Stony Brook University

Automatic Incrementalization of Queries in Object-Oriented Programs

A Dissertation Presented

by

Thomas Michael Rothamel

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

December 2008

Stony Brook University

The Graduate School

Thomas Michael Rothamel

We, the dissertation committee for the above candidate for
the degree of Doctor of Philosophy
hereby recommend the acceptance of this dissertation

Himanshu Gupta — Chairperson of Defense
Professor, Computer Science, Stony Brook University

Yanhong A. Liu — Dissertation Advisor
Professor, Computer Science, Stony Brook University

Michael Kifer — Committee Member
Professor, Computer Science, Stony Brook University

Scott D. Stoller — Committee Member
Professor, Computer Science, Stony Brook University

Jacob T. Schwartz — External Comittee Member
Professor, Computer Science and Mathematics
New York University

This dissertation is accepted by the Graduate School.

Lawrence Martin
Dean of the Graduate School

Abstract of the Dissertation

Automatic Incrementalization of Queries in Object-Oriented Programs

by

Thomas Michael Rothamel

Doctor of Philosophy

in

Computer Science

Stony Brook University

2008

High-level query constructs help greatly improve the clarity of programs and the productivity of programmers, and are being introduced to increasingly more languages. However, the use of high-level queries can come at a cost to program efficiency, because these queries are expensive to compute and may be computed repeatedly on slightly changed inputs. For efficient computation in practical applications, an automatic method is needed to incrementally maintain query results with respect to updates that may change those results.

This dissertation describes a general and powerful method for automatically generating incremental implementations of high-level queries in object-oriented programs. These queries may be over objects, sets, tuples, and maps; and may contain aggregation and grouping constructs. The method generates coordinated maintenance code and invocation mechanisms for all updates that may affect the query results, ensuring that the results are computed correctly and efficiently even though object references may be aliased arbitrarily. We present implementations and experimental results showing the effectiveness of our method when applied to a variety of problem domains.

For my parents,
who supported me throughout.

Contents

Acknowledgments

First, I'd like to thank my advisor, Annie Liu. Without her years of support and guidance I probably wouldn't even have become a grad student, let alone a doctor of philosophy.

I'd further like to thank Himanshu Gupta, Michael Kifer, Scott D. Stoller, and Jacob T. Schwartz for serving on my committee and showing interest in my work. This dissertation owes much to the way their presence and feedback inspired me to do my best.

Some of the members of my group were there the day this dissertation was born as scribbles on a chalkboard. Others came later, and all helped me to give the ideas a final form. Michael Gorbovitski, Katia Hristova, Tuncay Tekle, and Yury Puzis—thanks for everything.

I'd also like to thank all the professors and staff of the Stony Brook University Computer Science department, who over the course of eight years of graduate and undergraduate education turned this programmer into something of a scientist. And everyone at Saint Philip's, Saint Anthony's and Farmingdale who helped me get that far.

I'd like to thank everyone at NRL Code 5546, especially Constance Heitmeyer and Elizabeth Leonard, for giving me the opportunity to work with and learn from them for several summers.

I'd like to thank the Office of Naval Research, the National Science Foundation, and Stony Brook University for their support throughout my research career.

I'd like to thank Chris, for his fifteen years of friendship, the months of time I've spent on the phone with him, and for the opportunity to give the best—and shortest—speech of my life. I'd like to thank Tony, Pete, and Lori for the decade they've spent trying to get me out of my shell. It seems to have worked. And I'd like to thank Scott and the members of the forum he runs. Without them, my graduate career would have been a bit shorter, but a lot less fun.

Finally, I'd like to thank my family, and especially thank my parents for all the support and love they've given me over the past thirty years.

Chapter 1

Introduction

An ongoing and welcome trend in the evolution of programming languages is the addition of high-level query constructs to languages. These query constructs, such as SETL set formers [47], Python and Boo generator expressions [41, 9], JQL for Java [53], and LINQ for C# [30], allow high-level data structures like sets, tuples, maps, and objects to be queried in a concise and abstract manner. This allows the programmer to focus on what is computed, rather than how a result can be efficiently computed.

However, the use of high-level queries can come at a cost to program efficiency, because they are expensive to compute and may be computed repeatedly on slightly changed inputs. Straightforward implementations of high-level query constructs always incur this penalty, limiting their acceptance. To improve efficiency, the results of queries need to be stored and incrementally maintained when values the queries depend on are updated. This can lead to a drastic, often asymptotic, improvement in program running time, especially in programs where queries occur more often than changes.

Currently, this incremental maintenance is mostly performed by hand. Rather than writing a clear high-level query, a programmer is forced to write code that maintains the result of a query in response to updates to values the query result depends on. This code can be complex and error-prone. Even worse, because the updates may be spread throughout a program, the code that incrementally maintains the query result may also be scattered all over the program, making the program difficult to understand and maintain. If

an update occurs without the corresponding incremental maintenance, the correctness of the query is compromised.

This leads to a conflict between clear high-level queries and efficient incremental maintenance. Programmers are forced to choose between clear yet slow and efficient yet complex implementations of the same queries. This typically results in performance critical code being hard to maintain, and less important code being slower than would be possible. Automatic incrementalization helps resolve this conflict, by transforming high-level queries into efficient implementations, allowing the programmer to express queries in a clear manner.

While much previous work has been done on automatic incrementalization of programs, little has been done for object oriented programs, as opposed to databases. The difference is in the flexibility one has in representing objects. In object-oriented databases, one has a large amount of flexibility in changing the representation of objects in order to be able to efficiently answer queries. In programming languages, the representation of objects is often governed by a platform-specific ABI. Even when it is not, efficiency and modularity considerations dictate specific representations, which systems that automatically incrementalize programs are forced to respect.

What work has been done has focused on applying hand-written incrementalization rules that match queries and updates. This is only a partial solution to the problem, as it requires writing one incrementalization rule for each kind of query to be incrementalized. While this method allows the incrementalization to be separated from the rest of the program, it still requires the programmer to undertake the tedious and error-prone task of writing incrementalization rules.

This thesis describes a general and powerful method for automatically generating incremental implementations of high-level queries over objects and sets, where a query may contain arbitrary set enumerators, field selectors, and additional conditions. Once we give the core method, we extend it to handle tuples, maps, aggregation, grouping. Along with a query we give that computes set union, this allows us to cover all of OQL. The method can handle any update to object fields and addition and removal of set elements, and generate coordinated maintenance code and invocation mechanisms to ensure that query results are computed correctly and efficiently. We also describe a prototype implementation of the method and experimental results that con-

firm the effectiveness of the method in supporting clear queries with efficient implementations.

Our method can be applied in dynamic and static manners. A dynamic application changes objects at runtime, allowing the method to only be applied to objects that change at runtime, and to code that is only loaded at runtime. A static application transforms the source code of the program. It requires that all program source code be present, and that the transformation be run in advance. Depending on the program, a static application can remove some of the overhead inherent in a dynamic one.

There are several reasons why our method is effective in practice. First, it processes one query at a time and handles any update to the values that the query depends on, using only local analysis, without requiring whole-program analysis. Whole program analysis is problematic because of libraries and plug-ins, which may not be present until the program is run, or even until sometime after the program is run. Treating queries individually allows the programmer to select which queries are processed, and also allows our method to scale to arbitrarily sized programs.

A second reason is that an object-set query implemented using our method only incurs overhead on objects that query depends on. We add maintenance code to objects directly, rather than to classes. This is important, as there may be classes, such as the set class, in which instances participate in many different queries, but each individual object participates in only a few queries. Similarly, we only incrementally update the results of queries that have been performed, rather than speculatively computing the results of queries that have yet to happen.

Finally, our method is fully automatic, allowing it to be applied without user interaction. We only require that the programmer write object-set queries, and to indicate which queries he wants incrementalized. Our method then takes these queries and generates code that maintains the result when data used by the query is updated. This automatic and repeatable process allows our method to be incorporated into a development workflow, and applied automatically when the program has changed.

Our method takes high-level object-set queries, and automatically generates efficient, incremental implementations from them. This empowers the programmer by allowing him to write programs in a convenient, concise, and

correct manner, without being forced to sacrifice efficiency in the name of clarity.

This dissertation is structured as follows: Chapter 2 introduces the language we will use to present our work. Chapter 3 describes tuple pattern based retrieval, a new language construct that we use as part of the incrementalization process. Chapter 4 describes our method for the automatic incrementalization of object-set queries, including the various extensions described above. Chapter 5 shows how the method can be applied statically. Each of these chapters includes descriptions of relevant related work, and experiments showing the effectiveness of the method described.

Chapter 2

Language

Our method can be applied to many object oriented languages. In this chapter, we present the syntax and semantics of the queries we add to these languages, and the types of changes that can affect the result of those queries. We will then discuss the intermediate language we generate the incremental update code in, a language that should easily map onto may of today's object-oriented languages. This language includes a special for-loop that performs tuple pattern based retrieval. These special for-loops can be translated into standard for-loops.

2.1 Queries and Changes

We consider queries of the following form, called *object-set comprehensions*, or *comprehensions* for short.

$$
\begin{aligned}
\mathit{comprehension} ::= {} & \mathit{parameter}^{+}\texttt{->} \\
& \texttt{\{}\,\mathit{result\_exp} : \mathit{enumerator}^{+}\ \mathit{condition}^{*}\texttt{\}} \\
\mathit{enumerator} ::= {} & \mathit{enumeration\_var}\ \texttt{in}\ \mathit{selector} \\
\mathit{selector} ::= {} & \mathit{variable} \mid \mathit{selector}.\mathit{field} \\
\mathit{parameter} ::= {} & \mathit{variable} \\
\mathit{enumeration\_var} ::= {} & \mathit{variable} \\
\mathit{result\_exp} ::= {} & \mathit{expression} \\
\mathit{condition} ::= {} & \mathit{expression}
\end{aligned}
$$

Intuitively, given values of parameters, a comprehension returns the set of values of the result expression for all values of the variables that satisfy the enumerator and condition clauses. We can see that a comprehension may contain arbitrary object field selections and set element enumerations. We require that every variable in a comprehension appear as either a parameter or an enumeration variable. We also require that the result expression and conditions be functions of the values of the variables in the comprehension, i.e., they can be any expressions whose values depend only on the values of the variables in the comprehension and that have no side-effect.

Precisely, the result of evaluating a query can be given in terms of the set of all possible variable assignments, called the *assignment set*, of the query. A *variable assignment* maps each variable in the query to a value. A variable assignment is in the *assignment set* of the query if and only if:

1. each parameter of the query is assigned the given value of that parameter.

2. each enumerator and condition clause in the query is satisfied when evaluated under the variable assignment.

The *result set* of the query is the set formed by evaluating the result expression under each variable assignment in the assignment set.

A variable may appear as both a parameter and an enumeration variable. We use *unconstrained parameters* to refer to parameters that are not enumeration variables, *constrained parameters* to refer to parameters that are also enumeration variables, and *local variables* to refer to enumeration variables that are not parameters. An enumerator whose enumeration variable is also a parameter is equivalent to a set membership test; we do not treat such tests as conditions, because we can handle them uniformly together with other enumerators for more efficient incremental computation.

Changes. We incrementally maintain the query result under all changes of these three kinds:

- adding an element to a set.
- removing an element from a set.
- assigning a value to a field of an object.

Example. We use the following query as an example of an object-set query:

```
1  wifi ->
2  { ap.ssid : ap in wifi.scan, ap.strength > wifi.threshold }
```

This query is performed on a wifi object, referenced by the variable $wifi$. The object has two fields: a signal strength threshold (*threshold*) and a set of access point objects (*scan*). Each access point object has two fields: a station id (*ssid*) and a signal strength (*strength*). The query has one parameter, $wifi$, the result expression, $ap.ssid$, an enumerator, ap `in` $wifi.scan$, and a condition clause, $ap.strength > wifi.threshold$. The result set of this query contains the *ssid* field of all *ap*s in $wifi.scan$ such that $ap.strength$ is greater than $wifi.threshold$.

The result of this query can be affected by many kinds of changes: adding to or removing from the set referenced by $wifi.scan$, and assigning to $ap.ssid$, $ap.strength$, $wifi.threshold$, and $wifi.scan$. These changes occur to three kinds of objects: wifi objects, access point objects, and sets. Furthermore, these changes can be spread anywhere in many components of the program.

2.2 Language for Generated Code

We write incremental maintenance code in an object-oriented language that supports operations on sets, maps, and tuples and where all values are considered to be objects. This language can be easily mapped onto many popular object-oriented languages, either directly or through the use of objects to box primitive types.

Figure 2.1 describes the operations we use on sets, maps, and tuples. All these operations take constant time, assuming that hashing and object equality comparison take constant time, and hashing is used in the implementation of sets and maps. The maps we use allow a single key to be mapped to multiple values. The set of values a single key maps to is called its image set. Maps support an operation that returns the image set corresponding to a key. When the map is updated, image sets so returned are also updated. Sets and maps are empty when first created. We only create tuples of a constant length.

We use $x == y$ to denote object identity comparison. It returns true if and only if two values refer to the same object. It is a constant-time operation.

s.`empty`()	set s to the empty set
s.`add`(x)	add element x to s
s.`remove`(x)	remove element x from s
s.*any*()	return any element of non-empty set s
x `in` s	true iff x is an element of s
x `not in` s	true iff x is not an element of s
m.`add`(x,y)	add mapping from x to y to map m
m.`remove`(x,y)	remove mapping from x to y from m
m.`get`(x)	return any member of the image set of x.
m.`img`(x)	return image set of key x under m
(x_1,...,x_k)	create a tuple with elements x_1,...,x_k

Figure 2.1: Operations on sets, maps, and tuples.

We use standard statements for assignment ($v = e$), sequencing (*stmt*1 *stmt*2), branching (`if` c`:` *stmt*), and looping (`for` v `in` s`:` *stmt*). During the process of generating incremental code, we also use special for loops, as described below. We use indentation to indicate scoping.

Special For Loops. During the process of generating incremental code, we use special for loops of the form:

```
1    for (x_1,...,x_k) in S:
2        ...
```

where s is a set of tuples of length k, and each x_i may, or may not, already be bound to some value before the loop. Such a loop iterates over the elements of s that *match the pattern* (x_1,...,x_k)—elements where each component corresponding to a bound component of the pattern equals the value of the corresponding variable in the pattern.

This can be done in time proportional to the number of matched elements, using tuple pattern based retrieval, as described in the next chapter. It should be noted that, unlike regular for-loops, the cost of a special for loop may change depending on which variables are bound in the surrounding scope. Our incrementalization method takes advantage of this to decrease the cost of incrementally maintenance.

These special for-loops are only used during the incrementalization processs. Before code is generated, they are turned into other statements: assignments, if statements, and traditional for loops.

Chapter 3

Tuple Pattern Based Retrieval

3.1 Language Construct

This chapter describes tuple pattern based retrieval, which is used to implement variants of the `while`, `if`, and `for` statements that retrieve tuples matching a pattern from a set of tuples. A powerful language construct in its own right, tuple pattern retrieval inspired our method for incrementalizing object set queries. We use it, combined with a method for dealing with objects, during the process of incrementalizing object-set queries. In this chapter, we present our full method for implementing tuple pattern based retrieval.

3.1.1 Example

An example of the use of tuple pattern based retrieval is given in Figure 3.1, which presents a program that topologically sorts the vertices of a graph. Apart from `for`, `while`, and `if` statements that use tuple patterns, the language contains statements that add and remove tuples to and from sets, as well as statements to read from input and print output. It also includes a function that constructs a new empty set. We represent tuples by comma-separated list of components enclosed in parentheses. To easily distinguish expressions from unbound variables in tuple patterns, we have underlined all expressions used in tuple patterns.

This example contains four statements involving tuple pattern based re-

```
1  read VERTICES, EDGES
2  INDEGREES = set()
3
4  for v1 in VERTICES:
5      indegree = 0
6      for (v2, v1) in EDGES:
7          indegree += 1
8
9      INDEGREES.add((v1, indegree))
10
11 while (v1, 0) in INDEGREES:
12     for (v1, v2) in EDGES:
13         if (v2, indegree) in INDEGREES:
14             INDEGREES.remove((v2, indegree))
15             INDEGREES.add((v2, indegree - 1))
16
17     INDEGREES.remove((v1, 0))
18     print v1
```

Figure 3.1: Topological sort, written using tuple pattern based retrieval.

trieval. There are two `for` statements, on lines 6 and 12, one `while` statement on line 11, and one `if` statement on line 13. Since the block starting at line 11 is the part of the program that actually computes the topological order (the rest of the program being initialization), we discuss the first three statements in that block in detail.

INDEGREES is a set of pairs, each of which consists of a vertex that has not been printed yet, and the number of edges into that vertex from other vertices that are the first component of a pair in *INDEGREES*. Throughout execution, we maintain the invariant that each vertex is the first component of at most one pair in *INDEGREES*. The `while` loop on line 11 continues as long as there is at least one pair in *INDEGREES* whose second component is zero. The first component of the pair is then assigned to the variable $v1$. In our example, this means $v1$ is given a vertex with an in-degree of zero, a

vertex that can be next in the topological order.

The purpose of the `for` statement on line 12 is to iterate through all of the edges leaving $v1$. It iterates through tuples in $EDGES$ with a first component equal to $v1$, and assigns their second components to the unbound variable $v2$. Each execution of the `for` statement iterates through each matching element at most once, which ensures that each edge will be considered at most once.

Finally, the `if` statement on line 13 finds in $INDEGREES$ a pair whose first component is equal to $v2$, and assigns its second component to *indegree*. The true block of the `if` statement executes only if such a pair is found, which is always the case in this example if the input is correct. In this case, the real purpose of the `if` statement is to find the tuple matching the pattern.

One important thing to note about this example is that we match against the second component of the tuples in $INDEGREES$ on line 11, and against the first component on line 13. This means that while this algorithm could be implemented using maps, there would have to be two maps corresponding to $INDEGREES$. Another two maps would need to correspond to $EDGES$. Using tuple pattern based retrieval, we halve the number of data structures and reduce the amount of update code that needs to be written by the programmer.

$$
\begin{array}{rcl}
\textit{retr\_statement} & ::= & (\texttt{while} \mid \texttt{if} \mid \texttt{for})\ \textit{retr\_clause}: \\
\textit{retr\_clause} & ::= & \textit{tuple\_pattern}\ \texttt{in}\ \textit{set\_expression} \\
\textit{tuple\_pattern} & ::= & (\ \textit{component}\ (, \textit{component})^*\ ) \\
\textit{component} & ::= & \textit{expression} \mid \textit{pattern\_variable}
\end{array}
$$

Figure 3.2: A grammar for tuple pattern based retrieval statements in our pseudocode language.

3.1.2 Syntax

The syntax we use for tuple pattern based retrieval is given in Figure 3.2. It consists of a retrieval clause, used as part of a `while`, `if`, or `for` statement.

Each retrieval clause consists of a tuple pattern, an `in` keyword, and an expression that evaluates to a set. During an execution of tuple pattern based retrieval, this set is known as the *accessed set*.

A *tuple pattern* consists of one or more comma-separated components. Each component is either an expression that has all variables bound before the retrieval, or a fresh variable that is not bound to anything before the retrieval, called a *pattern variable*. Pattern variables may not be used in expressions that are part of the same tuple pattern. Although our method as presented here does not allow nested tuple patterns, in Section 4 we discuss what would be involved in adding them.

Retrieval clauses can be used as part of `while`, `if`, and `for` statements. A retrieval clause replaces the entire condition of a `while` or `if` statement, or the iteration clause of a `for` loop. When used as the iteration clause in a `for` loop, tuple pattern based retrieval guarantees that each matching tuple is retrieved exactly once.

3.1.3 Semantics

The operational semantics of tuple pattern based retrieval can be given in terms of binding sets. We describe what these binding sets are, and how they can be computed for a given pattern and set. We then show how they can be used to execute a retrieval as part of the `while`, `if`, and `for` statements. Before we can do these, however, we must first define what it means for a pattern to match a tuple, a concept that we have used informally up until this point.

A tuple *matches* a tuple pattern if the tuple and pattern consist of the same number of components, and if, at the time of the matching, the value of each expression in the pattern is equal to the corresponding component of the tuple. For our purposes, equality here refers to structural equality or user-defined equality, not equality of object identity. If there are no expressions in the tuple pattern, that pattern trivially matches all tuples of the same length. Since the value of the expression in the pattern may change over the course of program execution as the variables that are used change, a matching is only valid at a particular point in program execution.

Using this definition, we can define binding sets. A *binding set* is a set

containing, for each tuple in the set matching a tuple pattern, a map from the pattern variables to the corresponding components in the tuple. Such a set could be computed by iterating over the accessed set and performing the matching operation, if we were to ever actually compute it. As a binding set involves matching, its value corresponds to a particular execution of a retrieval.

When a binding set is computed, and how it is used, are determined by the statements in which a retrieval occurs. In a `while` or `if` statement, a binding set is computed each time the condition is evaluated. If this binding set is empty, then the condition is false, and no variables are bound. This will cause the a `while` loop to terminate, or an `if` statement's `else` clause to execute, when present. If the binding set is non-empty, an arbitrarily selected map is taken from it, and the variables in it are bound to their associated values. Such bindings are in effect until the end of the body of the `if` or `while` statement, at which point they become unbound.

In a `while` statement, the fact that the binding set would be recomputed each time the condition is evaluated means that the loop continues as long as a matching element exists in the accessed set. This property makes such a `while` loop useful for accessing a workset. As the `while` loop would recompute the binding set each time through the set, `while` loops are suitable for use with sets that can be changed in the body of the loop.

