Removing Runtime Overhead for Optimized Object Queries *

Jon Brandvein
Computer Science Department
Stony Brook University
Stony Brook, NY 11794, USA
jbrandve@cs.stonybrook.edu

Yanhong A. Liu
Computer Science Department
Stony Brook University
Stony Brook, NY 11794, USA
liu@cs.stonybrook.edu

Abstract

Powerful optimizations of object queries can lead to reduced asymptotic running times. However, such queries are often used in dynamic languages, and the required generality of the optimizations in handling a dynamic language can lead to significant runtime overhead as well as significantly increased code size.

This paper studies combinations of optimizations for reducing this runtime overhead and code size. We describe two new optimizations — counting elimination and result set elimination, their effectiveness when combined with inlining and when using specialized data structures, and additional optimizations enabled by type analysis and alias analysis. The two new optimizations are enabled by the high-level nature of queries, even though they are difficult and not supported by general compiler optimizations. We have run a variety of benchmarks, including distributed algorithms and benchmarks from prior best systems, obtaining a speedup of up to 56% and code size reduction of up to 37%.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming; D.3.4 [*Programming Languages*]: Processors—Compilers, Optimization, Code generation; H.2.3 [*Database Management*]: Languages—Query languages; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Invariants; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Languages, Performance

Keywords Automatic Incrementalization, Program Optimization, Query Constructs, Dynamic Types

* This work was supported in part by NSF under grants CCF-1414078, IIS-1447549, CCF-1248184, CCF-0964196, and CCF-0613913; and ONR under grants N000141512208 and N000140910651.

PEPM'16, January 18–19, 2016, St. Petersburg, FL, USA
ACM. 978-1-4503-4097-7/16/01...
http://dx.doi.org/10.1145/2847538.2847545

1. Introduction

High-level queries over objects and sets are important for improving the clarity of programs, but are expensive if computed straightforwardly. There exist sophisticated methods for generating efficient implementations of these queries by 1) computing them incrementally as the values they depend on are updated, and 2) making this computation demand-driven, i.e., restricting it to only the part of the data that is queried. In order to cover all cases in a dynamic language, these implementations check whether the input data is well-typed and account for all possible forms of aliasing, at the cost of added runtime overhead and code size.

This paper examines several optimizations for the IncOQ system [13]. Compared to best available previous systems JQL [23, 24] and OSQ [20, 21], IncOQ is notable because it is able to handle all possible kinds of updates to an expressive query form, while at the same time providing competitive performance and a systematic way of reasoning about the generated implementation's correctness and costs. The optimizations we discuss are not applicable to JQL due to its restrictions on the query, and while some would apply to OSQ, OSQ did not study them.

We discuss two new optimizations, counting elimination and result set elimination. Counting elimination deletes code that tracks the number of reasons for an element to be in the query result. It is analogous to determining whether counts are needed for an incrementally maintained materialized view in a relational database. Counting elimination is not supported by standard optimizing compilers. Result set elimination is a form of dead code elimination that applies to sets optimized by counting elimination. Unlike general dead code elimination, it does not require a fixed-point analysis and is exceedingly simple to implement.

We also discuss two optimizations enabled by type and alias information: type check elimination and maintenance case elimination. Although both could be implemented by analyzing the generated code and specializing its conditional statements, this is not ideal. The generated code is larger and more complicated than the original program, yet has no new information that is not already present in the original program or in the invariants created during transformation. We therefore wish to apply these optimizations to the original, high-level program. This paper presents sufficient conditions for determining when these optimizations are safe, based only on analysis results for the original program and for the transformation invariants.

The final optimizations we consider are procedure inlining and native set optimization. Procedure inlining is a standard program optimization, which we apply to inserted maintenance code. We include it to discuss its particular interaction with generated maintenance code and to benchmark it in comparison with the other optimizations. Native set optimization chooses a more efficient im-

Table 1: Primitive set and object operations. s, o, and v stand for variables and f for a field name.

Expression	Return value
s `isset`	whether s is a set
v `in` s	whether v is in set s
o `hasfield` f	whether o is an object and has field f
$o.f$	the value of field f of o
Update	**Effect**
`s.add(`v`)`	add element v to set s
`s.remove(`v`)`	remove element v from set s
$o.f$ `=` v	assign value v to field f of object o
`del` $o.f$	remove the field f from o
Statement	**Effect**
`for` v `in` s`:` *body*	run *body* once for each element v of s

plementation of the set datatype where possible; it is also included primarily to show the combined results.

Section 2 gives background on the relevant aspects of the IncOQ method. Sections 3 through 6 describe the optimizations, including the high-level conditions, information needed (supplied by either automated inference or user annotation), and the extent to which the optimization is implemented in IncOQ. We present our empirical findings in Section 7, particularly, that these optimizations can improve running time by as much as 56% and code size by 37%. Section 8 discusses related work.

2. Language and Transformation

This section presents the language and a summary of the IncOQ method for incrementalizing object queries. We first describe the object model and query construct, before going into the transformation and an example of the generated code. For brevity, we do not state all aspects of the transformation; the reader is directed to [13] for a more thorough explanation. IncOQ is hosted within the Python runtime environment as described at the end of the section.

Object Model and Query Aside from primitives and tuples, the two kinds of values are sets and objects. A set is an unordered collection without duplicates. An object is an unordered mapping from field names to values. Sets and objects are mutable and stored by reference (as in Python and Java), therefore expressions that retrieve these values may be aliased to one another. There are no type restrictions on elements and field values.

Table 1 lists the primitive operations on sets and objects. For each update, we assume that the value of s and o is a set and object, respectively. We also assume that updates are not redundant, e.g., v is not already in s immediately before `s.add(`v`)` executes. More complex updates (e.g., those affecting many values at once, containing arbitrary expressions, or that would be redundant) can be preprocessed into these updates.[1] If the expression v `in` s or the loop `for` v `in` s`:` is evaluated when s is not a set, a `TypeError` occurs, and similarly when the expression $o.f$ is evaluated when o is not an object having field f.

The query construct is an *object-set comprehension*, which we illustrate with our running example.

```
{user.email : user in celeb.followers,
              user in group, user.loc == "NYC"}
```

[1] Throughout, we use the shorthand of writing expressions in updates, which should be preprocessed into fresh variables.

Given a particular celebrity user `celeb` and a group `group`, this query returns the set of emails of users who follow `celeb`, belong to `group`, and have a location field `loc` value of `"NYC"`. More generally, an object-set comprehension returns the set of values of its result expression (here `user.email`) for all combinations of values of variables that 1) satisfy its clauses and 2) are consistent with the given values for its parameter variables, which we denote with underlines. Values that cause a `TypeError` to occur are ignored; no other error cases are allowed.

Overview of IncOQ: Although the query construct seems simple, it is challenging to compute it incrementally due to duplicate result elements, ill-typed input data, and aliasing.

(1) Different objects for `user` may have the same `email` field.

(2) `group`'s value might not be a set, or some of its elements might not be objects having the `loc` and `email` fields.

(3) A field assignment for `followers`, `loc`, or `email` anywhere in the program can affect the result. A single set update may affect the result in two different ways if `celeb.followers` can be aliased to `group`.

The IncOQ method transforms programs so that they compute object-set comprehensions incrementally, accounting for all of the above cases. It does this by inserting a block of *maintenance code* immediately before or after each update that can affect the query. The code uses auxiliary data structures to efficiently compute changes to the query's stored result; these structures are also incrementally computed by their own maintenance code. The query's computation is made demand-driven by specifying at runtime a (usually small) set of parameter values that are considered *demanded*, e.g., the values that will actually be passed to the query. The incremental computation is then restricted to only manipulate values that are directly or indirectly reachable from these values by retrievals of elements and fields.

