Demand-Driven Incremental Computation of Object Queries

Jonathan Glenn Brandvein

May 18, 2016

Abstract

Object queries are a powerful construct for writing clear and concise high-level code, but are difficult to implement efficiently. A programmer who is not satisfied with the performance of a straightforward implementation may craft an ad hoc low-level solution. However, this forfeits the productivity benefits that declarative queries bring.

We present a novel static transformation for generating demand-driven incremental implementations of object queries. The queries may involve objects and sets that are arbitrarily nested, and may also use aggregate operators and nested queries. All possible updates are handled, even in the presence of aliasing, without requiring sophisticated runtime support. The method defines invariants for the query result and auxiliary values, and provides precise asymptotic bounds for running time and space usage. Different high-level choices in the selection of the demand strategy and join orders lead to different invariants and different time and space tradeoffs. Previous methods do not handle such expressive queries while obtaining precise cost bounds for the generated code.

We demonstrate IncOQ, a prototype implementation of our method, and verify that the transformed programs conform to their predicted costs. We compare our method both analytically and experimentally with prior work and alternative approaches, and justify our formulation of demand. Finally, we show successful applications to queries from a variety of problem domains including distributed algorithms, access control, and approximate probabilistic inference.

For my parents,
without whom I would not be here,
financially, academically, or existentially;

and for my sister,
who gives wisdom, support and unconditional love.

Contents

Acknowledgements

The canonical, age-old wisdom that is passed down from graduate student to prospective graduate student is, "Don't do it. Run now, while you still can." And so we're a stubborn bunch, composed almost entirely of people who ignored this very sound advice.

I can't begin to name all the people I've encountered in my time at Stony Brook. Here at the other end of it, I'm not sure I'd even recognize myself from when I arrived. It is because of my professors, students, friends, labmates, and roommates that I was able to grow as both a researcher and a person. Grad school has taught me how to be thorough and conservative — how to think before I speak. It's shown me how to write, how to construct an argument, how to judge an idea by its merits. It's also taught me not to be intimidated by big ideas and projects with broad scope.

I'd like to thank my advisor, Prof. Annie Liu, for her guidance throughout the years. It was her animated passion for good ideas that drew me to her, and I've relied on her vision, critical feedback, and objectivity ever since. Likewise, I thank Professors Scott Stoller, Michael Kifer, David Warren, and external member Dr. Peter Norvig, for gratiously agreeing to serve on my committee. Scott in particular I thank as a coauthor with Annie and myself.

This work is an evolution of work done by Tom Rothamel and others, and I thank Tom for letting me stand on his shoulders. I also owe a debt of gratitude to my labmates, including in particular Michael Gorbovitski, Tuncay Tekle, Puneet Gupta, Bo Lin, Xuetian Weng, and Christopher Kane, for letting me bounce ideas off them and giving me some fantastic conversations.

Some of my happiest times over the past several years have been at the Center for Talented Youth in Lancaster, PA, where I met so many vibrant and brilliant people. They let me exorcise the teaching daemon, and gave me a sense of purpose beyond day-to-day research.

To my roommates at one time or another — Navid, Roozbeh, Heraldo, Riccardo, Estefy, Behrad — I've never felt as accepted as I did in our group.

Whether it was talking science, politics, or pizza, you guys always made it worthwhile.

I could not have done any of this without the constant support of my family: My father Michael with his sound advice, my mother Robin with her persistent optimism, and my sister Carolyn with her perspective and encouragement. And finally, I'm thankful for my fiance Bhumika, who has given me something new to look forward to now that I'm graduated.

Chapter 1

Introduction

The principle advantage of high-level programming languages is that computations can be written in a simpler and more concise manner. This makes code more understandable, and by extension, reduces bugs, increases programmer productivity, and enables faster development. Query constructs help to achieve this goal by enabling the programmer to declaratively write the desired result of a computation, rather than giving one specific way of computing it.

This is indeed the rationale behind dedicated query languages such as SQL and Datalog [1]. Many general purpose programming languages support high-level queries as well. For example, Haskell and Erlang have list comprehensions, Python has comprehensions for various datatypes,[1] and C# has LINQ [55]. Other languages may only support the use of low-level loops in place of high-level queries. Whereas SQL and Datalog are relational, meaning that they operate only on flat sets of tuples, the query constructs in object-oriented languages typically permit expressions that retrieve object fields. This is especially useful because objects are better than flat sets at modeling the real world.

The straightforward way to implement a comprehension query is to turn each membership clause into a nested loop, executed at the same point in code where the query appears. However, a programmer who is worried about performance might decide to hand-code an alternative implementation, trading code clarity for execution efficiency. Our aim is to make this tradeoff obsolete by providing better automatic translations of queries into efficient low-level code.

Specifically, the implementations we derive are *incremental* and *demand-driven*. "Incremental" means that the query result is stored and made available

[1] `https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions`

for fast retrieval, and incrementally maintained when updates to data values affect the query result. "Demand-driven" means that we prune the stored data and maintenance computation, so that resources are only expended for data that is relevant to the queried values.

The queries we consider may, in a nested manner, retrieve elements from sets and fields from objects. Extensions add support for maps, tuples, aggregates, and negated membership conditions. Maintenance code is generated for all updates that can possibly affect the query result. The method works even in the presence of aliasing that may exist among the objects that are queried and updated.

Our method lends itself naturally to producing asymptotic cost bounds, both on the running time of the maintenance, and on the space used for storing the query result and the needed auxiliary values. In particular, due to our use of demand, we can place a bound on these costs that is a function of the size of the relevant data. When queries are not nested, this bound is only linear in the total size of the relevant data and the query result.

As an example, consider an access control query that decides whether or not a given user is authorized for a given permission. The determination is based on finding all roles that the user is assigned, and checking whether any of these roles has the requested permission. Since this kind of authorization check occurs frequently, it is too expensive to reevaluate from scratch each time, but the objects queried over can change at any time. Here are a few compelling strategies, each more difficult to implement than the previous.

(1) A caching (memoizing) approach would create a partially-filled two-dimensional lookup table that maps from a user and a permission to a Boolean answer, if one is available. Each table entry starts off uninitialized, and is populated by a from-scratch computation when its answer is first needed. We have to detect changes to the access policy (e.g., role updates) that potentially invalidate cached answers, and purge the affected entries.

(2) An incremental approach also uses a table, but its cells hold integers representing the number of different roles that justify the user having that permission. The query answer is `True` so long as the count is greater than zero. At updates, we need to increment or decrement affected counts appropriately, but no scratch recomputation is required.

(3) A demand-driven incremental approach works as an incremental approach, but only tracks table entries that are demanded. The decision of

which entries should be considered demanded is a matter of policy, but it should at least include those entries for which the query result needs to be readily available. For instance, if we are only processing access requests for a particular subset of permissions, then the table should be pruned of all rows for permissions outside that subset.

Demand for query parameters induces demand for other query variables. If p is a demanded permission, all of the roles that have p are tagged as demanded roles. We know whether an update to a role may be relevant by checking whether the role is demanded. If not, no work beyond that simple test is done.

Opting for one of the above designs may be worthwhile, but they are each tedious to implement by hand. The programmer would have to locate all updates in the program source, and, at each one, write logic to determine what entries need to be purged or incrementally maintained. For a demand-based strategy, there would also have to be logic to propagate (and un-propagate) demand to other values reachable from the queried parameters. Through it all, there are plenty of opportunities for a bug caused by one missed update or one overlooked dependency. And without following a systematic method, the programmer would be on his or her own to determine the cost of the program. These problems are all amplified for larger queries, especially if aliasing is possible.

Our solution to this problem is a systematic method comprising a static transformation and a cost analysis. The method is invariant-based, where the invariants are for the different auxiliary results that are materialized. This allows us to reason about costs and correctness at a high level. To the extent that there are choices in how the invariants are defined and implemented, we can produce a family of generated programs, all of which are correct but may have different cost tradeoffs.

Our approach incorporates ideas from previous work, but combines and extends them in a novel way to achieve new results. Among the most important techniques we draw on are a rewriting that flattens nested sets and objects, and a method for generating code to incrementally compute simple set comprehensions. We add to this an invariant-based formulation of demand, whereas prior techniques either did not use demand or dealt with it in an ad hoc way. No previous method is able to handle queries that are as expressive as ours while still giving precise cost bounds on time and space. Additionally, the generated code does not require the addition of a sophisticated runtime environment.

The rest of this dissertation is organized as follows. Chapter 2 presents the transformation method and cost analysis. We discuss the implementation and perform benchmarks in Chapter 3. These benchmarks confirm our asymptotic cost predictions and also illustrate smaller constant-factor tradeoffs related to the low-level details of the generated code. Comparison with related work, both theoretically and empirically, is done in Chapter 4. In Chapter 5 we apply our method to queries from distributed algorithms, access control, approximate probabilistic inference, and a student information management system. Finally, we conclude and discuss future work in Chapter 6.

This work was supported in part by NSF[2] and ONR[3]. Parts of this work appeared in [12] and [45].

[2]Grants CCF-1414078, CNS-1421893, IIS-1447549, CCF-1248184, CCF-0964196, and CCF-0613913.

[3]Grants N000141512208 and N000140910651.

Chapter 2

Incrementalizing object queries

This chapter describes our method for generating demand-driven, incremental programs with cost bounds. Our approach is based on three main ideas.

(1) The complex and expressive nature of the original object query can be simplified by converting it into an equivalent relational query, in which updated values cannot be aliased.

(2) The relational query can be computed incrementally by storing its result in a fresh variable, and maintaining the invariant that this stored value equals the result of the query. This transformation inserts *maintenance code* around updates that would violate the invariant. The maintenance code performs incremental computation as is needed for maintaining materialized views in a relational database.

(3) The computation can be made demand-driven by introducing new auxiliary structures to index just the relevant part of the input data. The auxiliary structures are themselves computed incrementally by maintaining invariants.

The first step is based on a translation given in [74, 72], but we extend it to handle aggregates, nested queries, and more forms of pattern matching. We also provide more precise requirements for determining when the query is well-formed and how the maintenance code deals with errors, particularly type errors. The second step is based on several previous techniques for computing set queries incrementally, as well as on the static transformation framework of finite differencing [66]; we add a number of optimizations and support for negated membership constraints. The third step is novel, in that it is the first invariant-based technique for computing demand-driven indices for object-set

queries. It supports high-level reasoning about time and space cost bounds, as well as generation of correct code for demand propagation in the presence of nested queries. It also provides an intuitive understanding of how values in the input data will be used by maintenance code, which leads to further optimization. See Chapter 4 for a full discussion of related work.

The overall time and space costs will depend on the query, its updates, and high-level choices that are made during the transformation. These high-level choices comprise a demand strategy for the query, and a join order for each piece of inserted maintenance code. We provide simple heuristics to make reasonable default choices, but these can be overridden by the user of our method — for instance, to take advantage of specialized domain knowledge, or to automatically generate multiple alternative implementations.

The queries, updates, and object model to which our method applies are given in Section 2.1. Sections 2.2 through 2.4 describe the three main ideas mentioned above. Section 2.5 explains how to add support for nested queries; aggregate queries; and queries that involve maps, nested tuples, negated membership constraints, exceptions, and pattern matching. Section 2.6 describes a number of optimizations to reduce the running time, space usage, and code size of the generated implementations.

2.1 Language

Our method applies to any language supporting the query and update operations described here. The two main kinds of values are sets and objects; other kinds include primitives and tuples. A set is an unordered collection without duplicates. An object is an unordered mapping from field names to values. All set and object values are stored by reference, so expressions that evaluate to these values may be aliased. Sets and objects may contain values of any type and may even contain themselves. For ease of presentation, we assume that the language is garbage-collected, although the method can be extended to languages with explicit memory management. We also assume that all distinct variables are renamed to ensure that their identifiers are distinct regardless of scoping.

The equality (`==`) and inequality (`!=`) operators for sets and objects are defined based on identity: If v_1 or v_2 is a set or object, then v_1 `==` v_2 iff they are the same value in memory. In other words, the "value" of a set or object is its identity, not its contents. This means that updates to sets and objects (i.e., to their contents) cannot cause two unequal values to become equal — a situation that would be problematic if the two values were originally elements

Expression	Return value	Precondition
`new set`	a new empty set	
v `isset`	whether v is a set	
v `in` s	whether v is an element of s	
v `not in` s	`not (`v `in` s`)`	
s `isempty`	whether s has no elements	
`arb` s	an arbitrary element of s	$\|s\| > 0$
`new obj`	a new object with no fields	
v `hasfield` f	whether v is an object and $f \in fields(v)$	
$o.f$	the value of field f of o	$f \in fields(o)$

Update	Effect	Precondition
s`.add(`v`)`	add element v to s	$v \notin s$
s`.remove(`v`)`	remove element v from s	$v \in s$
$o.f$ `=` v	create f on o and give it value v	$f \notin fields(o)$
`del` $o.f$	remove field f from o	$f \in fields(o)$

Table 2.1: Set and object operations. An exception is raised if any precondition is not satisfied. Any operation using s or o has the implicit precondition that it evaluates to a set or object value, respectively; v may evaluate to any value. f is a field name. $fields(o)$ denotes the set of field names for the object referred to by o. All operations are considered to take constant time.

of the same set, since they would become duplicates. We say that a value v is *accessible* from u if, starting at u, we can obtain v by moving zero or more times from a set or object to any of its elements or field values.

Operations on sets and objects are given in Table 2.1. More complex operations such as set-wise updates (union, difference, etc.), and element-wise updates without the preconditions, can be preprocessed into iterated or guarded forms of the operations in Table 2.1. We assume that sets are implemented as hash tables and have constant-time lookups. We assume that all expressions appearing in updates are deterministic, side-effect free, and that their values are not changed by the update. A preprocessing step can ensure this by inserting code to evaluate expressions before the update and store their result in a fresh variable.

The query construct we use is an *object-set comprehension* [74, 72], which

has the following abstract grammar.

$$\begin{aligned} \textit{object-set-comp} &::= \texttt{\{}\ \textit{result}\ \texttt{:}\ \textit{clause}^*\ \texttt{\}} \\ \textit{clause} &::= \textit{membership} \mid \textit{condition} \\ \textit{membership} &::= \textit{retrieval}\ \texttt{in}\ \textit{retrieval} \\ \textit{retrieval} &::= \textit{variable} \mid \textit{retrieval}.\textit{field} \\ \textit{condition} &::= \textit{expression} \\ \textit{result} &::= \textit{expression} \end{aligned}$$

It returns a new set consisting of all the values that its result expression can take on, for each combination of variable values that satisfies all of its clauses. Some of the variables in the query are designated as *parameters*, which will be denoted using underlines; the rest are locals. In practice, the programmer can declare parameters explicitly or they can be determined automatically using the host language's scoping rules. To ensure that object-set comprehensions are declarative, and to avoid assuming any particular evaluation order for the clauses, we impose a *safety requirement* and a *reachability requirement*.

The safety requirement ensures that our incremental maintenance code will be able to properly handle any kind of update. It says that for each condition expression and for the result expression, the expression's behavior is terminating, deterministic, and dependent only on the values of its maximal retrieval subexpressions (including variables).[1] The safety requirement can be subtle; for example, the expression `x in s` depends not just on the values of (i.e., the identities of) `s` and `x`, but also on the elements of `s`, and hence is not suitable as a condition clause. This is why the grammar does not treat memberships as a subcase of conditions.

The reachability requirement ensures that the incremental computation can be made demand-driven. We say that an expression in the query is *reachable* from the parameters if one of the following cases holds, inductively: (1) the expression is a parameter variable; (2) the expression is e_2 and there is a membership e_2 `in` e_1 where e_1 is reachable; or (3) the expression is $e.f$ for some field name f, and e is reachable. The requirement simply states that each retrieval expression (including variables) in the query must be reachable. This ensures that all values that can satisfy the query are accessible from the parameter values. It prohibits queries like

`{o : o.f in` $\underline{\texttt{s}}$`}`,

[1] By maximal retrieval subexpression we mean any retrieval that has at least one occurrence that is not nested inside another retrieval. For example, `o.f` is maximal in `o.f + o.f.g`, but `o` is not.

where there is no straightforward way to obtain values for `o` from the given value for `s`.

To give precise semantics for object-set comprehensions, define a *valuation* to be a total function from the query's variables to values. At the time a query is evaluated, a valuation is said to be *valid* if it maps each parameter to its given value, causes each clause to be satisfied, and does not cause any exceptions to be raised. The result of the query is a new set holding all values of the result expression under each valid valuation. Note that the query result may be changed not only by assignment to the parameter variables, but also by updates to sets and objects that are accessible from the parameter values. For the sake of simplicity, we will start off assuming that the only kinds of exceptions that can occur are when a field retrieval is evaluated where the left-hand side is not an object providing that field, or when a membership is evaluated where the right-hand side is not a set. We will eliminate this assumption in Section 2.5.

When discussing maintenance code and its cost bounds, it is often useful to talk about the possible values that a variable or expression may take on to satisfy or partially satisfy the query. We make this notion concrete as follows: A value v is *relevant* (negation: *irrelevant*) for a retrieval expression if, inductively, (1) the expression is a parameter and v is a value for that parameter; (2) the expression is e_2, there is a membership e_2 `in` e_1, and v is a member of a set that is relevant for e_1; or (3) the expression is $e.f$, and v is the value of field f of an object that is relevant for e. Notice that these cases mirror the reachability requirement. We may omit the expression when it is clear from context, or when we wish to say that a collection of values are all relevant for their respective expressions. A value that is inaccessible from the parameter values is always irrelevant for all expressions in the query.

An object-set comprehension can be computed using a straightforward nested-loop join. The clauses would be translated into imperative code consisting of nested `for` and `if` statements with appropriate type checks. Different nesting orders for the clauses' translated statements may be possible, yielding different running times. Assuming that the conditions and result expression take constant time to evaluate, this strategy would take worst-case time proportional to the product of the iterated sets' sizes.[2] In contrast, if computed using an incremental approach, the result can be retrieved from a data structure in constant time, and it takes only linear time in the size of the result to

[2]Our use of big-O notation assumes all variables are increasing, so $O(x + y + (x \times y))$ can be simplified as $O(x \times y)$ without worrying about whether one variable is zero. All space costs are for added data structures, i.e., not counting space usage of the original program's data.

generate a fresh copy for the user code. The copy and its linear-time cost are not needed if the user code does not directly update the retrieved result, and does not update the values the query depends on until after it is done using the retrieved result.

Example 1. Consider the following social network query, which takes in a special celebrity user and a group of users, and finds the email addresses of everyone who follows the celebrity, belongs to the group, and has a location of `"NYC"`.

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

Suppose that the value `celeb` is an object `alice`, whose `followers` field is a set containing an object `bob`. Then `alice` is relevant for `celeb`, `alice.followers` is relevant for `celeb.followers`, and `bob` is relevant for `user`. This is true regardless of whether or not these values are actually used in any valid valuation to satisfy the query.

Here is one possible straightforward computation. The type checks are written in slanted text and can be deleted if the data is known to be well-typed.

```
result = new set
if celeb hasfield followers:
  if celeb.followers isset:
    for user in celeb.followers:
      if group isset:
        if user in group:
          if user hasfield loc:
            if user.loc == "NYC":
              if user hasfield email:
                if user.email not in result:
                  result.add(user.email)
```

This code takes $O(|\texttt{celeb.followers}|)$ time. If we had instead chosen to iterate using the second membership first, it would take $O(|\texttt{group}|)$ time. Either cost may be prohibitive if the query is to be performed frequently and if the size of the iterated set is large. Memoization based on the parameters is also not a solution because the sets and object fields retrieved from the parameters may change in-between runs.

If the programmer attempts to write an incremental implementation by hand, he or she may be caught off guard by the number of cases that need

to be handled. Without assuming any particular constraints on how the program data is arranged and manipulated, possible updates affecting the result are: additions and removals to follower sets and groups, reassignment to a celebrity's `followers` field (i.e. replacing their set of followers with an entirely different set value), changing a user's location or email, and reassignment to the parameters themselves. All of these updates except the last may happen via aliasing from arbitrary expressions.

Aliasing can also exist between different expressions in the query, leading to potentially surprising results. The same set could serve as both a value for `celeb.followers` and `group`, so that an update to it affects the query in two different ways. Two celebrities may share the same set of followers, and the celebrity may even follow itself. Furthermore, two users may share the same email address, so that the size of the result is smaller than the number of users satisfying the query. Finally, mixed in with the well-typed data, there may be ill-typed data (e.g., users with no email) that should not contribute to the query result. Even after accounting for all these cases, the computation still has not been made demand-driven.

Our method automatically derives correct code for all of these cases, with systematic calculation of cost bounds, regardless of which of the above scenarios are actually used in the particular application. It also performs optimizations made possible when some of these cases do not occur. Furthermore, it avoids taking any storage space, or more than a constant factor of time, for all the data and updates pertaining to users who do not follow the given celebrity or belong to the given group. □

2.2 Rewriting to relational queries

Object-set comprehensions can have a complex, hierarchical structure that makes it difficult to see how to generate incremental maintenance code. In order to handle all possible kinds of updates correctly and systematically, the query is rewritten as a *relational comprehension* before incrementalizing it. There are three main steps to this rewriting:

(1) *Flatten* retrieval expressions and memberships, turning them into memberships over non-nested sets of tuples.

(2) *Strengthen* the query by rewriting the result expression to include all parameters, and changing these parameters to local variables. The new comprehension represents the original query's result for all values of its parameters.

(3) *Restrict* the former parameters to a specific domain, called the *demand set*, so that the comprehension is only concerned with results for combinations of parameter values in this set.

These steps are adapted from [74, 72], and discussed with attention to how each step preserves the semantics of the use of the query. Flatten is the main step that allows our method to work on such expressive queries. Strengthen makes it so a reassignment update to the parameters does not always require a recomputation of the query result. Restrict ensures that the relational comprehension has finite domain while giving all needed results.

We begin by describing the form of relational comprehensions and their operations. We then discuss each step in turn. By the end of this section, we will have converted the query to an expression whose value can be incrementally maintained by the invariant-based transformation.