A `for` statement has slightly different semantics. If implemented using binding sets, each execution of a `for` statement would cause a binding set to be computed once, before the first iteration. The iteration then occurs over the maps in the binding set, with the variables in each map being assigned their associated values while executing the body of the iteration.

We impose on `for` statements the restriction that neither the contents of the accessed set nor the values of the expressions in the tuple pattern change over the course of the iteration. These restrictions are not overly burdensome, as they are similar to the prohibition in languages such as Java and Python against changing collections while iterating over them. They also allow us to perform important optimizations, such as those given below. This restriction may be enforced by the language using some combination of program analysis and runtime checks, or it may be left as the programmer's responsibility.

3.2 Efficient Implementation

As mentioned above, binding sets only exist as a way to give an operational semantics of tuple pattern based retrieval. For tuple pattern based retrieval to become a useful language feature, we must find an efficient and practical implementation.

3.2.1 Local Implementation

Perhaps the most direct implementation of a `for` loop involving tuple pattern based retrieval is one that intermixes the computation of the elements of the binding set with the use of those elements. Lacking a better name, we call this a *local implementation*. A local implementation consists of iterating through each of the elements of the associated set. For each element that is a tuple matching the tuple pattern, the pattern variables are bound to their corresponding elements in the tuple. The body of the iteration is then executed with such bindings. This continues until all elements of the accessed set have been exhausted.

`while` and `if` statements, are implemented similarly. For these statements, however, the iteration is performed once for each time the condition is evaluated, and the iteration terminates once a matching tuple is found. If the iteration proceeds to completion without a matching tuple being found, then the tuple pattern based retrieval has failed. In this case, no variables are bound as a result of the retrieval, and the condition is false.

An example of a local implementation is given in Figure 3.3, showing the code that implements the initialization part (lines 4 through 9) of the example topological sort program in Figure 3.1.

The advantage of a local implementation is its simplicity. If one was asked to implement tuple pattern based retrieval, by hand and without regard to efficiency, something similar to a local implementation is what would likely arise. This method requires only a constant amount of memory, and requires us to only modify the statement containing the retrieval, without touching the rest of the program. Its main problem is inefficiency. The cost of a single tuple pattern based retrieval implemented using the local method is proportional to the size of the accessed set, rather than the number of elements in

```
1 for v1 in VERTICES:
2     indegree = 0
3     for (v2, tmp0) in EDGES:
4         if tmp0 != v1:
5             continue
6
7         indegree += 1
8
9     INDEGREES.add((v1, indegree))
```

Figure 3.3: Local implementation of initialization.

the set that match the tuple pattern. This can lead to asymptotic slowdowns in common algorithms. These slowdowns, while clearly undesirable, are occasionally accepted by programmers when rewriting for performance would unduly complicate the code.

3.2.2 Bound-Unbound Maps

The inefficiency of a local implementation stems from having to iterate over the entire accessed set on each tuple pattern based retrieval. To avoid this, we must develop a data structure that allows us to quickly iterate over only the tuples in a set that match a given pattern.

A data structure that allows this is a *bound-unbound map*. A bound-unbound map is a multimap (explained below) in which the keys are groups of values corresponding to the expressions in a pattern, while the values associated with a key are groups containing the values corresponding to pattern variables, taken from tuples matching the key. (We expect that groups will be implemented as tuples. Here, we refer to them as groups to avoid confusion with tuples taken from the accessed set.) A bound-unbound map can be created from a given tuple pattern and set, and can exist for the life of that set.

A multimap is a map where a single key may be associated with a number of values. When accessed with a key, a multimap returns a set containing all

values associated with the key. If no values are associated with the key, then an empty set is returned. An obvious implementation of a multimap is as a map from keys to sets of values. Care must be taken with this implementation to ensure that keys are garbage collected when their associated sets are empty.

The bound-unbound map for a given tuple pattern and set can be constructed in time proportional to the size of the set. This is done by iterating through all elements of the set. Each element of the same length as the pattern has its components divided into two groups, those corresponding to expressions and those corresponding to pattern variables. These groups become the key and value, respectively, of an association that is added to the map. This construction method results in at most one entry being added to the map for each element of the set, ensuring that the size of the bound-unbound map is proportional to the size of the set.

The use of a bound-unbound map allows us to break the evaluation of tuple pattern based retrieval into three steps. The first step is the construction of the bound-unbound map, given above. The second step evaluates the expressions in the pattern, and uses their values as a key to access the bound-unbound map. Such a lookup can be easily implemented using hashing as an expected constant-time operation (with hashing amortized over the cost of creating the tuple), and returns a (possibly empty) set giving the values of the unbound variables in matching tuples. The third step is to use the contents of this set in a manner appropriate to the construct being executed. When executing a `for`-statement, this entails iterating over the contents of the set, an operation that takes time proportional to the size of the set. `while`- and `if`-statements cause the set to be checked for emptiness and, if it is not empty, an arbitrary element is taken from it. Both operations are constant time, as is assigning the values from such an element to the pattern variables. In all three steps, the only operation that takes time proportional to the size of the accessed set is the construction of the bound-unbound map.

If we recompute the bound-unbound map each time a retrieval is executed, then the asymptotic running time of such an implementation is no better than that of a local implementation. However, an important difference is that a local implementation requires matching to be performed, while computing a bound-unbound map requires only knowledge of the contents of the set. Information about the actual values of the expressions in the pattern is not needed to

```
1  EDGES_ub = map()
2  for (a, b) in EDGES:
3      EDGES_ub.add(b, a)
4
5  for v1 in VERTICES:
6      indegree = 0
7      for v2 in EDGES_ub.img(v1):
8          indegree += 1
9
10     INDEGREES.add((v1, v2))
```

Figure 3.4: Bound-unbound map implementation of initialization.

compute the bound-unbound map. Because significantly less information is used, it is more likely that the computation of a bound-unbound map will be inside an enclosing loop in which the value of the accessed set, and therefore the bound-unbound map, does not change. In this case, we can move the computation of the bound-unbound set to the outside of the enclosing loop. We can move the computation of the bound-unbound map outside of any loop in which the accessed set does not change, potentially reducing the asymptotic running time of the program.

Figure 3.4 shows the initialization phase of Figure 3.1 when implemented using a bound-unbound map. The bound-unbound map is kept in the variable *EDGES_ub*, so named because the first component of the pattern is a pattern variable (and therefore unbound), while the second is an expression (and hence bound). To realize this example, we have added multimaps to our language, with methods to add and remove associations, and get the set of values associated with a key. Sets have a method "any", that returns an arbitrary element. The implementation given in Figure 3.3 takes time proportional to the number of vertices times the number of edges in the graph, while this implementation takes only time proportional to the number of edges. This asymptotic improvement is possible because we can move the computation of *EDGES_ub* outside of the iteration over *VERTICES*, as the contents of *EDGES* do not change during the iteration.

3.2.3 Incremental Update

While previously we have recomputed the contents of the bound-unbound map each time its associated accessed set has changed, this is neither necessary nor desirable. It is possible to incrementally update the contents of a bound-unbound map when changes to its accessed set occur. Doing so allows us to further move the computation of the bound-unbound map to the outside of loops where all updates to the accessed set have been incrementalized. If all updates to a set can be incrementalized, then the only times at which a from-scratch computation of the bound-unbound map is necessary is when it is initially constructed. If the accessed set starts off empty, then we can exploit the fact that an empty accessed set produces an empty bound-unbound map (regardless of the pattern) to eliminate such recomputation entirely.

We update the bound-unbound map by exploiting the property that each entry in the map corresponds to an entry in the accessed set, and each element in the set produces at most one entry in the bound-unbound map. This means that it is simple to create incrementalization rules for the two set update operations, add and remove. When a tuple corresponding to the pattern is added to or removed from the set, the entry in the bound-unbound map representing that tuple is computed in the same manner as is done when the map is computed from scratch. This entry is then added to or removed from the bound-unbound map, as appropriate. Updating a bound-unbound map takes constant time per update, allowing the asymptotic running time of the add and remove operations to remain constant.

If incrementalization is complete, and all recomputation eliminated, then the only operation that takes non-constant time is iteration over the retrieved result, which takes time proportional to the number of matched elements in the accessed set, the minimum time that operation can take. All other operations (incremental addition, incremental removal, lookup in the bound-unbound map, and retrieving a single element) take expected constant time. As a result, the incrementalized implementation of tuple pattern based retrieval is asymptotically optimal for a given input program.

```
1  read VERTICES, EDGES
2
3  INDEGREES_-bu = map()
4  INDEGREES_-ub = map()
5
6  EDGES_-bu = map()
7  EDGES_-ub = map()
8
9  for (a, b) in EDGES:
10     EDGES_-bu.add(a, b)
11     EDGES_-ub.add(b, a)
12
13 for v1 in VERTICES:
14     indegree = 0
15     for v2 in EDGES_-ub.get(v1):
16         indegree += 1
17
18     INDEGREES_-bu.add(v1, indegree)
19     INDEGREES_-ub.add(indegree, v1)
20
21 while true:
22     tmp0 = INDEGREES_-ub.get(0)
23     if not tmp0:
24         break
25
26     v1 = tmp0.any()
27
28     for v2 in EDGES_-bu.get(v1):
29         tmp1 = INDEGREES_-bu.get(v2)
30
31         if tmp1:
32             indegree = tmp1.any()
33
34             INDEGREES_-bu.remove(v2, indegree)
35             INDEGREES_-ub.remove(indegree, v2)
36             INDEGREES_-bu.add(v2, indegree - 1)
37             INDEGREES_-ub.add(indegree - 1, v2)
38
39     INDEGREES_-bu.remove(v1, 0)
40     INDEGREES_-ub.remove(0, v1)
41     print v1
```

Figure 3.5: Topological sort, implemented using static association of bound-unbound maps.

3.2.4 Associating Maps with Sets

One thing we have neglected up until this point is the precise way in which tuple patterns and accessed sets are associated with bound-unbound maps. In this section, we first describe the conditions under which it is possible that a single bound-unbound map can be associated with multiple tuple patterns. We then discuss static and dynamic approaches for associating bound-unbound maps with tuple patterns.

Above, we detailed how a bound-unbound map is constructed from a tuple pattern and an associated set. The only information used in this process is information that can be determined from the tuple pattern statically, specifically the length of the pattern and which components of the pattern are bound expressions. We can call this information the *bound-unbound pattern*, which for each component of the tuple pattern, contains information about whether that component is an expression or a pattern variable. It is possible that a program contains more than one tuple pattern based retrieval from a given set, such that both retrievals have the same bound-unbound pattern. In these cases, all retrievals can use the same bound-unbound map, thus saving space and time by reducing the number of bound-unbound maps that need to be maintained.

Static Approach. Even with a reduced number of bound-unbound maps, however, there is still the question of how these maps are associated with accessed sets. If we have a finite number of sets, and the sets are always accessed by a single name, then it's easy to do this statically. We simply create bound-unbound maps corresponding to each of the bound-unbound patterns that are used in retrievals from a set, and insert the code to update these maps whenever the set is updated. While this is a simplistic approach, it works well in practice, especially when dealing with modules of programs that do not pass sets to other modules.

Figure 3.5 gives an example of the static approach in action, showing how the topological sort example given in Figure 3.1 can be translated into working code, with bound-unbound maps statically associated with sets. Since $INDEGREES$ is initialized to an empty set, the two bound-unbound maps corresponding to it must also be empty, and so there is no need to produce code to compute their initial value. In addition, as all access to $INDEGREES$ is

done through $INDEGREES\_bu$ and $INDEGREES\_ub$, we were able to eliminate $INDEGREES$ itself in favor of maintaining only the bound-unbound maps.

Dynamic Approach. For more complex programs, a dynamic approach is called for. In this approach, we associate with each set object certain bound-unbound maps. Instead of attempting to statically determine which maps need to be constructed for which sets, we only associate a map with a set once that map has been used for a tuple pattern based retrieval. While this means that we always need to compute the contents of a bound-unbound map on its first use, this does not harm the asymptotic performance, as the amount of work to do this once is asymptotically less than the amount of work done to add elements to the set in the first place, and we only need to compute the bound-unbound map once. After its initial computation, a bound-unbound map is incrementally updated by hooks that are called by the add and remove operations on the set.

This dynamic method has the advantage of not requiring much in the way of static analysis, since all updates are performed by hooks, at runtime. This means that it works well in the presence of library code that cannot be changed, and with dynamic languages where static analysis is difficult, or even impossible in the face of code that can change at runtime.

Memory Usage. Our method requires that each set maintain a bound-unbound map for each of the bound-unbound patterns that is used to access that set. We do not maintain bound-unbound maps for patterns that are not used in the program, or for patterns that will never be used to access a set. As any program will have a constant number of bound-unbound patterns, the memory overhead will be a constant factor. In the case where a small number of patterns are used to access each set (as was the case in our examples), the memory overhead will be a small constant factor.

3.3 Discussion

Maps and Multimaps. It is often the case that we have a set and a pattern such that any tuple pattern based retrieval can match at most one tuple. Such a property is the equivalent of a key constraint on a database table, and is exhibited in the $INDEGREES$ table of our running example, which has one indegree for each vertex. The bound-unbound map for a pattern in which the vertex is bound will contain at most one entry per vertex. In this case, implementing the bound-unbound map as a multimap can be wasteful, as each of the sets in the multimap will contain at most one element, an unnecessary overhead. In this case, implementing the bound-unbound map as a simple map suffices.

One solution to this problem is to have the programmer declare key constraints on sets of tuples, and to use this information to select the appropriate implementation of a bound-unbound map. While this method is effective in practice, it does add to the burden of the programmer, and can lead to faulty programs if a set ever violates a declared constraint.

Another answer is to implement bound-unbound maps as data structures that can change their representation. Such a data structure would be implemented as a map as long as the key-constraint holds, but would automatically convert its representation to a multimap if the key constraint is ever violated. As the conversion automatically occurs when elements are added to the set, such a data structure is efficient when the key constraint holds, and robust to cases where it doesn't.

Eliminating Updates. One property of bound-unbound maps is that, for a given length of tuple, every tuple in the set has a corresponding entry in the bound-unbound map. It is therefore unnecessary to store the tuple in the set itself, as it can always be reconstructed when the set is accessed. By updating only the bound-unbound maps, and not the set itself, when elements are added to or removed from the set, we can reduce the cost of add and remove operations.

Nested Tuple Patterns. While the description of our method only deals with single-level tuples, our method can also be used to implement tuple pat-

tern based retrievals involving nested tuple patterns. This is done by flattening nested tuple patterns before finding the components that correspond to expressions and pattern variables, and similarly flattening nested tuples when building or updating the bound-unbound map.

Extension to Lists. While we have been discussing the retrieval of tuples from sets, our method is not limited to sets. We have also developed a data structure that allows one to efficiently perform tuple pattern based retrieval from lists, subject to limitations on the update operations that are performed on the accessed list. The limitation we impose is that addition to and removal from the accessed list must occur at the head or tail of the list, and not at arbitrary points in the middle of the list.

The data structure we use to implement this is an ordered bound-unbound map. This is an ordered multimap where entries can be added to the start or the end of the map, and are returned as a list in the order in which they appear in the multimap.

An ordered multimap can be implemented as a map from keys to lists, in the same way that a normal multimap can be implemented as a map from keys to sets. The limitation imposed above makes it possible to determine if the addition or removal of an entry in the bound-unbound map should occur at the start or the end of a list. If we allowed addition of an element to occur at an arbitrary point in an accessed list, it would be impossible to determine, in constant time, where in the associated list in the bound-unbound map to add the new entry. When subject to this limitation, however, addition and removal can be done incrementally in constant time, making tuple pattern based retrieval that accesses a list as efficient as that which accesses a set.

Alternative Syntax. The syntax proposed in this paper is by no means the only syntax that is possible for tuple pattern based retrieval. There are alternative syntaxes that are potentially more appropriate for specific languages.

When used with a dynamic language, it may make sense to add a keyword or other syntax element that indicates which components of a tuple pattern are bound expressions. This can reduce confusion in languages where variable bindings can leave a block. It may be necessary in languages (such as Python) where our proposed syntax is already legal, but has a different meaning.

Alternatively, one may indicate which components are pattern variables. This may be desirable in languages that require type annotations, as the type annotation can serve both to indicate that a variable is unbound, and to declare its type when it becomes bound.

3.4 Implementations

To gain experience with tuple pattern based retrieval, we have developed two systems that allow programs to be written using it. Our first system takes programs written in a high-level language and transforms them into efficient C++ code. This system has been successfully used in the implementation of a number of algorithms. At the same time, it suffers from a number of limitations. Instead of extending this language to address these, we have chosen to develop a second tool that adds tuple pattern based retrieval to an existing programming language. Our second tool takes as input a program written in Python extended with the three tuple pattern based retrieval statements, and outputs efficient standard Python code. In this section, we discuss the history of these systems, the differences in the generated implementations, and the advantages and disadvantages of each.

3.4.1 High-Level Language

Our high-level language, named PATTON, was originally written to assist in the implementation of parametric regular path queries, as described in [31] and the experiments section below. The inspiration for tuple pattern based retrieval originally came from the pseudocode found in that paper. This system was developed with two goals in mind: to allow us to efficiently try variants of the algorithms by translating the high-level language to C++, and to allow us to compare implementations of bound-unbound maps. Specifically, we compared based representations (using records and linked lists, as given in [48, 45, 46, 21] and [11]) with hash-table representations. While we originally added tuple pattern based retrieval to minimize the differences between an algorithm's implementation and its pseudocode, we quickly began to appreciate it as a language construct in its own right.

The high-level language is a simple one, but one that allows a number of algorithms to be easily expressed. Along with all three tuple pattern based retrieval statements, it includes statements for reading input and writing output, and for adding elements to and removing elements from sets. As data it supports sets, tuples, strings, and integers. The support for the latter two is limited to the equality comparisons needed for tuple pattern based retrieval. This is because the construction of new values through mathematical or string operations can interfere with based representations. The high-level language supports two kinds of variables: normal variables that can refer to strings, integers or tuples of such values, and set variables that can only refer to sets. Each set variable refers to a single set, and there is no way to create sets besides declaring set variables. As a result, programs written in this language support only a finite number of sets, all known statically. This, in conjunction with add and remove statements, makes it easy to insert code to maintain the bound-unbound maps, without needing complicated static analysis. An additional statement in the language allows specification of key constraints, which are used to select between map and multimap implementations of the bound-unbound maps.

Our high-level language proved to be a success, both on its own and when used as a target to simplify code generation. As we will discuss in the experiments section, we were able to use it to implement parametric regular path queries and Datalog rules. Although not discussed in this paper, we also used it to implement relational calculus queries. In all three cases, tuple pattern based retrieval reduced significantly the amount of effort needed to produce efficient code.

As we moved into the area of security policies, however, some issues with our language became apparent. A lack of support for function or procedure calls makes it impossible to implement security policy frameworks, such as role-based access control. While it would be possible to extend this language to include these features, we felt that it would be more useful to add tuple pattern based retrieval to a popular language. This decision led to the creation of our second system.

3.4.2 Python Preprocessor

We then implemented tuple pattern based retrieval as a preprocessor that takes as input Python programs augmented with the three tuple pattern based retrieval statements, and outputs efficient standard Python code. Python was chosen as a source and target language because of its built-in support for tuple construction and set iteration, both of which are syntactically similar to what is used in our tuple pattern constructs. Indeed, this similarity is to such an extent that we chose to give the added tuple pattern constructs different keywords, to prevent code that uses our extensions from being run by Python itself.

Our preprocessor generates code that uses callbacks to update the bound-unbound maps, with the maps themselves being stored, when present, in fields on the set objects themselves. This allows modifications to the underlying code to be minimized. The `Set` class is modified to automatically invoke callbacks when elements are added to or removed from it, a modification that only needs to be done once, regardless of the number of tuple pattern based retrieval statements and the number of add and remove call sites in the program. All of the other modifications to the program are confined to the immediate vicinity of the tuple pattern based retrieval statements. We do not need to find or modify statements that add and remove elements of the set, as such operations are detected using callbacks. This means that the preprocessor does not require a global static analysis, making it suitable for programs that may use arbitrary library code, and for programs that modify themselves (by constructing and evaluating code) at runtime. The code generated by our preprocessor represents all bound-unbound maps with a data structure that changes from a map to a multimap when necessary.

In the experiments section, we present the results of applying the preprocessor to the running example of topological sort, to graph reachability, and to role-based access control. The experiments revealed a number of optimizations that can be added, such as eliminating the construction of 1-component tuples, and keeping references directly to the bound-unbound maps. At the same time, the experiments show that the preprocessor is able to generate asymptotically efficient implementations of Python programs written using tuple pattern based retrieval.

implementation	size	Running time for number of edges 1,000	2,000	3,000	4,000	5,000
local	30	678 (19.9)	3,012 (43.0)	7,190 (66.6)	13,580 (97.0)	23,750 (132.7)
dynamic b-u map	63	56 (1.65)	116 (1.66)	176 (1.63)	234 (1.67)	296 (1.65)
optimized dynamic b-u map	63	34 (1.00)	74 (1.06)	110 (1.02)	150 (1.07)	185 (1.03)
static b-u map	29	34 (1.00)	70 (1.00)	108 (1.00)	140 (1.00)	179 (1.00)

Table 3.1: Program size in lines and running times in milliseconds for implementations of topological sort. The numbers in parenthesis are running times relative to the static implementation.

3.5 Experiments and Applications

In this section, we present the results of a number of experiments we conducted on programs written using tuple pattern based retrieval. These experiments were conducted over a variety of problem domains: graph algorithms and queries, program analysis, and security policy frameworks. We conducted these experiments to evaluate the advantages of having tuple pattern based retrieval as a language construct, and to confirm that the predicted efficiency of bound-unbound maps can be actually achieved. This section includes experiments done using both of our tools, with a mix of experiments that were performed as part of research into other areas, and experiments that are original to this paper. The latter allows us to demonstrate on well-known examples the effectiveness of tuple pattern based retrieval, while the former shows its applicability to a range of problems.