The stored result of the query is held in a *counted set*, which is a set that associates each element with a positive integer. Counted sets support all the operations of normal sets (where the `add` operation also initializes the element's count to 1), plus the operations listed in Table 2. The $\texttt{add}_\texttt{c}$ and $\texttt{remove}_\texttt{c}$ macros manipulate an element's count, only adding or removing the element when the count would switch between 0 and 1. Counted sets differ from multisets (bags) in that iterating over a counted set produces each element only once regardless of its associated count. IncOQ preserves the invariant that the count for each element is the number of combinations of values of query variables that produce that element.

Transformation The transformation done by IncOQ has three stages:

(1) rewrite the query into a flat relational form,

(2) define new invariants for demand-driven relations to help compute the query, and

(3) emit maintenance code for the stored result of the query and the auxiliary data structures, and replace occurrences of the query with uses of the stored result.

Stage (1) models all memberships and fields as pairs in new sets. We refer to these sets as *relations* because, just as in relational databases, they hold tuples of the same arity. The flattened query is called a *relational comprehension*; it is similar to object-set comprehensions except there are no field retrievals, and each membership clause has a tuple of variables on its left-hand side and a relation with the same arity as that tuple on its right-hand side. Relations are never aliased or reassigned, so their only updates are additions and removals of tuples, and it is trivial to determine where these updates occur.

Table 2: Counted set operations. s and v stand for variables; s must hold a counted set. The two macros are syntactic sugar, and are immediately replaced by their expansions in the AST.

Expression	Return value
`s.getcount(v)`	the count for v in s
Update	**Effect**
`s.inccount(v)`	increment the count for v in s
`s.deccount(v)`	decrement the count for v in s
Macro	**Expansion**
`s.add_c(v)`	⇒ `if v not in s: s.add(v) else: s.inccount(v)`
`s.remove_c(v)`	⇒ `if s.getcount(v) == 1: s.remove(v) else: s.deccount(v)`

Table 3: Some of the tag and filter definitions created for the running example query; not shown are dM_2, $dF_{\texttt{loc}}$, $dF_{\texttt{email}}$, $T_{\texttt{group}}$, $T_{\texttt{user}_1}$, and $T_{\texttt{user}_2}$. The two filters for M correspond to its two distinct uses in the query.

Tag or Filter	Meaning and Definition
$T_{\texttt{celeb}}$	Demanded `celeb` values `{celeb : (celeb, group) in` U`}`

$dF_{\texttt{fol}}$	$F_{\texttt{fol}}$ restricted to demanded `celeb` values
	`{(celeb, celeb_fol) : celeb in` $T_{\texttt{celeb}}$`,`
	`(celeb, celeb_fol) in` $F_{\texttt{fol}}$`}`

$T_{\texttt{celeb\_fol}}$	`followers` values of demanded `celeb` values `{celeb_fol : (celeb, celeb_fol) in` $dF_{\texttt{fol}}$`}`

dM_1	M restricted to `followers` values of demanded `celeb` values
	`{(celeb_fol, user) :`
	`celeb_fol in` $T_{\texttt{celeb\_fol}}$`,`
	`(celeb_fol, user) in` M`}`

For memberships, there is a single relation M that holds a tuple (s, e) whenever s is a set in the original program and e is in s. For each field f, there is a relation F_f holding (o, v) whenever o is an object having field f and $o.f = v$. There is also a relation U that holds a tuple $(v_1, \ldots, v_n)$ for each combination of demanded parameter values $v_1, \ldots, v_n$. The flattened query corresponding to the given example query is

```
{(celeb, group, user_email) : (celeb, group) in U,
    (celeb, celeb_fol) in F_fol, (celeb_fol, user) in M,
    (group, user) in M, (user, user_loc) in F_loc,
    user_loc == "NYC", (user, user_email) in F_email},
```

abbreviating `followers` as `fol`. Flattening reduces the complex membership and field retrievals of the original query to a relational join, whose dependencies on updates are uniform and easier to understand. The parameters are turned into local variables and inserted into the result expression, so that the relational comprehension expresses all of the original query's results over each combination of demanded parameter values.

Stage (2) defines *tag* and *filter* relations to help make the query's maintenance demand-driven. A tag is a set of possible values for a variable in the query. A filter is a subset of M or F_f where the first component of each tuple is in a tag set. Both are defined in a dovetailing manner, and both can hold only values that can be obtained from U by retrieving elements of sets and fields of objects. This helps bound the space usage for auxiliary data structures and time for query result maintenance. Some of the tags and filters for the running example are listed in Table 3.

We illustrate Stage (3) with the example update,

```
current_user.loc = new_loc,
```

where a `loc` field does not previously exist on `current_user`. The corresponding relational update is

$F_{\texttt{loc}}$`.add((current_user, new_loc))`.

The incremental maintenance of the query iterates over all values of the query variables such that `user` and `user_loc` match the values referred to by `current_user` and `new_loc`, respectively. This code, called a *block*, is formed by arranging the clauses in a valid order, and then turning each clause into a nested loop or test that, given values for the variables already determined (the *bound variables*), finds matching values for the remaining variables in that clause (the *unbound variables*). A valid order is one where each clause over M or F_f has at least one variable bound. Such orders always exist because all variables in the query are reachable from the parameters. The body of the code for the last clause performs the actual addition or removal on the stored result of the query. It is guaranteed that all values for a query variable are in at least one tag set corresponding to that variable.

The generated maintenance code for this update is given in Table 4. It is inserted immediately after the addition to $F_{\texttt{loc}}$; for removal updates the code would be inserted before, and the body would have a $\texttt{remove}_{\texttt{c}}$ update. Variable `Q` holds the counted set containing the stored result. The variables $dF_{\texttt{loc}}$, $dM_{1\texttt{\_in}}$, $dF_{\texttt{fol\_in}}$, and $U_{\texttt{\_out}}$, hold auxiliary data structures; their maintenance code is not shown. $dF_{\texttt{loc}}$ is a filter and the rest are *maps* that serve to index filters or U for efficient retrieval of values for unbound variables. The maintenance code begins with a test for whether the updated pair is in a filter; this is called a *demand check* and ensures that the maintenance code does not run for values not in the appropriate tag sets. The use of the query itself is replaced by a use of a map lookup expression `Q_bbu[(celeb, group)]`, which gets the set of third components of tuples in `Q` (the original query's result) matching the given values for its first and second components (the original query's parameters).

The lines marked with an asterisk (*) in Table 4 are type checks, and the line marked (**) uses reference counting. Type checks are needed to skip over values that would otherwise lead to `TypeError`. Table 5 lists all the cases where type checks are needed. In general, when the set or object variable is bound, we use a type check to see if its value does in fact have the expected type, before a retrieval or test uses this value. The remaining cases do not require type checks because they use retrievals from U and auxiliary data structures, which cannot cause `TypeError`.

Implementation The IncOQ system operates at the module level, taking an input program written in a subset of Python and producing a generated program that depends on a small runtime library. All updates to values that the queries depend on must be preprocessed into one of the forms listed in Table 1, and must appear inside the transformed module. This prohibits updates that occur via a library call or via reflection (`exec()`). An external module is allowed to import the transformed module and update its values by calling functions defined in the transformed module. In Section 5 we relax this restriction so that an external module can also directly update any value that is not in any tag set.

