to identify the memory usage caused by the AspectJ load-time weaver. In this situation no caching is taking place, but the AspectJ load-time weaver is still weaving classes as though it were. This contrasts with "no caching", where the the AspectJ load-time weaver is not weaving any classes.

together. Thus, they destroy each other at a much higher rate, which lowers the query/update ratio and decreases the length of the game. We can see that, when there are few robots and the arena size is small, the non-caching implementation slightly wins out overall. We would expect that, if the arena size decreased further, this difference would become more pronounced at the lower end. Likewise, we would expect that, as the arena size increased further, the *advantages* of incrementalised query caching would become more pronounced.

Another observation from Figure 2 is that using the *always-caching* heuristic is never really worse than with caching disabled. While this may seem surprising, it simply reflects the fact that, for this application (i.e. Robocode), the query/update ratio never falls low enough for caching to be detrimental.

Finally, we investigated the memory consumption of our incrementalised caching scheme. The results are reported in Figure 3. From this we can see that, while the memory overheads are high, this is almost entirely caused by the AspectJ load-time weaver (see the points marked "weave only"). We also confirmed this independently using the HProf profiler [39]. We would expect these overheads could be significantly improved using a bytecode weaver customised for the JQL system. Indeed, we would not expect large memory overheads for caching in this application since there are only four result sets (arising from the four queries) being cached, each with no more than 150 robots. Finally, while not shown, the overheads remain very much the same irrespective of Arena size.

Name	Details
One Source	[Attends a:attends | a.course == COMP101] This query has a single pipeline stage, and a single domain variable (COMP101 is a constant value for a Course object). Removing, inserting or updating an object in attends requires the system to simply check the affected object's course field against COMP101.
Two Sources	[Attends a:attends, Student s:students | a.course == COMP101 && a.student == s] This query requires two pipeline stages and has two domain variables. Removing, inserting or updating an object in attends requires the system to check the affected object's course against COMP101, and to compare its student field against all student objects in students.
Three Sources	[Attends a:attends, Student s:students, Course c:courses | a.course == COMP101 && a.student == s && a.course = c] This query requires three pipeline stages, and has three domain variables. Removing, inserting or updating an object in attends requires the system to check the affected object's course against COMP101, to compare its student field against all student objects in students, and then to compare those matching with all courses.

Table 1. Details of the three benchmark queries

4.3 Study 2: Cache Incrementalisation

Caching query result sets can greatly improve performance. Once a query's result set has been cached, it must then be incrementally updated when changes occur which may affect the cached results. Therefore, we have conducted experiments exploring the trade-off of querying performance against the overhead of incremental updates and we now report on these.

There are two ways that query results can be altered between evaluations: either objects can be added to or removed from one of the source Collections for a query; or, the value of an object field used in the query can be changed. In either case, the process for dealing with the change is the same: the affected object is passed through the existing query pipeline to ascertain whether it should be added or removed from the cached result set. We have constructed a benchmark which varies the ratio of query evaluations to updates to explore the trade-offs involved. The benchmark constructs three initial collections containing n Students, n Courses and n Attends objects respectively. The benchmark performs 5000 operations consisting of either a JQL query or the random addition and removal of an Attends object from its collection (which is a source for the JQL query). The size of the Attends collection is kept constant as this would otherwise interfere with the experiment, since the cost of a query evaluation depends upon the size of the Attends collection. Finally, we considered (in isolation) three queries of differing length to explore the effect of query size on performance. The three queries are detailed in Table 1.

Experimental Setup. For each query in Table 1, we measured the time to perform 5000 operations whilst varying the ratio of evaluations to updates (in steps of 0.02 between $0 \ldots 0.2$, and 0.1 between $0.2 \ldots 1$). As discussed above, each operation was either: a random addition and random removal from the Attends collection; or an evaluation of the query being considered. This was repeated 50 times with the average being taken. These parameters were sufficient to generate data with a variation coefficient of ≤ 0.15 — indicating low variance between runs.