In this section, when lines of code are given, the value provided is the number of lines of code that solve the problem, excluding comments and blank lines. This does not include the size of the library code included by an implementation, nor does it include the size of the test harnesses used to collect statistics. Unless otherwise noted, performance measurements were collected on an AMD Sempron 3100+, running at 1.8 GHz. When preprocessor-generated Python programs were run, Python 2.3.5 was used. The measurements are the average of five runs on the same input data set.

3.5.1 Topological Sort

The first experiment we perform is on the running example of topological sort. We created four Python implementations of the topological sort algo-

implementation	size	Running time for number of edges 50,000	100,000	150,000	200,000
dynamic b-u map	24	790 ms (0.98)	1,604 ms (1.00)	2,396 ms (0.99)	3,218 ms (0.99)
static b-u map	16	803 ms (1.00)	1,607 ms (1.00)	2,414 ms (1.00)	3,240 ms (1.00)

Table 3.2: Program size in lines and running times in milliseconds for implementations of graph reachability.

rithm given in Figure 3.1. Two of these implementations are the local and bound-unbound map implementations automatically created by the current version of the preprocessor. A third implementation consists of the bound-unbound map implementation, hand optimized to eliminate unnecessary tuple construction and store bound-unbound maps in local variables to prevent repeated field lookups. This version represents optimizations that are planned for a future version of the preprocessor, but are not yet implemented. The final implementation was a hand implementation of the version of the program given in Figure 3.5, similar to what would be generated from our pseudocode language, except in Python rather than C++. This was implemented by hand so we could compare a static implementation with the dynamic implementation generated by the preprocessor, under similar conditions.

Table 1 shows the results of these experiments. The first thing to note is the expansion of the program sizes. The topological sort algorithm, written in Python with tuple pattern based retrieval statements, is 13 lines long. The implementations varied in length from 29 to 63 lines long, so using tuple pattern based retrieval leads to a program that is less than half the size.

To evaluate the performance of the generated code, we ran the implementations on topological sort problems of varying size. In all cases, the input data consisted of a linear chain of vertices, such that there is one unique topological sort.

The first thing to note is that the local implementation of the program is asymptotically slower than the other three versions, as predicted. The unoptimized dynamic version, while still linear, is much slower than the other two versions, while the static version is slightly faster than the optimized dynamic one. We attribute this difference to the increased indirection required in the dynamic version, and while we are working on decreasing this indirection penalty, we recommend using a static implementation when possible.

```
1  read start, EDGES
2
3  WORKSET = set()
4  RESULT = set()
5
6  WORKSET.add(start)
7  RESULT.add(start)
8
9  while v1 in WORKSET:
10     for (v1, v2) in EDGES:
11         if v2 not in RESULT:
12             WORKSET.add(v2)
13             RESULT.add(v2)
14
15     WORKSET.remove(v1)
16
17 print RESULT
```

Figure 3.6: Graph reachability, written using tuple pattern based retrieval.

3.5.2 Reachability

To attempt to determine the cause of the difference in the running times of the static and dynamic versions of the program, we used the preprocessor to generate implementations of the graph reachability algorithm given in Figure 3.6. This algorithm contains two tuple pattern based retrievals, one from $WORKSET$ and one from $EDGES$. The retrieval from $WORKSET$ does not require a bound-unbound map, as no bound expressions are used in it. The only bound-unbound map is the one used for retrievals from $EDGES$. As $EDGES$ is not updated in the loop, this bound-unbound map is not changed from when it is created, allowing us to consider the performance of the queries themselves without needing to also consider the cost of updating bound-unbound maps.

We implemented two versions of graph reachability. The first version was a dynamic bound-unbound map implementation automatically generated from

a Python translation of the code given in Figure 3.6. We did not create a hand-optimization of this version, as the optimizations only pertain to bound-unbound map update. For comparison purposes, we also created by hand a static version of the graph reachability, again similar to what would be generated from our pseudocode language, except in Python instead of C++.

The results are given in Table 2. The first thing to note is that the dynamic implementation doubles the size of the initial 12-line program, while the static implementation adds a mere four lines. This is because the dynamic implementation still has to include the update code, even if it is never called, while the static version can eliminate it entirely. This has no effect on the running time, as the dynamic implementation is marginally faster than the static one. Achieving similar results on a program that does not contain any updates suggests that it is the update functions that differ in performance between the static and dynamic versions of the program.

3.5.3 RBAC — Role-Based Access Control

An example of a realistic system that benefits from tuple pattern based retrieval is role-based access control. RBAC controls access by assigning permissions to perform operations on objects to roles, and then assigning users to those roles. When a user activates a role in a session, that session has all the permissions of the role. This simplifies the management of permissions in systems with many users, objects, and operations.

In the ANSI standard for RBAC [17, 6], RBAC is specified using eight sets and four maps from values to sets. Five of the sets are uninteresting from our perspective, containing only values of a given type, and not tuples. The remaining three are sets of pairs: $PRMS$ contains all possible permissions in the system as object-operation pairs, PA relates roles with the permissions the role has, and UA assigns users to roles. Of the maps used in the core RBAC specification, only *user_sessions* and *session_roles* are fundamental, mapping a user to his sessions and a session to its roles, respectively. The other two, *assigned_permissions* and *assigned_users*, duplicate information contained in PA and UA, respectively.

To help evaluate tuple pattern based retrieval, we translated the administrative and system functions of core RBAC from the variant of Z used in

[17] to the dialect of Python that our preprocessor can process. As part of the conversion, we eliminated the two redundant maps, and turned the other two maps into sets of pairs. We were able to eliminate 10 of the 30 map and set updates from the 13 functions we translated. The resulting translation consists of 85 lines of python, comprised of 36 assertions, 19 set updates, 13 function definitions, 8 tuple pattern based retrievals, 3 function calls, 2 set membership tests, 2 set iterations, and 2 returns.

When translated by our preprocessor into a dynamic bound-unbound map implementation, the size of the program swelled to 211 lines. Inspection of the generated code showed that the preprocesor correctly generated bound-unbound maps corresponding to *assigned_permissions* and *assigned_users*. By using a tuple pattern based retrieval, we were able to eliminate two maps and a third of the update operations from the specification of RBAC.

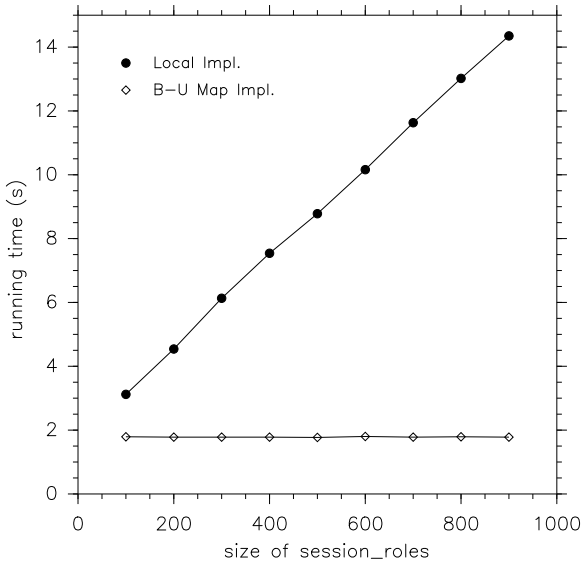

Figure 3.7: Running time of 100000 RBAC check_access operations.

Figure 3.7 demonstrates how an efficient implementation of tuple pattern based retrieval improves asymptotically the running time of our program. It shows the time it takes to perform 100000 `check_access` operations while vary-

ing the size of `session_roles`, the set that maps a session to the roles used by that session. In this graph, the number of roles per session is fixed at 10, while the number of sessions increases. The local implementation takes running time proportional to the total number of session-role pairs in the system, while the dynamic bound-unbound map implementation remains constant, scaling only with the number of roles per session. This beats the asymptotic performance of a straightforward implementation of the Z specification, which takes time proportional to the number of roles in the system, of which the roles per session is a subset. This demonstrates that tuple pattern based retrieval not only simplifies the implementation of RBAC, it also allows us to generate an implementation that is an asymptotic improvement.

3.5.4 Other Applications

In addition to the experiments performed above, tuple pattern based retrieval has been successfully applied to a range of problems, each complex enough to have merited its own paper. Here, we discuss how tuple pattern based retrieval was used in the experiments in those papers. In contrast with the experiments given above, our focus here is on describing how tuple pattern based retrieval simplified the creation of those implementations, rather than on performance measurements.

Parametric Regular Path Queries. Parametric Regular Path Queries [31] match a regular-expression-like pattern containing variables against paths in a graph containing labels. An existential regular path query returns the set of all vertex-substitution pairs such that there exists a path from the start vertex to the returned vertex where the labels on that path match the pattern, after the substitution has been applied to the pattern. This has a number of applications to program analysis, as simple queries can find uses of uninitialized variables, violations of locking disciplines, and other properties of the program.

Tuple pattern based retrieval was used in the implementation of multiple algorithms that perform parametric regular path queries. Indeed, it was the need to efficiently generate implementations of the algorithms in [31] that lead to the creation of our high-level language, and the pseudocode found in that paper that inspired the design of that language. By automatically generating

extremely efficient C++ code that uses based[45, 11] representations of the bound-unbound maps, we significantly reduced the effort required to implement variants of the parametric regular path query algorithms. This allowed us to give experimental performance results for a number of variants, enabling us to give guidance as to when each variant should be used.

When written using tuple pattern based retrieval, various algorithms for performing parametric regular queries range from 19 to 34 lines of code, counted as described in this section, including 3 tuple pattern based retrievals. When our pseudocode language is translated into C++, the code size expands substantially. The 19 line example was translated into 696 lines of C++, the 34 line variant to 833 lines. The generated code is fast, processing over 400,000 worklist entries per second. The running time of the implementations scales only with worklist size, further showing how bound-unbound maps can be used to implement tuple pattern based retrieval in an asymptotically optimal manner.

Datalog Rules. Tuple pattern based retrieval also shows promise as a construct in intermediate languages that tools can target. A method [32] is described that transforms a set of Datalog rules into efficient low-level implementations with guaranteed time and space complexities, avoiding dependency on a potentially large interpreter. One implementation of this method uses our high-level language, PATTON, as an intermediate language, to simplify the code generation process. It first generates code in PATTON, and then that code is translated to C++. Another implementation that generates C directly consists of 327 lines of Python, and uses a library of 2,000 lines of C code. The generator that targets our high-level language was written in two days, and consists of only 114 lines of Python code, and does not require a custom library.

To show the effectiveness of this approach, we give two examples. A transitive closure algorithm consisting of two Datalog rules was translated into 27 lines of high-level code, which in turn became 595 lines of efficient C++. A pointer analysis algorithm was translated into 93 lines of high-level code, and 1,944 lines of C++.

Chapter 4

Object-Set Queries

4.1 Overview of the Method

With tuple pattern based retrieval behind us, we are ready to give our method for the automatic incrementalization of object-set queries. Our method processes one query at a time, and incrementally maintains the result over every change that affect the result of the query.

To compute the result of a query efficiently in the presence of all possible changes to the parameters of the query, we maintain the result incrementally with respect to the changes. To do this, we first note that the result of a query can be computed from scratch straightforwardly in two steps. Step 1 computes the assignment set: it creates all possible variable assignments allowed by the enumerators and puts each such variable assignment that also satisfies the conditions in the assignment set. Step 2 computes the result set: it iterates through the assignment set and puts the result of evaluating the result expression under each variable assignment into the result set.

There are five main ideas for efficient incremental computation. (1) We can compute the precise change to the assignment set after any change to the query parameters, called *differential assignment set*, denoted D. (2) We can maintain a query result efficiently using D by keeping a reference count with each element in a result set. (3) We can return query results efficiently for possibly arbitrary parameter values by maintaining a map from combinations of parameter values to query results. (4) To avoid maintaining query results for

all possible values of unconstrained parameters, which would be impractical, we keep combinations of values of unconstrained parameters that have been queried on. (5) To efficiently retrieve objects from field values, and sets from members, as needed for efficient incremental computation, we can maintain inverse maps.

The differential assignment set, D. The differential assignment set, D, is a set of variable assignments that would be added to or removed from the assignment set by an update. Generating code to compute it efficiently under all possible changes is at the core of our method. We use D in maintaining the result set efficiently, avoiding maintaining the entire assignment set. Note that, in general, we can not compute the change to the result set efficiently without computing D, because multiple variable assignments may lead to the same value in the result set.

Reference counts for elements in a result set. Exactly because multiple variable assignments may lead to the same value in the result set, to determine whether a value should be in the result set when the assignment set is updated, we must keep a count of the number of variable assignments that produce that value. Addition and removal operations to a result set maintain the reference counts, and do the actual addition and removal of an element only if its reference count changes from 0 to 1 and 1 to 0, respectively. This is important for us to update a result set correctly and efficiently.

Result map, R. Instead of creating a new result set for each query instance, i.e., a query with a combination of parameter values, we maintain a map, R, called the *result map*, that maps combinations of parameter values to the result sets of the query. We can retrieve the result set of a query from R, based the parameter values, using a constant-time access operation. This yields a result set that changes as the result map does, which we call a *live* result set. In many cases, this is acceptable, as the set is used transiently and then discarded. Where necessary, we make a copy of the result set, at cost linear in the size of the result, and return that. Techniques exist for determining where copying is necessary [21].

Values of Unconstrained Parameters, U. We can not maintain query

results for all possible values of unconstrained parameters, because we do not know what the values might be. So we keep combinations of values of unconstrained parameters that have been queried on, as a set of tuples, denoted U, and only maintain query results for these values of the unconstrained parameters. Note that for each tuple in U, we maintain query result for all possible values of constrained parameters, because they are determined by values of unconstrained parameters. So we look up the query result in constant time if the values of unconstrained parameters are found in U.

Inverse maps, inv_m and inv_f's. We also maintain the following inverse maps. The map inv_m, where m stands for member, maps an object to the sets that contain it; it is the inverse of the usual mapping from a set to its members. The maps inv_f, one for each field f, maps an object to the objects referring to it through field f; it is the inverse of field selection.

These sets and maps are manipulated by the generated code. D, R, U, and the inverse maps are all stored in variables that are unique to a comprehension; these variables are bound to a single object in all the maintenance code generated for a comprehension, but are bound to different objects in code generated from other comprehensions.

4.2 Generating Code for Computing the Differential Assignment Set

We need to generate code to compute the differential assignment set, D, for each possible change to the data used by the query. With the current representation of queries, called the *object-domain* representation, changes include assigning new objects to all chains of selected fields, and adding and removing elements of all sets, in the query. To make it much easier to enumerate all possible changes and generate code, we translate the query into a pair-domain representation, enumerate changes and generate code in the pair domain, and then translate the code back to the object domain.

4.2.1 Translating to the Pair Domain

The *pair domain* uses sets of pairs to represent the field-value relations and the set-membership relation, though these sets do not exist in the final generated code. This allows us to consider only the addition and removal of pairs as changes. The translation also replaces each field selection with a fresh variable, so every object that the query depends on is referred to by a pair-domain variable. This makes it easy to enumerate all possible changes that can affect the result of a query.

Precisely, we use the following sets in the pair domain:

- For each field f, a set, $field_f$, is used to relate any object with the value of the field f of the object:

$$(o, v) \in field_f \iff v == o.f.$$

- A single set, $member$, is used to relate any set with any member of the set:

$$(s, o) \in member \iff o \in s.$$

Note that $field_f$ is used for same-named fields of different objects in the pair domain, just like $.f$ i is used to access same named fields of different objects in the original object domain.

We translate a comprehension into the pair domain by applying the following two rules repeatedly until they do not apply:

- For each variable o and field f, replace all occurrences of the field selection $o.f$ with a fresh variable, say v, and add a new enumerator (o,v) `in` $field_f$.
- Replace each enumerator v `in` s, where v and s are variables, with a new enumerator (s,v) `in` $member$.

This eliminates all fields and sets in the object domain.

For the wifi query, this yields:

```
1  wifi ->
2  { ap_ssid : (ap, ap_ssid) in field_ssid,
3              (wifi, wifi_scan) in field_scan,
4              (wifi_scan, ap) in member,
5              (ap, ap_strength) in field_strength,
6              (wifi, wifi_threshold) in field_threshold,
7              ap_strength > wifi_threshold }
```

This replaces all 4 fields and 1 set in the object domain with 5 sets in the pair domain and increases the number of variables used from 2 to 6.

4.2.2 Generating Code for All Possible Changes

Recall we need to generate code to compute D for each possible change to the data used by the query. Now we do this in the pair-domain.

First, we explain that each change we must handle corresponds to an element addition and/or removal based on an enumerator in the pair-domain comprehension. It is obvious, from the translation, that each occurrence of a field selector, and each retrieval of a set element, corresponds to an enumerator. So, each assignment to a field of an object and each element addition or removal that can affect the query result corresponds to an enumerator—each is indeed changing the object referred to by the left variable in the enumerator. In particular, we have the following:

- An enumerator of the form (o,v) `in` $field_f$ means that if the field f of an object, say o_0, that o refers to is assigned a value v_2, where the value of field before the change is v_1, then the corresponding changes we must handle are removing the pair (o_0,v_1) from $field_f$ followed by adding the pair (o_0,v_2) to $field_f$.
- An enumerator of the form (s,o) `in` $member$ means that if an element, say o_0, is added to a set, say s_0, that s refers to, then the corresponding change we must handle is adding (s_0,o_0) to $member$, symmetrically for removing an element.

Next, for each enumerator, we generate a block of code for computing D for either adding an element or removing an element from the set being

enumerated. The same block of code is used for both element addition and removal because, for each enumerator, the variable assignments added to the assignment set when an object is added to the set being enumerated are the same as the variable assignments removed from the assignment set when the object is removed from the set.

Note that the generated code refers to the element added or removed. Thus, for removal, the generated code must be run before the removal, and for addition, the generated code must be run after the addition.

Generating code here has two steps. Step 1 creates the clauses that compute the assignment set; this is independent of the enumerator considered. Step 2 determines a nesting order of these clauses that minimizes the cost of executing all clauses, based on enumerator considered.

Generating clauses. We generate one clause for each enumerator and each condition in the pair-domain comprehension. For each enumerator (x,y) `in` s, a for-clause of the following form is generated:

```
1  for (x,y) in s:
```

For each condition c, an `if`-clause is generated:

```
1  if c:
```

We also generate a single for-clause that ensures the values of the unconstrained parameters are in U:

```
1  for unc_params in U:
```

where *unc_params* is a tuple of the unconstrained parameters of the comprehension.

For the wifi query, consider the change that adds ap to $wifi\_scan$. The following clauses are created:

```
1  for (ap, ap_ssid) in field_ssid:
2  for (wifi, wifi_scan) in field_scan:
3  for (wifi_scan, ap) in member:
4  for (ap, ap_strength) in field_strength:
5  for (wifi, wifi_threshold) in field_threshold:
6  if ap_strength > wifi_threshold:
7  for (wifi) in U:
```

Nesting clauses. The basic idea of choosing a nesting order is to use the bound values of the variables in the given change to maximize the number of constant-time map lookups and field dereferences, which take constant time, and to minimize the number of interations, which take linear time. The high-level effect is to minimize the amount of work in incremental computation caused by a change. Doing this exploits the fact that bound variables in a special for-statement use lookups to reduce the amount of iteration needed.

To always ensure an optimial ordering would require knowing precise set sizes for the enumerators and costs for the conditions. Even using a join-order optimization algorithm [14] would require knowing the selectivity of each join. Both would require additional annotations added to the program, which we attempt to avoid. Instead, we find that a simple greedy algorithm works well in practice.

The greedy strategy picks a clause that has the minimum asymptotic running time to execute next, given the set of variables bound so far. The set of bound variables initially contains the variables that appear in the change. This set is used to analyze each clause, using the rules below, to determine if a clause is *runnable*, and if so, what the asymptotic running time is. A runnable clause with the lowest asymptotic running time is chosen, and added to the nesting order. The variables bound by that clause are added to the set of bound variables, and the process is repeated until all clauses are added to the order. Rules for analyzing the clauses are as follows:

- A special for-loop that iterates over a $field_f$ set is runnable if at least one variable in the pattern is bound; this avoids iterating over every object in the program. This for-loop takes constant time if the first variable in the pattern is bound, because it is a field selection, and linear time in

all other cases.

- A special for-loop that iterates over the *member* set is runnable if at least one variable in the pattern is bound; this avoids iterating over every set in the program. This for-loop takes constant time if both variables in the pattern are bound, because it is a set membership test, and linear time otherwise.
- The special for-loop that iterates over U is always runnable. It takes constant time if all variables in the pattern, i.e., all unconstrained parameters of the query, are bound, and linear-time otherwise.
- A `if`-clause is runnable if all of the variables in it are bound. All `if`-clauses are considered to take constant time by default.

A nesting order will always be computed, ensured by the clause that iterates through U. This is because the definition of object-set comprehensions ensures that every variable is reachable, through a path containing selection and enumeration, from at least one unconstrained parameter. As each selection and enumeration corresponds to a pair-domain clause, there is a path of pair-domain clauses from the unconstrained parameter to the variable. When the unconstrained parameters become bound by the statement iterating over U, all for-loops will become runnable, allowing all variables to be bound. This then ensures that every `if`-statement is runnable, allowing every statement to be placed in the nesting order.