Object-set comprehensions are represented as Python set comprehensions by translating membership and other clauses to `for` and

Table 4: Maintenance code block for the running example and an assignment to the `loc` field. Comments denote which clause each piece of code was created for, with bound variables underlined. The map lookups $map[key]$ act as indices, returning sets of matching component values from the corresponding relations; if key is not in map the lookup returns the empty set.

```
  if (current_user, new_loc) in dF_loc:                  // initial demand check and binding for update
    user, user_loc = current_user, new_loc
    if user_loc == "NYC":                                // user_loc == "NYC"
*     if user hasfield email:                            // (user, user_email) in F_email
        user_email = user.email
        for celeb_fol in dM_1_in[user]:                  // (celeb_fol, user) in M
          for celeb in dF_fol_in[celeb_fol]:             // (celeb, celeb_fol) in F_fol
            for group in U_out[celeb]:                   // (celeb, group) in U
*             if group isset:                            // (group, user) in M
                if user in group:
**                Q.add_c((celeb, group, user_email))    // Update result
```

Table 5: Generated code for each case where a type check is needed. Bound variables in the tuples are indicated by underlines.

Case	Code
$(\underline{s}, e)$ in M	if s isset: for e in s:
$(\underline{s}, \underline{e})$ in M	if s isset: if e in s:
$(\underline{o}, v)$ in F_f	if o hasfield f: v = $o.f$
$(\underline{o}, \underline{v})$ in F_f	if o hasfield f: if v == $o.f$:

`if` clauses respectively, with some minor rewriting to ensure Python can parse the clauses correctly and to account for Python's lack of pattern matching. Since Python's built-in `set` datatype cannot contain nested sets due to its combination of mutability and value equality semantics, IncOQ uses its own set datatype, `Set`, having reference semantics. There is also a `CSet` datatype for counted sets. Objects are required to be hashable and implement reference semantics. Objects have optional constructors and destructors, which in Python are implemented by defining `__init__()` and `__del__()` methods. In Sections 4 and 5, when we refer to a constructor, we mean any sequence of updates that initially assigns all the fields that an object of a particular type is expected to have, regardless of whether these updates actually occur inside a designated method. Likewise a destructor is any sequence of updates that deletes all fields immediately before the object is destroyed.

3. Counting Elimination and Result Set Elimination

Counting elimination uses a normal set in place of a counted set to hold the stored query result whenever possible. This saves a constant factor of space by not storing the count for each element, and a constant factor of time and code size by using `add` and `remove` operations instead of the expansions of $\texttt{add}_\texttt{c}$ and $\texttt{remove}_\texttt{c}$. Result set elimination deletes the set holding the query result when it is made redundant by a map. Counting elimination is partially implemented in IncOQ as described below, and result set elimination is fully implemented.

In general, a counted set is not dead code if it is used in any $\texttt{remove}_\texttt{c}$ updates, because this macro expands to code that reads the stored count to make control flow choices. However, if counting elimination can replace the counted set with a normal set, and if afterward the only uses of the set are in `add` and `remove` updates, then result set elimination can safely delete the set and its updates. Thus, result set elimination should be used in conjunction with counting elimination.

There are two cases under which counting elimination can be performed. The first is when there are no removal updates to any relation appearing in the query's flattened comprehension. Since $\texttt{remove}_\texttt{c}$ updates are only produced inside maintenance code for removal updates, and `getcount` operations only appear inside $\texttt{remove}_\texttt{c}$ updates, this means there are no operations that read the counts of the stored query result. The `inccount` statement in the expansion of $\texttt{add}_\texttt{c}$ is then dead code. If Q is the stored query result variable, we can simplify $Q.\texttt{add}_\texttt{c}(v)$ to

```
if v not in Q:
    Q.add(v).
```

The second case is when we can statically determine that the counts in the set will never be greater than 1. This means that the `if` tests in the expansions of $\texttt{add}_\texttt{c}$ and $\texttt{remove}_\texttt{c}$ will always succeed, and so they can be simplified to `add` and `remove` statements respectively, with no need to maintain a count. Define the *inputs* of a result expression to be the maximal field retrieval expressions that appear in it; for instance, the inputs to the tuple expression `(a, b.c.d)` are `a` and `b.c.d`, but not `b` or `b.c`. Then the following two conditions, taken together, are sufficient to ensure the counts will never exceed 1.

(1) The inputs of the result expression, together with the query's parameters, determine the values of all remaining variables in the query.

(2) The comprehension's result expression is an injective mapping from its inputs to its output, i.e., no two distinct combinations of values for its inputs produce the same output value. In particular, a variable or tuple of variables is such an expression.

Composing these two conditions ensures that, for a given result element and given parameter values, there is exactly one possible combination of values for the inputs, and in turn, the remaining query variables. These conditions can also be applied to relational comprehensions, where the inputs are just variables (not field retrieval expressions), including the parameters. In particular, the conditions are enough to eliminate counts for all filters, but not for tags.

Example 1. In the running example query, the result expression `user.email` has one input, itself, and it is trivially injective. If we know that each user has a unique email address, then the result expression's value determines the value of `user`. Together with values of parameters `celeb` and `group`, the values of all query variables are determined, so it is possible to use a normal set for the query result. Without the uniqueness assumption, counts are required in order to incrementally maintain the query result — we

need to know how many users share a given email address so that we can remove the address when the last user no longer satisfies the query. □

The uniqueness assumption can be expressed as an annotation alongside the query, e.g., `unique user.email`. This is analogous to a key constraint in a relational database system. Our prototype implementation in IncOQ does not support such an annotation, but it can automatically determine cases where conditions (1) and (2) follow from just the structure of the query, in particular, when the result expression is a tuple containing at least all non-parameter variables.

Regardless of the techniques used to analyze unnecessary counts, we perform the analysis for this optimization prior to generating maintenance code. This way, we can emit the optimized `add` and `remove` operations directly instead of the expansions of $\texttt{add}_\texttt{c}$ and $\texttt{remove}_\texttt{c}$.

4. Type Check Elimination

For each clause over M or F_f, the generated code includes a type check to ensure that `TypeError` does not occur, as indicated in Table 5. Type check elimination deletes these tests when they can be statically determined to always succeed. This saves a constant factor of running time and code size. Although type-check elimination can be performed by running type analysis on the generated code, the point of this section is not to talk about type analysis, but rather to specify how the type information present in the original program relates to the safety of omitting type checks in the generated program. IncOQ does not implement a type analysis, but for benchmarking purposes supports omitting type checks if directed to by the user.

The main idea is to first introduce the notion of a *safe typing* for the query's variables: For each clause in the query, if we evaluate the clause expression using values having the types prescribed by a safe typing (we say the values are *well-typed* with respect to the typing), a `TypeError` will not occur. Next, we show that if each demanded parameter value is well-typed at all program points where maintenance code for the query's stored result runs, then, by the definitions of tags and filters, these auxiliary data structures will hold only well-typed values. This in turn means that each value retrieved during the execution of maintenance code is well-typed and therefore satisfies the type check.

We give a simple type system for set and object types, and define a $hasType(v, T)$ relation inductively over type terms as follows:

- $hasType(v, T)$ holds when T is a primitive type and v is an instance of T.
- $hasType(v, \texttt{set<}T\texttt{>})$ holds when v is a set and for all elements e in v, $hasType(e, T)$ holds.
- $hasType(v, \texttt{obj<}f_1\texttt{:}T_1\texttt{, ..., }f_n\texttt{:}T_n\texttt{>})$ holds when v is an object having (at least) fields $f_1, \ldots, f_n$, and for each value u_i of f_i, $hasType(u_i, T_i)$ holds.