Relational comprehensions: A *relation* is a global,[3] unaliased variable that is never reassigned and that holds a set of tuples having the same arity. A relational comprehension is similar to an object-set comprehension, but each membership has a tuple of variables on its left-hand side and a relation on its right-hand side, and relations do not appear anywhere else in the query. In addition, the result expression must be a tuple expression. There are no retrieval expressions. Relational comprehensions must have at least one membership clause.

$$
\begin{aligned}
\textit{relational-comp} &::= \texttt{\{}\ \textit{result}\ \texttt{:}\ \textit{membership}^{+}\ \textit{condition}^{*}\ \texttt{\}} \\
\textit{membership} &::= \texttt{(}\textit{component}^{*}\texttt{)}\ \texttt{in}\ \textit{relation} \\
\textit{component} &::= \textit{variable} \\
\textit{condition} &::= \textit{expression} \\
\textit{result} &::= \textit{tuple-expression}
\end{aligned}
$$

All variables besides the relations are local variables; we do not underline the relations because they are always understood to be parameters. Since relations are global, this means that the query's value is well-defined everywhere in the program. Relational comprehensions must satisfy the safety requirement. There is no reachability requirement, but there is a *domain requirement*, which states that all variables must appear on the left-hand side of at least one membership.

Relational comprehensions are essentially conjunctive queries [16, 1], or in relational database terms, select-project-join queries, except that the result expression can perform computations on the projected variables. The only

[3] A global variable is accessible in any scope and its value is always defined.

kinds of updates that may affect relational comprehensions are addition and removal of tuples to and from the relations. The technique for computing them incrementally, described in Section 2.3, is similar to techniques for maintaining materialized views in databases [26].

Example 2. The `CheckAccess` query in Role-Based Access Control (RBAC) can be modeled with the following expression,[4] which is almost a relational comprehension except that it has (non-relation) parameters.

```
{(role,) : (role,) in ROLES, (session, role) in SR,
           (op, obj, role) in PR}
```

Following Python, we write a singleton tuple (a tuple of arity 1) with a comma after its single component, and the unit tuple (a tuple of arity 0) as empty parentheses. The relations `ROLES`, `SR` and `PR` represent the domain of all roles, a session-role mapping, and a permission-role mapping, where a permission is a combination of an operation and an RBAC object. The return value is the set of all (singleton tuples of) roles that afford a given session a given permission. □

We need some notations for projecting tuples and for indexing over relations. First, given a tuple value $t = (t_1, \ldots, t_k)$, and integers $S = s_1, \ldots, s_n$ in increasing order with each s_i in range 1 to k, let t_S denote the projected tuple $(t_{s_1}, \ldots, t_{s_n})$. The same notation applies to tuple expressions, so that if e is a tuple expression `(`e_1`, ..., `e_k`)`, the projected tuple expression e_S is `(`e_{s_1}`, ..., `e_{s_n}`)`. (For readability in our examples, we write S as a concatenation of digits.) If E is an expression evaluating to a set of singleton tuples $\{(v_1,), \ldots, (v_n,)\}$, then the expression `unwrap(`E`)` returns a new set containing $\{v_1, \ldots, v_n\}$.

Next, suppose that E is an expression evaluating to a set e of tuples of arity k. Let I and J be two sets that partition $\{1, \ldots, k\}$, and let V be an expression evaluating to a tuple v of arity $|I|$. We define the *image-set expression*

$$E\,.\,J\{I = V\},$$

to return a new set (the image set) containing all tuples t_J such that $t \in e$ and $t_I = v$. We may also write $\{I_1 = V_1, \ldots, I_{|I|} = V_{|I|}\}$ in place of $\{I = V\}$. I and J are called the bound and unbound components, respectively. For cost purposes, the notation $|E\,.\,J\{I = V\}|$ refers to the cardinality of a particular image set, and $|E_{J/I}|$ refers to the largest cardinality that can be obtained

[4] Adapted from [52].

over any choice of V. As shorthand, we write E`.out{`V`}` for E`.2{`$1 = V$`}` and E`.in{`V`}` for E`.1{`$2 = V$`}`, and similarly we write $E_{\texttt{out}}$ for $E_{2/1}$ and $E_{\texttt{in}}$ for $E_{1/2}$.

Example 3. If `E` is an edge relation of a graph, then the image-set expressions `E.out{x}` and `E.in{x}` return the sets of all successor nodes and predecessor nodes of `x`, respectively, with each returned node wrapped in a singleton tuple. The terms `E.out{x}` and $|\texttt{E}_{\texttt{out}}|$ refer respectively to the outdegree of `x` and the largest outdegree of any node in the graph. □

Section 2.6 gives an optimization for eliminating singleton tuples where they occur in the generated code. For the sake of readability, our code examples will make use of this optimization.

Flatten: New special relations are introduced to model the relationship between sets and their elements, and between objects and their field values. Updates to the original program's sets and objects are treated as updates to these relations. The query comprehension is rewritten in terms of these relations, making it easier to see all the ways in which updates affect the query.

Two kinds of special relations are introduced: a single relation M for sets, and a family of relations F_f for each field name f. They are defined as:

$$(s, e) \in M \iff e \in s$$
$$(o, v) \in F_f \iff o.f = v$$

These relations are not actually materialized in the final program. Updates in the original program are treated as if they were updates to M and F_f.[5]

s`.add(`e`)`	$\Leftrightarrow$	M`.add((`s`, `e`))`
s`.remove(`e`)`	$\Leftrightarrow$	M`.remove((`s`, `e`))`
$o.f$ `=` v	$\Leftrightarrow$	F_f`.add((`o`, `v`))`
`del` $o.f$	$\Leftrightarrow$	F_f`.remove((`o`, `$o.f$`))`

To flatten comprehensions, first, all non-variable retrieval expressions are eliminated. For each distinct field retrieval $e.f$ (where e may itself be a retrieval), a fresh variable v is introduced, and all occurrences of the retrieval are replaced with v. A new membership `(`e`, `v`) in` F_f is inserted to define v. Next, each membership e_2 `in` e_1 of the original query is replaced with

[5]The last rule requires additional rewriting to first move the field retrieval $o.f$ into a fresh variable v, since its value is affected by the update.

(`e1, e2`) `in` M. Finally, the result expression r is replaced with the singleton tuple (`r,`), and the entire query is put inside a call to `unwrap()`. All of these steps ensure that the flattened comprehension fits the form of a relational comprehension, except that it still has (non-relation) parameters and might not satisfy the domain requirement. Any value that is relevant for a retrieval expression in the original query is also relevant for the corresponding variable in the flattened comprehension.

Example 4. The social network example is flattened as follows. (We abbreviate `followers` as `fol`.)

```
unwrap({(user_email,) :
        (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})
```

Note that there is exactly one membership for each of the kinds of updates mentioned in Example 1 except assigning to `celeb` and `group`. The self-join on M corresponds to how, when both `celeb.followers` and `group` are aliased to the same set, an update to that set affects the query in two different ways. If a value is relevant for `celeb_fol` in this comprehension, it is also relevant for `celeb.followers` in the original query. □

Strengthen: After flattening, there are two kinds of updates that can affect the query result: changes to the underlying relation, and reassignment to the parameter variables. The latter update is problematic for incremental computation because there need not be any similarity between the old and new results. In Example 2, it's easy to see that for any particular combination of a session and permission, the set of matching roles is mostly the same when we add or remove tuples to any of the relations. But if we then change the session or permission under consideration, we may expect a completely different result.

Strengthening takes care of this by changing occurrences of the parameters inside the comprehension into local variables. This makes the comprehension result include all of the original query's results for all its parameter values simultaneously. (Here, "all" is an infinite domain of possible parameter values, but that will be rectified in the next step.) The result expression is rewritten to mark each result entry with the parameter values that it corresponds to. The use of the query is wrapped inside an image-set expression to select just those entries corresponding to the current values of the parameters at the time the query operation actually occurs. However, the results for many parameter

values will be incrementally maintained. Thus, reassigning to a parameter variable will not affect the stored result of the comprehension, although it will affect how this stored result is retrieved.

Let the parameters be $p_1, \ldots, p_n$, and let `(`r`,)` be the result expression (which must be a tuple due to the Flatten step). Strengthening renames the occurrences of the parameters inside the query into local variables $p'_1, \ldots, p'_n$, and prepends these to the result expression to make it `(`p'_1`, ..., `p'_n`, `r`)`. The use of the comprehension is rewritten as the image-set expression

$$E.(n+1)\{1 = p_1, \ldots, n = p_n\},$$

where E is the comprehension. In the following two examples, the parameters are underlined while the new local variables are not.

Example 5. After applying stregthening to Example 2, we obtain

```
{(session, op, obj, role) :
    (role,) in ROLES, (session, role) in SR,
    (op, obj, role) in PR}.4{1 = session, 2 = op, 3 = obj}.
```
□

Example 6. After stregthening the flattened query from Example 4, we obtain

```
unwrap({(celeb, group, user_email) :
        (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
       }.3{1 = celeb, 2 = group}).
```
□

Restrict: The domain requirement says that all variables must appear on the left-hand side of at least one membership. This is needed because the incremental maintenance code will enumerate values for these variables. The Restrict step guarantees that this is the case. It also introduces the first aspect of demand: limiting what results are stored. This improves time and space costs in and of itself, but further improvement will come after we introduce demand filtering in Section 2.4.

A new relation, U, is introduced, called the *demand set* for the query. Let $p_1, \ldots, p_n$ be the query's parameters. A subset of these will be chosen to be the *demand parameters*, each of which corresponds to a component of U's tuples. Without loss of generality, assume that the first k parameters are the demand parameters. Restricting adds a new membership

`(`p'_1`, ..., `p'_k`) in `U

to the comprehension, where p'_i is the local variable corresponding to p_i. This membership is called the *demand clause*.

The meaning of this step is that the comprehension will include results of the original query only for the combinations of demand parameter values that appear in U. We say that such combinations and their corresponding results are *demanded*. The notion of relevant values is extended so that all values in the i^{th} component of tuples in U are considered relevant for p_i; this in turn means that more values are relevant for other expressions in the query.

So long as the current combination of values of the demand parameters are present as a tuple in U at the time the query occurs, the Restrict step does not change the semantics of the use of the query. To ensure this, new code is inserted just before the query to add ("demand") the current combination to U if not already present. Subject to this one requirement, our method permits any strategy for deciding when to demand and undemand parameter values — e.g., predictively demanding values long before the query occurs, or using a least-recently-used cache to undemand values when they do not appear to be needed.

The maintenance code generated for updates to U will do the actual work associated with demanding and undemanding parameter values. For an addition to U, the maintenance code is comparable to the code that would be run to compute the query from scratch in a non-incremental implementation. There is a tradeoff concerning how many tuples we keep in U. The more tuples, the more queries we will be able to answer immediately, without paying any time cost to demand new parameter values. However, we will pay a higher space cost to store all those demanded results, as well as a higher time cost at all other updates to maintain them.

It is required that the demand parameters are chosen such that, if all non-demand parameters were local variables, the query would still satisfy the reachability requirement. (Note that it is always valid to use all parameters as demand parameters.) This has two effects. First, it ensures that the relational comprehension we get after this step satisfies the domain requirement, since any variable appearing only in a condition or in a result expression must be a parameter and furthermore a demand parameter, and hence be added to the demand clause. Second, it makes it so that the relational comprehension is *connected*, meaning that if we put an edge between every pair of memberships that have at least one variable in common, the resulting graph is connected. This is important for generating maintenance code that does not rely on enumerating the unmaterialized special relations M and F_f.

Example 7. Restricting the query from Example 6 adds `(celeb, group) in` U as the first clause. Both of the parameters must be demand parameters.

Suppose that `alice` and `charlie` are both values in the first component of tuples in U, and that `bob` and `dave` are users following `alice` and `charlie` respectively. Then `alice` and `charlie` are both relevant for `celeb`, while `bob` and `dave` are both relevant for `user`. □

2.3 Transforming into incremental computation

In the last section we rewrote the query as a relational comprehension in order to make its structure more uniform, and to make it less dependent on arbitrary changes to its parameters. This section shows how to transform the program to compute the relational comprehension incrementally. There are three main steps.

(1) *Incrementalize* the comprehension by replacing its occurrence in the program with a new variable that holds its stored result. To keep this stored result up-to-date, maintenance code is inserted next to each update to a relation appearing in the comprehension. The maintenance code uses *maintenance joins*, which are a slight extension of relational comprehensions, to express the set of changes to the stored result.

(2) *Expand Joins* by turning them into equivalent low-level code that performs an index join using image-set expressions. The final asymptotic costs are affected by the choice of *join order* in this step.

(3) *Implement Indices* by incrementalizing image-set expressions and unwrap expressions.

There may be several relational comprehensions to be transformed, either because the original program had several queries to compute incrementally, or because steps for demand filtering or optimization generated multiple comprehensions from a single query. The first two steps apply once per relational comprehension, while the third step is done at the end of the transformation, after all comprehensions have been processed.

Both the Incrementalize and Implement Indices steps rely on finite differencing [66], an invariant-based transformation for incremental computation. The technique in this section is similar to the one used in [74, 72], but ensures that the generated joins are disjoint, which can save an asymptotic amount of

temporary storage space. It also handles ill-typed data, and rewrites equalities in memberships for safe transformation. In contrast to previous work, our presentation separates the Incrementalize step from the Expand Joins step because it allows us to better reason about demand filtering.

Assuming that no new parameter values need to be demanded, the asymptotic time for obtaining a query result is linear in the size of the result, since it needs to be copied into a fresh set that the user can interact with. If it can be proven that the user does not write to the returned set, and does not read from the set after any update that would cause the stored result to be modified, then the copy can be eliminated. In this case the cost is constant time. The asymptotic time for maintaining the result at an update depends on the particular join orders used for that update's maintenance joins. The space cost for added data structures depends on both the size of the stored results for all demanded parameter values, as well as which image-set expressions are used to implement all joins.

Our generated code will make use of *counted sets* and *maps*, whose operations are shown in Table 2.2. Counted sets are sets that associate each element with a positive integer count; they are similar to bags (multisets) except that they do not produce duplicate elements during iteration. They have operations for incrementing and decrementing the count associated with an element, and macro operations for adding or removing the element when the count goes between 0 and 1.[6]

The generated code will also use *tuple-decomposing* versions of assignment statements and `for` loops. For both, instead of having a single variable on their left-hand side, they have a tuple of variables `(`x_1`, ..., `x_k`)`. The expression on the right-hand side must evaluate to a tuple of arity k for assignment, or a set of tuples of arity k for a `for` loop. Upon execution, the individual tuple components are assigned to their respective variables.

The rest of this section describes the three steps in detail. We defer examples until the end of the section.

Incrementalize: The main idea of this step is to store the comprehension result in a counted set and keep an invariant:

> A value v appears in the set with count c iff the comprehension has c many valid valuations for which the result expression evaluates to v, with c greater than 0.

[6]Macros are for notational convenience only, and are always immediately expanded in the abstract syntax tree.

Expression	Return value	Precondition
`new map`	a new empty map	
k `in` m	whether $k \in keys(m)$	
m`[`k`]`	the value that m associates with k	$k \in keys(m)$
m`[`k`] default` v	m`[`k`]` if $k \in keys(m)$, otherwise v	
`new cset`	a new empty counted set	
s`.getcount(`v`)`	the count that s associates with v	$v \in s$

Update	Effect	Precondition
m`[`k`] =` v	associate key k with v in m	$k \notin keys(m)$
`del` m`[`k`]`	remove key k and its value from m	$k \in keys(m)$
s`.inccount(`v`)`	increment the count for v in s	$v \in s$
s`.deccount(`v`)`	decrement the count for v in s	$v \in s$

Macro		Expansion

s`.add`$_{\texttt{c}}$`(`v`)`	$\Rightarrow$	`if` v `not in` s`:`
		  s`.add(`v`)`
		`else:`
		  s`.inccount(`v`)`

s`.remove`$_{\texttt{c}}$`(`v`)`	$\Rightarrow$	`if` s`.getcount(`v`) == 1:`
		  s`.remove(`v`)`
		`else:`
		  s`.deccount(`v`)`

Table 2.2: Map and counted-set operations. Counted sets support all the primitive operations of ordinary sets in Table 2.1. An exception is raised if any precondition is not satisfied, including if s or m does not evaluate to a counted set or map value, respectively. k and v evaluate to arbitrary values. $keys(m)$ denotes the set of keys for the map referred to by m. All operations take constant time.

At every update to one of the comprehension's relations, a piece of maintenance code consisting of one or more maintenance joins is inserted next to the update. Each join finds all the valuations that have become valid or invalid due to the update's effect on one of the comprehension's memberships. The code iterates over all valuations found by the joins. For each one, it performs an `add`$_{\texttt{c}}$ (for addition updates) or `remove`$_{\texttt{c}}$ (for removals) on the stored result.

Maintenance joins are themselves a variant of relational comprehensions. The result expression is a tuple of all variables in the query. The right-hand side of each membership may be either a relation as before (e.g., `S`), a singleton set (`{e}`), or a set difference of a relation and a singleton set (S `- {`e`}`). Here the element in the singleton set is the same variable or expression that

represents the element being added or removed by the update. Intuitively, the join is formed by plugging in this element in place of an occurrence of the updated relation in the comprehension.

Precisely, let $R_1, \ldots, R_n$ be the n occurrences of relation variables in the query, and let the update be either S`.add(`e`)` or S`.remove(`e`)`. For each i such that $R_i = S$, a join is formed from the comprehension as follows:

- The result expression is replaced by a tuple expression of all the query's variables, in some canonical order.
- The right-hand side of the i^{th} membership is replaced by `{`e`}`.
- For all other memberships j where $R_j = S$ and $j > i$, the right-hand side is replaced by R_j `- {`e`}`.

For an addition update, the code is inserted after the update, while for a removal, it is inserted before. This ensures that e is present in S at the time the joins are evaluated.

The join generated for the i^{th} membership returns the set of all valuations that map the left-hand side of this membership to e, and that do not satisfy any subsequent membership over S using e. This exclusion ensures that the joins are disjoint. Without it, a temporary set of valuations would need to be materialized in order to detect valuations that have been double-counted, as in [74, 72]. Such a temporary set can take asymptotically more space than the query result itself. (Note that although we do not want to double count any valuations, we still want to count distinct valuations that derive the same entry in the result set.)

In the case where a single membership has a variable that occurs more than once on its left-hand side, we must rewrite the query prior to incrementalizing it: For each such variable, rename its duplicate occurrences into fresh variables, and create new conditions to equate the fresh variables to the original variable. For example,

```
(x, y, x, x) in R
```

becomes

```
(x, y, x2, x3) in R, x == x2, x == x3.
```

This step is required because an image-set expression cannot by itself enforce an equality constraint. See Section 2.6 for an alternative rewriting that leads to faster performance.

Expand Joins: Each maintenance join is implemented as imperative code that performs, in relational database terms, an index join. First a join order is chosen for the clauses. (We denote join orders as lists of clause indices, i.e., $[i_1, \ldots, i_n]$ means the $i_j{}^{\text{th}}$ clause of the comprehension appears at position j in the order.) Then, each clause is turned into imperative code, nested inside the code for the previous clauses. The innermost code evaluates the result expression and performs the $\texttt{add}_\texttt{c}$ or $\texttt{remove}_\texttt{c}$ operation.

As we proceed from the first clause in the join order to the last clause, the variables on the left-hand side of each membership become bound. The code for a membership is responsible for iterating over all values of its unbound variables that are consistent with the current values of its bound variables. The code for a condition just checks whether that condition is true; its variables must be bound by previous memberships.

Table 2.3 lists all the cases for turning clauses into code. For a membership over a generic relation R, where R is not one of the special relations M or F_f that were introduced during flattening, the code either uses R itself or an image-set expression over R. For a membership over one of the special relations, if the variable corresponding to the set or object is already bound, the generated code will operate on directly on that value. Otherwise, if the second value is bound, an image-set expression will be used. The case where neither variable is bound is not supported since that would require iterating over the unmaterialized special relation. For a membership S `-` `{`e`}` where S may or may not be special, the code is the same as for a membership over S except that an `if` statement is inserted to exclude e. Finally, a membership over `{`e`}` just uses a tuple decomposing assignment; in practice only the first case for this kind of membership is needed.

There are two requirements for join orders. It is always possible to find a join order satisfying both requirements.

(1) Each variable in a condition must be bound. This is satisfiable due to the domain requirement; in the worst case, the condition can be run after all memberships.

(2) Each membership over M or F_f must have at least one variable bound. This is satisfiable because the Restrict step ensures that the query is connected.

Subject to these two requirements, we can use any method for deciding the exact join order. We follow [74, 72] and use a simple greedy heuristic: At any step, if any clause would run in constant time given the current set of bound variables, it is a candidate for the next clause. Otherwise, pick any

Case	Code	Cost
$vars$ `in` R $(I = \emptyset)$	`for` $vars$ `in` R`:`	$O(1)$
$vars$ `in` R $(J = \emptyset)$	`if` $vars$ `in` R`:`	$O(1)$
$vars$ `in` R (otherwise)	`for` $vars_J$ `in` $R.J\{I = vars_I\}$`:`	$O(\lvert R.J\{I = vars_I\}\rvert)$
`(`$\underline{s}$`,` e`)` `in` M	`if` s `isset:` `for` e `in` s`:`	$O(\lvert s\rvert)$
`(`$\underline{s}$`,` $\underline{e}$`)` `in` M	`if` s `isset:` `if` e `in` s`:`	$O(1)$
`(`s`,` $\underline{e}$`)` `in` M	`for` s `in` M`.in{`e`}:`	$O(\lvert M$`.in{`e`}`$\rvert)$
`(`s`,` e`)` `in` M	— unrunnable —	
`(`$\underline{o}$`,` v`)` `in` F_f	`if` o `hasfield` f`:` v `=` $o.f$	$O(1)$
`(`$\underline{o}$`,` $\underline{v}$`)` `in` F_f	`if` o `hasfield` f`:` `if` v `==` $o.f$`:`	$O(1)$
`(`o`,` $\underline{v}$`)` `in` F_f	`for` o `in` F_f`.in{`v`}:`	$O(\lvert F_f$`.in{`v`}`$\rvert)$
`(`o`,` v`)` `in` F_f	— unrunnable —	
$cond$ (all vars bound)	`if` $cond$`:`	$O(1)$
$cond$ (otherwise)	— unrunnable —	
$vars$ `in {`e`}` $(I = \emptyset)$	$vars$ `=` e	$O(1)$
$vars$ `in {`e`}` $(J = \emptyset)$	`if` $vars$ `==` e`:`	$O(1)$
$vars$ `in {`e`}` (otherwise)	$vars_J$ `=` e_J`:` `if` $vars$ `==` e`:`	$O(1)$
$vars$ `in` S `- {`e`}`	same as $vars$ `in` S, then: `if` $vars$ `!=` e`:`	same as $vars$ `in` S

Table 2.3: Code and costs for each kind of clause and each kind of binding environment. S can be any relation, and R can be any relation besides one of the special relations M or F_f. Underlines indicate bound variables, $vars$ is a tuple of variables, I and J are the indices of bound and unbound variables, and $cond$ is a condition.

clause that has at least one variable bound. Finally, fall back on picking any clause that is not unrunnable. (The last fallback is never necessary for the relational comprehensions that are obtained from flattening an object-set comprehension, unless the unflattened-sets optimization from Section 2.6 is used.) The time cost of running a piece of maintenance code is the sum of the costs for running each of its joins; the cost of a join is bounded by the product of the costs for each of its clauses' code as per Table 2.3.