Once all variables are bound to some values, these variables and values are used to create a variable assignment, which is then added to D. Let $var\_asgn()$ be a function that creates a variable assignment for these variables using their bound values. Figure 4.1 shows the generated code in the pair domain for computing D under one update that affects the result of the wifi query.

4.2.3 Translating Back to the Object Domain

This translation eliminates field and member sets in the pair domain, and replaces special for-loops with standard statements. The translation uses the rules in Table 4.1. It gives code for special for-loops over $field_f$ sets and over the *member* set, and for all three possible combinations of boundness of the two variables in the pattern—recall that we do not have the case when both variables are unbound.

after adding ap to a set that wifi_scan refers to:

```
1       for (ap, ap_ssid) in field_ssid:
2         for (ap, ap_strength) in field_strength:
3           for (wifi_scan, ap) in member:
4             for (wifi, wifi_scan) in field_scan:
5               for (wifi, wifi_threshold) in field_threshold:
6                 if ap_strength > wifi_threshold:
7                   for (wifi) in U:
8                       D.add(var_asgn())
```

Figure 4.1: Generated pair-domain code for computing D for the running example.

- When both variables are bound, loops over $field_f$ sets are field-value tests, and loops over the *member* set are membership tests.
- When the first argument is bound but the second is not, loops over $field_f$ sets are field selections, and loops over the *member* set are element retrievals.
- When the second variable is bound but the first is not, inverse maps are used for reverse retrievals for both field selections and element retrievals.

In the third case, we also generate code for incrementally maintaining the inverse maps, as given in Table 4.2; it is easy to see that these maps takes constant time to maintain and has a constant-factor space overhead.

Note that computing D for a change uses these inverse maps. Thus, for element addition, inverse maps must be updated before computing D, and for element removal, the inverse maps must be updated after computing D.

The special for-loop over U is implemented similarly. If some variables in the pattern are not bound, we maintain a map from values of bound variables to values of unbound variables. These maps are incrementally updated when U changes. This allows each matched element to be retrieved in constant time, and the entire for-loop to take linear time in the number of matched elements.

For the wifi query and the update we considered, the code in Figure 4.2 is generated.

pair-domain construct	`for` (x,y) `in` $field_f$`:` *block*	`for` (x,y) `in` $member$`:` *block*
x bound y bound	`if` $y == x.f$`:` *block*	`if` y `in` x`:` *block*
x bound y unbound	$y = x.f$ *block*	`for` y `in` x`:` *block*
x unbound y bound	`for` x `in` inv_f.`img`(y)`:` *block*	`for` x `in` inv_m.`img`(y)`:` *block*

Table 4.1: Rules for translating back to the object domain.

`for` (x,y) `in` $field_f$`:` *block*	`for` (x,y) `in` $member$`:` *block*
when an object is first referred to by x: inv_f.`add`$(x.f, x)$ **before assignments to x.f** inv_f.`remove`$(x.f, x)$ **after assignments to x.f** inv_f.`add`$(x.f, x)$	**when an object is first referred to by x:** `for` y `in` x`:` inv_m.`add`(y, x) **before x.remove(y):** inv_m.`remove`(y, x) **after x.add(y):** inv_m.`add`(y, x)

Table 4.2: Rules for generating code for maintaining inverse maps.

after adding ap to a set that wifi_scan refers to:

```
1   ap_ssid = ap.ssid
2   ap_strength = ap.strength
3   if ap in wifi_scan:
4     for wifi in inv_scan.img(wifi_scan):
5       wifi_threshold = wifi.threshold
6       if ap_strength > wifi_threshold:
7           if (wifi) in U:
8             D.add(var_asgn())
```

Figure 4.2: Generated object-domain code for computing D for the running example.

4.3 Generating Code for Maintaining the Result Map

Once D is computed, it is used to update the result map R. For each block of code that computes D, we generate another block of code that updates R.

For maintenance after assigning a value to a field or adding an object to a set, we generate the code below, where $params(a)$ takes a variable assignment a and returns a tuple containing the values of the parameters of the query, and $eval(e,a)$ evaluates expression e under the variable assignment a. D is reset to empty after it is used for updating R, thus is empty and takes no space when we are not executing maintenance code.

```
1   for a in D:
2     R.add(params(a), eval(result_exp, a))
3   D.empty()
```

For maintenance before assigning a value to a field or removing an object from a set, the generated code is the same except with `add` replaced with `remove`.

The result map maintenance code must be run after all code for computing D for a given change has been run. Aliasing could cause problems otherwise. For example, in the query below, suppose $p.S$ and $p.R$ are aliased to the same

set, say called SR. When an object, say o, is added to set SR, two updates occur: one adds an object to a set that $p.S$ refers to, and the other adds an object to a set that $p.R$ refers to. The D for both updates will contain the variable assignment $\{x \mapsto o, y \mapsto o\}$.

```
1    p -> {x : x in p.S,  y in p.R,  x == y }
```

By running the result map maintenance code after all code for computing D, we ensure that each variable assignment causes a result mapping to be added to the result map once, and only once. This maintains the invariant that the reference count of a mapping in the result map equals the number of variable assignments in the assignment set projecting onto that mapping.

4.4 Organizing Maintenance Code

Three kinds of maintenance code have been generated: (1) code that maintains inverse maps, (2) code that computes D, and (3) code that maintains R. They must be run in response to updates to objects, including set objects.

We organize maintenance code based on the pair-domain variables. This is for two reasons, from Section 4.2: (1) each object that the query result depends on is referred to by a pair-domain variable, and (2) each block of maintenance code generated is for an update to an object that a pair-domain variable refers to, or when the object is first referred to by the variable. We put together, conceptually, all maintenance code that handles updates to the object referred to by a pair-domain variable v, and we call it *an obligation* any object referred to by v must fulfill; we use v as the id of the obligation. Note that an object may have multiple obligations, because it may be referred to by more than one pair-domain variable, a.k.a. aliasing.

Recall that, among the three kinds of maintenance, R must be maintained after D is computed, and for addition, inverse maps must be maintained before D is computed and R is maintained, and all three must be done after the addition, while for removal, inverse maps must be maintained after D is computed and R is maintained, and all three must be done before the removal. When an object has multiple obligations, we run the maintenance code of the same kind from all obligations, before running maintenance code of another kind,

for the same reasons as before. Note that R is only updated once, by the first block of maintenance code for R, because D is reset to empty at the end of it, and later blocks of code have no effect.

Obligations are assigned to objects using the function *assign_obligation*. It takes an object o and an obligation 'v' as arguments. It does nothing if o is already assigned obligation 'v'. Otherwise, it (1) runs any maintenance that needs to be run when o is first referred to by variable v, and (2) registers the maintenance code corresponding to v, separately for addition and removal of course, with o, so that it is called when addition and/or removal occurs. Maintenance code for (1) includes code for updating inverse maps, as in Section 4.2.3, and assigning obligations to other object, as described below. Implementation for (2) depends on the host language, which we will describe for Python, in Section 4.8; similar ideas apply to other languages.

Two mechanisms are used to assign obligations to objects. Obligations are assigned to unconstrained parameters by the query execution code discussed in the next section. Obligations are assigned to enumeration variables, i.e., constrained parameters and local variables, by maintenance code associated with other obligations.

Assigning obligations to enumeration variables. The code that assigns obligations to enumeration variables is generated following a reachability-based approach. We start by initializing a set, called the set of supported variables, to the unconstrained parameters of the comprehension. We then search for a pair-domain enumeration of the form:

```
1    (x,y) in s
```

where x is in the set of supported variables, and y is not. This clause is then used to create obligation assignment code, as given below, and y is added to the set of supported variables. This process repeats until all variables are added to the set of supported variables. This process will always complete because all variables are reachable from the unconstrained parameters.

The obligation assignment code generated depends on the set s in the enumeration. If s is a $field_f$ set, we generate the following code:

when obligation 'x' is assigned to an object referred to by x

1 *assign_obligation*($x.f$, 'y')

when an object with obligation 'x' has field f assigned

1 *assign_obligation*($x.f$, 'y')

If it is the set *member*, we generate:

when obligation 'x' is assigned to an object referred to by x

```
1  for i in x:
2    assign_obligation(i, 'y')
```

when an object with obligation 'x' has element i added

1 *assign_obligation*(i, 'y')

Note that obligation assignment code is classified as maintenance code of kind (1), because it is done when an object is first referenced by a variable, which also causes inverse maps to be updated.

The method above ensures that if an object is ever referred to by a variable v, the object is assigned obligation v. An obligation assigned to an object is never removed from the object. So, the cost of assigning obligations is constant amortized over object creation, element addition, and field assignment. However, the maintenance code may be run even after an object can no longer affect the result of a query, until it is garbage collected.

4.5 Generating Code for Executing the Query

Finally, we generate code for query executing. Recall that we keep combinations of values of unconstrained parameters that have been queried on. Each query, for a combination of values of unconstrained parameters, is computed once from scratch—the first time it is encountered; after that, the query result is incrementally maintained.

The query execution code first determines if the query is being incrementally maintained, i.e., if a tuple consisting of the values of the unconstrained parameters is in the set U. If it is, then the incrementally maintained result is returned. If not, the query execution code computes the result from scratch,

and begins incremental maintenance; this has four steps:

1. Call *assign_obligation* to assign $obligation_p$ to the object referred to by each unconstrained parameter p. This will then ensure that obligations are assigned to every object the query depends on.

2. Add a tuple of the values of the unconstrained parameters to the set U.

3. Compute D for the addition to U in Step 2, using the method in Section 4.2.

4. Maintain the result map, using the method in Section 4.3.

At the end, the values of the unconstrained parameters are used to retrieve a live result set from the result map. This set is the result of the query.

For our running example, the generated query execution code is given in Figure 4.3.

4.6 Discussion

Correctness. The correctness of our method is guaranteed by the invariants maintained by the generated code for each kind of code generated. The cost of our method is linear in the size of the given program, because all analyses are local, and all transformations are 1-1 correspondence. The costs of individual operations in the generated code are described with the method; we summarize below the overall performance of the generated code.

Our method allows many extensions and additional optimizations, such as handling tuples in queries (by translating them into objects and back), supporting aggregate operations (such as count and sum), and simplifying the generated code (by eliminating pair-domain variables that only do copying). We describe below an interesting join optimization.

Aliasing. Our method uses the D set to ensure that the reference count is maintained correctly in the face of aliasing between objects. For example, consider the query:

```
1   if (wifi) not in U:

2       assign_obligation(wifi, obligation_wifi)
3       U.add((wifi))

4       wifi_scan = wifi.scan
5       wifi_threshold = wifi.threshold
6       for ap in wifi_scan:
7           ap_ssid = ap.ssid
8           ap_strength = ap.strength:
9           if ap_strength > wifi_threshold:
10              if (wifi) in U:
11                  D.add(var_asgn())

12      for a in D:
13          R.add((wifi), eval(ap_ssid, a))
14      D.empty()

15  return R.img((wifi))
```

Figure 4.3: Generated code for executing the query for the running example.

```
1    p -> x : x in p.S,  y in p.T,  x == y
```

When $p.S$ and $p.T$ are aliased to the same set, say called ST. When an object, say o, is added to set ST, maintenance code for additions to both $p.T$ and $p.S$ must be run. The set D will have the variable assignment $\{x \mapsto o, y \mapsto o\}$ added to it twice, while the result x will only be added to R once. This is important if $p.S$ is changed to point to a set other than ST. In that case, the assignment will be only added to D once, and so x will be removed from R once. If R was updated directly instead of using D, the reference count of R would be incorrect.

If it is proved that no aliasing can occur between the pair-domain variables used by the query, then it's possible to have the maintenance code update R directly. This eliminates the time required to iterate over and clear D, as well as the memory used by that set.

Memory usage. As incrementalization improves program performance by storing and updating the results of comprehensions, it will increase memory usage. R takes the space required to store the result of each query for which we maintain results; we maintain results if the unconstrained parameters are in U. The inv_m and inv_f maps require space proportional to the size of the objects assigned obligations; this adds constant overhead. While inside maintenance code D uses space equal to the number of assignments generated, it is empty when the program executes outside of maintenance code.

Weak references can be used to eliminate query results for objects that are no longer alive, preventing memory leaks through the results sets. Further memory reduction is possible by reaping the least-recently used queries, as described in [54]. We expect programmers using this method to understand its memory use, and only request incrementalization when the memory overhead is acceptable.

Performance. To allow the second and later executions of a query to occur in constant time, our method requires that updates to data maintain the result map. The cost of this maintenance depends on the structure of the comprehension and the update.

For example, for the wifi query, assuming that each *ap* is in the *scan* field of exactly one wifi object, the maintenance corresponding to the following

changes will be performed in constant time: (1) assigning to $ap.strength$, (2) assigning to $ap.ssid$, and (3) adding an ap to or removing an ap from set $wifi.scan$. Other updates require linear time to maintain the result: assigning to $wifi.threshold$, and assigning to $wifi.scan$, i.e. assigning a new set to the $wifi.scan$ field, not updating the content of the set. The linear time for this maintenance is because a single value for ap cannot be determined from the update, so iteration over the $wifi.scan$ set is needed.

Our maintenance code is asymptotically faster than recomputation code when the values of the variables bound at an update allow some iterations to be eliminated, and thus, our method will always produce an asymptotic speedup as long as the frequency of updates is not asymptotically higher than that of queries. In no case does our method produce incremental update code that is asymptotically slower than code that executes the query from scratch, because the worst-case maintenance code is identical to recomputation code; therefore, when the frequency of queries is asymptotically the same as that of updates, our method will never produce an asymptotically slower program.

When the frequency of queries is asymptotically less than that of updates, our method may produce slower programs, depending on the running times of maintenance code. Currently, we rely on the programmer to not choose incrementalization in these circumstances. As our method analyzes the asymptotic running times of maintenance code when deciding the nesting of clauses during code generation, it can be easily extended to report these times statically. We can extend our method to use these times to help decide what queries to incrementalize.

Optimizing joins. Equality joins on object identity asserts that two fields must refer to the same object. They are expressed in object-set comprehensions as conditions of the following form—recall that a selector is a variable optionally followed by one or more field selections:

```
1    selector == selector
```

Our method already handles joins as conditions, but we can optimize them and further decrease the asymptotic running time.

We optimize joins by modifying the pair-domain representation of the query. After translating the query into the pair domain, we search for con-

ditions of the form $v1 == v2$. Because selectors are always translated to pair-domain variables, all joins will be of this form. When a join is found, it is eliminated from the pair-domain comprehension. In the remaining conditions, enumerations, and the result expression, $v1$ and $v2$ are replaced with the new variable $v1\_v2$. We repeat this process until no joins remain.

This optimization reduces the number of variables in the comprehension. It usually increases the number of generated clauses containing bound variables, and thus can decrease the asymptotic running time of the code for computing D.

4.7 Extensions

4.7.1 Multiple Field Retrieval

While we believe our pair-domain method for performing object-set queries to be adequate for most purposes, there are some classes of queries for which it performs suboptimally. One such class of queries is those for which it is beneficial to retrieve objects using more than one field. In this chapter, we will extend our method to deal with these queries.

An example of such a query is as follows:

```
1   S, R -> { (a, b) : a in S; b in R;
2                      a.first == b.first; a.last == b.last }
```

This query finds objects in S and R that have matching $first$ and $last$ fields, at might be useful when normalizing two data sets. Our pair-domain method may take a suboptimal amount of time to maintain this query. For example, when an object is added to set S, the corresponding objects in set R are looked up. When the pair-domain method is used, the objects in set R are accessed by their first or last field, but not both simultaneously. When many objects in set R share the same first or last field, this can lead to an asymptotic slowdown. These conditions are the case with people's names, where there are common first names (Tom, Michael), and last names (Liu, Smith), but very few people share both names.

Our method needs some changes in order to support retrieving objects using multiple fields. The largest change is that the pair domain needs to be

extended to support dealing with multiple fields of an object at once. We call this extended pair domain the tuple domain, with the changed method being called the tuple-domain method. We also need to extend the definition of the inverse maps so that they can be used to support our tuple-domain method.

As the extension of the inverse maps is fairly straightforward, we will address that first. In the pair-domain method, an inverse map of the form inv_f maps an object to the set of object having that object as the value of field f. The tuple-domain reverse map extends inverse maps to support multiple fields. We name these inverse maps $inv_{f,g}$. These multiple-field inverse maps map tuples to objects. A tuple is mapped to an object if the value of the first field on an object is equal to the value of the first component of the tuple, the value of the second field on a object is equal to the second value of the tuple, and so on. We ensure that the field names always appear in some total order, as this minimizes the number of inverse maps we need to maintain.

Note that these extended inverse maps are similar to the bound-unbound maps we maintain when optimizing tuple pattern based retrieval. Both maps use known values of a data structure to retrieve the remaining values of the data structure, although they differ in the data structure they maintain this information about.

The remaining changes introduced by our tuple-domain method are to the way we generate the code that computes the differential binding set. There are four changes we must address. First, we define the tuple domain, and how it differs from the pair domain. We then show how to translate a pair-domain comprehension into a tuple-domain comprehension. After this, we generate and order the tuple-domain statements used to compute the differential binding set. Finally, we translate these tuple-domain statements back into the pair domain, using the extended inverse maps if necessary.

The tuple domain.. The tuple domain differs from the pair domain in the definition of the $field$ set. The new field set is a set of that satisfies the following condition:

$$(o, v_1, v_2, ..., v_n) \texttt{ in } field_{f_1,f_2,...,f_n} \iff \forall i <= 0 <= n, o.f_i == v_i$$

This definition of the tuple domain allows us to deal with one or more fields being accessed on an object, which is the complete set of interesting cases. The

tuple-domain definition of $field$ is a superset of the pair-domain definition. In the pair-domain, n could only be 1.

Translating to the tuple domain.. We must now show how to translate a comprehension from the pair domain to the tuple domain. This is a straightforward process. We find each group of pair-domain enumerations of the form:

```
1  (o, v_1) in field_f_1
2  (o, v_2) in field_f_2
3  ...
4  (o, v_n) in field_f_n
```

Because of the way we translate comprehensions into the pair domain, there are no duplicate fields. We translate each group of pair-domain enumerations into a single tuple-domain enumeration:

```
1  (o, v_1, v_2, ... v_n) in field_f_1,f_2,...,f_n
```

We do this for each group of pair-domain enumerations we can find. Conditions and enumerations over the *member* set are the same in the tuple and pair domains, and so do not need to be translated.

Generating and nesting clauses.. The next step in our method is to use generate clauses from the tuple-domain enumerations and conditions, and to nest these clauses in a way that minimizes the cost of executing them. Clause generation for conditions and enumerations over the *member* set is the same as for the pair-domain method. Enumeration over the $field$ sets requires us to take into account multiple fields. When encountering the enumeration clause:

```
1  (o, v_1, v_2, ... v_n) in field_f_1,f_2,...,f_n
```

we generate the special for-loop:

```
1  for (o, v_1, v_2, ... v_n) in field_f_1,f_2,...,f_n:
```

The semantics of these special for-loops are those of a tuple pattern based retrieval for-loop.

We then need to alter our heuristics for nesting the clauses to take into account the more complex for-loops. In the tuple-domain method, as with

the pair-domain, we treat a for-loop over a field set as runnable if at least one variable is bound. If the variable comprising the first component of the tuple pattern is bound, then we treat the enumeration as taking constant time, otherwise, it takes linear time.

Since these for loops allow for there to be runnable for-loops with more than one free variable, it is no longer appropriate to treat all linear-time for-loops as equal. Instead, we choose a runnable for-loop with the least number of free variables. This increases the likelyhood that we have bound a field or set of fields that form a key. If all variables comprising a key are bound, then the for-loop will run in constant time.

As with the pair-domain, tuple-domain for-loops benefit from more precise cost models. Knowing which fields of objects form a key with respect to a given set is one type of information that can help with choosing a nesting order.

Translating back to the object domain.

Finally, we need to translate back to the object domain. This is done by applying the technique for tuple pattern based retrieval to the special for loop, and then creating the obligations needed to maintain the inverse maps, should one be needed.

We translate the for-loop:

```
1    for (o, v_1, v_2, ..., v_n) in field_{f_1,f_2,...,f_n}
```

in the following way. First of all, we reorder the variables and fields such that the bound variables come before the unbound variables. Reordering the variables and fields in the same way preserves the semantics of the special for-loop. With this reordering, we can assume that variables $v_1, v_2, ..., v_m$ are bound, and variables $v_{m+1}, v_{m+2}, ..., v_n$ are unbound.

If o is not bound, our first step is to retrieve it. We generate the code:

```
1    for o in inv_f_1,f_2,...,f_m.get((v_1, v_2, ..., v_m)):
2        ...
```

We also need to generate obligations that maintain $inv_{f_1,f_2,...,f_n}$. These obligations are of the form:

when object o is assigned obligation o

```
1    inv_f_1,f_2,...,f_m.add(((o.f_1, o.f_2, ..., o.f_m), o))
```

when object o is about to be garbage collected

```
1    inv_f_1,f_2,...,f_m.remove(((o.f_1, o.f_2, ..., o,f_m), o))
```

when a field o.f_i is assigned to...

...before the assignment

```
1    inv_f_1,f_2,...,f_m.remove(((o.f_1, o.f_2, ..., o.f_m), o))
2
```

...after the assignment

```
1    inv_f_1,f_2,...,f_m.add(((o.f_1, o.f_2, .., o.f_m), o))
```

When o is bound, we need to check to see that the values of the bound variables correspond to the values of the field on object o. For 1 ¡= i ¡= m, we generate the code:

```
1    if o.f_1 == v_1 and o.f_2 == v_2 and ... and o.f_m == v_m:
2      ...
```

Either way, once the value of o is known, we must assign values to the unbound variables. This is done by generating the code:

```
1    v_k = o.f_k
2    ...
```

where $m < k <= n$.

