Automatic Incrementalization of Queries in Object-Oriented Programs

Tom Rothamel and Yanhong A. Liu

Computer Science Department
Stony Brook University
rothamel,liu@cs.sunysb.edu

Abstract

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, a powerful method is needed to incrementally maintain query results with respect to updates to query parameters.

This paper describes a general and powerful method for automatically generating incremental implementations of high-level queries over objects and sets in object-oriented programs, where a query may contain arbitrary set enumerators, field selectors, and additional conditions. 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. Our implementation and experimental results for example queries and updates confirm the effectiveness of the method.

1. Introduction

High-level query constructs in programming languages help greatly improve the clarity of programs and the productivity of programmers, and are being introduced to increasingly more languages. These query constructs, such as SETL set formers [29], Python comprehensions [25], and JQL for Java [34], allow queries over sets and objects to be written in a concise and abstract manner. This allows the programmer to focus on what is computed, rather than how to compute efficiently.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © ACM [to be supplied]. . . $5.00

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. To improve efficiency, the results of the queries need to be stored and incrementally maintained when values the queries depend on are updated. This can lead to drastic and often asymptotic improvements 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 implementations 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 unnecessarily slow. Automatic incrementalization helps resolve this conflict, by transforming clear high-level queries into efficient implementations.

This paper describes a general and powerful method for automatically generating incremental implementations of high-level queries over objects and sets in object-oriented programs, where a query may contain arbitrary set enumerators, field selectors, and additional conditions. 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 implementation and experimental results for example

queries and updates that confirm the effectiveness of the method.

Our method is practical for several reasons. First, it processes one query at a time and handles any updates to the values that the query depends on, without requiring program analysis, so it scales to programs of large sizes that may depend on libraries and plug-ins. Second, the method does not change the representations of sets and objects, so incrementalized modules can interact with the remaining modules. Finally, the method incurs overhead only on objects that an object-set query depends on.

We expect the user to indicate queries that can be profitably incrementalized. Once this is is done, all required code is automatically generated. This allows the user to determine, perhaps experimentally, cases where incrementalization will be profitable.

There is a large amount of prior work on incrementalization of programs and on incremental query evaluation, as discussed in Section 10. However, to the best of our knowledge, there is no existing method that can automatically incrementalize queries over arbitrarily aliased sets and objects in object-oriented programs without changing representations of objects in the programs. This is a challenging problem because even simple queries can be affected by a large number of different updates to the objects the query depends on.

2. Language

Our method applies to any language that supports the following query and update constructs.

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

$$
\begin{aligned}
\mathit{comprehension} &::= \mathit{parameter}^{+}\texttt{->} \\
&\quad\ \{\mathit{result_exp} : \mathit{enumerator}^{+}\ \mathit{condition}^{*}\} \\
\mathit{enumerator} &::= \mathit{enumeration_var}\ \texttt{in}\ \mathit{selector} \\
\mathit{selector} &::= \mathit{variable} \mid \mathit{selector}.\mathit{field} \\
\mathit{parameter} &::= \mathit{variable} \\
\mathit{enumeration_var} &::= \mathit{variable} \\
\mathit{result_exp} &::= \mathit{expression} \\
\mathit{condition} &::= \mathit{expression}
\end{aligned}
$$

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 iff (1) each parameter of the query is assigned the given value of that parameter and (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. Note that nested queries can be formed by supplying the result set of one query as a parameter to a second query.

As a running example, we take the following query, modeled on an authentication query found in the Django web framework. To better demonstrate the method, we have both simplified this query by removing one iteration, and made it more complex by adding a check that the group is active. 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.

$$
\begin{aligned}
&\mathit{users}, \mathit{uid}\ \texttt{->} \\
&\{\ p.\mathit{name} : u\ \texttt{in}\ \mathit{users}, g\ \texttt{in}\ u.\mathit{groups}, p\ \texttt{in}\ g.\mathit{perms}, \\
&\quad u.\mathit{id} == \mathit{uid}, g.\mathit{active}\ \}
\end{aligned}
$$

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.

The query has two parameters, $users$ and uid. Its result expression is $p.name$. It has three enumerators: u `in` $users$, g `in` $u.groups$, and p `in` $g.perms$; and two conditions: $u.id$ == uid and $g.active$.

We classify the variables used in the query into three kinds. A *constrained parameter* is a parameter that is equated to a selector, as uid is in the condition $u.id$ == uid. All other parameters to the query are unconstrained parameters. Non-parameter variables are known as $local\ variables$, as they are entirely local to the query scope. In the running example, $users$ is an unconstrained parameter; uid is a constrained parameter; and u, g, and p are local variables.