Implement Indices: After all relational comprehensions have been incrementalized, we need to implement image-set expressions and unwrap expressions. A straightforward but inefficient approach is to iterate over the given set, constructing the result on-the-fly. This takes linear time in the size of the given set, and can be much larger than the output of an image-set expression. For an image-set or unwrap expression appearing around a query result, the cost could be amortized over the cost of copying the result set, if such a copy is needed; but a copy is never needed for an image-set expression inside maintenance code, and any expensive operations there are compounded by appearing inside nested loops. Furthermore, there is no way to compute an image-set expression over M or F_f straightforwardly since the operands are not materialized in the final program.

The solution is to incrementalize image-set and unwrap expressions, just as we did for relational comprehensions. Each image-set expression $R.J\{I = V\}$, where R is a relation, requires an *auxiliary map* $R_{J/I}$. Two image-set expressions that differ only in V share the same auxiliary map. The invariant for each auxiliary map is that

$$R_{J/I}[V] = R.J\{I = V\}$$

for all V such that $R.J\{I = V\}$ is non-empty, and V is not a key of $R_{J/I}$ otherwise. The use of the image-set expression is replaced by

$$R_{J/I}[V] \texttt{ default new set},$$

where the `default` covers the case where the image set for V is empty. We abbreviate $R_{2/1}$ and $R_{1/2}$ as $R_{\texttt{out}}$ and $R_{\texttt{in}}$ respectively.

The maintenance rules for auxiliary maps are straightforward: For an addition update to the operand relation, create the stored image set if it doesn't already exist and then add the corresponding sub-tuple, and for a removal update do the opposite. The exact code is given in Table 2.4. For consistency with the rules for relational comprehension maintenance, auxiliary map maintenance code is inserted after addition updates and before removal updates.

Relation update		Maintenance
R.add(t)	⇒	if t_I not in $R_{J/I}$: $R_{J/I}[t_I]$ = new set $R_{J/I}[t_I]$.add(t_J)
R.remove(t)	⇒	$R_{J/I}[t_I]$.remove(t_J) if $R_{J/I}[t_I]$ isempty: del $R_{J/I}[t_I]$

Table 2.4: Rules for maintaining an auxiliary map $R_{J/I}$.

Maintenance takes constant time. The space for storing an auxiliary map is linear in the size of the relation it indexes. Since image-set expressions inside maintenance code never need to make a copy, these will run in constant time.

An unwrap of an image-set expression is handled using a slightly modified invariant,

$$R_{J/I}[V] = \texttt{unwrap}(R.J\{I = V\}).$$

For the unwrap to be applicable, J must have a single component j. The maintenance rules are as in Table 2.4 except that we replace the singleton tuple expression t_J with its component expression t_j.

Because the Implement Indices step runs after the Incrementalize step, all maintenance code for auxiliary maps runs closer to the program point of the update than all maintenance code for comprehensions. This means that the auxiliary map invariants are always satisfied at the program points where they are used — a property guaranteed by finite differencing. Implement Indices is run after *all* comprehensions have been incrementalized so that the same auxiliary maps may be reused in the maintenance code for multiple queries.

Examples: We give incremental code and cost bounds for three examples: the `CheckAccess` query, a simple graph query involving a self-join, and finally the running example query. The cost analysis of the running example motivates the use of demand filtering in the next section. Singleton tuples are eliminated from the final generated code as described in Section 2.6.

Example 8. Working from the relational comprehension in Example 2, we derive maintenance code for the update `SR.add((s, admin))`. Assume that the query's stored result is held in a variable `Q`. After the Incrementalize step, the high-level code is

```
SR.add((s, admin))
for (session, op, obj, role) in {(session, op, obj, role) :
        (role,) in ROLES, (session, role) in {(s, admin)},
        (op, obj, role) in PR}:
    Q.add_c((session, op, obj, role)).
```

The result expression of the join happens to be the same as the result expression of the relational comprehension being maintained. Note that the variables introduced by the `for` loop are separate from the local variables inside the join even though they have the same names.

The join is implemented using the join order $[2, 1, 3]$. The other two joins for updates to `ROLES` and `PR` will use an image-set expression over `SR` to obtain `session` from `role`, and the use of the query will become

$$\mathtt{Q}_{4/123}\mathtt{[(session, op, obj)]\ default\ new\ set},$$

so that the auxiliary maps $\mathtt{SR_{in}}$ and $\mathtt{Q}_{4/123}$ are needed. After expanding joins and implementing indices, the final maintenance code is obtained.

```
SR.add((s, admin))
if admin not in SR_in:
    SR_in[admin] = new set
SR_in[admin].add(s)
(session, role) = (s, admin)
if role in ROLES:
    for (op, obj) in PR_12/3[role] default new set:
        if (session, op, obj, role) not in Q:
            Q.add((session, op, obj, role))
            if (session, op, obj) not in Q_4/123:
                Q_4/123[(session, op, obj)] = new set
            Q_4/123[(session, op, obj)].add(role)
        else:
            Q.inccount((session, op, obj, role))
```

The running time for this maintenance is $O(|\mathtt{PR.12\{3 = admin\}}|)$, that is, the number of permissions that the added role has. The space usage is

$$O(|Q| + |SR| + |PR|),$$

that is, linear in the size of the result (for all parameter values) plus the sizes of the session and permission information. □

Example 9. The following relational comprehension takes in an edge relation `E`, and a relation of source nodes `S`, and returns all nodes that are two hops away from a source node.

```
{(z,) : (x,) in S, (x, y) in E, (y, z) in E}
```

The high-level maintenance code for an addition to `E` is as follows (not showing the expansion of the $\texttt{add}_\texttt{c}$ macros):

```
E.add((a, b))
for (x, y, z) in {(x, y, z) : (x,) in S, (x, y) in {(a, b)},
                                         (y, z) in E - {(a, b)}}:
    Q.add_c((z,))
for (x, y, z) in {(x, y, z) : (x,) in S, (x, y) in E,
                                         (y, z) in {(a, b)}}
    Q.add_c((z,))
```

The set difference `E - {(a, b)}` is required for the case of inserting a reflexive edge; when `a` and `b` both evaluate to the same value v, we must only count the valuation that maps all of `x`, `y`, and `z` to v once. The fact that the maintenance code runs after the update is important for this example, since if it ran before the update then this valuation would not be found at all. After expanding joins and implementing indices, using the orders $[2, 1, 3]$ and $[3, 2, 1]$ for the two joins respectively, the final code is obtained.

```
E.add((a, b))
if a not in E_out:
    E_out[a] = new set
E_out[a].add(b)
if b not in E_in:
    E_in[b] = new set
E_in[b].add(a)
(x, y) = (a, b)
if x in S:
    for z in E_out[y] default new set:
        if (y, z) != (a, b):
            Q.add_c(z)
(y, z) = (a, b)
for x in E_in[y] default new set:
    if x in S:
        Q.add_c(z)
```

The time cost for this maintenance is $O(|\texttt{E.out\{b\}}| + |\texttt{E.in\{a\}}|)$, and the space cost is $O(|\texttt{Q}| + |\texttt{E}|)$. In other words, whenever we add or remove an edge, we pay the time cost of checking all the adjacent edges, and the space costs are the size of the query result plus the size of the edge relation (for effectively building an adjacency list representation to use in place of an edge list representation). □

Example 10. We consider the running example query in Example 7. We show the incremental maintenance for the update

```
bob.followers = alice.followers,
```

which causes `bob.followers` to alias `alice.followers`, where `bob` previously did not have a `followers` field. Treating the update as if it were to $F_{\texttt{fol}}$, incrementalizing using the join order $[2, 3, 5, 6, 7, 4, 1]$, and assuming no auxiliary map maintenance is needed for $F_{\texttt{fol}}$, we get the following code, where `Q` holds the saved result. The expansion of $\texttt{Q.add}_\texttt{c}$ and the maintenance of $\texttt{Q}_{3/12}$ is not shown.

```
F_fol.add((bob, alice.followers))
(celeb, celeb_fol) = (bob, alice.followers)
if celeb_fol isset:
    for user in celeb_fol:
        if user hasfield loc:
            user_loc = user.loc
            if user_loc == "NYC":
                if user hasfield email:
                    user_email = user.email
                        for group in M_in[user] default new set:
                            if (user, group) in U:
                                Q.add_c((celeb, group, user_email))
```

The time cost is at most

$$O(|\texttt{alice.followers}| \times |M_{\texttt{in}}|).$$

In other words, it is proportional to the number of sets each of `alice`'s followers is a member of, regardless of whether or not those sets are actually demanded as values for `group`. This cost is incurred even if `bob` is never demanded. The space cost, assuming no other auxiliary maps are needed, is $O(|M| + |\texttt{Q}|)$ — the sum of the sizes of all sets in the original program, plus the sizes of all possible query results. □

As the above example demonstrates, the costs of incremental computation without demand filtering can be drastically high relative to the amount of data that is being queried. Maintenance code can take time to traverse values that are not only irrelevant, but also completely inaccessible from any demanded parameter value. Likewise, auxiliary maps take up space to index all the sets and objects in the original program — a steep constant factor cost, when one considers that the implementation of auxiliary maps is likely nowhere near

as optimized as the host language's native representation of its own sets and objects. In the next section we will see how to address these concerns so that no space and no asymptotic time is spent for indexing and traversing irrelevant values.

2.4 Filtering using demand

Due to the Restrict step, our generated code only stores and maintains query results corresponding to demanded parameter values. However, this by itself is not enough to make the whole computation demand-driven. This section introduces demand filtering, an optimization that curtails the number of entries retrieved by image-set expressions, so that auxiliary maps take up less space and the maintenance code runs in less time.

The basic idea is to define *tags* and *filters* based on the structure of the relational comprehension, and in the maintenance code, replace uses of relations in image-set expressions with uses of their corresponding filters. A tag is a set containing every value that a valid valuation can map a variable to, as well as some other relevant values for that variable, but no irrelevant values. A filter is a subset of a relation where all the tuples have a specific component whose value is in a specific tag set.

Both tags and filters are themselves represented as relational comprehensions and incrementalized using the method in Section 2.3. The maintenance for tags and filters, called *demand propagation*, incurs its own overhead. There are multiple ways to form filters for a given query, leading to different tradeoffs between demand propagation cost and the cost of incrementally computing the query. We refer to the particular choice of tags and filters as the *demand strategy*.

Although the specific cost bounds for demand-driven programs vary depending on the query and on the choice of demand strategy, a few generic bounds always hold.

(1) For an update that changes an element or field value v, the time cost for demand propagation is at most linear in the total size of the relevant values that are accessible from v.

(2) For an update to a value v that is irrelevant both before and after the update, the time cost for both query maintenance and demand propagation is constant.

(3) The space cost for storing tags, filters, and auxiliary maps over filters is at most linear in the total size of all relevant values. Since the only other space cost is for storing and indexing the query result, the overall space used by the method is at most proportional to the size of all demanded query results plus the sizes of all relevant values.

(4) As a very coarse upper bound, the time for query maintenance is at most a polynomial function of the size of all relevant values. (The same bound applies to the total cost of computing the query from scratch over all demanded parameter values.)

These last two points are notable because they mean that the overall cost of demand-driven incremental computation is in fact small whenever the amount of demanded data is small. Since inaccessible values are always irrelevant, we can reword each of these bounds to be more intuitive but less precise by changing "relevant values" to "accessible values".

Demand filtering adds a new step, *Filter*, to be run after the Incrementalize step but before the Expand Joins step. The Filter step replaces uses of relations that require image-set expressions with uses of their corresponding filter. The effect is that all image-set expressions will be over filters. The precise order in which the Incrementalize step is applied to tags and filters, both relative to each other and relative to the query, is important for ensuring correctness.

Our design for tags and filters is based on the strategy for demand-driven indices used in [74, 72]. However, unlike their approach, our indices have well-defined invariants that allow us to reason about cost bounds. Our demand strategies are also more precise in several ways regarding how selective our filters are. Most notably, we avoid a demand "leak" whereby a value continues to remain demanded until it is garbage-collected, even after it is no longer relevant. This selectiveness makes it possible to relate the behavior of maintenance code to the relevant values for each variable (Theorem 2), making certain optimizations possible that would otherwise be unsafe. Formally, demand filtering can be justified by reasoning over conjunctive queries (Theorem 1) or even magic set transformation [10, 11]. See Chapter 4 for further discussion.

The rest of this section discusses the particulars of forming tags and filters, the Filter step, how the Incrementalize step applies for the query and for the tags and filters, the cost analysis of demand propagation, and the correctness of filtered maintenance code.

Intuition: Consider a simple field selection query,

$$\texttt{\{o.f.g : o in s\}},$$

which after flattening, strengthening, and restricting, is

$$\texttt{\{(s, g) : (s,) in } U\texttt{, (s, o) in } M\texttt{, (o, f) in } F_{\texttt{f}}\texttt{, (f, g) in } F_{\texttt{g}}\texttt{\}}.$$

The maintenance code for an update to the `f` field of an object o_1 (i.e., an update to $F_{\texttt{f}}$) would iterate over all sets containing o_1, using the auxiliary map $M_{\texttt{in}}$. This requires $O(|M\texttt{.in\{}o_1\texttt{\}}|)$ time and $O(|M|)$ space. We can instead define a filter

$$dM = \texttt{\{(s, o) : (s,) in } U\texttt{, (s, o) in } M\texttt{\}},$$

and substitute uses of M in the maintenance code with uses of dM. This is correct because the only entries that are lost would have failed the test `(s,) in` U, and such entries could not satisfy the query anyway. The cost is only $O(|dM\texttt{.in\{}o_1\texttt{\}}|)$ time and $O(|dM|)$ space, where dM is bounded by the total number of elements of sets in U. In the case where U has only one element, the maintenance is constant time.

Likewise, at an update to the `g` field of an object o_2 (an update to $F_{\texttt{g}}$), the maintenance code would use $F_{\texttt{fin}}$ to find all objects containing o_2. We define a tag for `o` and a filter for $F_{\texttt{f}}$,

$$\begin{aligned} T_{\texttt{o}} &= \texttt{\{(o,) : (s, o) in } dM\texttt{\}} \\ dF_{\texttt{f}} &= \texttt{\{(o, f) : (o,) in } T_{\texttt{o}}\texttt{, (o, f) in } F_{\texttt{f}}\texttt{\}}, \end{aligned}$$

and replace the use of $F_{\texttt{f}}$ in the query maintenance code with $dF_{\texttt{f}}$. Now the query maintenance cost is at most $O(|dF_{\texttt{f}}\texttt{.in\{}o_2\texttt{\}}| \times |dM_{\texttt{in}}|)$, instead of $O(|F_{\texttt{f}}\texttt{.in\{}o_2\texttt{\}}| \times |M_{\texttt{in}}|)$. The filtered code iterates over only the objects and sets that are relevant for `o` and `s`.

$T_{\texttt{o}}$ is conservative in that it includes at least any value for `o` that is actually used to satisfy the query, as is required for correctness. At the same time, it does not include any value that is irrelevant for `o`, which is important for cost bounds.

In this example of an update to $F_{\texttt{g}}$, it happens that the query maintenance always takes constant time whenever the updated value is irrelevant. This is because an irrelevant value for `f` fails to produce any results for the lookup $dF_{\texttt{f}}\texttt{.in\{f\}}$. In general, if this query were more complicated, we would insert an `if` statement to check whether the changed tuple is in $dF_{\texttt{g}}$ before running

the rest of the maintenance join's code. This ensures that no non-constant time is taken for updates to irrelevant values.

For the update of demanding a new value s_1 for parameter `s`, the time cost of all demand propagation is $O(|s_1|)$, with the filter dM being the cause of the linear complexity due to the many-to-many nature of M. For all other updates, the cost of demand propagation is at most constant, as can be predicted from the fact that the only relevant values that are accessible from the value in the update are field values, and all F_f relations are many-to-one.

Defining tags and filters: In general, we represent a demand strategy as an edge-labeled directed multigraph. In essence, demand strategies are an instance of sideways information passing (SIP) strategies [11]. The nodes are all the memberships of the relational comprehension. Each arc label is a variable that appears in both the source and destination memberships. It is required that the graph is acyclic, with all memberships reachable from U; we use topological sortings of the graph to reason about the dependencies of the tags and filters. Intuitively, the nodes correspond to filters, while the arcs indicate what tags each filter depends on. Different choices of demand strategies will provide different definitions of filters (and therefore different overall costs), although the definitions of tags are the same regardless.

Precisely, let's number the memberships and use an unambiguous notation to refer to all the tags and filters that can be created: d^iR means the filter associated with the i^{th} membership in the query and that has R on its right-hand side, while T^ix means the tag associated with the use of the variable x in the i^{th} membership. Suppose that the i^{th} membership is $vars$ `in` R, and that the demand strategy has incoming arcs to i from the nodes $j_1, \ldots, j_n$ respectively labeled with variables $y_1, \ldots, y_n$. We call these incoming arcs the *predecessors* of i. Then the tags for i are:

$$T^ix = \texttt{\{(}x\texttt{,) : }vars\texttt{ in }d^iR\texttt{\}}.$$

for each x in $vars$. The filter for i is

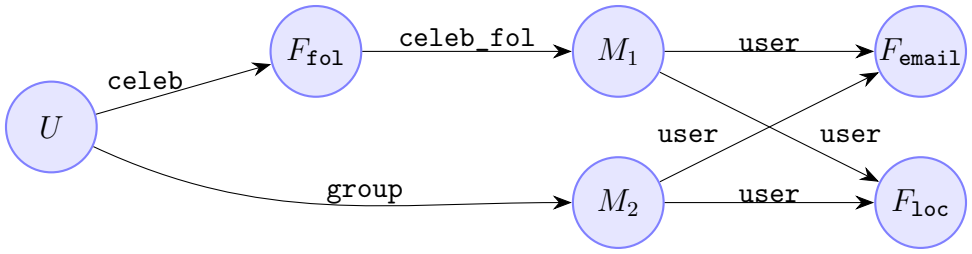
$$d^iR = \texttt{\{}vars\texttt{ : (}y_1\texttt{,) in }T^{j_1}y_1\texttt{, ..., (}y_n\texttt{,) in }T^{j_n}y_n\texttt{, }vars\texttt{ in }R\texttt{\}}.$$

Note that since U is the root, it has no predecessors and acts as its own filter. In relational algebra terms, we can think of each tag as a projection of a filter, and each filter as a join of a relation with one or more tags, where the attributes of the tags are subsumed by the attributes of the relation so that it acts as a selection.

Tags that are not used to constrain any filter can be eliminated; these tags have no corresponding arcs in the demand strategy. In addition, when there is only one demand parameter, its tag can be replaced by U itself. For the above field selection query, we would have generated tags for `s` and for `g`, but the tag over `g` is never needed and the tag over `s` gets replaced by U.

We place two restrictions on the demand strategy for memberships of the form `(`x`, `y`) in` M or `(`x`, `y`) in` F_f, where x and y are distinct variables: An arc leaving the membership cannot be labeled with x, and an arc entering the membership cannot be labeled with y. The first restriction helps to avoid creating low-information tags that are less likely to be worth their maintenance overhead. Intuitively such tags corresponds to the constraints "the set is non-empty" (for M) or "the object has field f defined" (for F_f). The second restriction avoids creating filters whose maintenance code would itself rely on unfiltered image-set expressions over M or F_f. Our default heuristic for choosing a demand strategy is to pick any maximal graph satisfying these restrictions.

Example 11. We use the following (maximal) demand strategy for the running example query:



M_1 and M_2 refer to the two occurrences of M. After eliminating unnecessary

tags, the remaining tags and filters are, in dependency order:

$$
\begin{aligned}
T_{\texttt{celeb}} &= \{\texttt{(celeb,) : (celeb, group) in } U\} \\
T_{\texttt{group}} &= \{\texttt{(group,) : (celeb, group) in } U\} \\
dF_{\texttt{fol}} &= \{\texttt{(celeb, celeb\_fol) : (celeb,) in } T_{\texttt{celeb}}\texttt{,} \\
&\qquad \texttt{(celeb, celeb\_fol) in } F_{\texttt{fol}}\} \\
T_{\texttt{celeb\_fol}} &= \{\texttt{(celeb\_fol,) : (celeb, celeb\_fol) in } dF_{\texttt{fol}}\} \\
dM_1 &= \{\texttt{(celeb\_fol, user) : (celeb\_fol,) in } T_{\texttt{celeb\_fol}}\texttt{,} \\
&\qquad \texttt{(celeb\_fol, user) in } M\} \\
T_{\texttt{user}_1} &= \{\texttt{(user,) : (celeb\_fol, user) in } dM_1\} \\
dM_2 &= \{\texttt{(group, user) : (group,) in } T_{\texttt{group}}\texttt{,} \\
&\qquad \texttt{(group, user) in } M\} \\
T_{\texttt{user}_2} &= \{\texttt{(user,) : (group, user) in } dM_2\} \\
dF_{\texttt{loc}} &= \{\texttt{(user, user\_loc) : (user,) in } T_{\texttt{user}_1}\texttt{, (user,) in } T_{\texttt{user}_2}\texttt{,} \\
&\qquad \texttt{(user, user\_loc) in } F_{\texttt{loc}}\} \\
dF_{\texttt{email}} &= \{\texttt{(user, user\_email) : (user,) in } T_{\texttt{user}_1}\texttt{, (user,) in } T_{\texttt{user}_2}\texttt{,} \\
&\qquad \texttt{(user, user\_email) in } F_{\texttt{email}}\}.
\end{aligned}
$$

The intuitive meanings of the tags and filters are as follows.