4.7.2 Tuples

While we use tuples throughout our method, until now our query language did not support them. They can be added by mapping tuples to objects,

performing the multiple retrieval method described above, and then mapping the field accesses back to tuple components. As tuples are immutable, there are several simplifications we can make when generating obligation code.

Syntax and Semantics. The first step in incrementalizing object set queries involving tuples is to change the syntax of comprehensions. We alter the definition of an enumerator to allow for enumeration over a set of tuples. The revised comprehension syntax is given below:

$$
\begin{aligned}
\mathit{comprehension} &::= \mathit{parameter}^{+}\texttt{->} \\
&\quad\ \texttt{\{}\mathit{result\_exp} : \mathit{enumerator}^{+}\ \mathit{condition}^{*}\texttt{\}} \\
\mathit{enumerator} &::= \mathit{enumeration\_var}\ \texttt{in}\ \mathit{selector} \\
\mathit{enumerator} &::= (\mathit{enumeration\_var}, +)\ \texttt{in}\ \mathit{selector} \\
\mathit{selector} &::= \mathit{variable} \mid \mathit{selector}.\mathit{field} \\
\mathit{parameter} &::= \mathit{variable} \\
\mathit{enumeration\_var} &::= \mathit{variable} \\
\mathit{result\_exp} &::= \mathit{expression} \\
\mathit{condition} &::= \mathit{expression}
\end{aligned}
$$

This new rule allows one to define a *tuple enumerator*, which matches a tuple pattern against a set. The tuple pattern consists of one or more enumeration variables. A tuple enumerator is satisfied by variable assignments that cause the tuple pattern to construct a tuple that is present in the set.

Changes to the generation of differential binding set computation code.. The first step in our method is applied when transforming tuple enumerators into the pair domain. This is done by adding a transformation rule that turns the tuple pattern:

1 (v_1, v_2, ..., v_n)

into the new variable v_{tuple}, while at the same time, adding the enumerations:

1 (v_tuple, v_1) `in` $field_{component1}$
2 (v_tuple, v_2) `in` $field_{component2}$
3 ...
4 (v_tuple, v_n) `in` $field_{componentn}$

Where $component1$, $component2$, ..., $componentn$ are unused field names that

represent the various components of the tuples. These will then be merged into the single enumeration:

```
1    (v_tuple, v_1, v_2, ..., v_n) in field_{component1,component2,...,componentn}
```

These transformed obligations will then be used to generate maintenance code. The maintenance code will access fields of the form $v\_tuple.componentk$. As these fields do not exist in the original program, they need to be transformed into component accesses of the form $v\_tuple[k]$. After this transformation, the code is compatible with the original program, and maintenance can be performed.

Impact on obligations. We must also take into account tuples when generating obligation code. Unlike other objects, tuples are immutable, which means that fields corresponding to components cannot be modified. This means that it's unnecessary to generate obligation code corresponding to modifications of these fields. Omitting such code can reduce the number of inverse maps that we need to maintain.

4.7.3 Maps

Adding maps to the class of queries our method supports poses many of the same issues as sets. We solve them in a similar way, by considering mappings as objects, generating the maintenance code, and then transforming the generated code so that it uses the original maps whenever possible.

Maps present several unique issues we must address, however. The most important is that unlike tuples, mappings are not objects with unique ids in the system. The second is that map objects support fast associative access, and so it does not make sense to include additional maps just to give us that property.

Syntax. The first thing we do is to alter the definition of a selector to include access to maps. This entails adding the grammar rule:

$$selector ::= selector\texttt{.get}(selector)$$

to our query language.

Changes to the generation of maintenance code. For the purpose of generating maintenance code, we treat each mapping (as well as each map) as an object. A mapping has three fields: map, key, and value. Map is the map the mapping is part of, while key and value are self-explanatory. It's important to note that these mapping objects are only used for the purpose of code generation. They do not exist in the generated code.

We transform the comprehension by replacing the code:

```
1    v_map.get(v_key)
```

with v_{value}, and adding the following three enumerations:

```
1    (v_mapping, v_map) in field_map
2    (v_mapping, v_key) in field_key
3    (v_mapping, v_value) in field_value
```

where $v_{mappping}$ and v_{value} are new variables. Note that as no variable or expression corresponds to $v_{mapping}$, it will not be used outside these three enumerations.

Code generation then proceeds as described in this section. Note that this will create the larger enumeration:

```
1    (v_mapping, v_map, v_key, v_value) in field_map,key,value
```

This, in turn, will become the special for-loop:

```
1    for (v_mapping, v_map, v_key, v_value) in field_map,key,value
```

In such a special for-loop, $v_{mapping}$ will always be unbound, and at least one of the other three variables will be bound. This means that there are five possible combinations for which variables are bound. We use different code to transform back to the object domain for each possible combination of bound variables.

1. When v_{map}, v_{key}, and v_{value} are bound, the special for-loop becomes equivalent to a map lookup. The code generated is:

```
1    if v_value in v_map.get(v_key):
2        ...
```

2. When v_{map} and v_{key} are bound while v_{value} is unbound, the special-for loop is equivalent to iteration over the contents of the map:

```
1  for v_value in v_map.get(v_key):
2      ...
```

3. When v_{map} and v_{value} are bound while v_{key} is unbound, we need to maintain a map back from the map and value to the key. We call this map $inv_{map,value}$. We replace the special-for loop with the following code:

```
1  for v_value in inv_map,value.get((v_map, v_value)]:
2      ...
```

We also generate the following obligations for v_{map}:

when the obligation is first assigned to map m

```
1  for k, v in m.items():
2      inv_map,value.get((m, v)).add(k)
```

when the mapping k -¿ v is added to map m

```
1  inv_map,value.get((m, v)).add(k)
```

when the mapping k -¿ v is removed from map m

```
1  inv_map,value.get((m, v)).remove(k)
```

4. When only v_{map} is bound, with v_{key} and v_{value} unbound, the special for-loop becomes equivalent to iteration over the items in the map:

```
for (v_key, v_value) in v_map.items():
    ...
```

5. When v_{key} and v_{value} are bound and v_{map} is unbound, it becomes necessary to use the $inv_{key,value}$ map to map a key-value combination to the maps containing it:

```
for v_map in inv_key,value.get((v_key, v_value)):
    ...
```

We also generate the following obligations for v_{map}:

when the obligation is first assigned to map m

```
for k, v in m.items():
    inv_key,value.get((k, v)).add(m)
```

when the mapping k -¿ v is added to map m

```
inv_key,value.get((k, v)).add(m)
```

when the mapping k -¿ v is removed from map m

```
inv_key,value.get((k, v)).remove(m)
```

6. When only v_{key} is bound, with v_{value} and v_{map} unbound, it becomes necessary to look up the value in inv_{key}, a mapping from the key to the maps that contain it, and the values it undertakes in those maps.

```
for (v_map, v_value) in inv_key:
    ...
```

We also generate the following obligations for v_{map}:

when the obligation is first assigned to map m

```
for k, v in m.items():
    inv_key.get(k).add((m, v))
```

when the mapping k -¿ v is added to map m

```
inv_key.get(k).add((m, v))
```

when the mapping k -¿ v is removed from map m

```
inv_key.get(k).remove((m, v))
```

7. When only v_{value} is bound, with v_{map} and v_{key} unbound, we look it up in inv_{value}, which maps the value to the maps and keys that correspond to it.

```
for (v_map, v_key) in inv_value:
    ...
```

We also generate the following obligations for v_map:

when the obligation is first assigned to map m

```
for k, v in m.items():
    inv_value.get(v).add((m, k))
```

when the mapping k -¿ v is added to map m

```
inv_value.get(v).add((m, k))
```

when the mapping k -¿ v is removed from map m

```
inv_value.get(v).remove((m, k))
```

Note that $v_{mapping}$ does not appear in any of the generated code. This is because it is redundant to the combination of v_{map}, v_{key}, and v_{value}, and hence we can remove it entirely from the program.

4.7.4 Aggregation

Throughout this dissertation, the results of queries have been sets of objects. While queries producing sets are useful, they are by no means the only useful type of query. Here, we will see how we can use the bindings we produce with other forms of aggregation.

The first thing we do is to extend the syntax of our queries to support aggregation. This is accomplished by adding aggregation operators to our query language, as well as a way to indicate that a query is performing aggregation rather than the usual form of set construction.

An aggregation expression is an expression containing aggregation operators. These aggregation operators look like functions that are optionally applied to result expressions over the variables defined in the query. These result expressions may not contain aggregation operators directly, although nested queries should be okay.

We also change the syntax of queries to indicate that aggregation is being used. Specifically, we replace the ':' with a '—' when aggregation is in effect. The query:

```
1  wifi ->
2  { ap.ssid : ap in wifi.scan, ap.strength > wifi.threshold }
```

is equivalent to the following query, which uses an aggregation expression:

```
1  wifi ->
2  { set(ap.ssid) | ap in wifi.scan, ap.strength > wifi.threshold }
```

This assumes that the 'set' aggregation operator has been defined to create a set containing the objects given to it by the result expression. That's what the reference-counted sets we use do.

Of course, there are many other operators that are interesting to us. For example, we can use a 'sum' aggregation operator to sum up numeric values, while a 'count' operator counts the number of operations produced. (Similar operator names are used in SQL.) This could be used to produce a query that incrementally maintains, for example, the average salary of professors at a college.

```
1  professors ->
2  { sum(p.salary) / count() | p in professors }
```

There are a large number of aggregations that are interesting. It's important to realize that it would be impossible for us to enumerate every one. Instead, we must provide a framework that allows programmers to define their own aggregations, so that programmers don't have to choose to not use incrementalization at all. At the same time, we should provide a library that contains the most commonly used aggregations, so that they don't constantly need to be reinvented from scratch.

Aggregation operators must be registered with the incrementalization sys-

tem before they can be used. An aggregation operator is a name that's bound to an object constructor, that's used to make aggregation objects. Aggregation objects are objects with the following three methods.

- add - used to indicate that an object should be added to the aggregate.
- remove - used to indicate that an object is being used for an aggregate.
- value - used to return a usable value from the aggregate.

The value method bears some explaining. It's used to return a value from the comprehension that differs from the object created by the aggregation operator. For example, an aggregation operator that computes the sum of the aggregated values will want to return an integer, even if an object is used to manage the aggregation.

Altering the incrementalization method.. Extending our incrementalization method to support aggregation involves two fairly minor changes.

The first is that we change R to be a map from parameters to aggregation objects, rather than always being a map from aggregation objects to reference-counted sets. This means that the map must construct new aggregation objects as it gains new keys.

The other change is to make the query code call the value method of the aggregation object when it is retrieved. This allows aggregation objects to chose the value that is returned to the user, rather than always returning the aggregation object itself.

With these two changes, the method supports arbitrary aggregation operators. However, there are several aggregation operators that we will discuss in greater detail, as we have used them in experiments.

Set. The first aggregation operator is 'set'. This is the reference-counted set that we use when performing object-set queries that do not support aggregation. The constructor creates a new reference-counted set, while the add and remove operators add and remove elements from the set, respecting the reference counts of the elements. Finally, the value method simply returns the set itself.

Sum. Another aggregation operator is 'sum'. This sums up the result of a query. Note that nothing forms a set before aggregation, so this will sum up the value of the result expressions as applied to all bindings that match the clauses in the query.

The implementation of the sum aggregation operator is straightforward. The constructor creates a new object with a total field set to 0. The add method adds the value of the result expression to the total, while the remove method subtracts it from the total. The value method returns the total as the result of the aggregation.

Note that the version of sum described works best in dynamic languages which support automatic promotion of numeric types. In more static languages, it may become necessary to have multiple versions of the sum operation, each supporting a different type.

The sum operation can also be used to implement counting, by applying it to a result expression consisting of the number 1. This means that queries that compute averages can be expressed entirely with sum. For example, the query to compute the average salary of professors can be written as:

```
1  professors ->
2  { sum(p.salary) / sum(1) | p in professors }
```

Count. Since many query languages support a count operation that counts the things, we suggest that the count aggregation operation be the equivalent of sum(1).

Minimum. The last interesting aggregation operator we present is 'min', which maintains the minimum value of a selected object. Min can be implemented by maintaining a tree of values. The add and remove operations add and remove values from the tree, while the value operation gets the minimum value in the tree. This allows the minimum value to be incrementally maintained. (Maximum can be implemented similarly, as can the median value.)

Unlike the other aggregation operations, incrementally maintaining the minimum requires O(lg n) time for each element of the differential binding set (D) produced during an update. Retrieving the result can still be done in O(1) time. While the other aggregation operators take O(1) time, we do not believe that allowing the running time to differ based on the aggregation operator is

a problem. Rather, it is something the programmer must be aware of, much as he must be aware of it when writing incremental code himself. Allowing aggregation operators with differing costs in time and space is necessary for allowing the automatic incrementalization of a wide range of problems.

User-defined aggregation operations. Our system allows us to automatically incrementalize any form of aggregation for which it's possible to write an aggregation operator— that is, where it's possible to independently give what happens when elements are added and removed. We believe that writing the small amount of incremental code required for each type of aggregation is much easier than incrementalizing queries themselves, making our system suitable even for cases where appropriate aggregation operators have yet to be defined. What's more, aggregation operators, being mere object constructors, lend themselves to inclusion in a library, making it easy for systems implementing our method to grow.

4.7.5 Grouping

Another operation that is important in queries is grouping, also known as nesting. In our query language, this operation produces a map from the value of a grouping expression, to an aggregation produced from the bindings with that value for the grouping expression.

We extend our syntax for aggregation by adding "@" as the group operator. This operator separates the aggregation expression from a group expression. We allow multiple grouping expressions to be specified, in which case they are processed from left to right, with the leftmost expression corresponding to the outermost nested map. For example, the following query

```
1  professors ->
2  { sum(p.salary) / sum(1) @ p.department @ p.rank
3  | p in professors }
```

first groups professors by department, then by rank. The average salary is computed for the professor in each group.

This query allows the average salary of full computer science professors to be retrieved using an expression like:

```
1  averagesalary.get('CS').get('full')
```

where *averagesalary* is the name of the variable containing the result of the query.

Incremental maintenance. Incrementally maintaining the result of queries using grouping requires a small extension to our method. Instead of adding or removing result expressions to aggregation objects directly, we first evaluate the grouping expressions, and use them to look up the appropriate aggregation objects, creating the aggregation objects if necessary. Once the aggregation objects are known, we add or remove the values of their result expressions, as appropriate.

For our grouped professor salary query, the code to maintain the result map, run in response to an add of an element to the professors set, is as follows:

```
1  for professors, p, p_department, p_rank, p_salary in D:
2      map_1 = R.get(professors)
3      map_2 = map_1.get(p_department)
4      agg_1, agg_2 = map_2.get(p_rank)
5      agg_1.add(p_salary)
6      agg_2.add(1)
```

This assumes that when a missing key is retrieved from R or map_1, a new map is created and associated with that key. When a missing key is retrieved from map_2, a tuple containing two sum new objects (corresponding to the two aggregation operators from the) is created and associated with that key.

When values are retrieved from these maps by user code, the value methods must be called on the aggregation objects, and then the aggregation expression evaluated with these values. For our example, this means that we would want the user-level expression:

```
1  result.get(department).get(role)
```

to

execute the code:

```
( r.get(department).get(role][0].value() /
  r.get(department).get(role)[1].value() )
```

where *result* is the object returned from evaluating the query, and r is the value of $R[professors]$ at query evaluation time. This can be accomplished by making result a wrapper object, or alternatively by making r and *result* the same object, but using a different set of methods to access the aggregation objects directly from inside maintenance code.

Grouping versus Constrained Parameters. Finally, it's important to note that some of the effects of grouping can be accomplished through the use of constrained parameters. For example, to get the average salary for full professors in a given department using grouping as given above, we would need to write:

```
salary = { professors ->
             ( sum(p.salary) / sum(1) @ p.department @ p.rank
             | p in professors )
           }.get(department).get('full')
```

using unconstrained parameters, one could instead write:

```
salary = { professors, department ->
             ( sum(p.salary) / sum(1)
             | p in professors, p.department == department,
               p.rank == 'full')
           }
```

Neither version of the query has an advantage in running time, as making *department* a constrained variable ensures that the query is maintained for all possible values of *department*, and that results can be retrieved in constant time.

4.7.6 Set Union

The final extension to our method is a technique for incrementalizing set union, the last remaining piece of the OQL algebra. We perform set union by

noting that the set union operation:

```
1    S union T
```

is equivalent to the object-set query

```
1    ST -> a : a in b; b in ST
```

where

```
1    ST = S, T
```

To allow reuse of this query, it's necessary that this transformation always return the same set ST for a given pair of sets S and T. This can be done by looking the sets up, by object identity, in a map.

While it's not strictly necessary to generate code for changes to ST, such code does not hurt anything, since no maps are required to handle these updates.

4.7.7 Redundant Variable Elimination

An optimization to our method is the elimination of redundant variables in maintenance and query optimization code. We define a variable to be redundant if its value can be determined from that of a non-redundant variables using a constant time optimization. For object-set queries, this is only the case with field accesses. More specifically, if we have the following pair-domain enumerator:

```
1    (a, b) in field_f
```

and a is not redundant, then b is redundant. b can be determined from a by evaluating the expression $a.f$. For each redundant variable, there is an expression that can compute its value from a non-redundant variable using only a series of field accesses.

For our running example, the reundant variables and the expressions that compute their values are:

variable	expression
wifi_signals	wifi.signals
wifi_threshold	wifi.threshold
ap_ssid	ap.ssid
ap_strength	ap.strength

The remaining non-redundant variables are $wifi$ and ap.

We can improve program performance by only including non-redundant variables into the differential assignment set. When the result is computed using these non-redundant values, uses of redundant variables can be replaced with expressions that compute their values.

The following block of query update code represents assignments as tuples containing the values of the non-redundant variables.

after adding ap to a set that wifi_scan refers to

```
1       ap_ssid = ap.ssid
2       ap_strength = ap.strength
3       if ap in wifi_scan:
4         for wifi in inv_scan.img(wifi_scan):
5           wifi_threshold = wifi.threshold
6           if ap_strength > wifi_threshold:
7               if (wifi) in U:
8                 D.add((wifi, ap))
```

The corresponding result set update code retrieves the values of the non-redundant variables from the differential assignment set, and uses them to compute entries that are added to the result map.

```
1   for (wifi, ap) in D:
2     R.add((wifi), ap.ssid)
3   D.clear()
```

Redundant variable elimination reduces the space needed by the differential assignment set, as smaller tuples can be placed into that set. In non-pathological

cases, it should also reduce the running time of the program, as hashing the smaller tuples takes less time, and this should make up for the time required to evaluate expressions corresponding to redundant variables.

4.8 Implementation and Experiments

To evaluate our method, we developed an implementation in Python. Our implementation consists of a single Python module, named `incr.py`. It consists of 617 lines of Python, and requires only the Python standard library to run. It provides as a public interface a single function, `run_query`, which takes as arguments a comprehension with values for its parameters, and returns the result of the query. This function first checks to see if it has encountered the query before. If not, it generates obligation and query execution code corresponding to the query, passes the parameters to the query execution code, and returns the result.

We implement obligations by creating classes. Each object in the system starts with an initial class, the class it was initially created with. For each combination of initial class and set of obligations, we create a new class that inherits from the initial class and runs the maintenance code in the obligations. When an obligation is assigned to an object, we find the class corresponding to the object's initial class and the set of obligations. We assign this new class to the object, an operation Python allows. We then run any initial code required by the obligation being assigned to the object.

Our experiments were run on a computer with an Intel Core 2 Duo processor running at 2.13GHZ. The machine has 2GB of RAM, although memory usage was not a concern in our experiments. Our experimental code is written in Python, using Python 2.4.4, running under Ubuntu Linux. All times reported are CPU usage. They do not include the time used for code generation, which was about 0.2 seconds for the running example.

All of the programs that our method has been applied to were written by us. Some may object to this, preferring that our method be applied to open source programs written by others. However, open-source programs were generally written with efficiency in mind, and the programmers generally incrementalized the repeated expensive computations by hand. Our method would

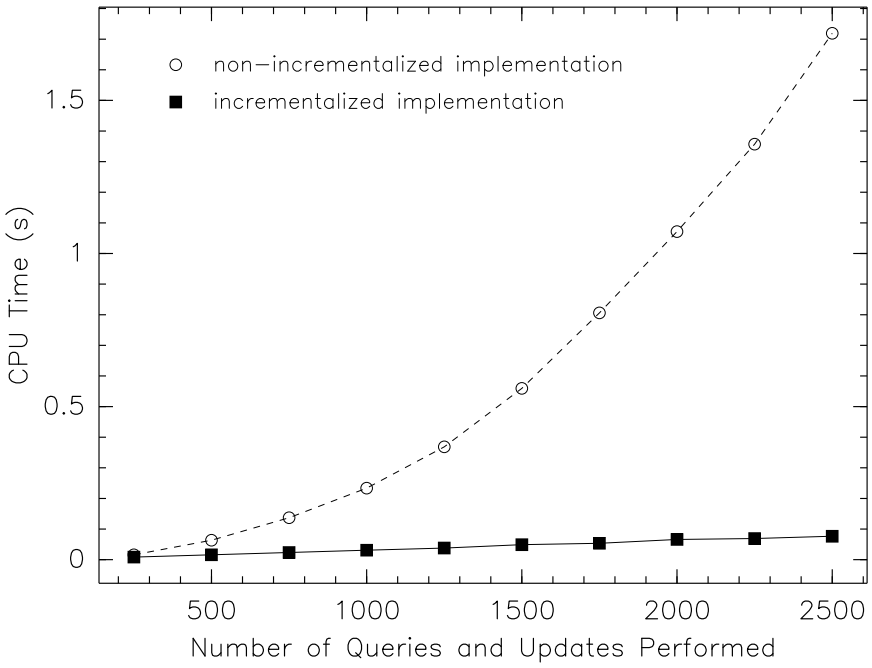

Figure 4.4: Time taken to perform a varying number of updates and queries, using our running example.

not improve the performance of such programs. Instead, our method would allow programmers to write simpler, more readable, and more maintainable programs, and perform incrementalization automatically.