Discussion. Figure 4 presents the data for each of the queries in Table 1. There are several observations which can be made from these plots. Firstly, incrementalised caching is not optimal when the ratio of queries to updates is low. This reflects the fact that, when the number of evaluations is low, the pay off from caching a result set is small compared with the cost of maintaining it. As the ratio increases, however, the advantages of caching quickly become apparent. For the *ratio* caching policy, we can clearly see the point at which it begins to cache the query result sets. The plots highlight both the advantages and disadvantages of this heuristic: when the query/update ratio is low, it can outperform the *always-on* policy by not caching result sets; unfortunately, however, the point at which it decides to cache result sets is fixed and, hence, it does not necessarily obtain the best performance on offer.

Another interesting observation from the plots is that the complexity of the query affects the point at which it becomes favourable to cache result sets. This is because the cost of an update operation is directly affected by the number of source collections in the query. If there is just one source collection, then each update corresponds to simply passing the affected object through the pipeline; however, if there is more than one source collection then, roughly speaking, we

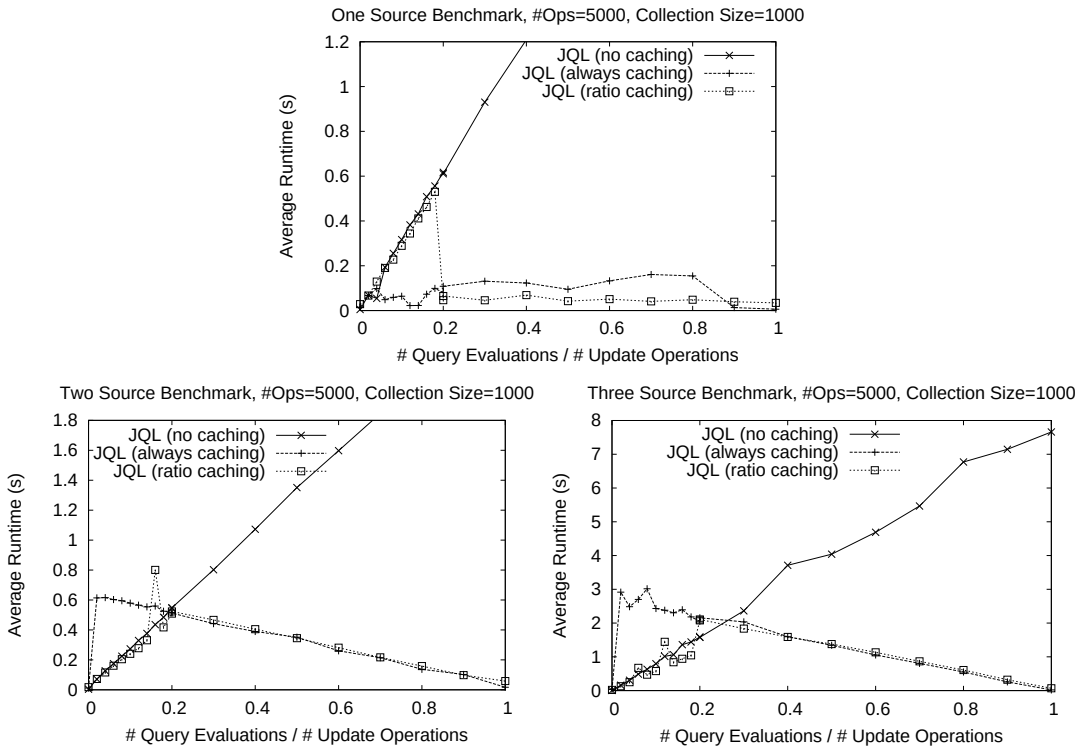

Figure 4. Experimental results comparing the evaluation time without caching, with *always-on* caching and with *ratio* caching.

must iterate through the product of those not containing the affected object. Clearly, this quickly becomes expensive and can easily outweigh the gains from incrementally maintaining cached result sets.

4.4 Study 3: Understanding Program Loops

Queries represent a general mechanism for replacing loops over collections. This raises the question of how many loops found in typical programs can, in fact, be replaced by queries. To understand this, we inspected every loop in the Java programs listed in Table 2. We found that the majority of loops can be classified into a small number of categories, most of which can be replaced by queries. For the purposes of this paper, we are also interested in which loops can potentially benefit from caching and incrementalisation.