$T_{\texttt{celeb}}$	Demanded celebrities
$T_{\texttt{group}}$	Demanded groups
$dF_{\texttt{fol}}$	$F_{\texttt{fol}}$ for demanded celebrities only
$T_{\texttt{celeb\_fol}}$	Follower sets of demanded celebrities
dM_1	Membership info for demanded celebrities only
$T_{\texttt{user}_1}$	Users following a demanded celebrity
dM_2	Membership info for demanded groups only
$T_{\texttt{user}_2}$	Users belonging to a demanded group
$dF_{\texttt{loc}}$	Location info for just the users following a demanded celebrity and belonging to a demanded group
$dF_{\texttt{email}}$	Email info for just the users following a demanded celebrity and belonging to a demanded group

□

Filtering joins: Given a join in the query's maintenance code, the Filter step replaces some of the relations with their corresponding filters. (Since U is its own filter it cannot be replaced.) Only the relations that will end up appearing in the expanded code are affected. For memberships over M and F, this means that the case where the first component is bound will be unaffected, since it is translated into code that uses the underlying set or object directly.

The Filter step requires knowledge of the join order that will be used in the Expand Joins step.

In addition to rewriting these memberships, the Filter step also adds a membership over the filter corresponding to the update; the generated code for this clause ensures that the updated values are actually relevant. Suppose that the maintenance join is being generated for the membership *vars* `in` R, so that this clause is replaced by *vars* `in {`e`}` where e is the expression for the tuple in the update. Let dR be the filter corresponding to that use of R. Then the new membership is *vars* `in` dR. The join order will always run the clause over the singleton set first; we furthermore require it to run this new membership second, so that no other clauses run when the updated values are irrelevant.

Example 12. Consider using filtering on the running example for an update $F_{\texttt{loc}}$`.add((bob, "NYC"))`, with the join order $[5, 6, 7, 3, 2, 1, 4]$. Filters are used for $F_{\texttt{fol}}$, the first occurrence of M, and the update to $F_{\texttt{loc}}$, giving us the join

```
{(celeb, group, user_email) : (celeb, group) in U,
      (celeb, celeb_fol) in dF_fol, (celeb_fol, user) in dM_1,
      (group, user) in M, (user, user_loc) in {(bob, "NYC")},
      (user, user_loc) in dF_loc, user_loc == "NYC",
      (user, user_email) in F_email}.
```

The other field relations and the other occurrence of M are not filtered because their set or object is bound by the time it appears in the join order, and U can never be rewritten.

Omitting type checks for brevity, the expanded query maintenance code after the Implement Indices step is

```
(user, user_loc) = (bob, "NYC")
if (user, user_loc) in dF_loc:
    if user_loc == "NYC":
        user_email = user.email
        for celeb_fol in dM_1in[user] default new set:
            for celeb in dF_folin[celeb_fol] default new set:
                for group in U_out[celeb] default new set:
                    if user in group:
                        Q.add_c((celeb, group, user_email)).
```

The time cost bound is only $O(|dM_1\texttt{.in\{bob\}}| \times |dF_{\texttt{fol}\texttt{in}}| \times |U_{\texttt{out}}|)$, compared with $O(|M\texttt{.in\{bob\}}| \times |F_{\texttt{fol}\texttt{in}}| \times |U_{\texttt{out}}|)$ for the unfiltered version. In addition, the filtered version takes only constant time whenever `bob` is not

both a follower of a demanded celebrity and a member of a demanded group. Finally, assuming no other auxiliary maps are needed by other pieces of maintenance code, the space cost is $O(|dF_{\texttt{loc}}| + |dM_1| + |dF_{\texttt{fol}}| + |U| + |\texttt{Q}|)$, as opposed to $O(|F_{\texttt{loc}}| + |M| + |F_{\texttt{fol}}| + |U| + |\texttt{Q}|)$. Observe that when U is empty, all of the sets in the first cost expression are empty. □

Transformation order: The three transformation steps in Section 2.3 are used for both the query's relational comprehension and the relational comprehensions for the tags and filters. The order in which the steps are applied to different comprehensions is critical for correctness, and is determined by the finite differencing method, which provides a systematic way of maintaining multiple dependent invariants. Specifically, the Incrementalize step must be applied in the order given below, and the Implement Indices step must come after all applications of the Expand Joins step are done.

The Incrementalize step must be applied to the query's comprehension first, followed by each tag and filter in some order that is consistent with a topological sorting of their dependencies (i.e., starting with the tags that are projections of U, then the filters that use these tags, and so on). This ensures that all updates affecting a tag or filter will exist in the intermediate program before that tag or filter is itself incrementalized. It also ensures that the invariants for filters will be satisfied at the program points where they are needed, i.e., inside the maintenance code for the query result.

We demonstrate how this sequencing of transformation steps leads to correct generated code. To avoid unnecessary complication, we summarize the maintenance code by showing just its overall structure. Each line in the code outline says what invariant is being maintained, for what updated variable, and at what cost. We use indentation to indicate that an invariant is being maintained in response to a change made by the preceding piece of maintenance code; the indented code executes once for each change the preceding code makes. The final asymptotic cost is obtained by multiplying the costs of all indented lines with their preceding lines, and taking the sum of these costs.

We'll show the code outline at several stages in the transformation of the running example for an update to U, since this update produces the largest amount of code. Before any Incrementalize steps have run, it is just

U`.add((c, g))`.

After incrementalizing the query, naming the stored result `Q`, we get the following. Inserted code is marked with asterisks.

```
   U.add((c, g))
*  maint Q for U                                  O(|c.followers|)
```

We could have also chosen a join order with cost $O(|\texttt{g}|)$. Next we apply the Incrementalize step to $T_{\texttt{celeb}}$, $T_{\texttt{group}}$, $dF_{\texttt{fol}}$, and $T_{\texttt{celeb\_fol}}$, in this order. Constant-time cost annotations are omitted.

```
   U.add((c, g))
*  maint T_group for U
*  maint T_celeb for U
*    maint dF_fol for T_celeb
*      maint T_celeb_fol for dF_fol
   maint Q for U                                  (O(|c.followers|))
```

The code for maintaining $T_{\texttt{group}}$ appears above that for $T_{\texttt{celeb}}$ because it is inserted immediately below the update to U, and after the code for $T_{\texttt{celeb}}$ has already been added. After this we incrementalize dM_1, $T_{\texttt{user}_1}$, dM_2, and $T_{\texttt{user}_2}$, in order.

```
   U.add((c, g))
   maint T_group for U
*    maint dM_2 for T_group                       (O(|g|))
*      maint T_user_2 for dM_2
   maint T_celeb for U
     maint dF_fol for T_celeb
       maint T_celeb_fol for dF_fol
*        maint dM_1 for T_celeb_fol               (O(|c.followers|))
*          maint T_user_1 for dM_1
   maint Q for U                                  (O(|c.followers|))
```

Finally we incrementalize $dF_{\texttt{loc}}$ and $dF_{\texttt{email}}$, in order. Each has maintenance code inserted at two separate points in the code.

```
   U.add((c, g))
   maint T_group for U
     maint dM_2 for T_group                          (O(|g|))
       maint T_user2 for dM_2
*        maint dF_email for T_user2
*        maint dF_loc for T_user2
   maint T_celeb for U
     maint dF_fol for T_celeb
       maint T_celeb_fol for dF_fol
         maint dM_1 for T_celeb_fol                  (O(|c.followers|))
           maint T_user1 for dM_1
*            maint dF_email for T_user1
*            maint dF_loc for T_user1
   maint Q for U                                     (O(|c.followers|))
```

Again, maintenance code for $dF_{\texttt{email}}$ appears above that for $dF_{\texttt{loc}}$ because they are both inserted below the same updates, but with the transformation for $dF_{\texttt{email}}$ coming after the transformation for $dF_{\texttt{loc}}$.

The overall time complexity of this code is $O(|\texttt{c.followers}| + |\texttt{g}|)$. The cost of query maintenance (which is only $O(|\texttt{c.followers}|)$) is amortized over the cost of demand propagation. Notice that all tags and filters are properly maintained and up-to-date at the point where they are needed in the maintenance of `Q`. The code outline for a removal update can be found by a similar derivation, but inserting maintenance above updates instead of below them. This results in a code outline that is similar to this one, but with the lines for $dF_{\texttt{email}}$ and $dF_{\texttt{loc}}$ occurring in the opposite order, and likewise for the four non-indented lines. The cost of removal is the same as for addition.

Demand propagation cost: It is possible to obtain a time cost bound for demand propagation by analyzing the generated maintenance code directly, or looking at its code outline as above. However, it may be more intuitive to describe these costs directly in terms of the demand strategy. One advantage of this higher-level approach is that we can compare demand propagation costs of alternative demand strategies before actually generating the incremental code.

The analysis is based on annotating each node and arc of the demand strategy with an asymptotic cost in a bottom-up (i.e., reverse-topological) manner. The following two rules are used.

(1) A node is annotated with the sum of the costs of each of its outgoing arcs. A node without any outgoing arcs is annotated with $O(1)$.

(2) An arc from i to j labeled x is annotated with the cost of j, multiplied by the number of different tuples that can satisfy clause j for a worst-case fixed value of x.

These costs can be simplified using standard algebraic rules for asymptotic cost formulas, as well as two special rules that take advantage of the meaning of M and F_f:

(1) A cost term $|M.\texttt{out}\{s\}|$ is simplified to $|s|$.

(2) A cost term $|F_f.\texttt{out}\{o\}|$ corresponding to a clause `(`o`, `v`) in `F_f is simplified to $O(1)$, and uses of v in other cost terms of the formula are replaced by $o.f$.

In general, since an object can have only one value per field, all non-constant work done for demand propagation originates from the need to iterate over the elements of a set, when that set is added to or removed from a tag set.

Once the cost annotations are filled in and optionally simplified, we can use them to determine demand propagation cost bounds for updates to the relations that tags and filters depend on. These costs are instantiated by substituting the specific values used in the update for the corresponding variables. We use three rules.

(1) The cost for updating a filter d^iR is the instantiated annotation for node i.

(2) The cost for updating a tag T^iv is the sum of the instantiated annotations for all arcs leaving i and having v as their labels.

(3) The cost for updating a relation R is the sum of the instantiated annotations of node i for all d^iR, that is, all filters for occurrences of R. Note that each summand is instantiated separately (each according to the variables appearing in that clause) before being combined into one cost formula.

The first two rules help us understand how individual demand invariants contribute to the overall cost. The last rule summarizes the total demand propagation cost for a single update in the original program.

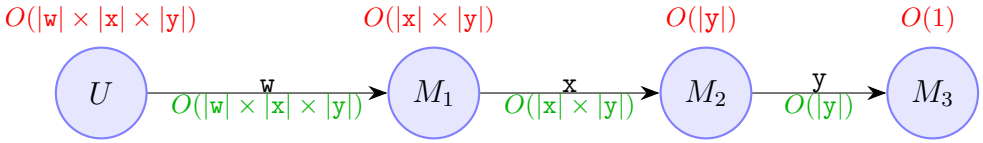
Example 13. Suppose we have a query over several levels of nested sets,

```
{z : x in w, y in x, z in y}
```

flattened, strengthened, and restricted to

```
{(w, z) : (w,) in U, (w, x) in M, (x, y) in M, (y, z) in M}.
```

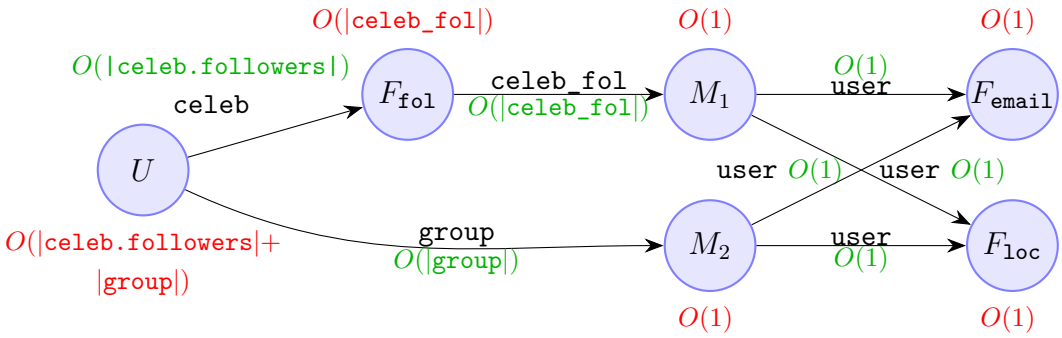
The demand strategy with annotated and simplified costs is shown below.



Based on this graph we arrive at definitions of filters dM_1, dM_2, dM_3 and tags Tw (equivalent to U), Tx, Ty, and Tz (unused).

- At an update that adds c to Ty, the instantiated cost is $O(|c|)$, reflecting the time needed to add pairs (c, d) to dM_3 for all $d \in c$. The same cost applies to an update that adds (b, c) to dM_2, the maintenance code for which entails adding c to Ty.
- At an update that adds (a, b) to dM_1, we can only instantiate the variable `x` to b, obtaining a cost of $O(|b| \times |\mathtt{y}|)$. This is the size of b times the worst-case size of a value for `y` (in actuality, an element of b).
- Finally, at an update that adds element e to set s, i.e., that adds (s, e) to M, the instantiated costs for the three M nodes are respectively $O(|e| \times |\mathtt{y}|)$ (mapping `x` to e), $O(|e|)$ (mapping `y` to e), and $O(1)$. Their sum simplifies to $O(|e| \times |\mathtt{y}|)$. □

Example 14. The annotated and simplified demand strategy from Example 11 is shown below.



The simplifications allow us to eliminate cost terms of form $|F_{\ldots\mathtt{out}}|$ for the fields `email`, `loc`, and `fol`; to replace some (but not all) occurrences of `celeb_fol`

with `celeb.followers`; and to replace cost terms of form of $|M$`.out{`s`}`$|$ where s is one of `celeb_fol`, `celeb.followers`, or `group` with these terms directly.

The analysis allows us to conclude, by reading off the graph above, that the only expensive (i.e., non-constant-time) operations as far as demand propagation is concerned are reassigning a celebrity's follower set and demanding or undemanding a celebrity or group. □

Correctness of filtering: Unlike the transformation steps of Sections 2.2 and 2.3, the Filter step is not drawn directly from previous work on object-set query incrementalization [74, 72, 85]. To show that it is correct, we interpret a set of membership clauses as a conjunctive query [16, 1] that returns all of its variables, and prove that substituting each relation with its corresponding filter does not change the set of valid valuations. A slight extension of this result demonstrates that the Filter step is sound.

Theorem 1. Let P be a set of membership clauses, and let P' be formed from P by replacing each relation with its filtered version.

$$P = vars_1 \texttt{ in } R_1\texttt{, } \ldots\texttt{, } vars_n \texttt{ in } R_n$$
$$P' = vars_1 \texttt{ in } dR_1\texttt{, } \ldots\texttt{, } vars_n \texttt{ in } dR_n$$

Then P and P' have exactly the same set of valid valuations. Furthermore, the same is true when P' is formed by using either R_i or dR_i for each i in $1, \ldots, n$.

Proof. The $P' \subseteq P$ direction follows immediately from the fact that each filter is a subset of its underlying relation, and from the monotonicity of conjunctive queries. To show the other direction, we need to construct a containment mapping from the unfolding of P' to P. Unfolding repeatedly replaces a clause over a tag or filter with the clauses that make up its definition, substituting fresh variables for the variables that appear only in the definition.

Fresh variables are only created when a tag is unfolded into a set of clauses that includes a filter dR_i. The containment mapping is defined to map these fresh variables to the corresponding variables from $vars_i$; all other variables map to themselves. After all unfolding is done and after renaming each variable according to the containment mapping, each clause has the form $vars_i$ `in` R_i, which exists in P.

Since P and P' have the same variables and since joins do not project away any of their variables, the fact that $P = P'$ means that they also have exactly the same set of valid valuations. The case where the set of membership

clauses uses either R_i or dR_i for each i is the same, except with fewer unfolding steps. □

Example 15. Consider the original and filtered versions of an abbreviated version of the join from the running example:

$$\begin{aligned} P = {} & \texttt{(celeb, group) in } U, \\ & \texttt{(celeb, celeb\_fol) in } F_{\texttt{fol}}, \\ & \texttt{(celeb\_fol, user) in } M \\ P' = {} & \texttt{(celeb, group) in } U, \\ & \texttt{(celeb, celeb\_fol) in } dF_{\texttt{fol}}, \\ & \texttt{(celeb\_fol, user) in } dM_1 \end{aligned}$$

Clearly $P' \subseteq P$. For the other direction, the first clause of P' is over a base relation and does not need unfolding. The second clause ($dF_{\texttt{fol}}$) expands to

$$\begin{aligned} & \Rightarrow \texttt{(celeb,) in } T_{\texttt{celeb}}\texttt{, (celeb, celeb\_fol) in } F_{\texttt{fol}}\texttt{,} \\ & \Rightarrow \texttt{(celeb, group}_1\texttt{) in } U\texttt{, (celeb, celeb\_fol) in } F_{\texttt{fol}} \end{aligned}$$

where $\texttt{group}_1$ is a fresh variable. The third clause (dM_1) likewise expands to

$$\begin{aligned} & \Rightarrow \texttt{(celeb\_fol,) in } T_{\texttt{celeb\_fol}}\texttt{, (celeb\_fol, user) in } M \\ & \Rightarrow \texttt{(celeb}_1\texttt{, celeb\_fol) in } dF_{\texttt{fol}}\texttt{, (celeb\_fol, user) in } M \\ & \Rightarrow \texttt{(celeb}_1\texttt{,) in } T_{\texttt{celeb}}\texttt{, (celeb}_1\texttt{, celeb\_fol) in } F_{\texttt{fol}}\texttt{,} \\ & \quad \texttt{(celeb\_fol, user) in } M \\ & \Rightarrow \texttt{(celeb}_1\texttt{, group}_2\texttt{) in } U\texttt{, (celeb}_1\texttt{, celeb\_fol) in } F_{\texttt{fol}}\texttt{,} \\ & \quad \texttt{(celeb\_fol, user) in } M \end{aligned}$$

where $\texttt{celeb}_1$ and $\texttt{group}_2$ are fresh. Putting them together, the three clauses collectively expand to

$$\begin{aligned} & \texttt{(celeb, group) in } U\texttt{,} \\ & \texttt{(celeb, group}_1\texttt{) in } U\texttt{,} \\ & \texttt{(celeb, celeb\_fol) in } F_{\texttt{fol}}\texttt{,} \\ & \texttt{(celeb}_1\texttt{, group}_2\texttt{) in } U\texttt{,} \\ & \texttt{(celeb}_1\texttt{, celeb\_fol) in } F_{\texttt{fol}}\texttt{,} \\ & \texttt{(celeb\_fol, user) in } M\texttt{.} \end{aligned}$$

Substituting each subscripted variable with its non-subscripted version, each of the six clauses appears in P. This is enough to show that $P \subseteq P'$. Moreover, since the set of variables appearing in P and P' are the same, the set of valid valuations for P and P' are identical. □

Theorem 1 says that any choice of relations may be filtered without changing the comprehension's meaning. To show that filtering maintenance joins is correct, we note that for an update to R_i that adds or removes e, e is actually in R_i at the time that the maintenance join runs. This gives us the following equality.

$$
\begin{aligned}
& vars_1 \texttt{ in } R_1\texttt{, ..., } vars_i \texttt{ in \{}e\texttt{\}, ..., } vars_n \texttt{ in } R_n \\
= \; & vars_1 \texttt{ in } R_1\texttt{, ..., } vars_i \texttt{ in } R_i\texttt{, ..., } vars_n \texttt{ in } R_n\texttt{, } vars_i \texttt{ in \{}e\texttt{\}} \\
& (e \in R_i) \\
= \; & vars_1 \texttt{ in } dR_1\texttt{, ..., } vars_i \texttt{ in } dR_i\texttt{, ..., } vars_n \texttt{ in } dR_n\texttt{, } vars_i \texttt{ in \{}e\texttt{\}} \\
& \text{(Theorem 1)}
\end{aligned}
$$

Not all of the R_i are necessarily replaced with dR_i in the actual maintenance join.

In addition to showing that filtered maintenance code is correct, we also show that every value that is retrieved for a variable x in any generated maintenance code (for either the query or for tags and filters) is a relevant value for x. This is important for the generic cost guarantee that query maintenance time is bounded by a function of the relevant values. It is also useful for the try-block elimination and type check elimination optimizations in Section 2.6.

Theorem 2. Consider a relational comprehension and one of its maintenance joins, created for an update that adds or removes e to R. Let $c_1, \ldots, c_n$ be the clauses of the join arranged according to the join order. Note that c_1 is always the membership over `{`e`}`, and c_2 is always the membership over dR that is inserted by the Filter step. Then for all $i = 3, \ldots, n$, if x is a variable that appears in some prior clause $c_j, j < i$, then each execution of the code for clause c_i will be with a bound value for x that is relevant for x.

Proof. First, note that every component of a tuple in a filter is relevant for the component's corresponding variable. This is because each component is either constrained to be in a tag set (which holds only relevant values), or else is constrained to be an element or field value of such a component.

We use induction on the clauses of the join order. For the base case of $i = 3$, clause c_2 tests that the only variables that are bound so far are in the filter dR. For the inductive case, assume that all variables bound so far satisfy the property. Case analysis on the kinds of clauses that can run in the i^{th} step shows that any newly bound variable y that can satisfy the clause must be relevant for y. If the code for the i^{th} clause is an `if` statement, it binds no new variables. If it is an iteration over U, or over an image-set expression

over U or a filter, then the retrieved values are necessarily relevant as per the above. If it is an iteration over a set's elements or a retrieval of an object's field value, then by the inductive hypothesis the set or object is already relevant, and therefore so are its elements or field values. □

2.5 Extensions

This section describes several extensions to the query language that make it more powerful and useful to the programmer. The most important are nested queries and aggregate queries. Nested queries are not specifically discussed in [72], except to say that the method can be iterated to apply to both the inner and outer queries. The situation is in fact more complicated due to demand. Not only must the inner query's demand set be managed properly to include certain values needed for the outer query, but the timing of when these values are added and removed needs to be coordinated with the behavior of the outer query's maintenance code. Aggregates over comprehensions are discussed in [72], but we generalize to handle aggregates that are nested inside comprehensions as well.