A type may be associated with a name. This allows us to give recursive types to objects that may directly or indirectly refer to themselves. A type T_1 is a subtype of T_2, denoted $T_1 \leq T_2$, if for every value v, $hasType(v, T_1) \Rightarrow hasType(v, T_2)$. It follows that set and object types are covariant in their element and field types, respectively.

We now apply the type system to comprehensions. A *typing* τ for a comprehension is a mapping from its variables to their respective types. Expressions are typed as follows:

- If x is a variable, $\tau(x)$ is the type that τ assigns to x.
- If α is a variable or a field retrieval expression, and $\tau(\alpha)$ is an object type $\texttt{obj<..., }f\texttt{:}T_f\texttt{, ...>}$, then $\tau(\alpha.f)$ is T_f, and undefined otherwise.

We say that a typing τ is *safe* for a query if the following two conditions hold:

- For each membership clause x `in` α, where α is a variable or field retrieval expression, $\tau(\alpha)$ is $\texttt{set<}T\texttt{>}$ where $T \leq \tau(x)$.
- For each field retrieval expression α, $\tau(\alpha)$ is defined.

A typing for an object-set comprehension naturally extends to a typing for its corresponding flattened comprehension: If $o.f$ is replaced by a new variable o_f during flattening, then $\tau(o_f) = \tau(o.f)$.

The key property that must hold for type check elimination is that, at the point that the query's maintenance code runs, each demanded parameter value must be well-typed with respect to some safe typing τ. Precisely, for each $(v_1, \ldots, v_n)$ in U corresponding to parameters $p_1, \ldots, p_n$, $hasType(v_i, \tau(p_i))$ holds. From this it follows that tags and filters hold only well-typed values, i.e., for each tag for a variable x, the comprehension defining the tag returns only values of type $\tau(x)$, and for each filter for a flattened clause $\texttt{(}x\texttt{, }y\texttt{) in }R$ where R is M or F_f, its comprehension returns a set of pairs of values with types $\tau(x)$ and $\tau(y)$ respectively. With this property, we can then show that all values retrieved for query variables in the query maintenance code are well-typed, by induction on the bound variables. The base case is that the variables appearing in the update must hold well-typed values if the demand check passes. The inductive case is that each clause's code retrieves values for unbound variables from either U, the filters, or the bound variables' values, all of which produce well-typed values.

This leaves us with the problems of finding a suitable τ, and determining whether the demanded parameter values are in fact well-typed. Both can be addressed in general by a combination of type annotation and type analysis (e.g., that of [7]). For the second problem, we observe that type checks can only be eliminated to the extent that the programmer provides the queries with well-typed input data. In a statically typed language, this would be trivial, but in a dynamic language the programmer must exercise discipline. It is unavoidable that an object will be ill-typed in at least two situations: at the program point between the deletion of a field and its reassignment to a new value, and during the object's construction and destruction, while fields are being assigned and deleted, respectively. The first case is not a problem because no maintenance code runs at that point. The second case can be addressed by the programmer making sure to not leak any references to an object before its construction completes, and symmetrically, ensuring all other references to it are gone before destruction begins. This way, the object is not reachable from demanded parameter values while it is in an ill-typed state.

Example 2. For the running example, we may use two type definitions,

```
User = obj<followers:set<User>,
           email:string, loc:string>
Group = set<User>,
```

and the typing,[2]

$$\texttt{celeb} \to \texttt{User} \quad \texttt{group} \to \texttt{Group} \quad \texttt{user} \to \texttt{User}.$$

It is easy to confirm that this typing is safe. Provided that the program ensures that all demanded values for `celeb` and `group` have `User` and `Group` type respectively, we can eliminate all type checks from maintenance code. This would not be the case if, say, we created a new user and immediately made them follow a demanded celebrity before assigning them an email address. □

[2] A more general safe typing can also be found, wherin `celeb` and `user` are assigned separate types with fewer fields.

5. Maintenance Case Elimination

IncOQ inserts maintenance code at all updates that could affect the query result. In principle, since arbitrary aliasing is possible, this requires adding code at all updates to all sets, and at all updates to fields that appear in the query. In practice, we can use alias and demand information to identify and eliminate dead maintenance that is guaranteed to not change the stored query result. This optimization reduces code size and saves a constant factor of running time. It can additionally save a constant factor of space if some auxiliary data structures are no longer needed after deleting the code. IncOQ does not itself implement the kind of alias analysis needed to automate maintenance case elimination; in our experiments we have applied the optimization manually.

Suppose that a program containing the running example query also gives `user` objects a `favoritefoods` field that stores a set of strings. Even though there is no mention of this field in the query, if we have the code

```
current_user.favoritefoods.add("chocolate")
```

IncOQ will insert two pieces of maintenance code at the addition update, just in case `current_user.favoritefoods` is aliased to some value for `celeb.followers` or `group`. A similar situation occurs when any celebrity follower set or group set is updated; instead of running one kind of maintenance code or the other, both kinds run just in case the same set is used in both capacities. Maintenance case elimination is simply the use of alias information, provided by an annotation or external analysis over the *original* program, to eliminate maintenance code by determining that such situations do not arise. Just as for type-check elimination, this optimization can be performed directly on the generated low-level code (by specializing the demand check for the case when it is false). However, optimizing the high-level code allows us to save some effort and to characterize the problem in terms of demand instead of aliasing: updates to values that are not in tag sets do not require maintenance.

We point out a few interesting uses of this optimization. First, suppose we know that the query's input data is well-typed with respect to some safe-typing as defined in Section 4. We can use static knowledge of the types of variables appearing in updates to eliminate any block of maintenance code for an update of an incompatible type. For instance, if we know that the running example query satisfies the safe typing from Example 2, then we know that the `favoritefoods` update above does not need any maintenance code, since the typing requires any demanded value for `user` to be a `User` rather than a string.

Second, recall from Section 4 that type checks can be eliminated from constructors and destructors, so long as no other references to the object exist during their execution. In fact, that condition is strong enough to eliminate not just the type checks but also the rest of the maintenance code, since we know the object cannot possibly be in any tag set.

Finally, maintenance case elimination justifies the correctness of using IncOQ in applications where we cannot intercept all updates. This situation arises in practice when we interface an IncOQ-transformed program with an external system, where the system may manipulate objects and sets without IncOQ's knowledge.[3] So long as the updates appearing in the external system occur while those values are not in any tag set, this does not cause any correctness problem, because any maintenance code that would have been inserted would not run due to its demand check.

[3] An example is the incrementalization interface we built between DistAlgo[15] and IncOQ.

6. Inlining and Native Set Optimization

In this section we discuss two optimizations that are not novel, but will be benchmarked in Section 7 for comparison with the other optimizations.

Inlining By default, IncOQ generates all maintenance code for a single combination of an invariant and an update inside its own procedure, and inserts a call to the procedure immediately before or after an update. We consider the effect of inlining these maintenance procedures. Inlining saves a constant factor of running time by avoiding the overhead of procedure calls, at the expense of larger code size. Inlining also allows type check elimination and maintenance case elimination to apply separately to each place that the maintenance code is run; without inlining, only optimizations that are safe for every call site can be performed. IncOQ supports inlining upon request of the user.

There is one maintenance procedure for each kind of update to each invariant, including invariants for auxiliary data structures (tags, filters, and maps). The number of nested calls to maintenance procedures is approximately bounded by the number of tags and filters, which is at most linear in the size of the query. Since the dependencies among invariants are non-cyclic, the procedures are non-recursive and complete inlining is always possible.