For a given combination of unconstrained parameters, our method incrementally maintains results such that queries involving any combination of constrained parameters can be answered in O(1) time. For our running example, when we are maintaining the query over a given $users$ set, we can retrieve the results for any uid in O(1) time.

Changes. We decompose all changes to query parameters into the following three kinds, and incrementally maintain the query result under all changes of these kinds:

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

For the running example, the incrementally maintained result of the query can be affected by the following updates: adding to or removing from the $users$, $u.groups$, and $g.perms$ sets; and assigning to $u.groups$, $u.uid$, $g.perms$, $g.active$ and $p.name$ fields. For incrementalization to be correct, we must maintain the result in response to all of these updates, which may be spread throughout the program.

Language for generated code. We use a standard object-oriented language that supports operations on sets, maps, and tuples and where all values are considered to be objects.

Table 1 describes the operations we use on sets, maps, and tuples. All these operations take constant time, assuming that hashing is used in the implementation of sets, including the key sets and image sets of maps. Sets and maps are empty when first created. The maps we use are multimaps that map a key to a set of values, known as the image set. Image sets are updated when the map is changed. We only create tuples of a constant length.

s`.empty()`	make s the empty set
s`.add(`x`)`	add element x to s
s`.remove(`x`)`	remove element x from s
x `in` s	return whether x is an element of s
x `not in` s	return whether 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`.img(`x`)`	return image set of key x under m
$(x_1,...,x_k)$	create a tuple with elements $x_1,...,x_k$

Figure 1. Operations on sets, maps, and tuples.

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.

For ease of presentation, in the generated intermediate code, we use special for-loops of the form `for` $(x_1,...,x_k)$ `in` s`:` $stmt$, where s is a set of tuples of length k, and each x_i may 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, by maintaining a map from values of bound components to values of unbound components, looking up the bound components in the map in constant time, and iterating only over the unbound components, as described in [27].

Other than the above, we use standard statements for assignment ($v = e$), sequencing ($stmt1$ $stmt2$), branching (`if` c`:` $stmt$), and looping (`for` v `in` s`:` $stmt$). We use indentation to indicate scoping.

3. Overview of the method

Our method processes queries indicated by the programmer one at a time, as queries can be incrementalized independently. It handles all updates to values the query depends on.

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 into the assignment set, provided it satisfies all conditions. 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 the result set. (3) We can return query results efficiently for arbitrary parameter values by maintaining a map from combinations of parameter values to query results. (4) Since we cannot determine all possible values of unconstrained parameters, 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. *D* must be a set because there is no efficient way, in the presence of arbitrary aliasing of objects, to generate each added or removed assignment exactly once.

Reference counts for elements in a result set. As multiple assignments may lead to the same value when evaluated under a given result expression, 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 or 1 to 0, respectively.

Result map, *R*. Instead of creating a new result set for each query instance, i.e., a query with a combination of actual 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 on 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, a copy of the result set can be made and returned, at cost linear in the size of the result. Techniques exist for determining where copying is necessary [11].

Values of Unconstrained Parameters, U. We cannot maintain query results for all possible values of unconstrained parameters, because we do not know what the values might be. We keep combinations of values of unconstrained parameters that have been queried once in a set of tuples, *U*, and only maintain query results for these values of the unconstrained parameters. Note that for each tuple in *U*, we maintain query results for all possible values of constrained parameters, because they are determined by the values of unconstrained parameters through the constraining enumerators. 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. 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.

4.1 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. This makes it easy to automate the translation, while not affecting the final code, since $field_f$ will be eliminated in the generated code.

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.

Finally, we look for conditions of the form $a == b$. When such a a condition is found, we eliminate it and replace all instances of a with b. This will merge a local variable with each unconstrained parameter.

For our running example, this yields:

```
users, uid ->
{ p_name :
   (users, u) in member,
   (u, u_groups) in field_groups,
   (u, uid) in field_id,
   (u_groups, g) in member,
   (g, g_perms) in field_perms,
   (g, g_active) in field_active,
   (g_perms, p) in member,
   (p, p_name) in field_name,
   g.active
 }
```

This replaces the 5 fields and 3 sets used in the object domain with 6 sets in the pair domain, and increases the number of variables used from 5 to 10. The advantage of this approach is that every object that can be used by the query must now be accessed through one of the variables.