In addition to these, we also describe how to handle queries involving map lookup expressions, tuple patterns, negated memberships, general exceptions, and additional syntactic sugar for pattern matching. Only the extensions for maps and tuples are described in [72]; our treatment of them also handles ill-typed values and discusses how they affect the notions of reachable expressions and relevant values.

2.5.1 Nested queries

This extension allows subqueries to appear as the right-hand side of membership clauses of an outer comprehension. This can make queries easier to write and to understand, and affects the performance characteristics of incremental computation. When used in combination with aggregates or negation, it also increases the expressive power of the query language. (Without them, expressive power remains the same since our comprehensions act as conjunctive queries [1].)

The main work in supporting nested queries is in extending the steps of Section 2.2 that rewrite the query as a relational comprehension. Once the nested input queries have been transformed into nested relational comprehensions, the Incrementalize and Filter steps of Sections 2.3 and 2.4 apply as

normal to one query at a time, with nested queries being processed innermost-first.

The cost bound calculation described in Section 2.4 applies to a single query being maintained in response to a single update, and does not necessarily hold for nested queries with demand. The time bound for propagating demand to an inner query, based on changes affecting the outer query, may be larger. In addition, the space usage for tracking demand for the inner query is not necessarily linear in the number of demanded combinations of parameter values for the outer query.

Syntax: For object-set comprehensions, the syntax of membership clauses is extended to allow another object-set comprehension on its right-hand side as a subquery.

$$\begin{aligned} \textit{membership} &::= \textit{retrieval}\ \texttt{in}\ \textit{rhs} \\ \textit{rhs} &::= \textit{retrieval} \mid \textit{object-set-comp} \end{aligned}$$

The safety and reachability requirements apply to both the outer and inner queries, but we must generalize reachability in order to model the dependence of a subquery on its parameters. Let $p_1, \ldots, p_n$ be the inner query's parameters and let e be the retrieval expression on the left-hand side of the membership over the subquery. Then, extending the definition on page 14, e is reachable if all $p_1, \ldots, p_n$ are reachable. The new requirement is that all retrieval and subquery expressions must be reachable, except for expressions that are local to a subquery, which will be covered by that subquery's own reachability requirement.

We also need to add a rule for the relevant values of e. If values $u_1, \ldots, u_n$ are relevant for $p_1, \ldots, p_n$, and if evaluating the subquery using parameter values $u_1, \ldots, u_n$ can produce a result set containing v, then v is relevant for e.

The syntax of relational comprehensions is extended similarly.

$$\begin{aligned} \textit{membership} &::= \texttt{(}\textit{component}^*\texttt{)}\ \texttt{in}\ \textit{rhs} \\ \textit{rhs} &::= \textit{relation} \mid \textit{relational-comp} \end{aligned}$$

For obvious scoping reasons, memberships of subqueries are not counted when considering the domain requirement of outer queries.

Flatten and Strengthen: We augment these two steps with a few simple rewriting rules. Consider an object-set membership over a nested query,

$$e\ \texttt{in}\ \{r\ \texttt{:}\ \ldots\}.$$

After flattening the inner query, we begin to flatten the outer query by replacing the retrieval e with a fresh variable x, giving us

$$x \texttt{ in unwrap(\{(}r\texttt{,) : ...\})}.$$

To finish flattening this clause of the outer query, x is wrapped in a singleton tuple to cancel out the `unwrap`.

$$\texttt{(}x\texttt{,) in \{(}r\texttt{,) : ...\}}$$

Strengthening the inner query will turn this into

$$\texttt{(}x\texttt{,) in \{(}{p_1}'\texttt{, ..., }{p_n}'\texttt{, }r\texttt{) : ...\}.}(n+1)\texttt{\{}1 = p_1,\ \ldots,\ n = p_n\texttt{\}}.$$

The image-set expression is eliminated by prepending the parameters to the left-hand side of the membership.

$$\texttt{(}p_1\texttt{, ..., }p_n\texttt{, }x\texttt{) in \{(}{p_1}'\texttt{, ..., }{p_n}'\texttt{, }r\texttt{) : ...\}}$$

Restrict: This step is the one most affected by nested queries. The restriction on the outer query's demand set is just like the case without nesting: The demand set contains the queried parameter values at the time the query is performed. For an inner query, however, the demand set must contain at least all parameter values that can be used to satisfy the outer query, i.e., all parameter values that are mapped to by some valid valuation for the outer query. This is so that a valuation for the outer query is not made invalid solely on account of an inner query's demand set being too small.

We address this by defining the inner query's demand set as a new comprehension, called the *demand comprehension*; the inner query's demand clause has this comprehension on its right-hand side in place of a demand set. The clauses of the demand comprehension are a subset of the outer query's clauses, and the result expression is a tuple of the inner query's demand parameters. All of the parameters in the demand comprehension are renamed into local variables. The outer query clauses that are used in the demand comprehension are called the *preceding clauses*. It is required that the inner query's demand parameters are reachable from the outer query's demand parameters using only the preceding clauses; this is always possible due to the reachability requirement.

Restricting is done outermost-first for nested queries, and in reachability order for queries at the same level. This ensures that, at the time we are determining the preceding clauses for any subquery, the demand clause of its parent query has already been added, and any sibling query that appears in a

preceding clause has already been restricted. The Incrementalize step applies in an innermost-first order, so that first it transforms the program to insert code for the demand comprehension of the inner query, then for the inner query itself, and then finally for the outer query.

Since the restricted inner query already reflects the demand of its outer query, there is no need to define a new filter for the outer query's membership over the inner query; it acts as its own filter, just like U. The time and space complexities for computing demand comprehensions can be calculated using the same method as for any other comprehension. Likewise, the demand comprehension is itself filtered. Its tags and filters may have the same definitions as the tags and filters of the outer query, but they are distinct invariants nonetheless; since they are maintained at different points in the final program, their stored results may differ during the execution of maintenance code.

The top-down nature of the Restrict step for nested queries is analogous to how magic-set transformation [10, 11] and demand transformation [78, 79] model top-down demand information by adding new predicates to be evaluated bottom-up.

Example 16. Consider the following nested object-set query, which finds users in a given group that have visited at least one large city.

```
{user : user in group,
        loc in {city : city in user.visited,
                       city.pop > 1000000}}
```

The inner query takes the outer query's `user` as a parameter and returns all large cities that the user has visited. After flattening both queries, strengthening both queries, and restricting the outer query, we end up with the following expression (not showing the unwrap and image-set expressions around the outer query).

```
{(group, user) : (group,) in U, (group, user) in M,
                 (user, loc) in {(user', city) :
                     (user', user_visited) in F_visited,
                     (user_visited, city) in M,
                     (city, city_pop) in F_pop,
                     city_pop > 1000000}}
```

The primed variable `user'` is local to the inner query. The preceding clauses for the inner query's demand comprehension are the outer query's memberships over U and M, so the final result after restricting the inner comprehension too is

```
{(group, user) : (group,) in U, (group, user) in M,
                 (user, loc) in {(user', city) :
                     (user',) in {(user'',) :
                         (group'',) in U,
                         (group'', user'') in M},
                     (user', user_visited) in F_visited,
                     (user_visited, city) in M,
                     (city, city_pop) in F_pop,
                     city_pop > 1000000}},
```

where the inserted demand clause with a demand comprehension on its right-hand side is in bold font, and the double-primed variables are local to the demand comprehension. Note that at this point, all three comprehensions have no parameters aside from relations — all other parameters were renamed into locals, either during the Strengthen step or when forming the demand comprehension.

The space costs for the demand comprehension, inner query, and outer query are, respectively, the number of users in demanded group sets, the number of big cities those users have visited, and the number of users in demanded group sets who have visited at least one big city. After considering the space cost for tags and filters, the overall space cost is at most the number of cities visited by users in demanded group sets, assuming each user visits at least one city.

Let's look at the code outline for what happens when we query a new group `g`. Call the demand comprehension, inner query, and outer query Q_D, Q_1, and Q_2, respectively. We show the update to U, maintenance for these three queries (but not their tags or filters), and the retrieval of the outer query's result.

U`.add((g,))`	
`maint` Q_2 `for` U	$(O(\|\mathtt{g}\| \times \|Q_{1\mathtt{out}}\|))$
`maint` Q_D `for` U	$(O(\|\mathtt{g}\|))$
`maint` Q_1 `for` Q_D	$(O(\|dM_{city\mathtt{out}}\|))$
`maint` Q_2 `for` Q_1	$(O(\|dM_{user\mathtt{in}}\|))$
`retrieve result from` Q_2	

We use dM_{user} and dM_{city} to refer to the filters for the uses of M that represent users in groups and cities in user visited sets, respectively. The image-set expression Q_1`.out{`u`}` returns just the big cities for a given user u, so $|Q_{1\mathtt{out}}|$ is dominated by $|dM_{city\mathtt{out}}|$. The total time bound is $O(|\mathtt{g}| \times |dM_{city\mathtt{out}}| \times |dM_{user\mathtt{in}}|)$, but we can simplify this by noticing that the maintenance of Q_2 for Q_1 only

runs in situations where `g` is the only group for the added user, giving us $O(|\texttt{g}| \times |dM_{city\texttt{out}}|)$, which is optimal. □

Discussion: It is tricky to handle demand for nested queries correctly. The naive approach would be to process each query in isolation, innermost-first, introducing a new demand set for each query. The demand set for each query would be updated just before the point that the query result is needed. This idea can be illustrated with the following code outline for an update to the outer query's demand set, where Q_1 and Q_2 are respectively the inner and outer queries, and U_1 and U_2 their demand sets.

```
1  add to U2
2  maint Q2 for U2
3    add to U1
4      maint Q1 for U1
5        maint Q2 for Q1
6    retrieve result from Q1
7  retrieve result from Q2
```

If the parameter values for the retrieval of Q_2's result (line 7) are not already demanded, we need to update U_2 (line 1), which triggers maintenance of Q_2 (line 2). This maintenance relies on the result of Q_1 (line 6) as part of the generated code for Q_2's clauses; we show the line indented because it may be needed many times in the course of performing the maintenance associated with line 2. Before we can use the result on line 6, we may need to update U_1 (line 3), which in turn triggers maintenance of Q_1 (line 4) and Q_2 (line 5).

This naive approach suffers from two problems. The less subtle one is that at line 3, it may not be clear what to add to U_1 because the parameters to Q_1 are not necessarily bound (depending on the join order used by the maintenance code for Q_2). The more insidious problem is that updating U_1 indirectly triggers maintenance for Q_2 on line 5, even though we have not yet resolved the maintenance of Q_2 for the original update to U_2. Our maintenance rules are not reentrant, but are rather designed to handle only one update at a time. The problem is not purely hypothetical; Appendix A constructs a specific instance where this kind of code structure does not correctly maintain the invariants. In general, this situation should not arise when using finite differencing because there are no cyclic invariant dependencies. It is only a problem here because the naive approach incorrectly updated a relation that a query depends on, while that query's maintenance code was still running in response to a different update. The existence of this problem underscores the importance of deriving code using a systematic method.

2.5.2 Aggregate queries

An aggregate query has the syntax

$$op\texttt{(}\textit{retrieval} \mid \textit{object-set-comp}\texttt{)}$$

where *op* is a supported aggregate operation. The operations presented here are `count`, `sum`, `min`, and `max`, but more can be added so long as there is a rule for incrementally maintaining the aggregate value as elements of the operand set are added and removed. All aggregate operators are defined over all inputs: `min` and `max` return a sentinel value, `None`, when the input set is empty; elements that are not appropriate for the aggregate operator (non-number values for `sum`, or incomparable values for `min` and `max`) are ignored; and retrieval operands that do not evaluate to a set are treated as if they evaluated to the empty set. An aggregate of a retrieval expression $op\texttt{(}e\texttt{)}$ is preprocessed into an aggregate of an object-set comprehension,

$$op\texttt{(\{}x\texttt{ : }x\texttt{ in }e\texttt{\}),}$$

where x is a fresh variable, so that all aggregates are over comprehensions. We need to extend Section 2.2's rewriting into relational queries for aggregates, and then extend the Incrementalize step for them.

In general, to compute an aggregate query incrementally we need some auxiliary information, which we call the aggregate's *state*. The exact nature of the state depends on the operator. For each operator *op* we define the following.

- $retrieve_{op}\texttt{(}s\texttt{)}$ is an expression that takes in a state s for *op* corresponding to some operand set T, and returns the value of $op\texttt{(}T\texttt{)}$.
- $\emptyset_{op}$ is an expression that returns a new state corresponding to the result of *op* applied to an empty set.
- $add_{op}(s, v)$ is an expression that takes in a state s corresponding to some operand set T, and returns the new state corresponding to $T \cup \texttt{\{}v\texttt{\}}$.
- $remove_{op}(s, v)$ is as above but returns the state corresponding to $T - \texttt{\{}v\texttt{\}}$.

The definitions of these operations for the four supported aggregates are given in Table 2.5. For `count` and `sum` the state is just the result itself, and all operations take constant time. For `min` and `max` we use a double-ended priority queue, where `new pqueue` constructs a new empty queue, s`.push(`v`)` and s`.pop(`v`)` add and remove v in the queue, and s`.min()` and s`.max()` return

op	$\emptyset_{op}$	add_{op}(s, v)	$remove_{op}$(s, v)	$retrieve_{op}$(s)
count(T)	0	s + 1	s - 1	s
sum(T)	0	s + v	s - v	s
min(T)	new pqueue	s.push(v)	s.pop(v)	s.min()
max(T)	new pqueue	s.push(v)	s.pop(v)	s.max()

Table 2.5: Rules for producing and updating a state corresponding to the computation of an aggregate. In the case where an invalid type of value is given for v — i.e., a non-number value for `sum`, or an incomparable value for `min` or `max` — the add_{op} and $remove_{op}$ operations just return s unmodified.

the smallest and largest elements (or `None` if the queue is empty). We assume that updating the queue takes logarithmic time and retrieving the min and max take constant time. Such a data structure can be implemented using a balanced binary search tree that keeps a pointer to its leftmost and rightmost elements.

We need to introduce a comprehension-like expression for representing maps in the intermediate code. Let $x_1, \ldots, x_n$ be variables and let R be an expression that evaluates to a set of tuples of arity n. Also let k and v be expressions that may have the x_i as free variables. Then the expression

$$\texttt{\{}k \mapsto v \texttt{ | (}x_1\texttt{, ..., }x_n\texttt{) in }D\texttt{\}}$$

denotes a map that associates the value of k with the value of v for each combination of values of $x_1, \ldots, x_n$ that is in D. The expression's value is undefined if any two tuples in D cause k to take on the same value.

Flattening the aggregate's operand will produce an expression of form

$$op\texttt{(unwrap(}E_1\texttt{))}$$

where E_1 is the flattened comprehension of the operand. Flattening does not affect the aggregate expression itself. Strengthening the operand then gives us

$$op\texttt{(unwrap(}E_2\texttt{.(}n+1\texttt{)\{}1 = p_1,\ \ldots,\ n = p_n\texttt{\}))}$$

where E_2 is the strengthened version of E_1, and where $p_1, \ldots, p_n$ are the comprehension's parameters (and hence, parameters of the aggregate expression too). To strengthen the aggregate expression, it is rewritten as a map lookup over a map expression,

$$\begin{aligned}&\texttt{\{(}p'_1\texttt{, ..., }p'_n\texttt{) } \mapsto op\texttt{(unwrap(}E_2\texttt{.(}n+1\texttt{)\{}1 = p'_1, \ldots, n = p'_n\texttt{\}))}\\ &\texttt{ | (}p'_1\texttt{, ..., }p'_n\texttt{) in } \mathcal{T}^n\texttt{\}[(}p_1\texttt{, ..., }p_n\texttt{)]},\end{aligned}$$

where each p'_i is a fresh local variable, and $\mathcal{T}^n$ represents an infinite domain of all tuples of arity n.

To perform the Restrict step, let C be either a demand set, or a demand comprehension as described in the steps for handling nested queries above, depending on whether or not this aggregate occurs at the top level or as a subquery of a comprehension. The restricted aggregate expresion is obtained by simply replacing $\mathcal{T}^n$ with C. This is followed by restricting the operand comprehension E_2 with the demand clause `(`p''_1`, ..., `p''_n`) in` C, where p''_i is the renamed version of p_i inside E_2, obtaining a relational comprehension E_3. This yields the following and completes the rewriting.

$$\texttt{\{(}p'_1\texttt{, ..., }p'_n\texttt{)} \mapsto op\texttt{(unwrap(}E_3\texttt{.}(n+1)\texttt{\{}1 = p'_1, \ldots, n = p'_n\texttt{\}))}$$
$$\texttt{ | (}p'_1\texttt{, ..., }p'_n\texttt{) in }C\texttt{\}[(}p_1\texttt{, ..., }p_n\texttt{)]}$$

Next, we need to incrementalize. The inner comprehension E_3 is incrementalized and replaced by a fresh relation Q holding its stored result. If C is a demand comprehension then it is incrementalized and replaced with a relation, otherwise it is a demand set and it is left alone; in either case call the resulting relation D. To incrementalize the aggregate expression itself, a variable A is introduced to hold a map from parameter values to the corresponding aggregate state, with the invariant that

$$retrieve_{op}\texttt{(}A\texttt{[(}p_1\texttt{, ..., }p_n\texttt{)])} = op\texttt{(unwrap(}Q\texttt{.}(n+1)\texttt{\{}1 = p_1, \ldots, n = p_n\texttt{\}))}$$

for all $(p_1, \ldots, p_n) \in D$. The aggregate result is then retrieved using

$$retrieve_{op}\texttt{(}A\texttt{[(}p_1\texttt{, ..., }p_n\texttt{)]).}$$

If D is a demand set and not a demand comprehension, then code is inserted to add a tuple of the current parameter values to D just before this retrieval.

The rules to maintain this invariant are as follows. (The singleton elimination optimization on page 74 can be used to simplify `(v,)` to `v` in both rules.)

after Q`.add(`t`)` or before Q`.remove(`t`)`:
```
    k = t_{1..n}
    (v,) = t_{n+1}
    if k in D:
        state = A[k]
        state = add_op(state, v)        for addition updates, or
                remove_op(state, v)     for removal updates
        del A[k]
        A[k] = state
```

after D`.add(`t`)`:
```
    (p_1, ..., p_n) = t_{1..n}
    state = ∅_op
    for (v,) in Q.(n + 1){1 = p_1, ..., n = p_n}:
        state = add_op(state, v)
    A[(p_1, ..., p_n)] = state
```

before D`.remove(`t`)`:
```
    del A[(p_1, ..., p_n)]
```

At a change to the operand Q, the maintenance time is proportional to the time taken by add_{op} or $remove_{op}$. When a new combination of parameter values is demanded, the time bound is the product of the time for add_{op} and the size of the newly demanded result of the operand query. When parameters are undemanded, the time is constant. These times do not include the time to propagate demand to the operand comprehension and the maintenance for additions to and removals from Q. For space usage, the size of A is the sum of the sizes of all states for each operand query result. As long as the size of a state does not exceed the size of the set value that produces it — as is the case for each of the four aggregate operators we have implemented — the cost of storing A is amortized over the cost of storing Q.

To support using aggregates inside a comprehension, we extend the syntax for object-set comprehensions to treat aggregates as retrievals.

$$\begin{aligned} \textit{retrieval} &::= \textit{variable} \mid \textit{retrieval}.\textit{field} \mid \textit{aggr} \\ \textit{aggr} &::= \textit{op}\ \texttt{(}\ \textit{object-set-comp}\ \texttt{)} \mid \textit{op}\ \texttt{(}\ \textit{retrieval}\ \texttt{)} \end{aligned}$$

An aggregate expression is reachable if all of its parameters (i.e., the parameters of its operand comprehension) are reachable. A value v is relevant for the

result of the aggregate if there exists a combination of relevant values for each of the parameters that, when the aggregate is evaluated for these parameter values, produces the result v.

The flattening step for comprehensions replaces an aggregate with a fresh variable, just like it does for a field retrieval. For each aggregate query A, $AGGR_A$ is a relation that holds tuples of form $(p_1, \ldots, p_n, r)$ exactly when

$$retrieve_{op}(A[(p_1, \ldots, p_n)]) = r.$$

The aggregate A with parameters $x_1, \ldots, x_n$ is replaced by a fresh variable y, and a new membership, $(x_1, \ldots, x_n, y)$ `in` $AGGR_A$, is added.

$AGGR_A$ is materialized and maintained in a straightforward manner using the following rules, applied immediately after A is incrementalized.

after A`[`k`] =` v:
    `(`p_1`, ..., `p_n`) =` k
    `r =` $retrieve_{op}$`(`v`)`
    A`.add((`p_1`, ..., `p_n`, r))`

before `del` A`[`k`]`:
    `(`p_1`, ..., `p_n`) =` k
    `r =` $retrieve_{op}$`(`A`[`k`])`
    A`.remove((`p_1`, ..., `p_n`, r))`

For the purpose of finding a join order and calculating a cost bound, a clause over $AGGR_A$ takes constant time when the parameters are all bound. Aside from that, an $AGGR_A$ clause is treated just like a clause over a comprehension subquery. In particular, $AGGR_A$ serves as its own filter.

2.5.3 Maps

So far we have only used maps as auxiliary indices in the generated code. We can add support for them in the input language by treating them in a similar manner as object fields. Recall that map operations are listed in Table 2.2. We add to these operations two expressions: m `ismap`, which returns whether m is a map value or not, analogous to `isset`; and m`.items()`, which returns a set of all key-value pairs in m.

Map lookups are added as another kind of retrieval expression for object-set comprehensions.

$$\begin{aligned} retrieval ::= {} & variable \mid retrieval\texttt{.}field \mid aggr \\ \mid {} & retrieval \texttt{ [} retrieval \texttt{]} \end{aligned}$$

For a map lookup m`[`k`]`, if m is reachable then both k and the whole lookup expression are reachable. A value is relevant for k or m`[`k`]` if it is a key or the value associated with a key, respectively, in a map that is relevant for m. For a given valuation, an exception is raised if m is not a map or if the value of k is not a key in m. The assumption from Section 2.1, that only certain exceptions occur, is relaxed to permit this case.