Naturally, inlining increases code size if the maintenance procedures are used more than once. Even when the number of updates in the original program is small, the expanded maintenance code for a single update can be asymptotically large (in the size of the query) when two particular extensions to IncOQ are used: nested queries and aggregates. For nested queries of depth k, the update may directly affect each of the k queries, and each inner query requires maintenance to propagate updates to all of its outer queries, yielding a factor of $O(k^2)$ in code size. For aggregates, the maintenance code has two separate updates to the stored result to delete the old aggregate result and assign the new one. These can multiply with each level of nesting, such that the code size is exponential. In practice, k is a small constant, and we have not seen examples where inlining causes code size to increase to unmanageable levels.

Native Sets IncOQ has two set-like classes in its runtime library. The first, `CSet`, provides a counted set by wrapping around a map (a Python dictionary) object, and incurs the overhead of a Python method call for each operation. The second, `Set`, subclasses Python's built-in ("native") `set` type, and has no additional overhead for its `add` and `remove` operations. When counting elimination is performed, we have the option to change the optimized set from a `CSet` to a `Set` and eliminate the overhead. IncOQ does this automatically unless directed not to by the user.

7. Experiments

We have implemented these optimizations and measured their effect on running time, space usage, and code size, using the existing IncOQ implementation. Our findings are summarized as follows:

- Running time can be reduced by as much as half (56% in Figure 1), with the effect greatest for workloads that spend more time on incremental computation. This particularly mitigates some of the overhead that is inherent to maintaining tags and filters.
- Code size of the output program is modestly improved by each optimization except inlining. For programs implementing the queries and updates of distributed algorithms, we obtain between 9% and 20% improvement for applying counting elimination, result set elimination, and native set optimization. For the running example, a 37% improvement is obtained by applying all optimizations even including inlining.

- Significant improvement in both running time and space can come from native set optimization. We obtain a 5x improvement in running time on a stress test benchmark (Table 8) that taxes only the set datatype, and a 4% improvement in overall space usage on the running example's benchmark.
- Overall space usage is not affected much by any optimization except native set optimization.

We perform five evaluations in this section.

(1) *Comparison with JQL* We put optimized and unoptimized IncOQ into context by comparing its performance with that of JQL, a prior best system for running object queries incrementally.

(2) *Running Example* We examine the cumulative effect of each optimization on the running example query.

(3) *Native Sets* We run a simple benchmark of different implementations and uses of the set datatype to determine how much overhead might be avoided.

(4) *Distributed Algorithms* We apply optimized IncOQ to distributed algorithms.

(5) *Wifi Query* We show that the optimizations are able to reduce the overhead for computing tags and filters for demand-driven computation. Compared to an implementation that is not demand-driven and does not use tags or filters, the optimizations reduce overhead from 3.3x to only 1.5x.

All experiments were run on an Intel Core i5 4200M CPU (2.5 GHz) with 8 GB memory, with the Python garbage collector disabled. Python 3.4.2, 64-bit, is used. For JQL experiments we used JQL 0.3.3, Java 1.6.0_45, AspectJ 1.7.2, and ANTLR 3.0.1. Unless otherwise noted, all reported times are CPU time, in seconds, as the mean of repeated runs until the standard deviation is less than 10% of the mean, with no fewer than 10 runs and no more than 50. All lines-of-code counts exclude lines consisting solely of whitespace or comments.

Comparison with JQL We compare the time performance of IncOQ, with and without optimization, against that of JQL, with and without caching. Although a direct comparison of absolute running times would be confounded by the two languages' drastically different runtime environments, it is still informative to examine the large (asymptotic) differences in scalability as the size of the test data increases.

We run all three queries from the JQL cache incrementalization experiment [24], and show results for the query that is over two sets. Expressed in our notation, this query is

```
{(a, s) : a in attends, s in students,
          a.course == COMP101, a.student == s}.
```

To show scalability, we create up to 30,000 objects in each set, and performs 5,000 operations, each being either a query or a random addition and removal of an element of `attends`. The ratio between queries and updates is 1:1.

Figure 1 shows the running times. IncOQ is more than two orders of magnitude faster than JQL. A straightforward batch implementation using a Python set comprehension was also produced, but takes quadratic time due to its join strategy, and times out (> 1 min) before the first datapoint. The high cost of JQL is because it executes a hash join, even for incremental maintenance, which requires scanning an input set, whereas IncOQ uses an incrementally computed auxiliary data structure as an index. The difference is even greater on the JQL query containing three membership clauses. On average the optimized IncOQ implementation is 56% faster than the unoptimized one.

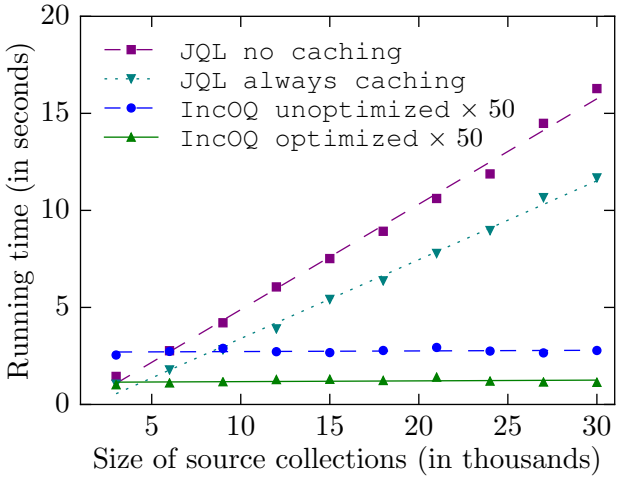

Figure 1: Running time for JQL and IncOQ. Note the scale magnifier for IncOQ implementations. For this benchmark, 50 trials were run per datapoint regardless of standard deviation, as was done in [24].

Running Example We took an input program from [13] that performs the running example query and its associated updates, and transformed it with varying levels of optimization, summarized in Table 6. The simplest implementation is incremental and demand-driven but does not use any of the optimizations discussed in this paper. Each subsequent implementation incorporates all the optimizations of the previous ones. We chose to apply these optimizations cumulatively to give a better idea of the total possible speedup, and because some of the optimizations depend on others. In particular, `Maint case elim` depends on `Inlining` because it removes code from different inlined instantiations of a procedure that cannot otherwise be optimized, and `Result set elim` and `Native sets` depend on `Counting elim` because they eliminate or improve non-counted sets. In general, the optimizations are in order of increasing sophistication and needed information, starting with inlining (which can always be done), then set optimizations, then type- and alias-based optimization, with native set optimization coming last because it is implementation-dependent. Our test data is generated such that each `user` has a unique `email`, no values for `celeb`, `celeb.followers`, or `group` are aliased, and all values conform to the typing from Example 2 (in fact, all values for `user` are valid values for `celeb` and vice versa). This information is needed to deduce respectively that counting elimination, maintenance case elimination, and type check elimination are safe to apply. Maintenance case elimination was applied manually by deleting the maintenance handling followers/group set aliasing, and the maintenance at the initialization of a new celebrity's fields.

Our procedure for generating the input data is designed to vary how "dense" the data is. This in turn affects how many changes there are to propagate to the stored result, and thus how much of the running time is affected by the part of the code that is optimized. There are N users, where N varies from 2,000 to 20,000, and 200 groups. Each user belongs to 5% of the groups and has one of 20 locations, one of which satisfies the location condition in the query. A particular group is singled out as demanded; the demanded parameter pairs are all N pairs of each user with this group. The total size of all follower sets is fixed at approximately 2,000,000 by varying the number of followers per user inversely with N (e.g., the followers per user are 1,000 and 100 at the extremes of $N = 2,000$

Table 6: Benchmarked implementations.