4.2 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 the 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:

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

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

`if` c:

We also generate a single for-clause to iterate through the values of the unconstrained parameters in *U*:

`for` unc_params `in` U:

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

For our running example, consider the change that adds g to u_groups. The following clauses are created:

```
for (users, u) in member:
for (u, u_groups) in field_groups:
for (u, uid) in field_id:
for (g, g_perms) in field_perms:
for (g, g_active) in field_active:
for (g_perms, p) in member:
for (p, p_name) in field_name:
if g.active:
for (users) 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 lookups, which take constant time, and to minimize iterations, which have a linear factor. 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. While an optimal ordering could be produced, using precise set sizes in the enumerators and costs of the conditions, by a powerful combinatorial optimization algorithm, we have found that using only the distinction between constant time and linear time (in any set size), based on boundness of the variables, with a simple greedy algorithm, works well when information about set sizes and selectivity is absent. If such information is available, this can be treated as a join-order optimization problem.

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 smallest 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 otherwise.

```
after adding g to a set that u_groups refers to:

for (g, g_active) in field_active:
  if g_active:
    for (g, g_perms) in field_perms:
      for (g_perms, p) in member:
        for (p, p_name) in field_name:
          for (u, u_groups) in field_groups:
            for (u, uid) in field_id:
              for (users, u) in member:
                if (users) in U:
                  D.add(var_asgn())
```

Figure 2. Generated pair-domain code for computing *D* for the running example.

- 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 2 shows the generated code in the pair domain for computing *D* under one update that affects the result of the example authentication query.

4.3 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 1. It gives code for special for-loops over the $field_f$ sets and over the $member$ set, and for all three possible combinations of boundness of the two variables in the pattern—note that our method for nesting clauses ensures that, for iteration over the $field_f$ sets and the $member$ set, we do not have the case that both variables are unbound.

- 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.

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 1. Rules for translating back to the object domain.

In the third case, we also generate code for incrementally maintaining the inverse maps, as given in Table 2; it is easy to see that these maps take constant time to maintain and have 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 our running example and update, the code in Figure 3 is generated, along with code that maintains the inv_{groups} and inv_m inverse maps.

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

Table 2. Rules for generating code for maintaining inverse maps.

```
after adding g to a set that u_groups refers to:

g_perms = g.perms
g_active = g.active
if g_active:
  for p in g_perms:
    p_name = p.name
    for u in inv_groups.img(u_groups):
      uid = u.id
      for users in inv_m.img(u):
        if (users) in U:
          D.add(var_asgn())
```

Figure 3. Generated object-domain code for computing *D* for the running example.

5. Generating Code for Maintaining the Result Map

Once the differential assignment set *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. Both $params$ and $eval$ can be replaced with simple executable code, so no language-level eval support is necessary. *D* is reset to empty after it is used for updating R, and thus is empty and takes no space when we are not executing maintenance code.

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

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

The result map maintenance code must be run after all code for computing the set *D* for a given change has been run, to prevent aliasing from causing problems. The discussion section elaborates on our handling of aliasing.

6. 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. Coordinating the invocation of different kinds of maintenance code is critical for the correctness of the method.

We organize maintenance code based on the pair-domain variables. This is for two reasons, from Section 4: (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, due to aliasing. Obligations are assigned to objects, not variables or fields, and so the obligation will be triggered whenever the object is updated, even in the presence of 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 4.3, and assigning obligations to other objects, as described below. Implementation for (2) depends on the host language, which we will describe for Python, in Section 9; 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. To bound the time spent assigning obligations, obligations are never removed from objects.

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:

(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:

$when\ obligation\ 'x'\ is\ assigned\ to\ an\ object\ referred\ to\ by\ x$:
  $assign_obligation(x.f, 'y')$

$when\ an\ object\ with\ obligation\ 'x'\ has\ field\ f\ assigned$:
  $assign_obligation(x.f, 'y')$

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

$when\ obligation\ 'x'\ is\ assigned\ to\ an\ object\ referred\ to\ by\ x$:
  `for` i `in` x:
    $assign_obligation(i, 'y')$

$when\ an\ object\ with\ obligation\ 'x'\ has\ element\ i\ added$:
  $assign_obligation(i, 'y')$

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.

7. Generating Code for Executing the Query

Finally, we generate code for query execution. 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. Note that this code will want to have variables that range over possible values of the unconstrained variables, so we introduce a new variable for each unconstrained variable. (For example, $uid_$ for uid.)
4. Maintain the result map, using the method in Section 5.

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.

8. Discussion

Correctness. To be correct, our method must maintain the invariant that the reference count of values in R are the number of assignments that would map to that value. The differential assignment set computation code determines the assignments added to or removed from the assignment set. This code is correct because it is simply an execution of the original query with the variables involved in an update bound; updates can only affect assignments containing the objects involved. The *D* set contains the assignments that are added to or removed from R, thus ensuring that the invariant holds.