4.4.1 Taxonomy of Loop Categories.

Our inspection used the following categories of loops in programs: Unfiltered, Level-1 Filters, Level-2+ Filters, Reduce, and Other — for loops which did not fit in to one of the other categories.

Unfiltered. Loops in this category take a collection of objects and apply some function to every element. This roughly corresponds to the map function found in languages such as Haskell and Python and is the simplest category of loops that can be expressed as queries. An example from Robocode is the following:

```
for(Robot r : robots) { r.out.close(); }
```

These loops do not stand to gain from caching and incrementalisation since they always operate over whole collections — thus, there is nothing to cache.

Level-1 Filters. These loops iterate over a collection and apply some function to a subset of its elements. We've discussed how the Robocode game frequently operates on every alive robot, by selecting from the main robot list:

```
for(Robot r : battle.getRobots()) {
  if(r!=null && r!=this && !r.isDead() && r.intersects(...)) {
    ...
  }
}
```

We have already seen how this loop can be turned into a JQL extended for-loop. The key point is that, in our system, the list of all robots which are not dead can be cached and incrementally maintained. This prevents us from scanning every robot every time the query is evaluated (as is done in the original code). Hence, this class of loops stand to benefit from caching and incrementalisation.

Level-2+ Filters. This category is a logical continuation from the previous, except that it identifies nested loops, rather than simple loops, which operate on a subset of the product of the source collections. A simple example is the

following code for operating on students who are also teachers:

```
for(Student s : students) {
  for(Teacher t : teachers) {
    if(s.name.equals(t.name)) { ... }
} }
```

This would be translated into the following JQL query:

```
for(Student s:students, Teacher t:teachers
     | s.name.equals(t.name)) {
 ...
}
```

The greater the level of nesting, the greater the potential cost of such a loop is. But, at the same time, the greater the potential gain from the optimising query evaluator used by JQL. Intelligent join ordering strategies and incrementalised query caching can all greatly speed up such nested loop operations.

In our categorisation, we distinguish level-1 from level-2+ filters for several reasons: firstly, level-1 filters are by far the most common in our benchmark programs; secondly, they stand to gain from the incrementalised caching technique presented in this paper, but not from the join ordering strategies outlined in our earlier work [41].

Reduce. The Reduce category consists of operations which reduce a collection(s) to either a single value or a very small set of values. Summing the elements of a collection is perhaps the most common example of this. Concatenating a collection into one large string is another. These operations cannot be expressed as queries in our query language and, hence, do not stand to benefit from incrementalised query caching. With a sufficiently expressive query language it is possible to maintain them incrementally, although this requires more complex techniques than we are considering here [24].

Other. The majority of loops classified under Other are related to I/O (e.g. reading until the end of a file, etc). The remainder are mostly loops on collections which either: depend on or change the position of elements in the collection; or operate on more than one element of the collection at a time. Collections.sort(), for example, cannot easily be expressed as a query since it relies upon the ordering of elements. Likewise, Collections.reverse() is also not expressible as a query.

4.4.2 Results and Discussion.

Using this taxonomy we examined, by hand, the set of open source Java programs listed in Table 2. The results of our analysis are presented in Figure 5. Roughly two-thirds of the loops we encountered were expressible as JQL queries and, of these, roughly half would stand to benefit from our incrementalised caching approach. Unfiltered is the most commonly occurring category of loops, which seems unfortunate as these cannot gain from incrementalised caching. An interesting observation, however, is that many of the Unfiltered loops may, in fact, already be operating on manually maintained query results. To understand this, consider the Battle.getAliveRobots() method from §2.1 and corresponding loop in Robot.scan(). Since this searches the entire collection returned by Battle.getAliveRobots(), it would be classified as "Unfiltered". However, this loop would not even exist in a system making use of cached, incrementalised queries! A deeper analysis of the structure of programs is needed to prove this hypothesis, however.