4.8.1 Running Example

Our experiment consisted of creating a single $wifi$ object, and looping n times, where n varied between 250 and 2500 at intervals of 250. In each iteration loop, we create an ap object and add it to $wifi.scan$, and then perform the wifi query. All ap objects were created with $ap.signal > wifi.threshold$, so the number of objects returned from this query equals to the number of iterations through the loop. For a given n, each experiment was repeated 20 times, with the reported times being the average of 20 runs.

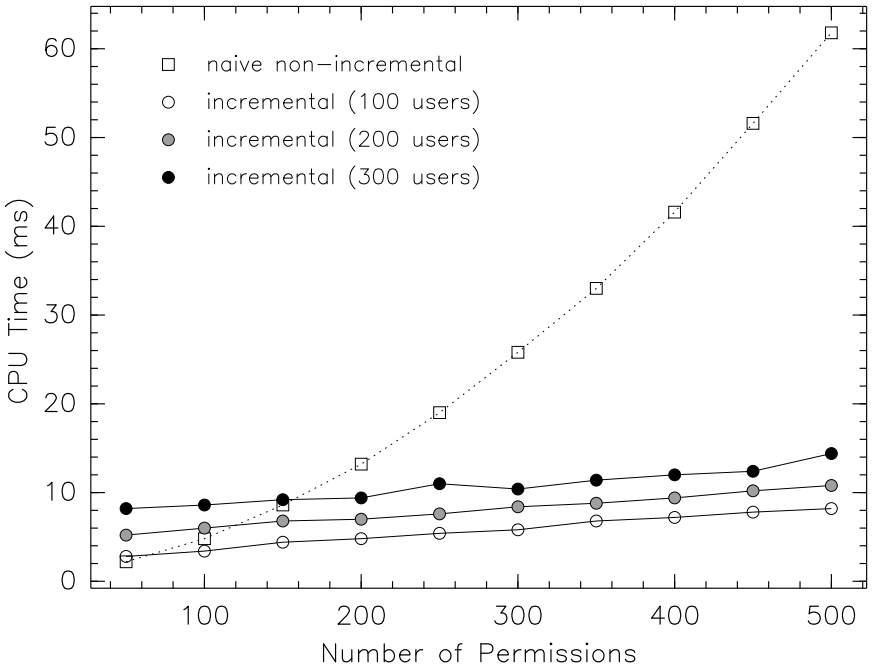

Figure 4.5: Time taken to perform a varying number of updates and queries, using our running example.

Figure 4.4 shows the results of this experiment. The incrementalized implementation is linear in the number of queries and updates performed. This is consistent with our prediction that the incrementalized code can perform the query and the update in constant time. The running time of the non-incrementalized incrementalization is quadratic, reflecting the increase in time it takes to perform a query as sets get larger.

4.8.2 Django Authentication Query

We then took the following query from the Django web framework. The result set of this query contains the names of all permissions of all groups of all given users whose user id is the given uid and that the groups are active.

```
1   users, uid ->
2    p.name : u in users, g in u.groups, p in g.perms,
3      u.id == uid, g.active
```

In this query *users* is expected to be a set of User objects. Each User object must have at least two fields: *id* containing the user id and *groups* giving the set of Group objects the user is in. Group objects also have two fields: *active*, a flag that is true if the group is active, and *perms* giving the set of permissions the group has. Permissions are represented by Permission objects, each of which must have a *name* field giving the name of that permission.

Our method predicts that running this query from scratch takes running time $O(users * groups_{user} * perms_{group})$, where $groups_{user}$ means the number of groups each user has. When the user is known, such as when adding or removing a user or updating $u.uid$ or $u.groups$, the running time is $O(groups_{user} * perms_{group})$. When the group is known, such as when added to or removed from a set or updating $g.active$ or $g.perms$, the running time is $O(users_{group} * perms_{group})$. Finally, when a permission is known, via an update to a permissions set or $p.name$, the running time is $O(groups_{perm} * users_{group})$.

The experiment we performed using the Django authentication query consists of creating a number of users sharing a single group, then adding a varying number of permissions to that group, performing a query to find the permissions granted to one of the users. This experiment was performed using both non-incremental and incremental implementations of the code. As the performance of the incrementalized implementation depends on the number of users, we varied the number of users between 100 and 300, fixing the number of users at 100 for the naive version. The number of permissions was varied between 50 and 500. For a given number of permissions the experiment was repeated 50 times, with the reported times being the average of 50 runs.

Figure 4.5 shows the running time for the naive non-incremental version is quadratic in the number of permissions created, while the running time of the incremental version is linear in both the number of permissions and the number of users. This is what is expected from our method for this experiment. We also note that the incremental implementation moves most of the running time from queries to updates, so in systems with many more queries than updates (such as the authentication system this query is based on), our method is much

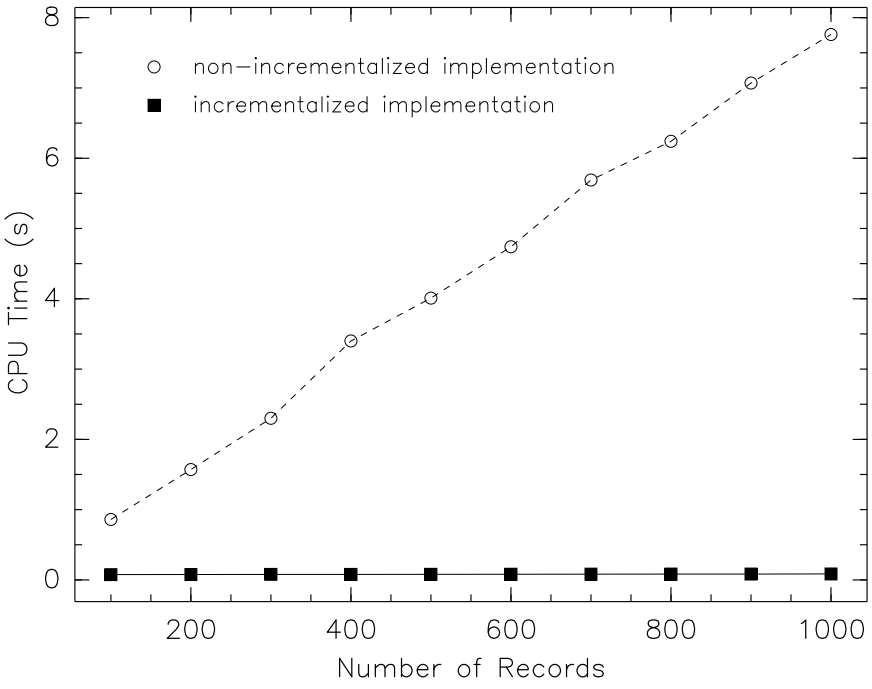

Figure 4.6: Time taken to perform 10,000 queries, for a varying number of electronic patient records.

faster than naive code.

We also ran the experiment against a Postgres database, which was far slower than our in-memory query code. For a query with 500 permissions and 100 users, the database query took 2570 milliseconds, versus 8 for the incrementalized code.

4.8.3 Electronic Health Record Policy

We have also written an object-oriented version of the United Kingdom's Electronic Health Record (EHR) service policy, based on the specification by Becker [7], and applied our automatic incrementalization method to it. The following query corresponds to a rule that determines if a clinician is authorized

to view a patient's electronic patient record into an object-set query:

```
org, cli ->
 record: tm in org.team_memberships,
         tm.cli == cli,
         tm.spcty == cli.spcty,
         record in org.records,
         record.group == tm.team
```

It finds the set of records a clinician is authorized to view, based on his team memberships and specialty.

To demonstrate how our method improves the query performance, we populated the database with a single clinician, team, and team membership, and with a varying number of records, all of which are accessible to members of the team. We then measured the time to perform 10,000 queries that determined the records accessible to the clinician.

As shown in Figure 4.6, the results of this experiment are similar to the results of the RBAC experiment. While both queries take linear time, the very low slope of the incrementalized version makes it look constant. In both EHR and RBAC, our method is able to replace expensive duplicate queries with constant-time retrievals, significantly improving program performance.

4.8.4 Student Information Management System

Finally, we have applied our method to a set of queries developed to manage the records of graduate students in our department. This system is highly object-oriented because of the temporal nature of much of the data. For example, each Student object has a programs set, which is a set of Program objects. Each Program object stores the program that the student is in (one of 'ms' or 'phd'), as well as start and end dates for the program. In this way, the history of a student can be maintained as the student proceeds through our graduate program.

This system is developed to extend an existing system, and as part of this effort, we have written a total of 54 comprehensions, most of which are queries in the original system, and the rest are queries for data conversion. We wrote each comprehension in as straightforward a manner as we could,

Query	Codegen time (s)	Updates	Lines of Code	AST Nodes
Current Students	0.43	11	526	3976
New Students	0.42	11	524	3853
Old Students	0.42	11	524	3870
Fresh Students	0.06	4	85	486
Fellowship Students	0.24	9	345	2298
TAs and Instructors	0.51	16	689	4657
TAs	0.03	3	45	237
Old TAs	0.09	5	126	813
New TAs	0.09	5	126	813
New TA Emails	0.26	11	412	2478
TA Waitlist	0.36	12	528	3423
Good TAs	0.18	8	267	1726
Qual Exam Results	0.46	13	614	4723
Advisors by Student	0.43	12	558	4236
Students w/o Advisor	0.32	10	452	3038
Advisor Overdue	0.41	11	522	3764
Prelim Exam Overdue	0.24	9	340	2274

Table 4.3: Code generation statistics for student management system.

without considering our method or any efficiency issues.

Our system is able to incrementalize 52 of the 54 queries. The two queries our method did not incrementalize involve aggregation of results, which we are yet to implement. This shows our method can handle many interesting queries.

Of the 52 queries our method can incrementalize, we identified 17 frequently used queries. Table 4.8.3 shows the statistics we collected when incrementalizing these 17 queries. For each query, we report the code generation time, the number of kinds of updates that can affect that query, and the lines of code and AST nodes generated. The number of updates approximates the number of methods that would need to be modified to run the maintenance code. It's a good approximation of the number of places in the program that need to

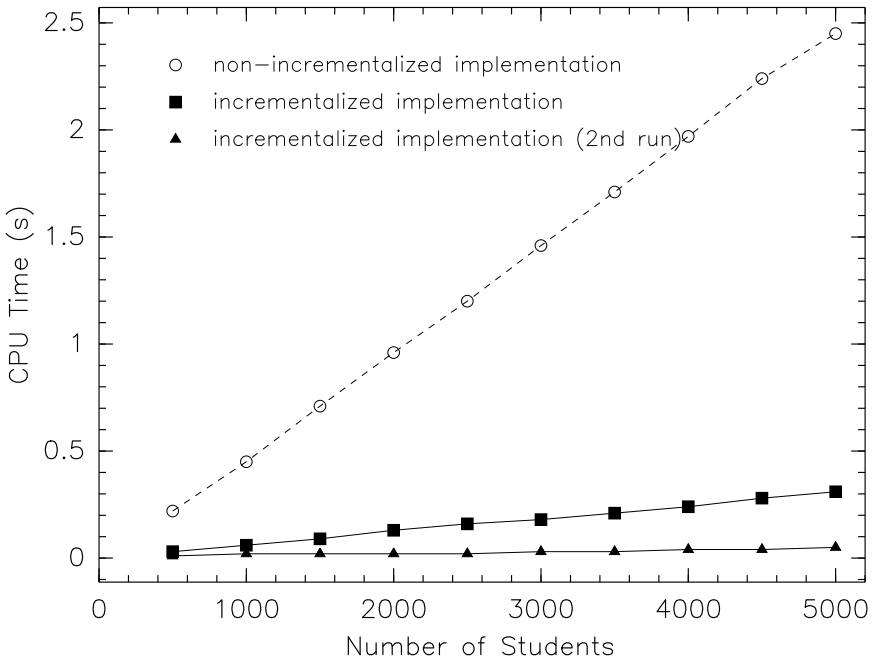

Figure 4.7: Time required for 100 student queries, for a varying number of students.

be changed to incrementalize it. All told, in less than 10 seconds we are able to generate the thousands of lines of code required to incrementalized these queries.

An example query from this system is the New Students query, which finds all students who either joined the department in a given semester or have changed program that semester. This query reads:

```
1  students, sem ->
2  s : s in students,
3        p in s.programs
4        if s.joined == sem or
5            p.start != null and p.start == sem
```

We store information about a student's program in a set of Program objects.

When a student changes program, those Program objects store the start and end dates of the old program and the start date of the new program. We use this query to experiment with changing fields. In this experiment, we created a varying number of Student objects. For each of 100 iterations, we choose a student from the set, choose one of its Program objects, and alter the starting semester. We then perform the query. Figure 4.7 shows that, as expected, the non-incremental version of the program takes linear time to run. The incremental version also takes linear time, but with a much lower slope. The bulk of the time is taken up in the initial computation of the incrementally maintained result, as the running time is constant when run a second time on the same data.

Chapter 5

Static Approaches

As currently proposed, our method is completely dynamic, requiring that obligations be inserted into objects at runtime. This is suitable for dynamic languages such as Python, but may be more difficult to accomplish in less-dynamic language such as C++, Java, and C#. For such languages, a dynamic approach would require language support for intercepting field accesses and method calls, something that can greatly slow down program execution. Generally, support for such interception is present in dynamic languages, and so a dynamic approach adds less overhead.

Statically generating incrementalization rules provides the following advantages:

1. It gives the programmer insight into the incrementalization process. Unlike the dynamic incrementalization process, the programmer can inspect the incrementalization rule, and the generated code after the incrementalization has been applied. This can give confidence in the correctness of the incrementalization, and also understanding of the performance of the generated code.

2. It can reduce the overhead of the incrementalization process. Unlike the dynamic method, there is no overhead for inserting code into the program. (But see below.)

3. Unlike the dynamic method, the static method can take advantage of

information known about the program. This will let us optimize the generate maintenance code in ways not possible with only dynamic code.

4. A static approach to generating incrementalization rules allows our method to be applied to languages which do not support the interception of method calls and field updates.

However, a static approach to incrementalization is not without downsides, however. Some potential downsides to the static method are:

1. The number of cases in which the incrementalization process can be applied is limited. Specifically, we must ensure that we can statically find all updates that can affect the incrementally maintained result. This limits the approach we can use in programs in which objects results depend on are updated by unanalyzable libraries and plug ins.

2. As there are many updates that may- but not must-alias objects that the query depends on, we may run at least some maintenance code for updates to objects that are not reachable from the query parameters. Compare this to the dynamic approach, where objects that are not reachable from query results incur no performance penalty.

In this chapter, we will discuss how to automatically generate static incrementalization rules for object-set queries. We will first describe what is required in a program transformation system that is capable of applying the rules we generate. This will be followed by two approaches for generating static rules, and a discussion of possible optimization opportunities presented by static incrementalization. Finally, we will give an experimental evaluation of the effectiveness of static and dynamic incrementalization techniques.

5.1 Program Transformation System

Rather than attempting to apply our program transformations to the program directly, we generate rules to be applied by a program transformation

system. This approach has two benefits: the rules become readable and understandable by users of the system, and much of the complexity of rule application can moved into the transformation system, rather than the incrementalization code proper. The latter is only an advantage if a system exists that can apply the generated rules. InvTS is one such system. [].

The rules our method generates are comprised of several types of clauses:

- An **inv** clause introduces the rule, giving a query to match and an expression that is used to replace that query. There can be only one **inv** clause per rule.
- A **de** clause allows code to be inserted into a scope, either in a class, written using **de in** *class*:, or in a specific module, written using plain **de global** *module*:.
- The **at** method causes code to be inserted before or after updates, into method definitions, and into property functions. Property functions are functions that are called before and after field updates. The syntax of the at function determines which form is used:
  - **at** *update* inserts code before or after an update.
  - **at method** *class.method(args)* inserts code at the start or end of a method.
  - **at property** *class.field* inserts code at the start or end of a property function that is called when a field is set. It's assumed that this property function always sets the field to the supplied value.
- The **if** *expression* clause may be a sub-clause of the **at** clause. Code is only inserted if the expression is true. The expressions is evaluated at the meta-level, using the meta-expression syntax defined below.
- The **do before** and **do after** clauses may be sub-clauses of **inv** and **at** clauses. The **do before**/**do after** clause inserts code before/after queries and updates, and at the start/end of methods or property functions.

- The **order** *number* clause can be a sub-clause of **at** clauses, and controls the order in which code is inserted. For a given rule, all code with a given order number is inserted at a given program point before any code with a higher order number is inserted at that point.

The **inv** and **at** clauses may introduce meta-variables which match expressions and types in the original program. Meta-variables introduced in an **inv** clause are scoped to the entire rule, while those introduced in an **at** clause are local to that **at** clause and its sub-clauses. Meta-variables may be used in **do** clauses, where they are expanded to their original values.

Meta-variables may also be used in meta-expressions as part of the if statements. Meta-expressions refer to three kinds of values: sets of objects, sets of types, and booleans. Meta-expressions consist of:

- $\$v$ - The set of objects that can be referred to by the expression matched by the given meta-variable.
- $e.f$ - The set of objects that can be referred to by the f field of objects contained in the set of objects e.
- $e.member$ - The set of objects that can be elements of objects in the set of objects e.
- $amay - aliasb$ - True if there exists an object in the set of objects a that may alias an object in the set of objects b.
- $amust - aliasb$ - True if every object in the set of objects a must alias every object in the set of objects b.
- $type(e)$ - The set of types of objects in the set of objects e.
- $a == b$ - True if the set of types a has a non-empty intersection with the set of types b.
- The **and** and **or** meta-operators have there conventional meaning, as does parenthesizing expressions.

Finally, if more than one **do** clause would cause code that is textually equivalent after meta-variable expansion to be inserted more than once at a single point in the program, we only insert that code a single time. This ensures that redundant code is not introduced into the program, and allows us to easily ensure that certain types of code are run only once.

5.2 Generating Static Incrementalization Rules

In order to statically incrementalize a program, we must follow a four step process. We first scan the program to find all queries that the programmer has selected for incrementalization. Next, we generate the obligation and query execution code for each query, using the technique described in the prior sections. We then transform this code into static static incrementalization rules corresponding to those queries, noting that queries with a similar enough form can be incrementalized using the same incrementalization rules. Finally, we use a program transformation system to apply the incrementalization rules to the program, yielding an incrementalized result program.

As incrementalizable queries are syntactically different from surrounding code, it's easy to scan through the source code of the program to find appropriate queries. This is made easier by the fact that the only penalty for misidentifying queries is that we will generate an incrementalization rule that cannot be applied to the program. Similarly, as we rely on an existing program transformation system, applying the rules involves invoking that system with the appropriate rule files.

The remaining operation is the generation of the static incrementalization rules. We do this by taking our technique for dynamic incrementalization, and using it to generate rules that can be applied statically. There are two approaches we can take for this generation process, distinguished by where we place the maintenance code. The internal approach places the maintenance code inside the classes of objects being updated, while the external approach places maintenance code at the location of the updates. Both have advantages and disadvantages, which will be discussed below. Much of the method is common between the internal and external approaches.

5.2.1 Commonalities

Comprehension-Local Variables. The first thing the two approaches to static incrementalization have in common is the need to be able to statically refer to the comprehension local variables: R, D, U, and the inv maps. Depending on the language, we either place these variables in a module dedicated for the purpose, or we place them as static fields of a class dedicated to this purpose. The module or class chosen should be unique to the query being incrementalized, as this ensures that the comprehension-local variables will be unique.

Knowing this, we can begin incrementalizing the running example query:

```
1  wifi ->
2   ap.ssid : ap in wifi.signals, ap.strength > wifi.threshold
```

We will be placing the comprehension global variables in the global module q, but note that this is an arbitrary choice, and should be different for other queries we incrementalize in the same program.

We can begin creating the query by introducing an **inv** clause, one that takes each parameter as a meta-variable, and replaces the query with an expression that retrieves the result from R. For the running example, we would introduce the clause:

```
1  inv q.R[$wifi] =
2        $wifi -> ap.ssid : ap in $wifi.signals,
3                            ap.strength > $wifi.threshold
```

We then introduce **de** clause, one that defines the comprehension-local variables used in this comprehension in the module q. The names of all of the comprehension-local variables can be determined from the generated obligation and query assignment code. For our running example, the comprehension-local variables are R, D, U, inv_{scan}, and $inv_{members}$. So we generate the following **de** clause.

```
1  de global q:
2      R = rc_multimap()
3      D = set()
4      U = set()
5      inv_scan = multimap()
6      inv_members = multimap()
```

If bound-unbound maps are needed for partially bound retrieval from U, those maps are also stored in the q module.

Obligation Meta-Expressions. Our next step is to determine, for each obligation, a meta-expression that computes the set of objects that can have this obligation. We use these meta-expressions to determine which classes and updates require the insertion of maintenance code. As obligations correspond to pair-domain variables, the set of objects that can have a given obligation is equivalent to the domain of that pair-domain variable.