```
if (users) not in U:

  assign_obligation(users, 'users')
  U.add((users))

  for u in users:
    uid_ = u.uid
    u_groups = u.groups
    for g in u_.groups:
      g_active = g.active
      if g_active:
        g_perms = g.perms
        for p in g_perms:
          D.add(var_asgn())

  for a in D:
    R.add(eval((users, uid_), a), eval(ap_ssid, a))
  D.empty()

return R.img((users, uid))
```

Figure 4. Generated code for executing the query for the running example.

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:

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, but it is empty when the program executes outside of maintenance code. We expect the programmer using the method to understand these, and only ask for 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.

To evaluate the query in our example 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 $O(groups_{perm} * users_{group})$.

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. 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.

Extensions. 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 mirror fields).

9. 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 711 lines of Python code, and requires only the Python standard library to run. It provides as a public interface a single function, `run_query`. This function takes as arguments a comprehension, repre-

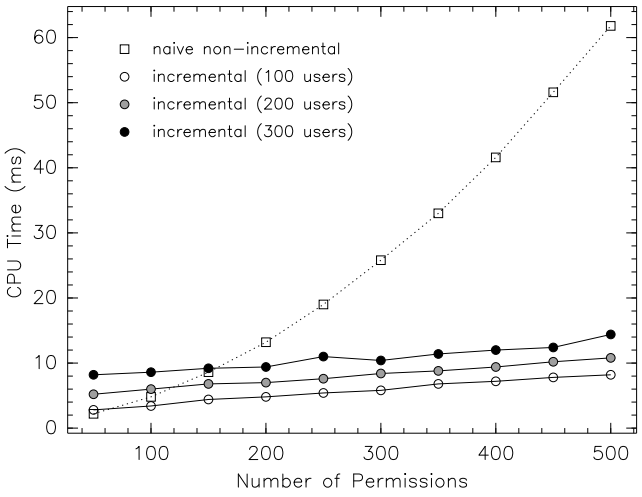

Figure 5. Time taken to perform a varying number of updates and queries, using our running example.

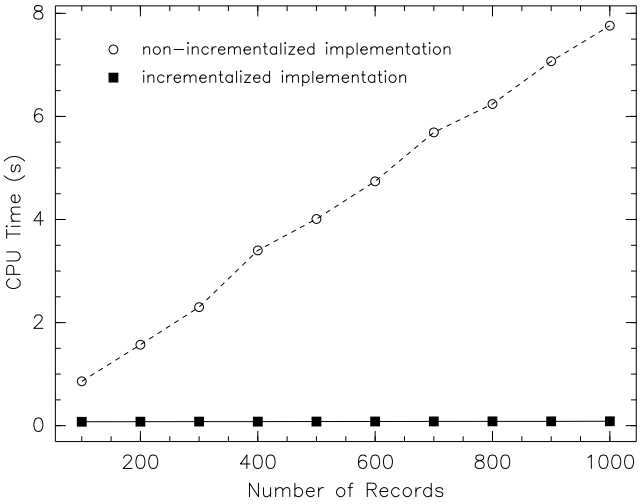

Figure 6. Time taken to perform 10,000 queries, for a varying number of electronic patient records.

sented as a string, and values for the parameters of the comprehension, and returns the result of the query. The 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. Our experimental code is written in Python, using Python 2.5.1, 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 and can be done before program execution.

All programs our method has been applied to have been written by us. Some may object to this, preferring that our method be applied to programs written by others. However, most programs are written with efficiency in mind, and the programmers generally incrementalized the repeated expensive computations by hand. Our method would not improve the performance of such programs. Instead, our method allows programmers to write simpler, more readable, and more maintainable programs, and perform incrementalization automatically.

Running example: authentication query. The experiment we performed using our running example 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 5 gives the result of this experiment. This graph shows that 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. 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 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.

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 [5], and applied our automatic incremen-

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 3. Code generation statistics for student management system.

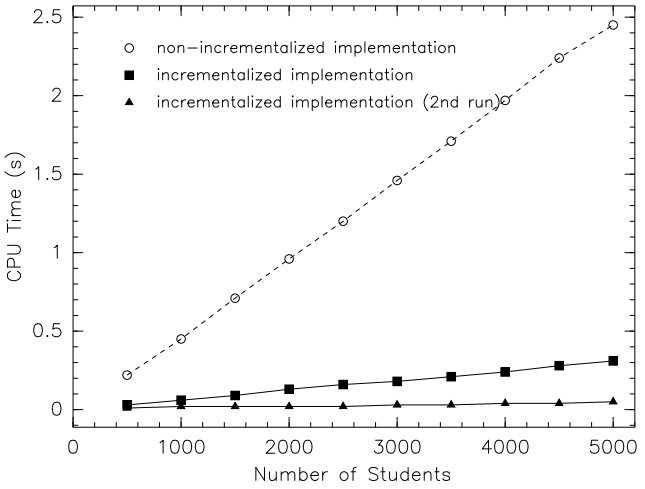

Figure 7. Time required for 100 student queries, for a varying number of students.

talization 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.

Figure 6 shows the results of this experiment. The non-incrementalized implementation obviously takes linear time. Our incrementalized implementation also takes linear time, although with a very shallow slope. Our method causes the first query to take linear time in the number of records, while the second and later queries take a constant amount of time. This gives substantial time savings when the database is large.

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, 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 that our method can handle many interesting queries.

Of the 52 queries our method can incrementalize, we identified 17 frequently used queries. Table 9 shows the statistics we collected for incrementalizing these 17 queries. For each query, we report the code generation time, the number of updates incrementalized, the number of lines of code generated, and the number of 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 be changed to incrementalize it. The number of lines of code and the number of AST nodes (used by Python's `compiler` module) provide a measure of the complexity of these changes. All told, in less than a total of 5 seconds, we are able to generate the equivalent of thousands of lines of code, code that is automatically added to objects that a query depends on.

An example query from this system is the New Students query, which finds all students who either joined the de-

partment in a given semester or have changed program that semester. This query reads:

```
students, sem ->
{s : s in students,
      p in s.programs
      if s.joined == sem or
         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 7 shows the result of this experiment. 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 can be seen by running the incremental version a second time on the same data, and noting that the running time remains constant.

Other languages. While our experimental implementation is for Python, our method is suitable for use with a variety of languages. We require the ability to intercept field modification and set access. Set access is easy to intercept, as one can add hooks to the various set methods. Intercepting field access is generally supported by dynamic languages, but may require VM or bytecode modification in languages like C# and Java. One approach would be to attach to each object a strategy object that controls how fields are altered. When an obligation is added to an object, the strategy object is replaced with one that runs the obligation code. This method can work with one strategy object for each class and each possible set of obligations that can be assigned to the objects of that class.

Our current implementation is a Python module that takes queries as strings. This is somewhat less than ideal, as it means that queries are neither syntax-checked nor type-checked before they are executed. If languages and associated tools had support for object-set query syntax, such checking can be easily done. The advantage of our current approach is that it can be dropped into an unmodified Python interpreter, making it more immediately useful.

10. Related Work and Conclusion

Incrementalization of programs has been a subject of much research, and automatic incrementalization techniques have been developed for queries in many areas. Our method improves over previous methods in two main respects, putting aside many finer distinctions. First, our method handles a query as a whole, while most previous methods decompose it into smaller queries that need to be maintained independently, which may have additional cost in time and memory. Second, our method incrementalizes high-level queries in object-oriented programs, while previous methods handle only sets and tuples, use representations of objects that are not suitable for incrementalized modules to interact with the remaining modules, or incrementalize only with the granularity of method calls.

The earliest work in this area was intended to provide a way of performing strength reduction on sets and maps [9, 10], ultimately yielding the finite differencing method [24, 31]. 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. Our method can derive most of those rules automatically.

In database area, techniques for incremental view maintenance over sets of tuples, as in relational databases, have been known for a long time [33, 6]. Finite differencing was applied to the problems of integrity constraints [17, 23] and incremental view maintenance [26, 14] in databases containing sets of tuples. It has also been extended to support views with duplicates [13]. These methods work by decomposing queries into incrementalizable parts. A method exists that handles a query as a whole and reference-counts intermediate results [15], as ours does the result map. However, all these methods do not handle objects.

Another approach used for incremental view maintenance, in both relational [7] and object-oriented [3, 2] databases, is to use a database engine to perform queries with some parameters bound. This is similar to our strategy that computes D without decomposing a query into subqueries. Unlike these techniques, our method generates code for computing D and does not require executing queries in a database engine.