In the Flatten step, the map key essentially acts like a field name except that it may appear as an unbound variable. A new special relation MAP is defined with the semantics

$$(m, k, v) \in MAP \iff m\texttt{[}k\texttt{]} = v.$$

For each distinct retrieval m`[`k`]` in the query, a fresh variable v is introduced, and all occurrences of the retrieval are replaced with v. A new membership `(`m`, `k`, `v`)` `in` MAP is inserted to define v. Updates to maps in the input program are interpreted as updates to MAP, which is not materialized in the final program. For any clause `(`m`, `k`, `v`)` `in` MAP, we require in the demand strategy that incoming edges are only labeled m and outgoing edges are only labeled k or v. The Strengthen, Restrict, and Incrementalize steps are unaffected by the extension for maps.

Table 2.6 gives the code and costs used for the Expand Joins step for MAP memberships. As in the cases of clauses over M and F_f, uses of image-set expressions will be filtered so that we do not actually take space proportional to MAP to materialize an auxiliary map over it. We require that the join order does not run a clause over MAP when all variables on the left-hand side are unbound; it is always possible to find such an order due to the reachability requirement.

2.5.4 Tuples

Tuples are already allowed in conditions and in the result expression. They are now also allowed on the left-hand side of memberships; we consider just these occurrences of tuples to be retrieval expressions. Tuples are still not allowed on the right-hand side of memberships, even though other retrievals are permitted there.

$$\begin{aligned} \textit{membership} &::= \textit{lhs}\ \texttt{in}\ \textit{rhs} \\ \textit{lhs} &::= \textit{retrieval} \mid \texttt{(}\ \textit{lhs}*\ \texttt{)} \end{aligned}$$

Since the tuple components can be variables or other retrieval expressions that are constrained by other clauses, we can think of these as tuple patterns. For

Case	Code	Cost
(<u>m</u>, <u>k</u>, <u>v</u>) in MAP	if m ismap: if v == m[k]:	$O(1)$

(<u>m</u>, <u>k</u>, v) in MAP	if m ismap:
	  if k in m:
	  v = m[k]
		$O(1)$

(<u>m</u>, k, v) in MAP	if m ismap: for (k, v) in m.items():	$O(\|m\|)$
(m, k, v) in MAP	— unrunnable —	
$vars$ in MAP, otherwise	for $vars_J$ in $MAP.J\{I = vars_I\}$:	$O(\|MAP.J\{I = vars_I\}\|)$

Table 2.6: Code and costs for clauses over MAP, analogous to Table 2.3. In the last case, I and J are the indices of the bound and unbound variables, respectively.

a tuple expresion (t_1, ..., t_n), each t_i is reachable if the whole expression is reachable, and a value is relevant for t_i if it is the i^{th} component of a value that is relevant for the whole expression. Our generated code will use a type check operation, t `hasarity` k, that returns whether t is a tuple and has arity k, analogous to `hasfield`.

The Flatten step rewrites tuples that appear on the left-hand sides of membership clauses, but leaves other occurrences of tuples alone, although their non-tuple subexpressions will still be rewritten. A new family of special relations TUP_k is defined for each $k \geq 0$ with the semantics

$$(\tau, t_1, \ldots, t_k) \in TUP_k \iff \tau = (t_1, \ldots, t_k).$$

For each distinct tuple expression (t_1, ..., t_k) appearing on the left-hand side of a membership (including nested tuple expressions), a fresh variable v is introduced, and all occurrences of that tuple expression on the left-hand sides of memberships are replaced by v. A new membership (v, t_1, ..., t_k) in TUP_k is inserted for v. In the demand strategy, all incoming edges to this new membership must be labeled v, and all outgoing edges must be labeled by some t_i. The Strengthen, Restrict, and Incrementalize steps are unchanged.

For the Expand Joins step, code and costs for TUP_k memberships are shown in Table 2.7. Unlike M, F_f, and MAP, there are no updates to TUP_k because tuples are immutable. In addition, whereas the other special relations are defined over sets, objects, and maps that have been created in the original program, TUP_k is an infinite domain of all tuples of arity k. This means that not materializing TUP_k isn't just a design choice for performance, but rather

Case	Code	Cost

$1 \in I$	`if` t `hasarity` k`:`
	$vars_J$ `=` t_J
	`if` $vars_J$ `==` t`:`
		$O(1)$

$I = \emptyset$	— unrunnable —	
otherwise	`for` $vars_J$ `in` TUP_k`.`J{$I = vars_I$`}:`	$O(\lvert TUP_k.J\{I = vars_I\}\rvert)$

Table 2.7: Code and costs for a clause $vars$ `in` TUP_k. t is a shorthand for $vars_1$, i.e., the component of TUP_k representing the tuple value itself. I and J are the indicies of the bound and unbound variables, respectively.

a necessity.[7] Our rules for generating maintenance code for TUP_k uses image-set expressions over TUP_k, but in the final code these will always be filtered and therefore finite. The case where no tuple components are bound at the time the TUP_k clause is run is not allowed; it is always possible to find a join order where at least one variable is bound due to our reachability requirement.

2.5.5 Negated memberships

At first glance, it might seem that a negated membership, e.g., `x not in s`, can be expressed as a condition clause. But this violates the safety requirement for condition clauses, just like `x in s` — at least in the general case where values that are relevant for `s` can be updated. To support negation in the general case, we extend the syntax of memberships. For object-set comprehensions, the change is as follows.

$$\textit{membership} ::= \textit{lhs}\ [\texttt{not}]\ \texttt{in}\ \textit{rhs}$$

The left-hand side of a negated membership is not reachable from its right-hand side, so it must be supported in some other way to satisfy the reachability requirement. Likewise, a value is not considered relevant for the expression on the left-hand side just because it is not in a set value that is relevant for the right-hand side. The semantics of a query with negated memberships are that a valuation satisfies e_2 `not in` e_1 if e_1 evalutes to a set and e_2 evaluates to a value that is not an element of e_1.

[7] The main reason we do not define TUP_k to include only tuples that exist in the original program is that tuples can be created inside queries, where we cannot insert any corresponding imperative maintenance code.

To extend relational comprehensions to handle negation, we again extend the syntax of memberships.

$$\textit{membership} ::= \texttt{(}\textit{component}^*\texttt{) [not] in } \textit{rhs}$$

The domain requirement is modified so that every variable appearing in the query must occur on the left-hand side of a non-negated membership.

The Flatten, Strengthen, and Restrict steps proceed as normal, leaving the negation alone. For example, a clause e_2 `not in` e_1 becomes `(`e_1`,` e_2`)` `not in` M, and a clause x `not in unwrap(`E`)`, where E represents a flattened subquery, becomes `(`x`,)` `not in` E.

To extend the Incrementalize step, for each relation R, we define a new relation NEG_R as its complement. Clearly NEG_R cannot be materialized. Additions and removals to R are treated as removals and additions to NEG_R, respectively. Right before a comprehension is incrementalized, each clause *vars* `not in` R is rewritten as *vars* `in` NEG_R, so that it fits the form already handled by the method. For each membership over NEG_R, a maintenance join will be generated and inserted before each addition to R and after each removal from R — the opposite of where maintenance is usually inserted relative to the update.

Note that the same update to R can cause maintenance code to be generated for both non-negated and negated clauses, but the two types of maintenance will be inserted on opposite sides of the update, and one will add result entries while the other will remove result entries. If there is a self-join on NEG_R, then just as for any other relation, the clauses to the right of the updated clause that are memberships over NEG_R will be rewritten as NEG_R `- {`e`}`. R and NEG_R are treated as entirely separate relations for the purpose of handling self-joins; this is because the same element cannot simultaneously satisfy a membership over R and another membership over NEG_R.

In the Expand Joins step, it is required that all variables on the left-hand side of a negated membership are bound in the join order before the clause can run. The code for the membership *vars* `in` NEG_R, where R is not one of the special relations M, F_f, MAP, or TUP_k, is simply `if` *vars* `not in` R`:`. In the case of `(`x`,` y`)` `in` NEG_M, the code is

```
if x isset:
    if y not in x:
```

The code for *vars* `in` NEG_S `-` `{`e`}` (where S can be M) is produced using the normal rule for a membership over the set difference of a relation and a singleton set: It's the same code as *vars* `in` NEG_S, followed by an inequality test. For example, if S is not M, this would be

```
if vars not in S:
    if vars != e:.
```

The other cases of memberships over NEG_R where R is F_f, MAP, or TUP_k do not arise.

The demand strategy does not introduce any filters or tags for negated memberships. For the maintenance join created for a negated membership `(`x_1`, ..., `x_n`)` `in` NEG_R and an update that changes whether e is in NEG_R (in actuality, whether e is in R), we cannot insert a clause `(`x_1`, ..., `x_n`)` `in` $dNEG_R$ because this filter does not exist. Instead, we add clauses that constrain each x_i to be in some corresponding tag Tx_i for that variable.

`(`x_1`, ..., `x_n`)` `in` `{`e`}`, x_1 `in` Tx_1, ..., x_n `in` Tx_n

These clauses are all run before any other clauses, to ensure that all values are relevant before proceeding with other parts of the maintenance join. The generated code has several new membership tests.

```
(x_1, ..., x_n) = e
if x_1 in Tx_1:
    ...
        if x_n in Tx_n:
```

If a variable x_i has several tags, any one of them may be used as the constraining Tx_i. Each x_i is guaranteed to have at least one tag, even though the clause over NEG_R does not introduce any tags for its variables, because of the reachability requirement.

Example 17. Consider the following algorithm that computes the nodes in a graph with edge relation `E` that are reachable from a source set `S` of nodes, using a relational comprehension.[8]

```
while not {(y,) : (x,) in S, (x, y) in E, (y,) not in S} isempty:
    v = arb {(y,) : (x,) in S, (x, y) in E, (y,) in NEG_S}
    S.add(v)
```

[8] Adapted from p.61 of [53], rephrasing the query in terms of comprehensions and negated memberships instead of quantifiers and set difference.

The query represents a worklist of nodes that are adjacent to, but not contained in, S. Assuming the query's result variable is Q, after applying the Incrementalize, Expand Joins, and Implement Indices steps, we get the following code. Initialization of S and E (and the corresponding maintenance) is not shown.

```
while not Q isempty:
    v = arb Q
    y = v
    for x in E_in[y] default new set:
        if x in S:
            Q.remove_c(y)
    S.add(v)
    x = v
    for y in E_out[x] default new set:
        if y not in S:
            Q.add_c(y)
```

This code has been optimized to eliminate singleton tuples, including rewriting v to hold a non-tuple value.

Further optimization is possible by observing that the first maintenance join simply removes v if it is present, regardless of how many incoming edges it has. This is an instance of the optimization for faster removals on page 78. Simplifying this join and eliminating the temporary variables x and y, we get the following.

```
while not Q isempty:
    v = arb Q
    if v in Q:
        Q.remove(v)
    S.add(v)
    for y in E_out[v] default new set:
        if y not in S:
            Q.add_c(y)
```

The statement `if v in Q:` always passes and can be specialized, i.e., replaced by its body. If we have the information, either by analysis or annotation, that everywhere in the program there are no removals from E at a time when S is non-empty, then there is no need to keep counts for the nodes in Q and the $\texttt{add}_c$ can be turned into a normal addition. This is very close to the optimal code that a programmer might manually write. The extra space used is at least proportional to E, for storing $\texttt{E}_{\texttt{out}}$, but not larger than the original graph, since Q is bounded by S. □

2.5.6 General exceptions

So far we have restricted ourselves to comprehensions where the only exceptions that are possible come from evaluating a membership, field retrieval, or map lookup. This makes it easier to generate succinct maintenance code, but it is not necessary in principle. The semantics of a comprehension are to simply ignore any exceptions that occur, treating them as a failure to satisfy all clauses. The maintenance code can be designed to treat exceptions in the same way.

Suppose we have a query that returns the multiplicative inverse of the elements of the given set that satisfy a condition `f`,

```
{1 / x : x in s, f(x)}.
```

Here, `1 / x` may cause a division-by-zero exception, while `f(x)` may cause an arbitrary exception. (However, the safety requirement still guarantees that the behavior of `f(x)` is determined entirely by `x`.) If we generate naive maintenance code that does not take these exceptions into consideration, we can end up generating incorrect code that produces an error during maintenance, while the original program, under the rather forgiving semantics of object-set comprehensions, had no error.

To address this, when generating the code for each nested clause in the Expand Joins step, the code for a condition clause is guarded by a `try` block to catch any exceptions that may be raised while evaluating the condition. Exceptions are treated in the same way as if the condition evaluated to false; the outer code may rerun the condition with different variable bindings. The evaluation of the result expression in the innermost code is also guarded. For the above example, the query maintenance code for demanding a new value v for `s` would be as follows.

```
s = v
if s isset:
    for x in s:
        try:
            if f(x):
                try:
                    Q.add_c((s, 1 / x))
                except:
                    pass
        except:
            pass
```

Here our syntax for `try` is borrowed from Python: All exceptions are caught and cause a no-op statement, `pass`, to run, immediately passing control back to the enclosing block.

The only code that needs to be guarded is code for conditions and the result expression. Membership clauses can only cause exceptions in a few cases, all of which are already accounted for by the checks in their generated code. In the next section we will look at sufficient conditions for eliminating both the `try` blocks and some of these other checks.

2.5.7 Equalities, wildcards, and general expressions in retrievals

Our queries already make use of pattern matching, in the sense that the left-hand sides of memberships act as constraints on what elements of the right-hand side may satisfy the clause. (Contrast this to Python's set comprehensions, where `for` clauses are imperative statements that assign to the terms on the left-hand side regardless of their existing values.) This pattern matching can be thought of as syntactic sugar for a query that replaces the expression on the left-hand side with a fresh variable, and that has a condition to equate this variable to the expression. For example,

```
{x : (x, y) in s}
```

is equivalent to

```
{x : z in s, z == (x, y)},
```

except that the second query does not satisfy the reachability requirement for `x` and `y`.

In order to allow our queries to be even more flexible, we add three kinds of syntactic sugar related to pattern matching: equality conditions as in the above example, wildcards in place of variables, and general expressions in place of retrievals. Each feature is rewritten so that it can be handled by the existing method. In Section 2.6 we also give related optimizations for generating asymptotically faster code.

Equality rewriting recognizes conditions of the form e_1 `==` e_2 where e_1 and e_2 are both retrieval expressions. (In this context, tuple expressions are treated as retrieval expressions, and are flattened just as those that are on the left-hand side of a membership.) The expression e_2 is reachable from e_1 and vice versa. Likewise, any value that is relevant for e_2 is also relevant for e_1 and vice versa. After the Flatten, Strengthen, and Restrict steps, the condition

has form x_1 `==` x_2 where x_1 and x_2 are variables. At this point the condition is eliminated and all occurrences of x_2 are replaced by x_1 (or, equivalently, vice versa). The overall effect is that the two equated retrieval expressions are modeled by the same variable in the relational comprehension, justifying why one is reachable when the other is. This also makes maintenance join code more efficient, because the value of one of the equated expressions immediately gives us the value of the other, whereas without our special treatment, the maintenance code might produce values for each side independently and then reject values that are unequal.

Equality rewriting does not apply when one or both sides of the equality is not a retrieval expression, or when the equality is not a condition clause but rather just a subexpression appearing inside a condition clause. For example, we cannot process a condition that is a disjunction of equality tests in this manner, since the query can be satisfied without the equality holding. (However, a condition that is a conjunction of equality tests can be handled by preprocessing them into multiple separate equality conditions.) Note that if two variables on the left-hand side of the same membership are turned into one variable by this rewriting, they will be split again by the Incrementalize step's preprocessing (see page 27), so that image-set expressions can be used safely.

Wildcards stand for local variables that are used only once in an object-set or relational comprehension. They are denoted with an underscore in place of a variable. For instance, the above query could be written as

```
{x : (x, _) in s},
```

to make it more obvious that the second component of the tuple is unused. Similarly, `_ in s` means `s` must be non-empty, `o.f == _` means `o` has a `f` field defined but we do not care about its value, and `m[_]` can be any value in the range of map `m`. Wildcards are supported by simply replacing each occurrence with a fresh variable and then processing the query as normal. Note that wildcards must still be reachable like any other variable, so the uses `x in _`, `_.f`, and `_[k]` are not allowed.

Finally, the extension for general expressions allows us to write an arbitrary expression in place of a retrieval expression, so long as the safety and reachability requirements are still satisfied. For example, we can use the following query to find all boxes having one of the desired perimeters.

```
{box : box in boxes, 2*box.width + 2*box.height in perimeters}
```

This is rewritten by introducing a fresh variable to replace the expression and equating the variable to the expression in a new condition.

```
{box : box in boxes, x in perimeters, x == 2*box.width + 2*box.height}
```

The above equality rewriting extension does not apply to this new condition because the expression on the right-hand side is not a retrieval, or else we would not have generated the new variable and condition in the first place.

2.6 Additional Optimizations

In this section we explore optimizations for reducing code size, running time, and space usage. The first three optimizations pertain to the overhead associated with modeling the original query as a relational comprehension. Specifically, we omit rewriting some of the query's set parameters during the Flatten, Strengthen, and Restrict steps; we omit the demand clause if possible; and we eliminate unnecessary uses of singleton tuples. These three optimizations allow our method to act as a sort of hybrid between object query optimization methods [74, 72] and purely set-based methods [52]. Next, we discuss four relational optimizations: pushing selections and projections through joins; replacing uses of filters to make them dead code; and two ways of performing faster maintenance for removal updates, based on whether all elements need to be cleared and based on the structure of the query.

After this we discuss five optimizations for constant-factor overhead removal that eliminate unnecessary counting operations, result sets, `try` blocks, type checks, and maintenance joins. The optimizations to eliminate `try` blocks and type checks are justified by Theorem 2, and are therefore enabled by the method's use of invariants and demand-filtering; in a non-invariant-based system they would have to be performed with more costly and difficult lower-level reasoning. Finally, we discuss inlining maintenance code and its effects on cost.

2.6.1 Omitting flattening of sets that are relations

The Flatten step ensures that correct code is generated regardless of how the program uses aliasing to update or reference the queried data. For sets, it does this by modeling the contents of the set with the special M relation. However, when it is known that a set in the input program already behaves as a relation, there is no need to model it with M, and it can instead be used directly in the relational comprehension. This simplifies the generated code for handling updates to the set, leading to reduced code size and lowering running time by a constant factor.

Intuitively, a set variable in the input program can be treated as a relation variable if it is never reassigned, and if it is (1) global, (2) unaliased, and (3)

holds a set of tuples of the same arity. Furthermore, any membership over the relation must have on its left-hand side a tuple expression of that arity. We can relax these three conditions as follows.

(1) If the set variable is not global, we need to pair the optimization with a rewriting that makes the set variable global without altering the semantics of any of its uses.

(2) If the set variable is aliased, it set value must not be updated through any of the other aliases. In a query, the only expression that the set value may be relevant for is the set variable itself, and no other value may be relevant for it.

(3) If the set variable does not hold tuples of the same arity, we need to pair the optimization with a rewriting that makes it so all elements of the set are wrapped in singleton tuples. This makes the relation have arity 1.

Since the set is global, all of its appearances in queries are as a parameter, so it is always reachable.

Once it has been determined that a set variable R acts as a relation variable — or once the program has been preprocessed to make this so — the transformation can proceed as normal, except that the membership over R is not flattened into a membership over M, and R is not treated like a parameter in the Strengthen and Restrict steps. The retrievals on the left-hand side of a membership over R are still flattened, except for the top-level tuple expression. In the Incrementalize step, updates that add to or remove from R will generate maintenance joins for memberships over R, not for memberships over M; conversely, updates to other sets in the program will not generate maintenance joins for memberships over R.

If a relation holds nested tuples, and if all memberships over the relation for all queries in the program agree on the nesting structure of the tuples on their left-hand sides, then the program can be preprocessed so that the relation holds flat tuples. For instance, in the original RBAC query [52], all memberships over `PR` have the structure

```
((operation, object), role) in PR,
```

so we can rewrite the pair-in-a-pair as a triple. This allows us to avoid the overhead of modeling the first component of `PR` using TUP_2. In general, TUP_k is needed to handle the uncertainty of whether a given value will have the expected tuple type and arity, if it cannot be determined statically.

Filters and tags are defined for memberships over R. Unlike memberships over the special relations M, F_f, MAP, and TUP_k, there is no restriction on the demand strategy regarding arcs entering and leaving the node for the membership over R. The membership over R is also allowed to act as a root in the demand strategy, i.e., with no incoming arcs and with it acting as its own filter, just like U.

Example 18. The following query is adopted from JQL [85], and is discussed in more detail in Chapter 4.

```
{(a, s) : a in ATTENDS, s in STUDENTS,
          a.course == COMP101, a.student == s}
```

If the set parameters `ATTENDS` and `STUDENTS` are global and never reassigned or aliased, they can be treated as relation variables provided that they are rewritten to hold singleton tuples. The only change to the query is that the uses of `a` and `s` on the left-hand sides of memberships are wrapped in tuples. After rewriting to obtain a relational comprehension (and eliminating variable equalities), we have the comprehension,

```
{(COMP101, (a, s)) : (COMP101,) in U,
                     (a,) in ATTENDS, (s,) in STUDENTS,
                     (a, COMP101) in F_course,
                     (a, s) in F_student}.23{1 = COMP101}.
```

This query does not depend on M, and so there is no need to generate maintenance code for updates to arbitrary sets that are not `ATTENDS` or `STUDENTS`. $F_{\texttt{course}}$ is filtered to hold only tuples for objects in `ATTENDS`. Since `COMP101` is reachable from the relations, we could also have chosen to omit using it as a demand parameter; see below for what happens when there are no demand parameters. □

Recall in Section 2.4 that when there is only one demand parameter, the tag for that parameter can be replaced by U itself. Generalizing, when a relation has arity 1, the tag introduced by that membership is equivalent to, and can be replaced by, the filter for that membership. In the case where the demand strategy has no incoming arcs for the membership, the filter is the relation itself. This means that in the above example, the `ATTENDS` relation acts as its own filter and also as a tag for `a`.

The optimization also applies to aggregate queries having the form $op(R)$ where the operand R is global and unaliased, i.e., it is almost a relation but does not necessarily hold tuples that are of the same arity. In this case we can

transform all uses of R so that its elements are wrapped in singleton tuples as above. This causes the query to be rewritten as

$$op(\texttt{unwrap}(R)),$$

which fits the form given on page 57 of an aggregate whose operand has been flattened. Effectively, we have replaced the Flatten step with the rewriting to make R hold singleton tuples.