Name	Version	LOC
Robocode (Game)	1.2.1A	23K
RSSOwl (RSS Reader)	1.2.3	46K
ZK (AJAX Framework)	2.2.0	45K
Chungles (File Transfer)	0.3	4K
Freemind (Diagram Tool)	0.8	70K
SoapUI (WebService Testing)	1.7b2	68K

Table 2. Java Programs Inspected for Study 3

Another observation from Figure 5 is that there are relatively few "level-2+ Filter" loops. Such operations are, almost certainly, actively avoided by programmers since they represent fairly expensive operations. The high proportion of "other" loops found in the Freemind benchmark may also seem somewhat surprising. Upon closer examination of the code it became apparent that the majority of these came from an automatically generated XML parser.

Finally, it is important to realise that, although our analysis of these programs indicates many of their loops could be transformed into queries which could benefit from incrementalised query caching, *this does not mean they necessarily will*. In many cases, for example, the loops in question may operate over collections which are small and, hence, the potential gain from incremental caching would be limited. Nevertheless we argue that, even if performance is not improved, the readability and understandability of the code will be.

5. Discussion

5.1 Designing for Querying

Optimised, first-class query support in programming languages should change the way programs are designed. To canvas these issues, consider the following interface for a Graph:

```
interface Graph {
  boolean addEdge(Edge e);
  boolean removeEdge(Edge e);
  Set<Edge> edges(Object n);
}
```

Operation	Expressible as Query?	Benefit from C/I?	Robocode	RSSOwl	ZK	Chungles	Freemind	SoapUI
Unfiltered	Yes	No	38	117	140	24	211	372
L1 Filter	Yes	Yes	92	109	124	18	160	154
L2+ Filter	Yes	Yes	8	2	4	0	2	1
Reduce	No	No	21	34	14	6	30	39
Other	No	No	66	67	62	31	696	126
Total			**225**	**329**	**344**	**79**	**1099**	**692**