Level	Implementation	Additional optimization	External information needed	LOC
0	`Unoptimized`	N/A	N/A	592
1	`Inlining`	Maintenance procedures inlined	None	854
2	`Counting elim`	$\texttt{add}_c$/$\texttt{remove}_c$ changed to `add`/`remove`	Key constraint on `user.email`	671
3	`Result set elim`	Result sets eliminated	None	655
4	`Type check elim`	Type checks eliminated	All pairs in U are well-typed	572
5	`Maint case elim`	Maintenance code eliminated	`celeb`, `celeb.followers`, `group` unaliased	374
6	`Native sets`	`CSets` replaced by `Sets`	None	374

and $N = 20,000$, respectively). The result is that at lower values of N there are more satisfying values for the query, and hence more changes to make to the stored result at updates.

Figure 2 and the left column of Table 7 show the measured running time for performing 200,000 pairs of operations consisting of alternately 1) querying over a random demanded user-group pair and 2) updating the location of a random user among those users that belong to the demanded group. On average, inlining and counting elimination lead to a modest improvement, as does type check elimination. The bulk of the improvement comes from `Result set elim`. However, we also observed a similar improvement (38% including the two previous optimizations) in an alternate experiment where native set optimization is performed before result set elimination, indicating that these two optimizations are competing for the same improvement (namely, removing an inefficient implementation of the result set). Maintenance case elimination had little or no observable effect because the removed blocks of maintenance code (for updates to sets of users) are not exercised by this benchmark. The effect of `Result set elim` in particular is stronger for small N because updates to the query result account for a higher proportion of the cost.

We measured the space usage of all implementations for this benchmark, measured after the timed operations were performed. We found that `Unoptimized`'s memory usage varies from 680 MB at $N = 2,000$ to 971 MB at $N = 20,000$ (average 812 MB). On average, the differential improvement of `Result set elim` over its previous optimization level is 2.0 MB (0.3%), and the differential improvement of `Native sets` over its previous level is 32.5 MB (4.0%); the rest of the optimizations make less than one megabyte of difference.

We ran a second benchmark that works as above, except instead of changing a random user's location, it changes one of the users they are following. The results are shown in the right column of Table 7; only averaged results are shown because they are essentially unchanged when N is varied. We can get a sense of how the effect of each optimization has changed, relative to the last benchmark, by comparing the differential speedups listed in the two columns. For instance, maintenance case elimination is now responsible for a 10% improvement because the measured update actually exercises a code path improved by this optimization. In general, this benchmark causes more changes to tags and filters, and fewer changes (at most two per update) to the stored query result. This explains why result set elimination is almost completely ineffective now (0% instead of 27%), while inlining, which affects all tag and filter maintenance procedures, is more effective (10% instead of 4%). Native set optimization is also more effective because of the extra time spent on filters (time spent on the result set does not matter because the result set has already been eliminated by the time this optimization is applied). Counting elimination does not save as much time as before because a smaller proportion of time is spent in total on the result set and filters.

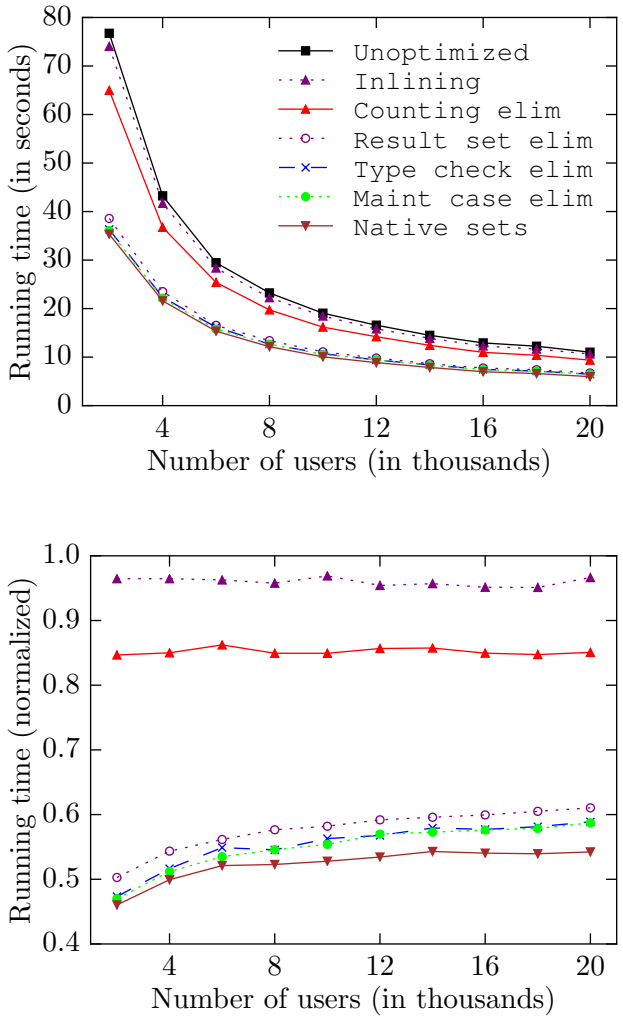

Figure 2: Running time for running example (querying + `user.loc` updates), (top) in seconds and (bottom) normalized to the time taken by `Unoptimized`. Time decreases for higher numbers of users because the data is sparser and therefore requires fewer incremental changes to the stored query result.

Table 7: Average improvement in running time over `Unoptimized` for running example, under the location-updating benchmark and the follower-updating benchmark. Parentheses indicate the differential improvement of each successive implementation.

	Speedup	
Implementation	`user.loc`	`followers`
`Inlining`	4%	10%
`Counting elim`	15% (+11%)	15% (+5%)
`Result set elim`	42% (+27%)	15% (+0%)
`Type check elim`	45% (+3%)	17% (+2%)
`Maint case elim`	45% (+0%)	27% (+10%)
`Native sets`	48% (+3%)	42% (+15%)

Table 8: Time to add a million unique integers to a data structure in random order. `s` and `n` hold the set and the number, respectively. `s_add` is a variable that is bound to the method `s.add`, an operation that is allowed by Python and which avoids repeatedly resolving the method lookup inside the timing loop.

Type	Operation	Time (s)
`CSet`	$\texttt{s.add}_c\texttt{(n)}$	1.14
`CSet`	`s.add(n)`	0.82
`Set`	`s.add(n)`	0.30
`Set`	`s_add(n)`	0.23

Native Sets In order to understand the effect of different low-level set implementations and ways of performing updates, we ran a benchmark that stress tests sets by performing addition updates without regard to any specific application. We generate a randomly shuffled list of one million unique integers and measure the time needed to add all numbers to the set. Results for four different configurations are shown in Table 8. By using `add` instead of $\texttt{add}_c$, switching from `CSet` to `Set`, and resolving the method lookup in advance of the timing loop, a speedup of up to 5x is achieved. This suggests that a comprehensive suite of optimizations should try to use native sets wherever possible. It is also worthwhile to develop a fast version of `CSet`, e.g. as a Python extension module, for situations where counting elimination does not apply.

Distributed Algorithms IncOQ has been used to incrementalize implementations of distributed algorithms written in DistAlgo [13, 15]. We have applied counting elimination, result set elimination, and native set optimization to these examples; result set elimination also has the effect of eliminating some relations that were in the input programs. In general, the optimizations reduce the size of the generated code, but do not reduce running time much, because the propagation of changes to the stored result tends to not be a bottleneck for these applications. The effect of the optimizations on code size is documented in Table 9.