After this, the Strengthen, Restrict, and Incrementalize steps apply as normal. Note that since R is not a query and does not have parameters, when the Strengthen step rewrites the aggregate as a map lookup over a map expression, it will use the unit tuple as the key. That is, it will yield the expression,

$$\begin{array}{l}\{() \mapsto op(\texttt{unwrap}(R.1\{\}))\\ \quad \texttt{| () in } \mathcal{T}^0\}[()].\end{array}$$

Recall that $\mathcal{T}^n$ represents the domain of all tuples having arity n; then $\mathcal{T}^0$ is a constant singleton set that contains the unit tuple. The Restrict step replaces $\mathcal{T}^0$ with a demand set or a demand comprehension, depending on whether the aggregate appears at the top level or nested inside a comprehension. In either case, it will have arity 0; see below for what this means.

2.6.2 Omitting the demand clause

When some parameters have been treated as relation variables, and the remaining parameters are all reachable from these relations, we have the option of not using any parameter as a demand parameter. In this case, the demand set (or demand comprehension, for a nested query) has arity 0 and is either empty or holds the unit tuple `()`. These correspond respectively to no result being demanded, and all results being demanded; essentially, demand is represented by a boolean. For a demand set, the unit tuple is added (if not already present) whenever the query is about to be performed, and for a demand comprehension, the unit tuple is present whenever the demand comprehension's preceding clauses are satisfied. Note that even when the demand set is empty, values may still be relevant by being accessible from the parameters that are treated as relations, but they cannot actually be used to satisfy all query clauses.

When there are no demand parameters, we have the option of omitting the demand clause entirely, i.e., omitting the Restrict step.[9] This is semantically the same as the query result always being demanded. If it is the user's intent to keep the demand set non-empty at all times, then omitting the demand clause is an optimization that improves running time and code size by a constant factor. Otherwise, if the user wants control over when the query result is stored and incrementally maintained, and when it is not, then the Restrict step should be applied as normal.

For an aggregate query whose argument behaves as a relation, as in the above optimization, if we omit the Restrict step then the set $\mathcal{T}^0$ will still be in the expression when we apply the Incrementalize step. This requires a slight modification of the maintenance rules for aggregates given on page 59. In particular, since $\mathcal{T}^0$ is non-empty at all times, there needs to be maintenance code that runs at the beginning of the program to initialize A`[()]` to $\emptyset_{op}$, the state corresponding to the aggregate operator *op*'s evaluation over an empty set. The maintenance for changes to the demand set or demand comprehension never runs because $\mathcal{T}^0$ does not have updates. The maintenance for changes to the operand will use a membership test over $\mathcal{T}^0$; any membership test v `in` $\mathcal{T}^0$ is replaced with v `== ()`.

2.6.3 Eliminating singleton tuples

This optimization removes unnecessary singleton tuples throughout the generated code. Such singleton tuples arise from relations with arity 1, and from image-set expressions where there is only one bound component or only one unbound component. Replacing singletons tuples with their contents saves a constant factor of running time and space. The optimization is applied to the generated code during the Expand Joins and Implement Indices steps.

For each relation R that has arity 1 (including relations holding comprehension results), each use of R is rewritten. In the statement R`.add(`e`)` or R`.remove(`e`)`, e is replaced by e_1, the expression that retrieves the single component in the singleton tuple e. The same is done for `if` e `in` R`:`. A tuple-decomposing loop `for (`x`,) in` R`:` is turned into an ordinary loop `for` x `in` R`:`. The expression `arb` R becomes wrapped in a tuple, `(arb` R`,)`. The tuple-decomposing assignment statement `(`x`,) =` e, which is generated

[9] Provided that there is at least one other membership in the comprehension. The only queries where this would not be true are pathological ones having no memberships and no retrieval expressions, e.g., `{100 : }`, which has no clauses and whose flattened form is `{(100,) : () in` U`}`. Such a query does not require incremental computation.

for running a membership over a singleton set containing a singleton tuple `(`x`,) in {`e`}`, is rewritten as just x `=` e_1.

In the maintenance rules for auxiliary maps given in Table 2.4, if I has only one component i, then the use of the singleton tuple expression t_I is replaced by its component expression t_i. In the rules given in Table 2.3 for a membership that uses an image-set expression, where J has only one component j, instead of generating

$$\texttt{for } vars_J \texttt{ in } R.J\{I = vars_I\}\texttt{:},$$

we generate

$$\texttt{for } vars_j \texttt{ in unwrap(}R.J\{I = vars_I\}\texttt{):},$$

which uses a normal loop instead of a tuple-decomposing loop. The rule for incrementalizing an unwrap expression of an image-set expression has already been given on page 31.

2.6.4 Pushing selection and projection through joins

A common optimization in relational databases is to eagerly perform select and project operations before joins, so that there are fewer tuples to join with. The equivalent optimization in our method is to rewrite a use of a relation so that the relation is replaced by a subquery. The subquery contains the original membership clause and may include condition clauses (selection) and return a subset of the original membership's variables (projection). This may improve running time asymptotically, and can only increase space usage by a constant factor. The optimization is applied just before we are about to run the Incrementalize step, after the preprocessing that splits multiple uses of the same variable in the same membership. This optimization generalizes a previous rewriting for equalities ("equal cards") and wildcards for Datalog [49, 47].

Precisely, the rewriting proceeds in two steps. First, each membership clause $vars$ `in` R that can benefit from rewriting is turned into a membership over a new comprehension subexpression,

$$vars' \texttt{ in \{}vars' \texttt{ : }vars \texttt{ in } R\texttt{, }conds\ldots\texttt{}\},$$

where *conds* includes each condition in the outer comprehension that uses only variables in $vars$, and where $vars'$ is a tuple of just the variables in $vars$ that are used in the outer comprehension, excluding the rewritten membership itself and the conditions in *conds*. Next, each condition in the outer comprehension that is used in at least one new comprehension subexpression's *conds* list is

deleted from the original comprehension. The newly obtained comprehension subexpressions are incrementalized first, then the outer comprehension.

A membership clause is considered to benefit from this rewriting if there is at least one condition applicable in *conds*, or if $vars'$ omits at least one variable from *vars*. Furthermore, R must not be a special relation (M, F_f, MAP, TUP_k, or NEG_R), because the space cost for maintaining the new comprehension subexpression could be on the order of the size of the special relation, which would otherwise not need to be materialized. This restriction ensures that this optimization does not add asymptotically to the space usage of the program.

Example 19. To illustrate how this rewriting works with the syntactic sugar for patterns from the previous section, let `E` be a variable holding a set of tuples representing edges in a weighted, labeled graph. Suppose that `E` can be treated as a relation, as per the optimization for omitting flattening, and that its tuples have components for source, destination, label, and weight, respectively. The following query finds nodes that can be reached from the given node `x` using two hops of `"red"` labeled edges, regardless of weight.

```
{y : (x, z, "red", _) in E, (z, y, "red", _) in E}
```

After rewriting wildcards and constant expressions, then flattening (treating `E` as a relation), strengthening, and omitting the Restrict step, we obtain

```
{(x, y) : (x, z, v1, w1) in E, v1 == "red",
          (z, y, v2, w2) in E, v2 == "red"}.
```

Without further optimization, the maintenance code at an addition of any edge would iterate over all adjacent edges of any color and weight. With optimization, the relational comprehension becomes

```
{(x, y) : (x, z) in {(x', z') : (x', z', v1, w1) in E, v1 == "red"},
          (z, y) in {(z', y') : (z', y', v2, w2) in E, v2 == "red"}},
```

where we have primed the uses of `x`, `y`, and `z` inside the nested comprehensions to make it explicit that they are local variables. Now if an edge is added, if the edge is not red then there is no change to either inner comprehension and no maintenance work for the outer comprehension. The same is true if the edge is red but there is already an existing red edge between the same end points; this would only result in incrementing the count associated with that edge in the inner query's result set. In the case where the inner comprehension result

is actually modified due to adding the first new red edge between two nodes, the maintenance for the outer comprehension will only iterate once for each adjacent vertex that is connected by at least one red edge. The space cost for storing the result of the inner comprehensions is amortized over the size of `E`, and can also be much smaller than `E`. □

2.6.5 Eliminating filters

We already eliminate tags that are not used in the definition of any filter. We can also eliminate filters that are not used in the definition of any tag, provided that there is no image-set expression that depends on the filter in any piece of query maintenance code. This saves space for the filter, and it may save asymptotically on demand propagation time.

Suppose that in the running example query, given a value for `user`, all join orders obtain values for `group` by using dM_1`in`, dF`folin`, and U`out`, rather than by using dM_2`in`. Also suppose that we use a demand strategy with no T`user`$_2$ tag, so that dM_2 does not introduce any live tags. Then no uses of dM_2 are live in the program except the membership test `if (group, user) in` dM_2 at set updates. This can be eliminated by replacing such tests with `if (group,) in` T`group` (since we know from the update that the `(group, user)` tuple is already in M). Eliminating all maintenance for dM_2 as dead code, the time cost of updates to U goes from $O(|$`celeb.followers`$| + |$`group`$|)$ down to $O(|$`celeb.followers`$|)$. This cost may be amortized over the query maintenance cost for U depending on its join order.

2.6.6 Clearing relations

For a relation R we can consider an additional kind of primitive update, R`.clear()`. This update removes all tuples from R at once, in constant time. It is useful to treat this update as a primitive rather than as a sequence of iterated element-by-element removals, because we can generate optimized maintenance code relying on the fact that the final state of R will be empty. This optimization can apply when a set parameter is treated as a relation as per the optimization for omitting flattening, or when the demand set U is cleared. Updates that clear ordinary sets modeled by M cannot be handled in this way.

In particular, for any relational comprehension that has a non-negated membership over R, the maintenance code for when R is cleared is to also clear the comprehension result; no maintenance joins are inserted. Likewise, any auxiliary map for a cleared relation is also cleared, using an analogous

constant-time operation for maps. For any comprehension that has negated memberships over R but no non-negated memberships over R, the optimization is not applicable for that query; it must instead be treated as an element-by-element removal from R (i.e., addition to NEG_R) that causes the corresponding maintenance joins to run.

For an aggregate query with operator *op*, operand relation R, and demand set or demand comprehension relation C, we have the following optimized rules. When R is cleared, the optimized maintenance is to reset each entry in the result map to $\emptyset_{op}$, in time proportional to $|C|$. When C is cleared, the maintenance is to simply clear the map in constant time.

2.6.7 Faster removals

If we have a relational comprehension where the variables of the result expression are a superset of the variables $vars$ in one of its membership clauses $vars$ `in` R, then we can generate faster maintenance code for removals from R. This faster code takes the place of the maintenance join for that clause over R.

The basic idea is that for any particular tuple r in the result set, all the valid valuations supporting this entry assign the same tuple value v to $vars$. When v is removed from R, we can delete r without having to repeatedly decrement its associated count down to zero first. To find all such result entries r that depend on the removed tuple v, an image-set expression over the query result is used to go from the values of the bound variables in $vars$ to the unbound remaining components of the result expression.[10] In the case where the variables in the result expression are exactly $vars$, the iteration over an image-set expression can be replaced by a membership test of whether $vars$ is in the query result.

This optimization applies in particular to any removal of a tuple from U, since the relational comprehension's result expression contain all demand parameters. It also applies in Example 17 (page 65) for the removal from NEG_S; no image-set expression is needed in that case because the variables in the result expression and the affected membership are the same.

[10] In languages like Python, a copy of this image set needs to be created since it will be updated while we are in the process of iterating over it. In the method without this extension, such a copy is never needed for image-set expressions inside maintenance code.

2.6.8 Counting and result set elimination

Ordinarily, the invariant for the Incrementalize step is that the result of a relational comprehension is stored in a counted set, where the count for an entry corresponds to the number of distinct valid valuations that produce it. Counting elimination, when possible, modifies this invariant so that the result is just a normal set: No `getcount`, `inccount`, or `deccount` operations are used, and the initialization of the result relation to `new cset` is replaced by `new set`. This is done during the Incrementalize step, based on the relational comprehension being processed.

Result set elimination is a special case of dead code elimination that applies to a query result relation. When counting elimination is used successfully on a relation, and when retrievals of the result occur through an auxiliary map rather than through the relation itself (as is the case for flattened object-set comprehensions), then the relation is dead code and can be removed. Result set elimination does not apply if the set is still counted, because the control flow of the maintenance code for the auxiliary map depends on the stored counts. Both result set elimination and counting elimination save a constant factor of running time, space, and code size. Both optimizations have previously been described in [12].

There are two cases where counting elimination can be done. First, if there are no counted removal updates to the result, then there are no tests of the counts. This makes the `else` in the expansion of $\texttt{add}_\texttt{c}$ dead code, so that each operation $s.\texttt{add}_\texttt{c}(v)$ simplifies to

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

We know that this case applies, without having to examine the generated code, if the flattened program has no removal updates to a relation appearing on the right-hand side of a membership, or if the only such removal updates can be optimized to use non-counted removals as per the previous optimization. Note that this condition is generally not satisfied if there are any reassignments to object fields (which entail a removal update to F_f).

The second case is when it can be statically determined that the counts never exceed 1, i.e., that no two valid valuations collide on the same result entry. This makes the `if` tests in $\texttt{add}_\texttt{c}$ and $\texttt{remove}_\texttt{c}$ always succeed so that they simplify to `add` and `remove`, respectively. For the relational comprehension, the following two conditions together are sufficient to show that this case holds.

(1) The values of variables in the relational comprehension's result expression determine the values of all other variables in the query. (In relational database terms, the variables in the result expression form a key.)

(2) The result expression is an injective mapping from inputs to outputs, i.e., no two distinct combinations of values for its variables produce the same output value.

Observe that under these rules, filters never need counting. By reasoning backward through the Strengthen and Flatten steps, we can restate these as conditions on the original object-set comprehension.

(1) The values of the maximal retrieval subexpressions appearing in the result expression, together with the values of the parameters of the query, determine the values of all other variables in the query.

(2) The result expression is an injective mapping from the values of its maximal retrieval subexpressions to its output value.

An automated reasoning for the second case can work by recognizing classes of injective expressions, such as variables and tuples, and functional dependencies inherent in certain relations, such as F_f, MAP, TUP_k, and $AGGR_A$. A programmer may also provide an annotation to express that a field acts as a key for its object. For our running example query, if it is known that `user.email` is a key for `user` (i.e., no two users may share the same email address), then its result does not need to be counted.

2.6.9 Try block elimination

In the previous section, we showed that handling comprehensions where arbitrary exceptions may occur requires inserting `try` blocks around the code for conditions and the result expression. In the general case, these blocks are needed to stop the maintenance code from failing with an error when it encounters values that do not satisfy the query due to an exception. We can eliminate these `try` blocks when it is known that the only values that can cause an exception to occur are values that the maintenance code will not explore. In particular, if the condition or result expression is safe to evaluate for all relevant values of its variables, then under Theorem 2, it will only run for these values and cannot raise an exception.

Reusing the example from the General Exceptions part of Section 2.5, if it is known that at the point where a particular update occurs, each element x of a demanded value for `s` is non-zero and does not cause `f(`x`)` to raise an

exception, then both `try` blocks can be removed. In general, this optimization saves a constant factor of running time and code size. An analogous result holds for eliminating type checks, below.

2.6.10 Type check elimination

The generated code for memberships over the special relations M, F_f, MAP, and TUP_k (as well as NEG_R where R is M) may use operations that directly access the underlying value, as opposed to using an image-set expression. In this case, the operation is guarded by a type check (respectively, `isset`, `hasfield`, `ismap`, and `hasarity`) to make sure that an exception does not occur. Type check elimination is an optimization that deletes these checks when they are guaranteed to succeed. This saves a constant factor of running time and code size. Type check elimination was described in [12], but the presentation here is more precise and ties in with our description of relevant values.

Although it is possible to perform a type analysis step over the generated code, any such analysis would have to reconstruct information that is already available in the original program and in the high-level invariants. For this reason we find it more intuitive to use an approach based on Theorem 2, similar to try-block elimination: Type checks are unnecessary when all relevant values for a variable x have a type consistent with how x is used in the query. The problem is to devise sufficient conditions for determining that this is the case.

The main idea is as follows. We define a simple type system, and the meaning of a typing for the query's variables. Then we introduce the notion of a *safe typing*, designed to ensure two properties: (1) if the demanded parameter values are well-typed, then all relevant values are well-typed; and (2) if the values of bound variables in maintenance code are well-typed, then the type checks in the code will pass. The point is to show that at the time some maintenance code runs, the demanded parameter values are well-typed with respect to some safe typing. By the above properties and Theorem 2, this implies that the type checks in the maintenance code for both the query and for the tags and filters will succeed.

The type system consists of the following type constructors:

- `set<`T`>`, for sets where all elements have type T
- `obj<`f_1`=`T_1`, ..., `f_n`=`T_n`>`, for objects defining at least fields $f_1, \ldots, f_n$ having types $T_1, \ldots, T_n$, respectively

- `map<`K`, `V`>`, for maps whose keys all have type K and whose values all have type V
- `tuple<`T_1`, ..., `T_n`>`, for tuples of arity n whose components have types $T_1, \ldots, T_n$, respectively
- `top` and `bottom`, the type of all values and no values, respectively

Types may also be given names so that recursive types can be defined, e.g., for objects that can directly or indirectly point to values of their own type. We say that type T_1 is a subtype of T_2, written $T_1 \leq T_2$, if the extent of values for T_1 is a subset of the extent of values for T_2. Thus, the above type constructors are covariant in their parameter types, and object types with more fields are subtypes of object types with fewer fields. `top` and `bottom` make the type lattice complete.

A typing τ is a function from each variable in the query to a type. Typings can naturally be extended to other expressions in the query, including field retrievals, map lookups, tuples, and subqueries. In each case, the type of the expression is the smallest type that is consistent with the types of its subexpressions. If a value v is relevant for an expression e, then we say that v is well-typed for e (with respect to τ) if v has the type $\tau(e)$.

The conditions on a safe typing τ are as follows:

- For each membership e_2 `in` e_1, $\tau(e_1) \leq$ `set<`T`>`, where $T \leq \tau(e_2)$. (Retrieving a set element produces a value that can be contained by e_2.)
- For each field retrieval $e.f$, $\tau(e) \leq$ `obj<`f`=top>`. (e is an object defining a field f.)
- For each map lookup e_1`[`e_2`]`, $\tau(e_1) \leq$ `map<`$\tau(e_2)$`, top>`. (e_1 is a map whose keys can be contained by e_2.)
- For each equality e_1 `==` e_2, $\tau(e_1) = \tau(e_2)$. (e_1 and e_2 can hold each other's values.)

The second property for safe typings, that type checks in maintenance code will pass if the values of variables are well-typed, follows from how each set membership, field retrieval, and map lookup constrain the associated expression to have the needed type. The first property, that the well-typedness of parameters implies the well-typedness of all relevant values, can be shown by induction on the definition of relevant values.

Theorem 3. Let τ be a safe typing, and assume that all demanded parameter values are well-typed for their corresponding parameter. Then for each value v and expression e, if v is relevant for e, then it is also well-typed for e with respect to τ.

Proof. For brevity we only show the base case and two of the inductive cases, which suffice when the extensions for maps, tuples, subqueries, and equalities are not used.

- *Base case:* v is a demanded parameter value for e.

  By assumption.

- *Inductive case for sets:* v is the element of some set u that is relevant for e_2, and there exists a membership e `in` e_2.

  By the definition of safe typings, $\tau(e_2) =$ `set<`T`>` where $T \leq \tau(e)$. By the inductive hypothesis, u is well-typed for e_2. Therefore, u has type `set<`T`>`, and v has type T. Since T is at least as small as $\tau(e)$, v also has type $\tau(e)$.

- *Inductive case for objects:* v is the value of field f of some object u that is relevant for e_2, and e is the field retrieval $e_2.f$.

  The type of $\tau($`e2.f`$)$ is the smallest T such that $\tau(e_2) \leq$ `obj<`f`=`T`>`. T exists because by the definition of safe typings $\tau(e_2) \leq$ `obj<`f`=top>`. By the inductive hypothesis, u is well-typed for e_2. Therefore, u has type `obj<`f`=`T`>`, and v has type T, which is $\tau(e)$.

Adding the cases for the other ways in which values may be relevant completes the proof. □

Example 20. For the running example, we can define the types

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

and use the safe typing,

`celeb` $\rightarrow$ `User` `group` $\rightarrow$ `Group` `user` $\rightarrow$ `User`. □

To determine that the demand parameter values are well-typed, we can rely on type analysis of the original program or user-supplied type annotations. There are two notable situations where an object will not be well-typed:

when fields are being assigned or deleted during the object's construction and destruction respectively, and when a field is in the middle of being reassigned. For the first case, the programmer should avoid making an object accessible from demanded parameter values before its fields have all been initialized, and symmetrically, should avoid deleting fields until the object is finally made inaccessible and is about to be destroyed. This will make it easier for a flow-sensitive type analysis to determine that at the points where the object is actually used, it has the required fields. For the second case, the only maintenance code that can run after a field deletion or before a field assignment is code for a negated membership over F_f (that is, a membership over NEG_{F_f}), but we do not generate such memberships. Consequently, no maintenance code runs while a field is in the middle of being reassigned.

2.6.11 Maintenance case elimination

Normally, the Incrementalize step inserts maintenance code at all updates to relations appearing in the relational comprehension. If the original query has any memberships, and if it has any retrievals of a field f, this means inserting code at all of the original program's updates to sets and to field f. Maintenance case elimination avoids adding a maintenance join where the values in the update are always irrelevant for the corresponding variables in the join. This saves a constant factor of running time and code size. It can also potentially save a constant factor of space if any auxiliary maps would have been used for those joins and nowhere else. Maintenance case elimination was described in [12].

For the running example query, usually any update to a set causes maintenance code to run two joins: one for `celeb.followers` and one for `group`. But if it is known that at the program point of the update, the affected set can only be a followers set and not a group, we can eliminate the second maintenance join.

In general, the problem of knowing that the value of an expression is irrelevant reduces down to the problem of knowing precise alias information. However, there are some interesting special cases that allow us to optimize without performing a separate analysis. First, if it is known that the demanded parameter values are well-typed with respect to some safe typing, and if it is known that the types of the expression in the update and its corresponding expression in the query are incompatible, then the values in the update are irrelevant. Second, if it is known that an object is not accessible from the demanded parameter values during its construction and destruction (as was advised as good practice in the above discussion), then all maintenance

code that runs during construction and destruction can be removed. This tells us that the method is able to work in languages that initialize and destroy all fields of an object at once.