A number of other methods deal with incremental view maintenance in object-oriented databases. These methods either create their own copies of objects used during incremental maintenance [35, 18], or require the system to represent objects in ways not suitable for program execution [4, 12, 22]. In contrast, our method can use existing objects and allow queries to return existing objects, without changing object representations, and we only require that objects support the ability to intercept method and field accesses. This allows our incrementalized components to be usable in the same way in all the contexts they were usable before.

The problem of choosing an optimal order for nesting clauses when generating code for computing D is similar to join order optimization [16]. The optimal solution to this problem is known to be NP-complete. While a number of approaches to solving this problem exist [32], these require

estimates of the size of each set used and the selectivity of each join performed. When the code for computing D is generated, much of this information might not be available. Our heuristic produces good and reasonable results for object-oriented programs when such information is not available, but the method can be extended to use such information when the information is available. Note that, in either case, our method takes into account the different costs of executing the same special for-loop in different code contexts.

There has also been work on incrementalizing object-oriented programs. One method is based on applying manually-written incrementalization rules which depend on comprehension structure [21, 19]. These rules need to explicitly deal with each possible update under all combinations of aliasing. Our method generates this code automatically.

While our method incrementalizes high-level queries in object-oriented programs, other methods in incrementalization of programs are developed for languages based on sets or bags [24, 31], functional programs [20, 1], logic programs [28], and queries [8] and functions in OO programs [30]. The last two are the most closely related and is discussed separately next.

The Ditto system [30] incrementalizes queries that consist of recursive functions in Java, using a method similar to those for incrementalizing functional programs [1]. This method works by memoizing the results of method calls, and recomputing a stored result of a method call when a field changes or when the result of a call changes. Recomputation is done by re-running the method call, hence Ditto has a method-call-level granularity. Thus, any update to data used by a high-level query in a method will result in recomputation of the entire method including the entire high-level query. Ditto also restricts the results of queries to be of primitive types, while our method allows the results of queries to be sets of elements of any type.

The research most similar to ours is the incrementalization for the JQL system [34, 8]. 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.

Areas of future work include adding support for multi-threaded computation, ending the maintenance of queries that are no longer useful to the program, and increasing the kinds of queries that our method can automatically incrementalize.

References

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

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

[4] M. A. Ali, A. A. A. Fernandes, and N. W. Paton. MOVIE: An incremental maintenance system for materialized object views. *Data and Knowledge Engineering*, 47(2):131–166, 2003.

[5] 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.

[6] P. A. Bernstein, B. T. Blaustein, and E. M. Clarke. Fast maintenance of semantic integrity assertions using redundant aggregate data. In *Proceeding of the 6th international conference on Very Large Databases (VLDB)*, Montreal, Canada, Oct. 1980.

[7] 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.

[8] D. J. P. Darren Willis and J. Noble. Caching and incrementalization for the java query language. Technical report, Victoria University of Wellington, 2007.

[9] J. Earley. High level iterators and a method for automatically designing data structure representation. *Journal of Computer Languages*, 1(4):321–342, 1976.

[10] 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 of Programming Languages*, pages 104–112, New York, NY, USA, 1976. ACM Press.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Transactions on Database*

Systems, 9(3):482–502, September 1984.

[17] S. Koenig and R. Paige. A transformational framework for the automatic control of derived data. In *Proceedings of the 7th international conference on Very Large Data Bases*, pages 306–318, Sept. 1981.

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] R. 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, 1984.

[24] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, 1982.

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

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

[27] T. Rothamel and Y. A. Liu. Efficent implementation of tuple pattern based retrieval. In *PEPM '07: Proceedings of the workshop on Partial Evaluation and semantics-based Program Manipulation*, pages 81–90, Nice, France, January 2007. ACM Press.

[28] 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.

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

[30] A. Shankar and R. Bodik. Ditto: Automatic incrementalization of data structure invariant checks (in Java). To appear in PLDI 2007.

[31] M. Sharir. Some observations concerning formal differentiation of set theoretic expressions. *ACM Transactions on Programming Languages and Systems*, 4(2):196–225, 1982.

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

[33] M. Stonebraker. Implementation of integrity constraints and views by query modification. In *SIGMOD '75: Proceedings of the 1975 ACM SIGMOD international conference on Management of Data*, pages 65–78. ACM Press, 1975.

[34] 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.

[35] 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.