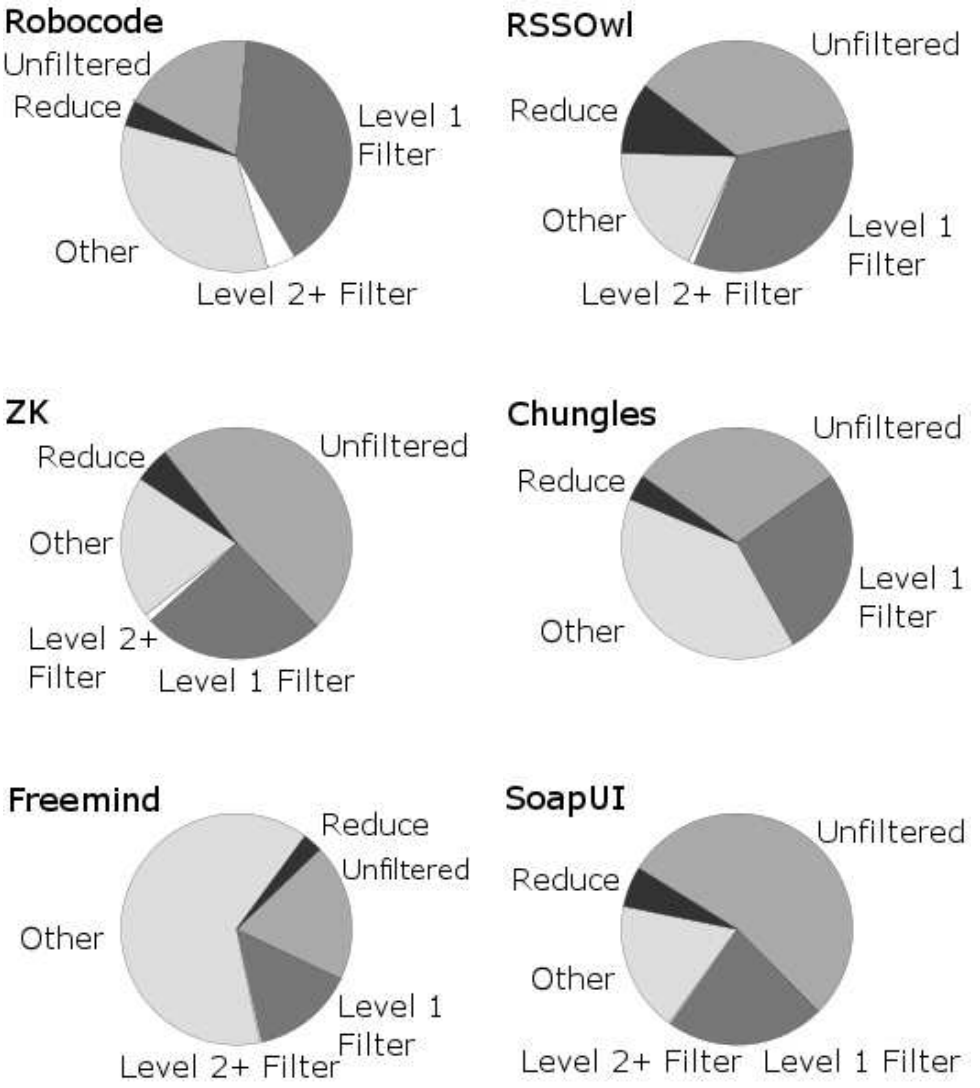

Figure 5. Categorisations of loop operations in Java programs. Note, "C/I" stands for "Caching and Incrementalisation". Hence, "L(evel-)1 Filter" and "L(evel-)2+ Filter" are the two loop classes which can benefit from the incrementalised query cache approach outlined in this paper.

This provides a fixed set of common operations for manipulating graphs: addEdge(e), removeEdge(e) and edges(n) (this returns the set of edges involving some object n). The add/remove operations are *updates*, whilst the edges() accessor is a *query*. But, what does it query over? In an abstract sense, a graph is simply a set of pairs E; the edges(n) operation is then a query over this set and can be formulated as a set comprehension:

$$\texttt{edges(n)} = \{(x, y) \in E \mid x = n \vee y = n\}$$

Thus, it becomes apparent that our Graph ADT fixes the set of possible queries when, in fact, many more are possible. For example, we might like to obtain the set of edges involving a node n, whilst excluding loops (i.e. edges to/from the same node):

$$\texttt{no\_loops(n)} = \{(x, y) \in E \mid (x = n \vee y = n) \wedge x \neq y\}$$

Since this operation is not part of our Graph ADT, a user wanting this functionality must obtain it manually. This is not hard to do; one simply iterates over edges(n) and uses a conditional to narrow it down appropriately. However, this is cumbersome and, we argue, most programmers expend considerable time writing such loops needlessly. First-class queries, on the other hand, provide a much cleaner and more general interface. The challenge is making them competitive with manual ADT implementations, which are typically optimised for the queries they support. We believe our incrementalised query caching scheme represents an important step in this direction. We hope it may allow programmers to write less optimised ADT implementations, with more general APIs, and rely on incrementalised query caching to boost performance. To see how incrementalised query caching can affect program design, compare two implementations of the Graph ADT. First, an adjacency list design provides an efficient edges(n) operation (iteration is linear in |edges(n)|), storing a set of edges for each node:

```
class AdjacencyList implements Graph {
  HashMap<Object,HashSet<Edge> > edges = ...;

  boolean addEdge(Edge e) {
    edges(e.head()).add(e);
    return edges(e.tail()).add(e);
  }

  boolean removeEdge(Edge e) { ... }

  Set<Edge> edges(Object n) {
    HashSet<Edge> rs = edges.get(n);
    if(rs == null) {
      rs = new HashSet<Edge>();
      edges.put(n,rs);
    }
    return rs;
  }
}
```

A simpler, somewhat naïve implementation can maintain edges as a single set:

```
class CompactGraph implements Graph {
  HashSet<Edge> edges = ...;

  boolean addEdge(Edge e) { edges.add(e); }
  boolean removeEdge(Edge e) { ... }

  Set<Edge> edges(Object n) {
    HashSet<Edge> rs = new HashSet<Edge>();
    for(Edge e : edges) {
      if(e.to() == n || e.from() == n) {
        rs.add(e);
      }
    }
    return rs;
  }
}
```

In the simpler design, answering the edges(n) query requires traversing the entire edge set whilst building up the result set. The advantage is a reduced memory footprint and faster add/remove operations (which avoid both HashMap lookups). Such trade-offs in ADT implementations are, of course, well understood.