Counting elimination applied to some but not all queries. In the case of Lamport's mutual exclusion (`lamutex`) [11], each of its three queries were optimized, but only two could be optimized automatically. A manual analysis of the third revealed that additional information is needed, at least for the particular implementation IncOQ generated. Namely, it requires that each process can receive at most one acknowledgement message from each of its peers for each request to enter the critical section. This assumption holds under our test conditions but not in general. An alternate implementation is also possible, in which no removal updates are performed for that query; in this case the optimization could be automated.

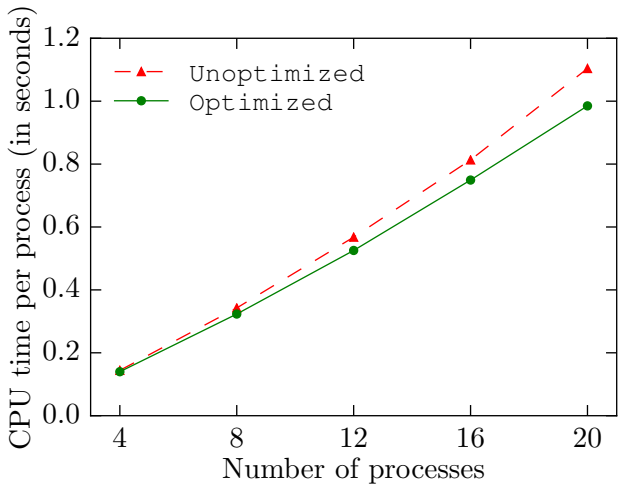

Figure 3: Running time for `ramutex` as the number of processes increase, and the number of requests to enter the critical section per process is held constant at 10.

Table 10: Running time improvement of optimized (including counting elimination, result set elimination, and native set optimization) vs. unoptimized, for distributed algorithms.

Program	`Unopt.` (s)	`Opt.` (s)	Improvement
`lamutex`	1.38	1.34	2.4%
`ramutex`	0.59	0.54	8.4%
`sktoken`	0.43	0.42	1.1%

Clear improvements were observed in three programs listed in Table 10; in the rest, the optimized and unoptimized versions were too similar to distinguish. Figure 3 illustrates the performance for `ramutex`, which had the biggest improvement.

Wifi Query Previous work on incrementalizing object queries has used the following query, which finds all access points of sufficient signal strength in a wifi object [20]:

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

The same query has also been used to benchmark IncOQ [13]. The benchmark procedure does N repeats of 1) creating an `ap` object satisfying the threshold condition and adding it to the set, and 2) performing the query. While running benchmarks for [13] it was determined that when IncOQ is used to generate two implementations — a filtered approach using tags and filters as described in this paper, and an unfiltered approach without these auxiliary data structures — the filtered approach runs slower on this example by a constant factor. This is because there is only one `wifi` value, and so every value is demanded, incurring the overhead of filtering with none of the benefit.

We re-examined this benchmark to study the effect of optimization on the runtime overhead of filtering. Because we are interested in comparing only incremental implementations and not the asymptotically slower straightforward (batch) implementation, we have drastically increased N from a maximum of 2,500 to a maximum of 100,000; the benchmark is otherwise unchanged. Figure 4 shows that the average ratio of the filtered approach's cost to the unfiltered's, without optimization, is 3.3x. With optimization applied to

Table 9: Optimization statistics for DistAlgo programs. The input program (Orig. LOC) is for a generated interface between the distributed algorithm and IncOQ, which includes all queries and updates. The number of relations removed includes both result sets introduced by IncOQ and relations that were already in the input program. Program names are as in [15].

Program	Orig. LOC	Unopt. LOC	Opt. LOC	# counted sets elim.	# relations elim.
`2pcommit`	74	1368	1226	8	14
`clpaxos`	152	1940	1754	10	26
`crleader`	27	156	125	2	4
`dscrash`	51	939	799	10	6
`hsleader`	64	421	357	4	7
`lamutex`	53	500	425	3	6
`lapaxos`	126	1382	1226	9	16
`ramutex`	51	762	693	3	6
`ratoken`	72	1059	952	8	5
`sktoken`	89	850	769	6	4

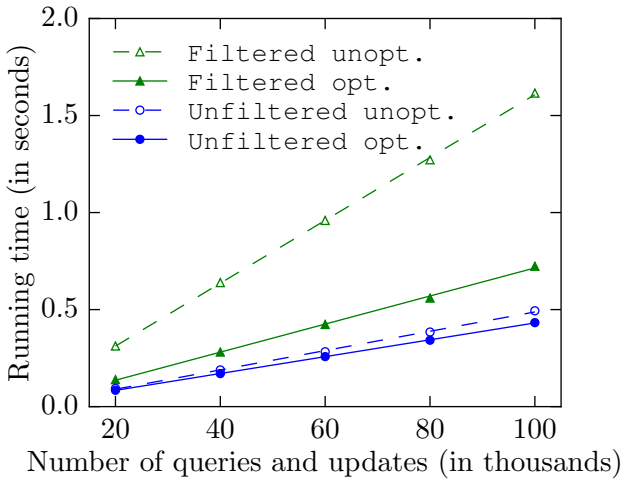

Figure 4: Running time for unoptimized and optimized versions of incremental implementations of the wifi query with and without demand filtering.

the filtered one (keeping the unfiltered one unchanged), this ratio goes down to 1.5x. Optimization has only a small effect on the unfiltered implementation, because counting elimination is applied only to the query result relation and not to the myriad filter sets that do not exist in this implementation. The effect is also bigger for the filtered one because type check elimination and maintenance case elimination were able to apply, whereas these optimizations are not in general safe for unfiltered versions.

8. Related Work

Incremental Object Query Systems We briefly discuss two alternatives to IncOQ for incrementally computing object queries; the reader is directed to [13] for more on IncOQ and its improvements over previous methods.

The Java Query Language (JQL) [23, 24] extends Java with object queries, however, it is not suitable for all applications that IncOQ is. JQL queries return ordered collections that may contain duplicates, so the user is responsible for duplicate elimination (and counts are not needed). Moreover, JQL cannot incrementally handle the case where a field of a nested object is updated, since it does not keep auxiliary data structures to efficiently find objects and collections that contain the updated object. Consequently, JQL cannot compute the running example query incrementally. Since Java is statically typed, JQL cannot express queries over ill-typed data.

The Object-Set Queries (OSQ) [20, 21] method is closely related to IncOQ, the main difference being that IncOQ is based on invariant-driven transformation. This makes it possible for IncOQ to provide asymptotic cost bounds as well as a high-level understanding of the actions to be executed at runtime, whereas OSQ requires a more intimate understanding of the dynamic interactions of its components. It is this high-level understanding that enables static optimization based on the original program. IncOQ also extends OSQ with a more general strategy for computing demand and with its ability to query ill-typed data.

Counting Elimination For a materialized view involving projection or join operators, it is common for maintenance algorithms to store a count for each tuple in the view. See for instance the aptly named Counting algorithm of Gupta et al. [9]. Blakeley et al. [2] identify that there is a choice between storing such counts, or else restricting the view in a way that guarantees that each count would be at most 1, in which case the counts are unnecessary. It is straightforward to generalize this choice so that views are not restricted yet counts are still omitted when possible.