The meta-expressions are computed in a manner similar to the set of supported variables. We start by declaring that the meta-expression for each parameter p is $\$p$. Next we find two variables, a and b, such that the meta-expression for a is known to be $expr$ and the meta-expression for b is unknown.

If a and b appear in the pair-domain selection:

```
1  (a, b) in field_f
```

then the meta-expression for b is $expr.f$. If they appear in the pair-domain selection:

```
1  (a, b) in members
```

the meta-expression should be $expr.member$. If they are not related in this way, we must pick another pair of variables. This process continues untill each pair-doman vairable, and hence obligation, is assigned a meta-expression. This process will always complete because we require that each pair-domain variable must be dependent on at least one unconstrained parameter.

In our running example, the meta-expressions corresponding to obligations are:

- wifi - $wifi

- wifi_scan - $wifi.scan
- ap - $wifi.scan.member

There is no need to compute meta-expressions for wifi_threshold, ap_strength, or ap_ssid, as there are no obligations added to objects that can be the values of these variables.

Obligation Assignment. The next thing we must do is to implement obligation tracking and assignment code. We track obligations using booleans added to the classes that contain the obligations. The boolean variables are named with the query's unique identifier, followed by the name of the obligation. For each obligation, we generate a **de** clause that adds the boolean to classes of objects in the domain of the corresponding variable.

The **de** clauses that add the booleans used by static versions of our running variables are:

```
1  de in type($wifi):
2      q_wifi = False

3  de in type($wifi.scan):
4      q_wifi_scan = False

5  de in type($wifi.scan.member):
6      q_ap = False
```

In module q, we define an obligation assignment function for each obligation. These assignment functions first check to see if an object already has an obligation. If it does, they simply return. If not, the obligation assignment code is executed. Calls to assign_obligation are rewritten to call the appropriate assignment function in q.

For our running example, the code to assign obligations is:

```
de global q:

    def assign_wifi(wifi):
        if wifi.q_wifi:
            return

        wifi.q_wifi = True

        assign_wifi_scan(wifi.scan)
        q.inv_wifi_scan.add(wifi.scan, wifi)

    def assign_wifi_scan(wifi_scan):
        if wifi_scan.q_wifi_scan:
            return

        wifi_scan.q_wifi_scan = True

        for ap in wifi_scan:
            q.assign_ap(ap)
            q.inv_members.add(ap, wifi_scan)

    def assign_ap(ap):
        if ap.q_ap:
            return
        ap.q_ap = True
```

Query Execution Code. The final type of code that is the same in both approaches to static incrementalization is the query execution code. This code must be run before the query occurs, using a **do before** clause. The only modifications needed are those required to access the variables and obligation assignment functions that are found in the comprehension specific module and class.

```
1   do before:
2     if ($wifi) not in q.U:

3       q.assign_wifi($wifi)
4       q.U.add(($wifi))

5       wifi_scan = wifi.scan
6       wifi_threshold = wifi.threshold
7       for ap in wifi_scan:
8         ap_ssid = ap.ssid
9         ap_strength = ap.strength:
10        if ap_strength > wifi_threshold:
11          if ($wifi) in q.U:
12            q.D.add(var_asgn())

13      for a in q.D:
14        q.R.add((wifi), eval(ap_ssid, a))
15      q.D.empty()
```

Placing this code in a **do before** clause ensures that the result of the query appears in $q.R$ when it is retrieved as part of the in clause.

5.2.2 Internal Approach

The internal approach is one of two approaches to the placement of maintenance code. In the internal approach, maintenance code is placed inside the classes of objects that are being updated, and run when methods or properties on those objects are called. This has the advantage of minimizing code duplication. It is also somewhat robust in the presence of libraries and plugins, since as long as the objects are updated using the supplied methods and properties, incremental maintenance can be performed correctly. The internal approach can determine where to insert the code using only type information.

The internal approach has some downsides, however. A small amount of inserted code runs whenever the appropriate update is performed by an object of the give class, even if there is no chance that object may be given the

corresponding obligation. For classes with objects that participate in multiple queries, such as the set class, this overhead may prove substantial. A second problem is that this approach demands that fields used in queries be represented as properties, which may not be supported in some languages, and may be an incompatible API change in others.

The first thing we must do to support the internal approach is to modify the add and remove methods of the set object so that our methods are only called when an object is actually being added or removed from the set. This code needs to be run before the set is actuall modified, which means that it is the highest-priority code the approach introduces. In the add and remove methods of the set class, we introduced the prefixed variable *q_changed*, which stores the fact that the set has been changed. A simple optimization would be to share a single *changed* variable for all static incrementalization. The clauses we introduce are:

```
1  at method set.add($s, $v):
2  order 1
3  do before:
4      q_changed = $v not in $s

5  at method set.remove($s, $v):
6  order 1
7  do before:
8      q_changed = $v in $s
```

With the *q_changed* variable present, we are able to generate an **at** clause for each block of code present in the obligations generated from a query. For each **at** clause, we also generate subordinate **if**, **order**, and **do** clauses. This generation can be done mechanically, as for each combination of update and code type, there's a single combination of clauses to be generated.

The **at** and **if** and **do** clauses are generated from the kind of update, according to the following rules. Note that in this table, a and b are the variables used in the obligations, f is a field, $\$a$ and $\$b$ are the corresponding meta-variables, and *expr_a* is the meta-expression determined for a. The meta-variables should not be the same as meta-variables used for parameters.

update	at clause	if clause	do clause
a add b	at method set.add($a, $b)	if True	do after
a remove b	at method set.remove($a, $b)	if True	do before
a.f assign	at property $a.f	if $a = type($expr_a$)	do after
a.f deassign	at property $a.f	if $a = type($expr_a$)	do before

The order clause is determined by a combination of the kind of update and the purpose of the code. The order numbers have been chosen so that code of a given kind executes in the same order it would in our dynamic method. This ensures that the differential assignment set is always computed before the result map is ever updated, and that the inverse maps are maintained before an addition/assignment but after a remove/deassignment.

purpose	add/assign	remove/deassign
inverse map maintenance	2	4
D computation	3	2
R update	4	3

When multiple at clauses insert code at the same point in the program, we expect that all code with a given order number will be inserted before any code with a higher order number.

Finally, we must give the code contained within the **do** clause. This code begins with a conditional that checks to see if the object being updated has been assigned the appropriate obligation. In the case of an add or remove update, the conditional also check to see if the set is being changed. As an example, we assume that the code has been inserted into object $\$a$, which may have obligation a, and that the unique identifier is q. For the add and remove updates, we generate the code:

```
1    if q_changed and $a.q_a:
2        ...
```

As the deassign and assign updates always occur, even if they do not change program state, it is only necessary to check the obligation:

```
1    if $a.q_a:
2        ...
```

The block of code is then inserted under this clause, substituting meta-variables

$a and $b for variables a and b, and replacing the comprehension-local variables with their equivalents in the q namespace.

As an example of this, take the differential assignment set computation code that runs when ap is added to a set that wifi_scan refers to:

```
1       ap_ssid = ap.ssid
2       ap_strength = ap.strength
3       if ap in wifi_scan:
4         for wifi in inv_scan.get(wifi_scan):
5           wifi_threshold = wifi.threshold
6           if ap_strength > wifi_threshold:
7               if (wifi) in U:
8                 D.add(var_asgn())
```

From the fact that this corresponds to an add update, we can determine the **at** and **if** clauses, and the fact that we need to have a **do after** clause. Knowing that it's a differential assignment computation code belonging to an add operation, we can determine that the **order** number is 3. So the final code is:

```
1   at method set.add($wifi_scan, $ap)
2   if True
3   order 3
4   do after:

5     ap_ssid = $ap.ssid
6     ap_strength = $ap.strength
7     if $ap in $wifi_scan:
8       for wifi in inv_scan.img($wifi_scan):
9         wifi_threshold = wifi.threshold
10        if ap_strength > wifi_threshold:
11            if (wifi) in U:
12              D.add(var_asgn())
```

Repeating this for every block of code in every obligation yields the complete internal static transformation rule.

5.2.3 External Approach

A second approach to the placement of maintenance code is the external approach, which places the maintenance code at the site the updates themsevels, rather than inside the classes of objects or sets that are being updated. The advantages and disadvantages of this approach mirror those of the internal approach. The primary advantage is that the internal approach uses may-alias information to only insert code at updates that potentially affect queries. While not as accurate as the dynamic method, using the external approach in conjunction with accurate must-alias information can reduce the cost of updating objects of classes that are shared between multiple queries.

A second benefit of the external approach is that it does not change the classes in incompatible ways, as the internal approach does. However, the external approach has the downside that it requires that code be inserted at any point in the program where a relevant update occurs, which in turns means that libraries and plug-ins must be analyzed and modified if they update objects involved in the query. This limits the usefulness of not modifying the API of the class itself.

The external approach requires that a conservative may-alias analysis be performed on the program. The conservativeness needs to be in the direction of over-estimating the may-alias relation. If a and b are expressions, then a must-alias b must be true if at least one object can be in the domain of both a and b. Overestimation will reduce performance as obligaitons will need to be checked unnecessarily, but underestimation will cause the application of the rule to be incorrect.

As with the internal approach, the first step we have is that we need to make available information about if a change actually occured to the set or map. We store this information in the *q_changed* variable, as we do in the internal approach. For every obligation a such that code for the update a add b exists, we generate the rule:

```
1  at method $s.add($v)
2  if $s may-alias expr_a
3  order 1
4  do before:
5       q_changed = $v not in $s
```

where $expr_a$ is the meta-expression corresponding to a. If the update a remove b exists, we generate the code:

```
1  at method $s.remove($v)
2  if $s may-alias expr_a
3  order 1
4  do before:
5       q_changed = $v in $s
```

Note that a single update may match under multiple obligations. The same code is generated for each match, so the program transformation system will only insert this check once.

The external approach mechanically generates **at**, **if**, **order**, and **do** clauses for each block of code found in the obligation, similar to the way the internal approach does it. We change the **at** and **if** clauses to match the updates, rather than the methods implementing updates. The new clauses are:

update	at clause	if clause	do clause
a add b	at $a.add($b')	if $\$amay - aliasexpr_a$	do after
a remove b	at $a.remove($b)	if $\$amay - aliasexpr_a$	do before
a.f assign	at $a.f = $ignored	if $\$amay - aliasexpr_a$	do after
a.f deassign	at $a.f = $ignored	if $\$a'may - aliasexpr_a$	do before

Note that $\$a$, $\$b$, and $\$ignored$ should all be fresh meta-variables that are not query parameters, this ensures that the at clause and the inv clause share no variable, save those used in $expr_a$.

The rest of the process is the same as with the internal approach. The order number is the same, the code generated is enclosed in the same conditional, and meta-variables are substituted for variables in the same way. The external static version of our sample block of code is:

```
1   at $wifi_scan.add($ap)
2   if $wifi_scan may-alias $ap.wifi_scan
3   order 3
4   do after:

5     ap_ssid = $ap.ssid
6     ap_strength = $ap.strength
7     if $ap in $wifi_scan:
8       for wifi in inv_scan.img($wifi_scan):
9         wifi_threshold = wifi.threshold
10        if ap_strength > wifi_threshold:
11            if (wifi) in U:
12              D.add(var_asgn())
```

5.3 Static Optimizations

There are several optimizations we can perform that improve the performance of the external approach to the static method. In this section, we present three such optimizations.

5.3.1 Inverse Map Elimination

Our methods, both static and dynamic, generate maintenance code for all possible programs that contain the query. However, the static method eventually applies rules to a single program.

It's possible to analyze the result of a rule application to see which inverse maps are actually added to the program. If an inverse map is never actually used by the transformed program, there's no need to maintain that inverse map. This lets us safely remove all code that maintains that map from the transformation rule.

For this purpose, we consider a map to be used if and only if at least one lookup occurs. We do not consider the addition or removal of mappings when determining if a map is used, as these additions and removals serve no purpose if the map is not later accessed. This can be considered to be a form of dead-

code elimination that takes into account the nature of the high-level map data type.

This optimization can reduce the running time of the program by a constant amount of time per update, as that's the cost of maintaining the unnecessary map. Our methods impose a constant amount of space overhead on the program, applying this optimization reduces that constant factor.

5.3.2 Obligation Check Elimination

A second optimization we propose the use of must-alias information to eliminate the overhead involved with checking that obligations are assigned to objects. We can perform obligation check elimination on obligations for which the following condition is true:

- When $\$a$ is the object being updated, and *expr* is the meta-expression that corresponds to the obligation, we require that at all updates which run mainteance code, if $\$amay - aliasexpr$, then $\$amust - aliasexpr$.

Note that this condition is trivially satisfied in the case where there are no updates that cause the maintenance code associated with a given obligation to be run.

When this conditions is the case, we can eliminate the need to perform obligation assignment. This is because we know that all objects that can have the obligation must have the obligation, eliminating the need to check, and the need to maintain the information required to check.

This is done in three steps:

- Eliminate the the definition of the q_a field in classes that are being updated.
- Eliminate the code that sets q_a to true in the $assign\_a$ method.
- Eliminate the check of $a.q\_a$ before each block of code generated from obligation a.

When no code exists in an obligation assignment method $assign\_a$, that method can be eliminated. To do so, remove the definition of the method,

and all code that calls it. When removing a method call removes the last remaining code in another obligation assignment method, that method should be removed as well.

This optimization simplifies the generated code and reduces the running time by a constant amount of time per update.

5.3.3 Known Parameters Optimization

Our next optimization applies to queries for which, at any given point in time, there are a limited number of combinations of unconstrained parameters for which we are maintaining the query. When the number of combinations of unconstrained parameters is small enough that it can be considered constant, we can take advantage of this to reduce query cost, and eliminate the maintenance of many inverse maps. In the case of our running example, this optimization can eliminate all use of inverse maps.

The known parameters optimization assumes that U is kept up to date, and that it contains only the combinations of the values of unconstrained parameters for which we are interested in maintaining queries. When we are no longer interested in incrementally maintaining query results for a given combination of query parameters, we remove that combination for U. We furthermore remove from R results corresponding to all combinations of parameters which we are no longer incrementally maintaining results.

Being able to treat U as a set with constant size means that we are able to place for-loops over it much earlier in the nesting order. Instead of being one of the last for-loops to be iterated over, the accesses to U will be one of the first. For two of the updates affecting our example, we generate the following pair-domain differential binding set computation code.

```
1   static wifi_signals add ap:

2   for wifi in U:
3     for wifi, wifi_signals in field_signals
4       for wifi, wifi_threshold in field_threshold
5         for ap, ap_ssid in field_ssid
6           for ap, ap_strength in field_strength
7             if ap_strength > wifi_threshold
8               D.add(assignment())

9   static ap.strength assign:

10  for wifi in U:
11    for wifi, wifi_signals in field_signals
12      for wifi, wifi_threshold in field_threshold
13        for ap, ap_ssid in field_ssid
14          for ap, wifi_signals in members,
15            if ap_strength > wifi_threshold:
16              D.add(assignment())
```

The advantages of the new nesting we've chosen appears when we translate back to the object domain. The object-domain differential binding set computation code is:

```
1   static wifi_signals add ap:

2   for wifi in U:
3     wifi_signals = wifi.signals
4     wifi_threshold = wifi.threshold
5     ap_ssid = ap.ssid
6     ap_strength = ap.strength
7     if ap_strength > wifi_threshold:
8       D.add(assignment())

9   static ap.strength assign:

10  for wifi in U:
11    wifi_signals = wifi.signals
12    wifi_threshold = wifi.threshold
13    ap_ssid = ap.ssid
14    ap_strength = ap.strength
15    if ap in wifi_signals:
16      if ap_strength > wifi_threshold:
17        D.add(assignment())
```

This code has several advantages over incrementalization that does not consider known parameters. First, note that when the size of U is small enough to be considered constant, which is the premise for this optimization, these blocks of maintenance code run in constant time. Furthermore, these blocks of code do not use any of the inverse maps. When the known parameters optimization is applied to the running example query, we do not generate any blocks of code that use inverse maps. This means that we no longer need to maintain any inverse maps, allowing us to reduce the running time and shrink the space usage of incrementalized programs.

5.4 Experiments

To compare the overhead of static and dynamic incrementalization, we used both methods to incrementalize one of the queries found in the ANSI

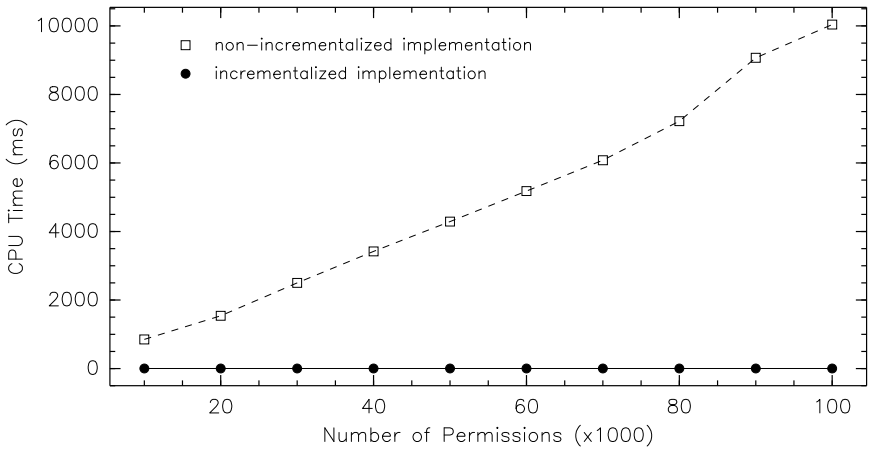

Figure 5.1: Time taken to perform 100 RBAC UserOperationsOnObject queries, for a varying number of permissions.

Role-Based Access Control standard. By comparing the time required for queries and updates under each variant of incrementalization, we are able to give guidance as to when to use each approach.

Role-Based Access Control (RBAC) [17, 6] is an ANSI standard for controlling access to operations on objects. Core RBAC controls access by assigning permissions to perform operations on objects to roles, and then assigning users to those roles. To demonstrate how our method can be used to simplify an implementation of Core RBAC, we created an object-oriented variation, OO-RBAC. In OO-RBAC, users, roles, sessions, operations, objects, and permissions are all implemented as objects. The role to permission and user to role assignment multimaps are implemented as fields on the user and role objects, respectively.

Our method was used to automatically create incremental implementations of the Core RBAC review functions. One such function is UserOperationsOnObject: given a user and an object, it returns the operations the user can perform on that object:

```
1  rbac, user, object ->
2   perm.operation : user in rbac.users,
3                    role in user.roles,
4                    perm in role.perms,
5                    perm.object == object
```

The *rbac* parameter and the first clause make *user* and *object* constrained parameters, rather than unconstrained parameters. This allows us to perform repeated queries involving the same *rbac* in constant time.

There are 11 possible updates that can affect the result of this comprehension: assigning to 5 fields, adding to 3 sets, and removing from 3 sets. Of these, OO-RBAC only uses the 6 set update operations. As each of these set updates corresponds to an administrative function, compared to a hand-incrementalized approach the use of our method lets us remove code from these 6 functions and centralize it in a single query.

To show how incrementalization improves this query performance, we populated the RBAC database with 1 user, 1 role (assigned to the user), and a varying number of permissions (all assigned to the role), with each permission allowing a different operation on the object. We then timed how long it took to perform 100 queries. Since all of the incremental versions use identical query evaluation code, we only present a single incremental case.

Figure 5.1 shows the results of this experiment. The non-incrementalized implementation obviously takes linear time, as each query needs to access every permission. Our incrementalized implementation also takes a constant amount of time to perform these queries. This gives substantial time savings when number of objects is large. For the largest case considered (100,000 operations), each query takes only 10 microseconds, versus 100 milliseconds for the non-incremental approach.

This decrease in query time is paid for by an increased cost of updates. For this query, adding a new permission to a role takes time proportional to the number of users assigned that role. This is exactly the size of the change to the result set. The precise overhead depends on the approach used to invoke the maintence code when the update occurs.

Figure 5.2 shows the time taken to insert a varying number of permissions into a role involved with a query. Since we're plotting the total time required to

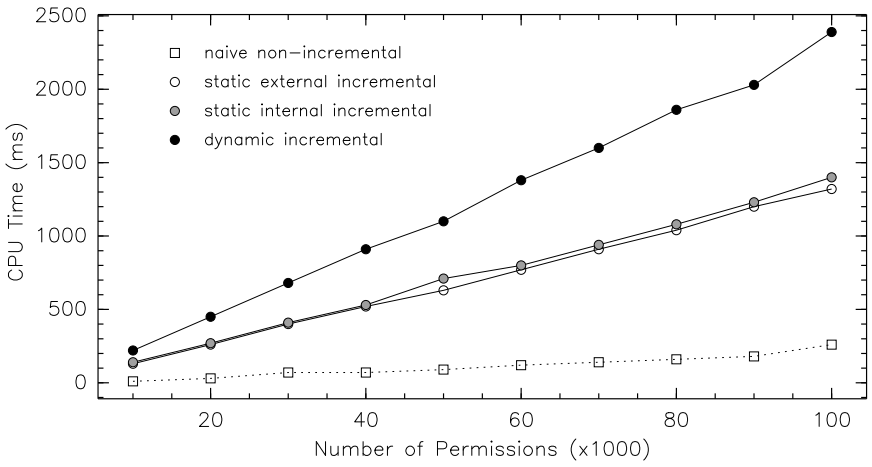

Figure 5.2: Time taken to add n permissions to a role that is involved with a query.

add a varying number of permissions to a role, all variants take a linear amount of time to complete. Each insertion takes a constant amount of time (assuming the number of users involved is fixed), with the constant revealed by the slope of that method's line. The non-incremental implemention is fastest, as it executes no additional code for an update. The two static approaches are faster than the dynamic approach, with the external static approach performing slightly but consistently better than the internal static approach. Even the dynamic approach takes less than 2.5 seconds to perform all 100,000 additions, which means that an incremental program will win out provided there is at least 1 query for every 4,000 updates.