The advantage of incremental querying incorporated into a programming language is that *suppliers* of ADTs can choose not to make these design tradeoffs: they can produce a straightforward design, and rely on incremental caches to improve the performance *of those queries that are actually executed by particular client programs*. If, for example, a graph is updated often but edges(n) is called relatively infrequently, then the time and space costs to maintain the hash map, not to mention the additional programmer effort required to implement the more sophisticated implementation, are all unnecessary.

So we can view the HashMap in AdjacencyList as precisely the kind of incremental cache that we discuss in this paper. If the ADT queries are cachable, even our limited prototype will cache the result of edges(n) in CompactGraph, and prevent its recomputation if edges(n) is called again for the same n. Of course, if the graph changes due to an update operation, JQL's cache will be incrementally updated to reflect the new graph. So, JQL's caches will correspond almost exactly to the AdjacencyList implementation — except that the work is done automatically within the caching system, rather than manually by the programmer. Over time, we expect that programmers will be able to design simpler and more straightforward programs, and rely upon incrementalised caching to provide acceptable performance.

5.2 Usage Patterns

An important feature of our caching implementation is that it can react to changing patterns of usage at runtime (assuming the query/update ratio policy is used). Manually implemented caching schemes do not typically do this

and, instead, assume fixed usage patterns. For example, the AdjacencyList implementation above makes a fixed assumption about the ratio of calls to edges(n) versus addEdge() (i.e. that it always favours caching edges(n)). A programmer is forced to choose either no caching (i.e. CompactGraph) or always caching (i.e. AdjacencyList) — there's no inbetween. Of course, one can in theory construct such a hybrid Graph implementation — but a typical programmer probably wouldn't (because of the additional complexity). Indeed, even the Java collections library doesn't provide such hybrid collections (e.g. of ArrayList and LinkedList) despite the potential benefits.

While our query/update ratio policy is certainly beneficial in this regard, it has some limitations. For example, while the heuristic can react to changing program usage, it may be slow to do this since it must wait until enough operations have occurred to push the ratio above (or below) the threshold. Whilst this is fine for most situations, it will be problematic in some. For example, when operations occur in short bursts, the heuristic may not react until near the end of the current burst, or not at all! Developing heuristics to deal with this kind of activity seems like challenging, but interesting future work.

5.3 Concurrency

Our prototype implementation of JQL deals with concurrency in a relatively simple manner. The cache of all result sets is implemented using a ConcurrentHashMap which supports concurrent access via non-blocking synchronisation. Individual result sets within the cache are only locked for the duration of a single update. This mechanism prevents ConcurrentModificationExceptions from being thrown by the JQL system. The desired semantics of JQL in a concurrent setting remains to be properly considered, however.

We also believe there is scope to exploit concurrency to optimise query execution further, although this is not currently supported by our prototype. Allowing query execution to take advantage of data parallelism would seem an obvious, and potentially valuable, strategy.

6. Related Work

Language queries and set comprehensions — generally without caching or incrementalisation — have been provided in many languages from SETL [36] and NPL [5] through Haskell and Python up to Cω [2], and LINQ in C♯ and VB [28].

Regarding optimisation of queries, an important work is that of Liu *et al.* [24] who regard all programs as a series of queries and updates. They developed an automatic system for transforming programs in an object-oriented language extended with set comprehensions; this operates at compile time, adding code to explicitly cache and incrementalise set comprehensions. Thus, their incrementalised caches are hard-coded into the program, which contrasts with our more dynamic approach. They demonstrate, for several Python list comprehensions, that the code produced by their system is significantly faster than the base implementation. This approach seems interesting since it takes the programmer closer to the goal of specifying complex operations, rather than implementing them laboriously by hand. In other work, Liu *et al.* consider efficiently evaluating Datalog rules using incrementally maintained sets [23] and, elsewhere, have demonstrated the value of this in the context of type inference [17]. They have also considered incrementalisation of more general computations, including array aggregation (essentially multi-dimensional reduce) [25] and recursive functions [26]. More recently, Acar *et al.* have designed a general incrementalisation framework as an extension to ML, and proved that their incremental computations have the same result as non-incremental computations with the same inputs [1].