Despite this, sometimes the added expressivity is deemed to not be worth the added conceptual baggage. This was the case for Ceri and Widom [3], which restricts the class of views it handles in order to avoid the complication of managing a separate count attribute in an SQL table. They point out that duplicate management is particularly tricky in SQL since there is no SQL operation that can delete some but not all duplicate tuples. In contrast, there is little conceptual baggage for tracking counts in our language, since counted sets are just an abstract data type extending normal sets. Our first case for counting elimination corresponds to how it is well known that counts are not needed when removal updates are not present; condition (1) of our second case corresponds to [3]'s algorithm for determining whether or not a particular view satisfies their requirements.

For nested queries, the Counting algorithm [9] has an optimization whereby changes that affect just the count of a tuple, but not the tuple's actual presence or absence, are not propagated to the outer query. This optimization is already inherent in IncOQ and therefore does not need to be exploited by this paper.

Other non-counting approaches are possible, such as the Delete and Rederive algorithm [9], or a strategy that rescans the projected relation [19]; see Gupta et al. [8] for a survey.

An earlier term for counted sets used by [20, 21] is "reference-counted sets", which brings to mind an analogy between counting the number of justifications for having an entry in a result, and the

number of justifications for keeping an object alive during garbage collection (GC). Counting elimination for stored query results can be thought of as analogous to statically replacing GC reference counts with explicit allocations and deallocations. This is a big performance gain that is hard to do in general. Other approaches seek to reduce the number of updates to the stored reference count [12], or to not track its value when it goes above a predetermined maximum size [22]. The former corresponds to eliminating some but not all $\texttt{add}_c$ and $\texttt{remove}_c$ operations, and generalizes to eliminating maintenance for sequences of updates that cancel themselves out before the next query operation occurs. The latter is not applicable to our use case because it would require recomputing the proper count at each query operation, and because the saved space (a fraction of a word per counted element) is quite small.

There is an unrelated "counting elimination" in probabilistic inference [5], where logical atoms are eliminated by means of a combinatorial counting argument.

Result Set Elimination The IncOQ method is based on finite differencing [18]. Result set elimination can be seen as a special case of the Clean stage of finite differencing, which applies a form of dead code elimination to the code for storing and maintaining intermediate results. In our case, the intermediate result is a query's result set, and the final result is a map acting as an index over it. When the result set is not counted, it is not needed by the code that maintains the map, and hence can be eliminated. Unlike traditional general-purpose dead code elimination [1, 17] which performs a fixed-point computation, we are able to eliminate result sets solely on the basis that 1) the set is not counted and 2) a use of the query is replaced by a map as opposed to by the set itself.

Result set elimination is neither applicable nor necessary for some other methods [16, 21] for generating incremental code, because those methods create the map directly, whereas IncOQ first creates a set and then generates a map to index it. IncOQ takes this approach because it makes it easier to support nested queries, since it means that the rule for transforming the outer query does not need to account for map lookups created by transforming the inner query.

Type Check Elimination Type check elimination can be seen as a form of partial evaluation [10] of the query maintenance code, where the inputs to maintenance can all be assumed to have the types expected by the type checks. The inputs to maintenance are the values participating in the update, the values reachable from the demanded parameter values, and the auxiliary data structures.

Optimization of dynamic object-oriented languages often focuses on inferring and using type information to statically resolve and inline method calls, e.g., for the SELF language [4]. In contrast to such general approaches, our optimization is based on high-level knowledge of the query, the queried data, and the way maintenance code is formed from queries.

A sophisticated type analysis, such as the one in Gorbovitski et al. [7], can be used to verify that the demanded parameter values do in fact have the required types. The type system we give in Section 4 is much simpler than the types used in such analyses because we only need enough information to determine whether an expression's evaluation can cause a type error.

Maintenance Case Elimination Our maintenance case elimination can be seen as a special case of the alias-based optimization done by InvTS [6, 14], another transformation system that can be used to generate incremental implementations of programs. The user of InvTS writes rules to specify what effect should occur when a value is updated, and it is up to InvTS to insert maintenance code at the appropriate locations, taking into account aliasing information. When there is no may-alias relationship between the expression in the update and the expression in the rule, no code is inserted; when there is a must-alias relationship, unconditional code is inserted; and otherwise, code guarded by a dynamic check is inserted.

Whereas InvTS uses sophisticated alias analysis [7] (which incorporates the aforementioned type analysis) to deduce alias relationships for general programs, our maintenance case elimination is based on high-level reasoning about demand and reachability from demanded parameter values.

Acknowledgment

We thank the anonymous reviewers for their detailed and helpful comments.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools, 2nd edition*. Addison-Wesley, 2006.

[2] José A. Blakeley, Per-Åke Larson, and Frank Wm Tompa. Efficiently updating materialized views. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pages 61–71, 1986.

[3] Stefano Ceri and Jennifer Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 577–589. Morgan Kaufmann, 1991.

[4] Craig Chambers and David Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. *ACM SIGPLAN Notices*, 24(7):146–160, June 1989.

[5] Rodrigo de Salvo Braz, Eyal Amir, and Dan Roth. Lifted first-order probabilistic inference. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pages 1319–1325. Morgan Kaufmann, 2005.

[6] Michael Gorbovitski. *A System for Invariant-Driven Transformations*. PhD thesis, Stony Brook University, Stony Brook, NY, USA, 2011.

[7] Michael Gorbovitski, Yanhong A. Liu, Scott D. Stoller, Tom Rothamel, and Tuncay K. Tekle. Alias analysis for optimization of dynamic languages. *ACM SIGPLAN Notices*, 45(12):27–42, 2010.

[8] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 18(2):3–18, 1995.

[9] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 157–166, 1993.

[10] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[11] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[12] Yossi Levanoni and Erez Petrank. An on-the-fly reference-counting garbage collector for Java. *ACM Transactions on Programming Languages and Systems*, 28(1):1–69, 2006.

[13] Yanhong A. Liu, Jon Brandvein, Scott D. Stoller, and Bo Lin. Demand-driven incremental object queries. *Computing Research Repository*, arXiv:1511.04583 [cs.PL], November 2015.

[14] Yanhong A. Liu, Michael Gorbovitski, and Scott D. Stoller. A language and framework for invariant-driven transformations. In *Proceedings of the 8th International Conference on Generative Programming and Component Engineering*, pages 55–64. ACM, 2009.

[15] Yanhong A. Liu, Scott D. Stoller, Bo Lin, and Michael Gorbovitski. From clarity to efficiency for distributed algorithms. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 395–410, 2012.

[16] Yanhong A. Liu, Chen Wang, Michael Gorbovitski, Tom Rothamel, Yongxi Cheng, Yingchao Zhao, and Jing Zhang. Core role-based access control: Efficient implementations by transformations. In *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 112–120, 2006.

[17] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 2015.

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

[19] Xiaolei Qian and Gio Wiederhold. Incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):337–341, 1991.

[20] Thomas Michael Rothamel. *Automatic Incrementalization of Queries in Object-Oriented Programs*. PhD thesis, Stony Brook University, Stony Brook, NY, USA, 2008.

[21] Tom Rothamel and Yanhong A. Liu. Generating incremental implementations of object-set queries. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, pages 55–66. ACM, 2008.

[22] Rifat Shahriyar, Stephen M. Blackburn, Xi Yang, and Kathryn S. McKinley. Taking off the gloves with reference counting Immix. *ACM SIGPLAN Notices*, 48(10):93–110, October 2013.

[23] Darren Willis, David J. Pearce, and James Noble. Efficient object querying for Java. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, pages 28–49. Springer, 2006.

[24] Darren Willis, David J. Pearce, and James Noble. Caching and incrementalisation in the Java Query Language. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, pages 1–18, 2008.