The amount of overhead incurred by an incrementalization approach can vary depending on which queries actually occur during program execution. To evaluate this, we experimented with programs in which the query may, but not must, occur. (For example, whether the query occurs can be controlled by a command line argument.) Figure 5.3 shows the result of this experiment. The two static approaches incur overhead even when the query does not occur, as they require inserting code at places in the program that can (during any possible execution) affect a query result, even if that update cannot affect a

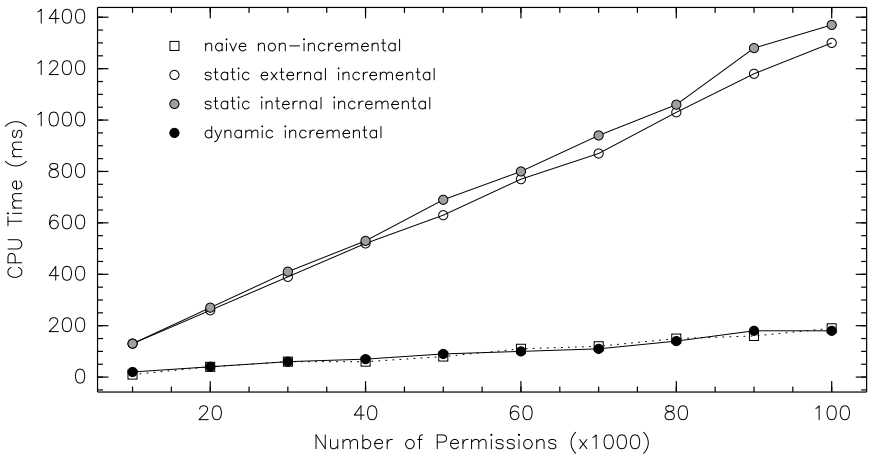

Figure 5.3: Time taken to add n permissions to a role that is involved with a query, when no queries occur.

result during a particular execution. The dynamic approach, while still somewhat of an over-approximation, is much more precise than dynamic methods, and so the time taken for insertions when no query occurs is identical to the non-incrementalized case.

Each approach has its own benefits and drawbacks. A static external approach requires all libraries and plug-ins that can change the query result be present at code-generation time, but yields the fastest code when every object can affect a query result. The static internal approach has the advantage of only modifying the classes of objects involved in the query, while leaving code that merely updates those classes unchanged. Both static approaches are suited to languages like C++, Java, and C# where changing classes is difficult or impossible. Finally, the dynamic approach requires no modification to the code, allows for runtime code-generation, and adds no overhead to objects that are not reachable from queries. The dynamic approach is therefore suitable for use in dynamic languages such as Python, Perl, PHP, and Ruby, when only a fraction of updates can affect query results.

Chapter 6

Related Work and Conclusion

6.1 Tuple Pattern Based Retrieval

Tuple pattern based retrieval is related to work in a number of fields of computer science. Since it involves querying data, it is related to databases. As a programming language construct, it is related to programming languages. It is also related to the tuple spaces used in distributed programming, and to indexing in Prolog. Lastly, our work can be considered to be in the area of data structure selection.

When working with sets of tuples, an obvious comparison is with relational databases. Tuple pattern based retrieval can be considered a restricted form of the select operation found in relational algebra. By focusing on only one operation, we gain a number of advantages over relational databases, which support more complicated queries. One advantage is that our query syntax is much more succinct than that of embedded SQL, and fits more naturally into programming languages. A second advantage is that we do not require a RDBMS, with the expense (in code size, running time, and occasionally currency) that implies. Finally, because of the low overhead of performing tuple patten based retrieval, it can be used in places where a database query would not be, such as the inner loop of the graph reachability example.

That said, there is much to be learned from relational databases. This paper leaves the query optimization possible with relational algebra up to the user. It is possible to automate such optimizations, even without the

information about set size that is known to a database query optimizer. In [40], it is shown how some relational queries can be translated into efficient code using tuple pattern based retrieval. A second issue is that we maintain bound-unbound maps, which are in many ways equivalent to indices in databases, for every tuple pattern based retrieval in the program. An area of research in databases is automatically determining which indices most benefit query performance [15, 20]. We may use similar methods to determine which bound-unbound maps most improve program performance. On a memory-limited system, when trading speed for memory we want to ensure that we make the best trade possible.

Moving on to programming languages, we should note that quite a few languages have support for tuple patterns. These include the ML family of languages, where there has been some work done on optimizing pattern matching [18], and dynamic languages such as Python and Perl. In these languages, pattern matching is against a single value, rather then a set of values. This disallows the asymptotic improvements we achieve by only retrieving matching values from a set.

The languages that we have found that contain the closest analog to tuple pattern based retrieval are Linda [12] and its successors, such as TSpaces [56]. They provide a simple model for distributed computing by providing shared tuple spaces, which are sets of tuples that can be distributed among multiple computers. Tuples in a space can be matched by providing the values of some of the fields, as in our tuple patterns. There is a difference in focus between Linda systems, which support distributed retrieval from a relatively small number of tuple spaces, and tuple pattern based retrieval, which provides fast centralized retrieval from a potentially large number of sets. Descriptions of Linda-like systems (such as those in [55]) focus primarily on retrieving a tuple from an appropriate distributed node, and do not address the problem of efficiently finding a tuple once that node has been found. In this way, we complement the work they have done.

One strategy that Prolog implementations use is to index facts to eliminate impossible unifications. By replacing sets of tuples with facts, and replacing matching with unification, our bound-unbound maps can be seen to accomplish a similar purpose as these Prolog indexes. More specifically, by indexing on all expressions, our method is similar to multiple argument indexing (also

called multiple position indexing), as found in [51, 10, 16]. Many Prolog systems do not support multiple argument indexing, instead indexing on only a single argument per fact. Systems that support multiple argument indexing either require the user to declare indices explicitly, or only generate indices in conjunction with other optimizations. Our method determines indices automatically from user-supplied code, even without analyzing the entire program.

Our work can be considered a case of data structure selection, and as such owes much to the pioneering data structure selection work performed with the SETL programming language [48, 45, 46, 21, 11]. Our work extends theirs by providing support for sets of arbitrary-length tuples, instead of sets of pairs, and by providing a syntax that allows us to use expressions to match any component of a tuple, rather than just the first component of a pair.

Finally, our method for incrementally maintaining bound-unbound maps is inspired by previous work in the area of incrementalization [39, 50], also known as finite differencing.

6.2 Object-Set Queries

Incrementalization of programs has been a subject of much research, and automatic incrementalization techniques has been developed for queries in many areas. Our method improves over previous methods in three main respects, putting aside many finer distinctions.

- Our method handles a query as a whole, rather then decomposing it into smaller queries that need to be maintained independently, which has additional cost in time and memory.
- Our method incrementalizes object-oriented programs, while previous work focused either on only sets and tuples, forced the user to decompose queries into recursive functions.
- Our method is compatible with the representations of objects in traditional (Java, C#, C++) and dynamic (Python, Perl, Ruby, PHP) object-oriented programming languages.

As our incrementalization method is focused on queries in object-oriented programs, this discussion of related work will be confined to systems that allow one to perform queries involving a comprehension over some combination of objects, sets, maps, and tuples. There are many techniques that incrementalize queries over other domains, such as functional programs [57, 35, 2, 1], logic programs [44, 33], and XML transformations [38]. There are also many cases where incrementalization was used to make programs more efficent without involving a general technique. We have not attempted to catalog all such cases, as the author encounters them all the time.

Early work with programming languages. Early work in this area was intended to provide a way of performing strength reduction on sets and maps [27, 19], ultimately yielding Paige and Sharir's finite differencing method [39, 50]. This work, while automatic, only considered sets and pairs, and decomposed queries to incrementalize them. Finite differencing also requires finite differencing rules to be given manually, while for a large class of queries our method derives those rules automatically.

Work involving databases. In the database area, finite differencing was applied to the problems of integrity constraints [28, 43] and incremental view maintenance [42, 24] in databases containing sets of tuples. It has also been extended to support views with duplicates [23], also known as bags. Again, this work assumes a relational model, where all sets are known in advance. Finite differencing also requires queries to be decomposed before incrementalizing them, which may require the storage of unnecessary intermediate results.

The basis of Ceri and Windom's technique for incremental view maintenance [13] is computing the updates to a materialized view by performing a query over the set with the parameters to the update bound. This is the method we use to compute the differential binding set, in the pair domain. They rely on the underlying database engine to compute the results of this query, whereas we generate code that directly computes the differential binding set, D. Due the fundamental assumptions of the relational model, Ceri and Windom's work assumes an environment in which all the sets are known. This makes aliasing impossible, and prevents one from having to deal with nested sets. By contrast, our method deals with arbitrarily nested sets.

The Gupta-Mumick-Subrahmanian method [26] (elaborated on in [25]) requires that queries be expressed as datalog rules. For each fact in each rule, they compute the changes to the stored reference-counted result, and use this to update the result. While it seems that it would be possible to build an object-oriented incrementalization technique around this method, it would require developing techniques for transforming object-oriented queries into datalog rules, and then transforming the resulting rules back into object-oriented code. For rules involving arbitrarily nested sets, we would also need to develop a technique for representing sets sets that doesn't encode a set as a datalog predicate. We believe that developing a technique similar to Ceri and Windom's was the simpler approach.

Several techniques have been proposed for incremental view maintenance in object-oriented databases.

Alhajj[4, 3] gives a technique that can only use objects from a single class in a query. This prevents their technique from being used to incrementalize more complicated queries, the queries that can most benefit from incrementalization.

Zhou et al. [58], Gluche et al. [22], Kuno et al. [29] and Ali et al. [5] decompose OQL queries into execution plans, and incrementalize at the plan level. At each level, they make query execution incremental. Many of their incrementalization rules require materializing parts of the query other than the final result. In our method, outside of maintenance code the only query results we store are either proportional to the size of the stored results (R, and U in pathological cases), or proportional to the size of program data (the various inverse maps). By contrast, these methods may all maintain large intermediate results for long periods of time.

Nakamura's work [37] requires that objects be transformed into a novel representation that allows finite differencing techniques that work on sets and tuples to be used. This representation is not suitable for the efficient execution of object-oriented programs.

Finally, we'll note that any technique requiring interaction with a database brings with it some drawbacks. While incremental view maintenance is a part of large database mangement systems such as Oracle and DB2 [8, 52], it is missing from popular open-source databases such as PostgrSQL and MySQL. Requring this feature to be present in the database limits the databases a program can run with. Using an external database requires that queries be

serialized and results unserialized, often with context switches and network latencies adding additional delay. Even if embedded databases supporting this technique, there would be some overhead as objects are converted between database and program representations. Allowing for incrementalization at the programming language level gives the programmer a choice as to how he incrementalizes his specific application.

Recent work with object-oriented programs. There has been some very recent work on incrementalizing object-oriented programs. The ditto system [49] incrementalizes queries that consist of recursive java methods, using a method similar to that used by Acar to incrementalize functional programs [2, 1]. This method works by memoizing the results of methods, and updating the stored result when a field accessed by the method changes, or the result of a method called by that method changes. Recomputation is done by re-running the method, hence ditto has a method-level granularity for incrementalization. Where our method uses a high-level query construct for querying, ditto requires the programmer to write recursive method calls. Ditto also restricts the result of queries to primitive types, while our method allows the result of queries to be method calls.

The research most similar to ours is the incrementalization for the JQL system [53, 54]. The incremental update code for JQL works by creating tuples of objects based on changes to sets. This works for JQL as all objects used by an incrementally maintained query must be contained in a set given as a query parameter. The class of queries our method supports is larger than that of JQL, as we support sets and objects that refer to sets, and incremental maintenance involving updates to fields of objects that are referred to by fields of other objects. Finally, the lack of result expressions in their language prevents aliasing from occurring, but may supply the programmer with the same object more than once. Our method includes result expressions, and handles aliasing correctly.

Our previous work with incrementalizing object-oriented programs [34, 36] requires incrementalization rules to be manually written. The work presented in this dissertation, on the other hand, automatically generates incrementalization rules for a large class of queries.

6.3 Conclusion

This dissertation presents a method for the automatic incrementalization of high-level queries in object-oriented programs. Derived from work on tuple pattern based retrieval, this method is able to incrementalize a broad class of queries involving object, sets, maps, and tuples. It is able to automatically incrementalize every query in this class, generating incremental maintenance code for every possible update that can affect the results of the query. This code remains correct even in the face of arbitrary aliasing between objects in the query. We have presented several ways of invoking maintenance code, allowing performance to be traded for flexibility. We have performed experiments showing that our method is useful in a variety of applications.

It is our hope that the techniques presented in this dissertation will allow automatic incrementalization of high-level queries to become yet another tool in the programmer's arsenal, replacing complicated manual incrementaliation with a clean automatic approach.

Bibliography

[1] U. A. Acar, G. E. Blelloch, M. Blume, and K. Tangwongsan. An experimental analysis of self-adjusting computation. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 96–107, New York, NY, USA, 2006. ACM Press.

[2] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 247–259, New York, NY, USA, 2002. ACM Press.

[3] R. Alhajj and A. Elnagar. Incremental materialization of object-oriented views. *Data Knowl. Eng.*, 29(2):121–145, 1999.

[4] R. Alhajj and F. Polat. Incremental view maintenance in object-oriented databases. *SIGMIS Database*, 29(3):52–64, 1998.

[5] M. A. Ali, A. A. A. Fernandes, and N. W. Paton. Movie: an incremental maintenance system for materialized object views. *Data Knowl. Eng.*, 47(2):131–166, 2003.

[6] American National Standards Institute, Inc. Role-based access control. ANSI INCITS 395-2004.

[7] M. Y. Becker. A formal security policy for an nhs electronic health record service. Technical Report UCAM-CL-TR-628, University of Cambridge, March 2005.

[8] R. G. Bello, K. Dias, A. Downing, J. Feenan, J. Finnerty, W. D. Norcott, H. Sun, A. Witkowski, and M. Ziauddin. Materialized views in oracle. In *Proceedings of the 1998 International Conference on Very Large Data Bases*, pages 659–664, 1998.

[9] The boo programming language. PDF, 2005. `http://boo.codehaus.org/BooManifesto.pdf`.

[10] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López, and G. Puebla. The Ciao Prolog system. Reference Manual. The Ciao System Documentation Series. TR CLIP3/97.1.10. School of Computer Science, Technical University of Madrid (UPM), August 2004.

[11] J. Cai, P. Facon, F. Henglein, R. Paige, and E. Schonberg. Type analysis and data structure selection. In *Constructing Programs From Specifications*, pages 126–164. North-Holland, 1991.

[12] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[13] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *VLDB '91: Proceedings of the 17th International Conference on Very Large Data Bases*, pages 577–589, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[14] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 34–43, New York, NY, USA, 1998. ACM.

[15] S. Choenni, H. M. Blanken, and T. Chang. On the selection of secondary indices in relational databases. *Data and Knowledge Engineering*, 11(3):207–, 1993.

[16] S. Dawson, C. R. Ramakrishnan, S. Skiena, and T. Swift. Principles and practice of unification factoring. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(5):528–563, 1996.

[17] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Trans. Inf. Syst. Secur.*, 4(3):224–274, 2001.

[18] F. L. Fessant and L. Maranget. Optimizing pattern matching. In *ICFP '01: Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming*, pages 26–37, New York, NY, USA, 2001. ACM Press.

[19] A. C. Fong and J. D. Ullman. Induction variables in very high level languages. In *POPL '76: Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, pages 104–112, New York, NY, USA, 1976. ACM Press.

[20] M. R. Frank, E. Omiecinski, and S. B. Navathe. Adaptive and automated index selection in RDBMS. In *Extending Database Technology*, pages 277–292, 1992.

[21] S. M. Freudenberger, J. T. Schwartz, and M. Sharir. Experience with the SETL optimizer. *ACM Transactions on Programming Languages and Systems*, 5(1):26–45, 1983.

[22] D. Gluche, T. Grust, C. Mainberger, and M. H. Scholl. Incremental updates for materialized OQL views. In *Deductive and Object-Oriented Databases*, pages 52–66, 1997.

[23] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of Data*, pages 328–339, New York, NY, USA, 1995. ACM Press.

[24] T. Griffin, L. Libkin, and H. Trickey. An improved algorithm for the incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):508–511, 1997.

[25] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *Bulletin of the Technical Commitee on Data Engineering*, 18(2):3–18, June 1995.

[26] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 157–166, New York, NY, USA, 1993. ACM Press.

[27] J. Earley. High Level Iterators and a Method for Automatically Designing Data Structure Representation. *Journal of Computer Languages*, 1(4):321–342, 1976.

[28] S. Koenig and R. Paige. A transformational framework for the automatic control of derived data. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 306–318. IEEE Computer Society, 1981.

[29] H. A. Kuno and E. A. Rundensteiner. Incremental maintenance of materialized object-oriented views in multiview: Strategies and performance evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):768–792, 1998.

[30] The LINQ project. web page, March 2006. `http://msdn.microsoft.com/netframework/future/linq/`.

[31] Y. A. Liu, T. Rothamel, F. Yu, S. D. Stoller, and N. Hu. Parametric regular path queries. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming Language Design and Implementation*, pages 219–230, New York, NY, USA, 2004. ACM Press.

[32] Y. A. Liu and S. D. Stoller. From datalog rules to efficient programs with time and space guarantees. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and Practice of Declarative Programming*, pages 172–183, New York, NY, USA, 2003. ACM Press.

[33] Y. A. Liu and S. D. Stoller. From datalog rules to efficient programs with time and space guarantees. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 172–183, New York, NY, USA, 2003. ACM Press.

[34] Y. A. Liu, S. D. Stoller, M. Gorbovitski, T. Rothamel, and Y. E. Liu. Incrementalization across object abstraction. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 473–486, New York, NY, USA, 2005. ACM Press.

[35] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Trans. Prog. Lang. Syst.*, 20(3):546–585, May 1998.

[36] Y. A. Liu, C. Wang, M. Gorbovitski, T. Rothamel, Y. Cheng, Y. Zhao, and J. Zhang. Core role-based access control: efficient implementations by transformations. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 112–120, New York, NY, USA, 2006. ACM Press.

[37] H. Nakamura. Incremental computation of complex object queries. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 156–165, New York, NY, USA, 2001. ACM Press.

[38] M. Onizuka, F. Y. Chan, R. Michigami, and T. Honishi. Incremental maintenance for materialized xpath/xslt views. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 671–681, New York, NY, USA, 2005. ACM Press.

[39] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst.*, 4(3):402–454, 1982.

[40] G. Priyalakshmi. Generating efficient programs for solving relational database queries. Master's thesis, Stony Brook University, August 2004.

[41] Python 2.4.2 documentation. web page, September 2005. `http://www.python.org/doc/2.4.2/`.

[42] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *Knowledge and Data Engineering*, 3(3):337–341, 1991.

[43] Robert Paige. Applications of finite differencing to database integrity control and query/transaction optimization. In H. Gallaire, J. Minker, and J.-M. Nicolas, editors, *Advances in Database Theory*, volume 2. Plenum Press, New York.

[44] D. Saha and C. R. Ramakrishnan. A local algorithm for incremental evaluation of tabled logic programs. In S. Etalle and M. Truszczynski, editors, *ICLP*, volume 4079 of *Lecture Notes in Computer Science*, pages 56–71. Springer, 2006.

[45] E. Schonberg, J. T. Schwartz, and M. Sharir. Automatic data structure selection in SETL. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 197–210. ACM Press, 1979.

[46] E. Schonberg, J. T. Schwartz, and M. Sharir. An automatic technique for selection of data representations in SETL programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3(2):126–143, 1981.

[47] J. Schwartz. Programming in SETL. web page. (draft in progress) `http://www.settheory.com`.

[48] J. T. Schwartz. Automatic data structure choice in a language of very high level. In *POPL '75: Proceedings of the 2nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 36–40. ACM Press, 1975.

[49] A. Shankar and R. Bodik. Ditto: automatic incrementalization of data structure invariant checks (in java). *SIGPLAN Notices*, 42(6):310–319, June 2007.

[50] M. Sharir. Some observations concerning formal differentiation of set theoretic expressions. *ACM Trans. Program. Lang. Syst.*, 4(2):196–225, 1982.

[51] T. L. Swift. *Evaluation of Normal Logic Programs.* PhD thesis, Stony Brook University, December 1994.

[52] B. Vialpando and V. Khatri. Comparing db2 materialized query tables and oracle materialized views. Web page, August 2007. `http://www.ibm.com/developerworks/db2/library/techarticle/dm-0708khatri/index.html`.

[53] D. Willis, D. J. Pearce, and J. Noble. Efficient object querying for java. In D. Thomas, editor, *ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 28–49. Springer, 2006.

[54] D. Willis, D. J. Pearce, and J. Noble. Caching and incrementalisation in the java query language. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 1–18, New York, NY, USA, 2008. ACM.

[55] G. Wilson. Linda-like systems and their implementation. Technical Report 91-13, Edinburgh Parallel Computing Centre, June 1991.

[56] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. T Spaces. *IBM Systems Journal*, 37(3):454–474, 1998.

[57] D. M. Yellin and R. E. Strom. INC: a language for incremental computations. *ACM Trans. Program. Lang. Syst.*, 13(2):211–236, 1991.

[58] G. Zhou, R. Hull, and R. King. Generating data integration mediators that use materialization. *Journal of Intelligent Information Systems*, 6(2/3):199–221, 1996.