The problem of incrementally evaluating database queries, known as the *view maintenance problem*, has received some considerable attention in the past (e.g. [3, 10, 9, 30, 18]). This problem differs somewhat from ours in several ways: firstly, it is usually assumed that the choice to incrementally maintain a table is made by the database administrator; secondly, certain operations (in particular, reduce) are not relevant in this setting. Nevertheless, it is useful to consider what has been done here. Gupta and Mumick examined the view maintenance problem in a traditional database setting [10]. They discuss a number of optimisations and algorithms found in the literature. For example, some algorithms operate when the source tables are only partially available and this limits the situations where incrementalisation can be safely performed; others (e.g. [11]) use something akin to reference counting to make delete operations more efficient. By counting the number of ways a tuple can enter the result set (known as *derivations*), these systems can avoid re-examining the whole source domain when a tuple is deleted. Another interesting work is that of Nakamura, who considered the incremental view problem in the context of object-oriented databases [30]. This setting is considered more challenging than for traditional databases as OODBs must handle more complex data structures and, presumably, queries.

Another relevant work is that of Lencevicius *et al.*, who developed a series of *Query-Based Debuggers* [21, 19] to address the *cause-effect gap* [6]. The effect of a bug (erroneous output, crash, etc) often occurs 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. They also considered the problem of incrementally maintaining cached

query result sets [20, 22]. Their system always chose to incrementalise queries, rather than trying to be selective about this as we are. Nevertheless, they observed speed ups of several orders of magnitude when caching and incrementalisation were used.

Several other systems have used querying to aid debugging and, although none of these support caching or incrementalisation, it seems likely they could benefit from it. The Fox [33, 34] operates on program heap dumps to check certain ownership constraints are properly maintained. The Program Trace Query Language (PTQL) permits relational queries over program traces with a specific focus on the relationship between program events [8]. The Program Query Language (PQL) is a similar system which allows the programmer to express queries capturing erroneous behaviour over the program trace [27]. Hobatr and Malloy [14, 15] present a query-based debugger for C++ that uses the OpenC++ Meta-Object Protocol [4] and the Object Constraint Language (OCL) [40]. 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. Our JQL system was originally inspired by these debugging systems, but was extended to support a range of join optimisations over single queries [41]. This paper describes how we extended JQL to cache results between queries, and then incrementally to update those caches to account for changes in the program as it executes between queries.

Finally, Ramalingam and Reps have produced a categorised bibliography of incrementally computation, which covers the diverse ways in which incrementalisation has been applied in computer science [35].

7. Conclusion

In this paper we have presented the design and implementation of a system for caching and incrementalisation in the Java Query Language. This improves the performance of object queries which are frequently executed, by caching queries' results and then updating their caches as the program runs. This means that subsequent queries can benefit from the work performed by earlier queries, even while the objects and collections underlying the queries are updated between each query execution.

An important aspect of our design is the choice of when to incrementalise a query. This is a challenge because incrementalisation is not for free: instrumentation is required to track updates to objects and to determine how these affect the cached result sets, and memory is required to store the caches. We have detailed an experimental study looking at different ratios of queries to updates in an effort to understand the trade-offs here. Furthermore, although we considered only relatively simple caching policies in this paper, it seems likely that many interesting heuristics could be developed to address this problem. We have also presented a study inspecting loops in Java programs, which indicates that many loops could be rewritten with queries and would stand to benefit from caching and incrementalisation.

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

Acknowledgements

Thanks to all the anonymous reviewers who have read 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] U. Acar, A. Ahmed, and M. Blume. Imperative self-adjusting computation. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, 2008.

[2] G. Bierman, E. Meijer, and W. Schulte. The essence of data access in cω. 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.

[3] 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.

[4] S. Chiba. A metaobject protocol for C++. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 285–299. ACM Press, 1995.

[5] J. Darlington. Program transformation and synthesis: Present capabilities. Technical Report Res. Report 77/43, Dept. of Computing and Control, Imperial College of Science and Technology, London, 1977.

[6] M. Eisenstadt. My hairiest bug war stories. *Communications of the ACM*, 40(4):30–37, 1997.

[7] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[8] 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.

[9] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proceedings of the international conference on Management of Data*, pages 328–339. ACM Press, 1995.

[10] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques and applications. *IEEE Quarterly Bulletin on Data Engineering; Special Issue on Materialized Views and Data Warehousing*, 18(2):3–18, 1995.

[11] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the international conference on Management of Data*, pages 157–166. ACM Press, 1993.

[12] 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.

[13] K. Hartness. Robocode: using games to teach artificial intelligence. *Journal of Computing Sciences in Colleges*, 19(4):287–291, 2004.

[14] 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.

[15] 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.

[16] J.-H. Hong and S.-B. Cho. Evolution of emergent behaviors for shooting game characters in robocode. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, pages 634–638. IEEE Press, 2004.

[17] K. Hristova, T. Rothamel, Y. A. Liu, and S. D. Stoller. Efficient type inference for secure information flow. In *Proceedings on Programming Languages and Analysis for Security*, pages 85–94. ACM Press, 2006.

[18] K. Y. Lee, J. H. Son, and M. H. Kim. Efficient incremental view maintenance in data warehouses. In *Proceedings of the conference on Information and knowledge management*, pages 349–356. ACM Press, 2001.

[19] R. Lencevicius. *Query-Based Debugging*. PhD thesis, University of California, Santa Barbara, 1999. TR-1999-27.

[20] R. Lencevicius. On-the-fly query-based debugging with examples. In *Proceedings of the Workshop on Automated and Algorithmic Debugging (AADEBUG)*, 2000.

[21] 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.

[22] 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.

[23] Y. A. Liu and S. D. Stoller. From datalog rules to efficient programs with time and space guarantees. In *Proceedings of the ACM Conference on Principles and Practice of Declarative Programming*, pages 172–183. ACM Press, 2003.

[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] Y. A. Liu, S. D. Stoller, N. Li, and T. Rothamel. Optimizing aggregate array computations in loops. *ACM Transactions on Programming Languages and Systems*, 27(1):91–125, 2005.

[26] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Transactions on Programming Languages and Systems*, 20(3):546–585, 1998.

[27] 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.

[28] E. Meijer, B. Beckman, and G. M. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *Proceedings of the ACM Symposium on Principles Database Systems*, 2006.

[29] P. Mishra and M. H. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1):63–113, 1992.

[30] H. Nakamura. Incremental computation of complex object queries. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 156–165. ACM Press, 2001.

[31] M. Nelson. Robocode, http://robocode.sourceforge.net, 2007.

[32] J. O'Kelly and J. P. Gibson. Robocode & problem-based learning: a non-prescriptive approach to teaching programming. In *Proceedings of the ACM conference on Innovation and technology in computer science education*, pages 217–221, 2006. ACM Press.

[33] A. Potanin, J. Noble, and R. Biddle. Checking ownership and confinement. *Concurrency and Computation: Practice and Experience*, 16(7):671–687, 2004.

[34] 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.

[35] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *Proceedings of the Conference on the Principles of Programming Languages*, pages 502–510. ACM Press, 1993.

[36] J. Schwartz, R. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, 1986.

[37] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal*, 6(3):191–208, 1997.

[38] 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.

[39] D. Viswanathan and S. Liang. Java virtual machine profiler interface. *IBM Systems Journal*, 39(1):82–95, 2000.

[40] J. Warmer and A. Kleppe. *The Object Constraint Language: precise modeling with UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[41] D. Willis, D. J. Pearce, and J. Noble. Efficient object querying for Java. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 4067 of *Lecture Notes in Computer Science*, pages 28–49. Springer-Verlag, 2006.