Going further, suppose that instead of transforming a whole program, the method operates on a single module, which other external modules may interact with through function calls. Any update or query operations that occur in external modules are not subject to transformation, so no maintenance code will run for them. This can cause inconsistencies if used irresponsibly. However, if the external modules only update values that are irrelevant, then this optimization tells us that the maintenance code was not needed anyway, so this arrangement is sound. (Indeed, this is how our implementation interacts with `DistAlgo` to generate efficient programs for distributed algorithms.)

2.6.12 Inline maintenance code

Maintenance code can either be generated inline, in which case it is inserted adjacent to each update, or non-inline, in which case a separate procedure definition is created for the maintenance code and a call is added next to the update. As is the case with procedure inlining in general, there is a trade-off between running time and code size. On the one hand, a small constant factor of running time is saved by avoiding the call overhead. On the other hand, the code size is increased, possibly by an asymptotic factor, and there may even be a running time slowdown due to more instruction cache misses. Inlining maintenance code may also allow specialization of maintenance for different occurrences of updates, say, to take advantage of type check elimination or maintenance case elimination at some program points where it does not necessarily apply to all updates of that kind. It is always possible to generate fully inlined maintenance code since all invariants are non-cyclic and all maintenance is non-recursive.

Let's look at the asymptotic code size for if all maintenance code is inlined. There is one maintenance procedure for each kind of update to each invariant, including invariants for tags, filters, and maps. For a single query, the number of nested calls is bounded by the number of tags and filters, which is at most linear in the size of the query. When incrementalizing nested relational queries, even without considering demand comprehensions and the Restrict step, the code size can be quadratic. This is because there can be nested queries to a depth k, and a single update may directly affect each of the k queries, each of which then propagates updates and incurs maintenance for its parent query. When the nested queries include aggregates, the maintenance code for each aggregate has two separate updates to the stored result, to delete the old

result and assign the new one. This can cascade into exponential code size. In practice, we observe k to be a small constant, and we have not seen examples where inlining causes code size to increase to unmanageable levels.

Chapter 3

Implementing and evaluating IncOQ

Having described the method in the last chapter, we now discuss our implementation and its application to the running example query. Our experiments confirm the asymptotic improvement predicted by cost analysis. Further experiments to compare against previous work and to show other applications are given in the following chapters.

3.1 Implementation

We have created a prototype implementation, IncOQ[1] ("Incremental Object Queries"), that takes a program written in a subset of Python and generates an output program where the set comprehension and aggregate queries are incrementalized. The system is written in Python 3.4 and the core of the compiler comprises approximately 9k lines of code (LOC), excluding whitespace and comments, and not counting an extensive suite of tests. This subsection describes various aspects of the architecture of IncOQ.

In general, small and medium sized examples can be transformed in a few seconds, while even the largest examples we have tried took under a minute. We expect that a production system could be made at least an order of magnitude faster by implementing it in a lower-level language and taking more care to optimize the number of passes over the AST. While these optimizations are not important for our prototype implementation, they may be needed in order to incorporate our method into a developer's workflow without negatively impacting their build times.

[1] `https://github.com/IncOQ/incoq`

We have implemented the extensions (Section 2.5) for nested queries, aggregates, maps, and nested tuples. We have also implemented the optimizations (Section 2.6) for not flattening sets that already act as relations, omitting the demand clause when there are no demand parameters, eliminating some uses of singleton tuples, clearing relations, counting elimination, result set elimination, type-check elimination (based on user annotations for when the query is type-safe), and inlining. The entire system may also be run in a relational mode where flattening is never performed, and where objects, nested sets, maps, and nested tuples are not permitted in queries; this mode is useful for incrementalizing simple queries where our system cannot infer by itself that the more complicated transformations are not needed. Likewise, the system may be run with or without using demand filtering.

Language: Our input language is a subset of Python, where each update to a value that can affect a query must fit one of the following syntactic forms. For sets we use the method calls e_1`.add(`e_2`)` or e_1`.remove(`e_2`)` where the call appears at statement level. Several convenience methods are also available for performing set union, difference, intersection, symmetric difference, and clearing. For objects and maps we allow attribute assignment/deletion and key assignment/deletion statements, but we do not permit the myriad of other ways that Python has for updating these values, such as directly accessing an object's `__dict__` attribute. Each operation besides the convenience methods requires that the appropriate precondition is satisfied — e.g., e_2 must not be in e_1 before attempting to add it. It is straightforward to add a rewriting step to eliminate this precondition, at the expense of additional code size and running time.

Arbitrary expressions may be used inside update statements. However, the update methods themselves may not be invoked by an alias. For instance, although the curried invocation `f = `e_1`.add; f(`e_2`)` is allowed in general Python programs, we do not permit it. Conversely, the syntax used by our updates may not be used for other operations having incompatible semantics. E.g., in general the names `add` and `remove` cannot be used for other methods in user-defined types, and the list element assignment statement `x[i] = v` cannot be used since it conflicts with map key assignment. These limitations can be worked around by using other aliases for these operations. Finally, values may not be updated inside external library functions.

Our query expressions are represented as Python set comprehensions, where memberships and conditions are written as `for` and `if` clauses, respectively. Due to Python's syntactic restrictions, the leftmost clause must be a member-

ship. Note that Python does not perform pattern matching, so in order for the untransformed program to have the correct semantics under the Python interpreter, the query should not use pattern matching. This can be achieved by taking any occurrence of an expression on the left-hand side of a membership where the expression is not an unbound variable or a tuple, and replacing it with a fresh variable that is equated to the expression. For instance, the query

```
{z : (x, y) in E, (y, z) in E},
```

which takes an edge relation `E` and a starting node `x` and finds all nodes two hops away, would be written in Python as

```
{z : for (x2, y) in E for (y2, z) in E
     if x == x2 if y == y2}.
```

When there are no unbound terms on the left-hand side of a membership, it can also be written as an `if` statement; it is still interpreted as a membership clause by our system. Our implementation does not make copies of comprehension results, so it is up to the user to do this if a copy is needed. Aggregate queries are syntactically represented as calls to the functions `count`, `sum`, `min`, and `max`.

Maintenance code is run around updates by inserting calls to functions immediately before or after the update statement. Query expressions are replaced by retrievals of their stored result from a variable. In cases where new parameter values may need to be demanded, we use an expression similar to the comma operator in C and C++, to first call a function handling the demand set logic before retrieving the stored result.[2]

We consider any variable that is initialized to an empty set at module-level scope (i.e., global scope) to be a relation. Such variables must not be reassigned or aliased.

Linking modules: Our system operates on a single Python module (i.e., file) at a time. This module must contain all queries to be incrementalized, and all updates that can affect these queries. In practice, the module generally provides a simple API to execute the appropriate queries and updates as function calls. A client module that is *not* processed by IncOQ can then choose whether to import the original module or its transformed version. In

[2]Simply calling a function on the line before the query occurrence does not suffice, because the query could appear inside a short-circuiting boolean expression, or in the guard condition of a `while` loop.

effect this lets the client choose between the original and incremental implementations of the same API. Since the client is not transformed, it may use arbitrary Python language features.

If demand filtering is not used, then all sets, objects, and maps that are manipulated by the transformed module must be constructed (i.e., populated with their elements, field values, and keys respectively) inside that module rather than in the client module. This is to ensure that the auxiliary maps over M, F_f, and MAP are properly maintained. When demand filtering is used, we can relax this restriction so that values may be constructed inside the client, so long as they are not directly manipulated by the client after they have been passed to the transformed module. This is because the appropriate time to maintain invariants for a program that uses demand filtering is when the values become demanded, not when they are initially constructed. We take advantage of this in some of our DistAlgo applications (see Chapter 5): Values may be constructed inside the DistAlgo runtime itself without running any IncOQ-generated code, and this causes no inconsistencies so long as the transformed module uses demand filtering.

User interface: When invoked, the system reads configuration information from the command line and from annotations in the input file. This includes per-query options such as what evaluation strategy to use, as well as directives that tell the system to perform certain optimizations even when they cannot be automatically inferred to be safe. It also includes global options like whether to perform inlining of maintenance code, and whether to transform all queries or just the ones that have been specifically annotated. We have scripted the system to automatically invoke it on our suite of sanity test programs and our benchmark programs, and to emit files containing transformation statistics such as the number of queries processed and the LOC of the output file.

Evaluation strategies and demand: Valid implementation strategies for comprehensions include (1) leaving it alone so that it uses Python's straightforward left-to-right strategy, (2) incrementalizing it without demand filtering, or (3) incrementalizing it with demand filtering. We also implemented a middle-of-the-road strategy that recomputes the comprehension each time its result is needed, but does so using incrementally maintained auxiliary maps and the join order heuristic.[3] Aggregate queries can use either strategies (1) or (2); demand filtering and the middle-of-the-road approach do not apply.

[3]This is used in Figure 5.3.

When determining the demand parameters for a query, by default we select a set of parameters that do not appear to be constrained by membership clauses. However, we also allow the user to say that all parameters are to be chosen, or to provide a specific list of which parameters should be chosen. The demand set for the query is updated to include the current parameter values immediately before the query operation is performed, unless the user provides an annotation at that occurrence of the query saying that the values are already present. The user may specify a maximum size for the demand set, in which case an LRU cache is used to evict stale entries. There is also a primitive statement that can be used in input programs to direct the incremental implementation to clear its demand sets at that program point.

When using demand filtering, the demand strategy is constructed by iterating over each membership in a left-to-right order. As each membership is processed, we decide what existing tags may constrain its filter, and what new tags may be introduced as projections of this filter. The fact that we go in a left-to-right order prevents any tag that is introduced to the right of the current clause from filtering this clause, and thereby ensures that the demand strategy is a DAG. However, it also means that the user must order the query's clauses such that all retrieval expressions are reachable with respect to this extra left-to-right processing constraint. For example, the clause order

```
for z in y for y in x
```

would need to be reversed since `y` gets its value (and hence its demand information) from the second clause. Python programmers are already used to this kind of restriction since Python comprehensions only allow their clauses to be evaluated left-to-right. We always insert the demand clause as the leftmost clause so that it is at the root of the demand strategy. There is also a configuration option that allows the user to specify an alternative order for propagating demand through the query's clauses without having to edit the source code.

Recall from Section 2.4 that demand graphs are subject to restrictions about what kinds of arcs may enter and leave a node. For example, for a clause `(x, y) in` M, an incoming arc must be labeled `x` and an outgoing arc must be labeled `y`. In general, we model this by associating each kind of membership with information about which of its positions allow incoming and outgoing arcs. Our algorithm for determining the demand strategy looks like this.

```
tags = ∅
for each membership (v1, ..., vn) in R, in left-to-right order:
    current_filter = new filter over R
    for each vi in v1, ..., vn where i allows incoming arcs:
        for each t in tags where t is for variable vi:
            add t as a constraint for current_filter
    for each vi in v1, ..., vn where i allows outgoing arcs:
        add to tags a new tag for vi based on current_filter
```

After this algorithm runs, all tags that are not used to constrain a filter are dropped. As an additional option, we allow the user to specify that at most one tag may be used to constrain each filter; this is used by the experiment for Figure 4.4. When the clause is over a relation in the input program that has not been flattened, all positions allow both incoming and outgoing arcs.

The join heuristic for ordering a comprehension's clauses is as described in Chapter 2, but when there are ties for the best next clause to run, the leftmost such clause is chosen. Since this approach and the approach for selecting the demand strategy are deterministic, the overall transformation is deterministic. This helps us avoid regressions in our system and keep the experiment results stable.

Runtime library: Both the input and output programs import a small runtime library. For the input program, this library provides straightforward linear-time implementations of aggregate queries and image-set expressions, as well as identity-based sets which are needed to support nested sets in Python. For the output program, the library provides essential data structures such as counted sets and balanced binary search trees.[4] The library also tracks the sizes of auxiliary data structures to facilitate bookkeeping for benchmarking purposes.

To explain identity-based sets, it is necessary to provide a little background on equality and hashing. Python data types may define custom routines for equality comparison and hashing, with the constraint that two equal entities must hash to the same value. The hash value is used by some collections to track their elements, and in that case, the elements' hash values must not change so long as they are in the collection. These two facts taken together imply that any data type that is stored in a hash-based collection may not have a mutable equality relation.

Unfortunately, standard Python sets do have a mutable equality relation. Two sets compare equal iff their elements compare equal in a one-to-one fash-

[4]Via the 3rd-party `bintrees` package. `https://bitbucket.org/mozman/bintrees`.

ion. Since sets may have elements added and removed during program execution, they cannot be given fixed hash values. Our solution is to not use Python sets, but rather a custom set type with identity semantics: It only compares equal to itself, and its hash is created when it is instantiated and never changes. The same approach is also taken to define a custom map (dictionary) type; objects already have identity semantics by default under Python.

There is some time overhead associated with our custom set and counted set classes. Since these are implemented in Python, their operations are more expensive than the equivalent built-in set class. An extension module written in C would help ameliorate the cost.

ASTs: Transformation occurs at the level of abstract syntax trees (ASTs), making extensive use of the visitor design pattern. Python provides a standard library (`ast`) for parsing and manipulating its own AST format, as well as a script (`unparse.py`) in its source distribution for unparsing. However, we found this by itself to be too limited for our purposes. We therefore created a suite of utilities for dealing with abstract syntax trees, and spun off the more generic and reusable contributions as a separate library, iAST.[5]

A key design decision of iAST is that node objects are immutable and have structural equality semantics. This eliminates bugs related to aliasing of ASTs, and makes it easier to compare two syntax trees for equality or store them in a hashed collection. Their child nodes are also automatically type checked, so that each level of the tree is of the appropriate syntactic domain. Instead of defining node classes using boilerplate code, they are generated from an abstract grammar written in Zephyr ASDL[80], similar to the `Python.asdl` file in the Python source distribution. iAST also supports more advanced transformation utilities than the built-in library, such as source code templating, pattern-matching over ASTs, and macro expansion. This reduces the amount of programmer effort needed to construct and traverse ASTs.

Separation of concerns and LOC breakdown: For ease of design we have segmented the system into several components. In order of their dependencies, they are as follows.

[5] `https://github.com/brandjon/iast`

Component	Approximate LOC
AST utilities	1,800
Type analysis	750
Cost analysis	750
Symbol table and config	800
Auxiliary map logic	600
Comprehension logic	1100
Aggregate logic	500
Demand filtering	200
Object/set flattening	600
Misc rewritings	2,000
Runtime library	340

AST utilities does not include iAST and `unparse.py`, and none of these figures includes tests.

We have tried to avoid instances where the behavior of one component for transforming queries depends on another. For instance, the main logic for generating maintenance joins should not have to worry about the behavior of demand, aggregates, or object/set flattening, except to the extent that this impacts the optimal join order and the emitted code. One way that we have achieved this is by defining separate handlers for different kinds of membership clauses, including memberships over: M, F_f, MAP, TUP, arbitrary relations, subqueries, singleton sets, set differences of singleton sets, and so on. Although these kinds of clauses each have their own distinct AST node type, each handler provides a uniform interface for determining what the clause's properties are — e.g., the variables on its left-hand side, the relation on its right-hand side, which components allow incoming and outgoing arcs for the demand strategy, how to generate cost and code given that certain variables are bound, whether the clause requires demand filtering, etc. This helps us reduce coupling between the consumer of this information (such as the logic for comprehensions and demand filtering) and extension features (such as aggregates and object/set flattening).

Transformation steps: The overall order of transformation steps is as follows.

(1) Parse the input source code into Python AST format, and then convert into our own internal AST format. Parse configuration directives and annotations, and create symbol table entries for the queries.

(2) Preprocess convenience operations (e.g., set union) into primitive updates. Convert `if` clauses in comprehensions to membership clauses if they test for set membership. Decide which variables can be treated as relations.

(3) Flatten objects, sets, maps, and tuples for all updates and queries.

(4) Perform scope analysis to determine what the parameters of each query are. Determine the demand parameters, and introduce demand sets and demand comprehensions.

(5) Repeatedly choose an innermost query marked for incrementalization and transform it. Repeat until no such queries remain. Maintenance joins will be ordered, filtered, and expanded, and tags and filters will themselves be incrementalized, all before proceeding to the next query. Counting elimination and type-check elimination are done as the query is processed.

(6) Implement image-set expressions using auxiliary maps. Run result set elimination and inlining, if requested. Convert back to Python AST format and emit code.

Output code size: The final output program emitted by IncOQ consists of

(1) a preamble of comments describing the invariants that were generated,

(2) header code consisting of definitions of maintenance functions for each invariant and kind of update, along with definitions of demand helper functions and the variables holding stored results, and

(3) body code from the input program, with calls to maintenance functions inserted next to updates, and uses of queries replaced by calls to query functions that may also update demand.

Most of the examples that we transformed ended up with hundreds of lines or more of generated code in the final program. For instance, the social network example (Table 3.1) went from 39 LOC to 237 without filtering, and 451 with it. The bulk of this increase is due to the header code; of the 412 additional lines in the filtered program, 367 is for maintenance functions. The preamble does not contribute since it consists only of comments, and the body does not add too much in practice, although it can be $O(\#invariants \times \#updates)$ in principle even without inlining. To give an idea of what all this added code is

for, we describe how to estimate the size of maintenance functions based on the invariants, using the filtered program to illustrate.

Auxiliary map invariants are the easiest to estimate because they always take the same number of LOC per map, since no other invariants depend on them and they have no complex internal structure. Currently this number is 15 LOC, but straightforward intraprocedural optimization could reduce this further. The example has 5 auxiliary maps which would give an estimate of 75 LOC, but in actuality it takes only 61 LOC because two maintenance functions end up being eliminated as dead code.

Comprehension maintenance code size is primarily driven by the number of membership clauses. The more membership clauses, the more kinds of maintenance, and the larger each maintenance join's expansion becomes. For a given query, let $\#clauses_{mem}$ and $\#clauses_{all}$ be the number of memberships and the total number of clauses in the query, respectively. Then the number of distinct kinds of maintenance joins is $2 \times \#clauses_{mem}$, each of which needs to run $\#clauses_{all}$ clauses and update the stored result. Assuming it takes 1 LOC to run each clause and updating the stored result takes 2 LOC, then since the example query has 6 memberships and 1 condition this gives us an estimate of

$$(2 \times 6) \times (1\ LOC \times 7 + 2\ LOC) = 108\ LOC.$$

In actuality the size is 99 LOC because one of the maintenance functions is dead code. In other examples the amount may be larger than this estimate due to taking more than one line per clause (e.g., for type checks) or more lines to update the query (to manage a counted set or call other maintenance functions).

Tags and filters, while technically comprehensions, can be estimated with simpler rules due to their small size. Each tag or filter maintenance function takes around 7 LOC. Each tag depends on one relation, and each filter depends on $1 + t$ relations where t is the number of incoming tags it has. Each dependency on a relation requires two separate functions to handle addition and removal. In the example, there are a total of five tags and five filters, where three of the filters have one tag and two of the filters have two tags. Three removal maintenance functions for tags and filters are eliminated as dead code. This yields a total of $5 + (3 \times 2 + 2 \times 3) = 17$ dependencies on relations, and $17 \times 2 - 3 = 31$ functions, for a total of 31×7 LOC $= 217$ LOC. The actual number is 207 LOC.

Aggregate maintenance functions are around 6–11 LOC apiece. As in the case of auxiliary map maintenance, this can be reduced by applying intraprocedural optimization such as copy propagation. One reason for the redundant

code is that our system separates the rules for handling the specific aggregate operator from the rules for managing the aggregate result map itself. This makes the system more extensible but it requires the use of temporary variables to pass information from one part of the maintenance function to the next. Our system also deletes the old aggregate result (and any other stored results that depended on it) before assigning the new one, whereas in some cases it would be possible to just directly update the result in place.

Cost and type analysis: Fundamentally, our method is able to provide time cost bounds because the values that our generated code iterates over are given by invariants. For instance, the change to a comprehension result is determined by maintenance code that iterates over image-set expressions. In the final code, these expressions are replaced with map lookups where the contents of the maps are known to equal (and hence be bounded by) the corresponding image-sets due to their invariants. This contrasts with other methods that maintain complex indices without using invariants and that do not give bounds for their indices' sizes.

To take advantage of this fact, we have implemented an automated asymptotic time cost analysis in IncOQ. (Space cost analysis is not automated since it is easy to do by hand by looking at the generated invariants.) The analysis consists of three components: an algebra of asymptotic cost terms, an analyzer that associates pieces of code with a cost bound on running time, and a simplifier that uses type information to clean up the bounds for presentation to the user. We discuss each in turn.

Cost algebra: The terms in our algebra are as follows.

Term	Meaning
1	the unit cost, i.e., constant time
R	the size of the atomic domain for variable R
$R.J\{I = K\}$	the size of the corresponding image set
$R_{J/I}$	as above, but the size for the largest image set for any value of K
$C_1 \times \ldots \times C_n$	the product of zero or more costs
$C_1 + \ldots + C_n$	the sum of zero or more costs
$min(C_1, \ldots C_n)$	the minimum of one or more costs
?	an unknown cost

An empty product or sum is equivalent to the unit cost. These terms are similar to the ones used for cost analysis in [49], and smaller terms may also be

subsumed into larger ones following similar rules, e.g. $R_{J/I} \leq R.J\{I = K\} \leq R$.

We keep costs in a normal form, as minimums of sums of products of other non-composite cost terms. For instance, $min(R_{\texttt{out}},\ S\texttt{.in\{}v\texttt{\}} + (T \times T))$ is in normal form. It is straightforward to get an arbitrary term into this form by recursively combining normalized costs in a bottom up fashion, distributing at product and sum operators and simplifying along the way where possible.

Cost analysis: Given a piece of code with no function calls, a cost term to bound running time is constructed in a syntax-directed manner. The overwhelming majority of nodes return either the unit cost or the sum of their child node's costs. While loops produce an unknown cost, as do several other node types that are not used in maintenance functions but may be used by user code. To handle `for` loops, we need to know a bound on the size of the expression on the right-hand side. This is produced by an analogous process that operates only on expression nodes and returns a cost term for their size. For the common case of a `for` loop over a variable or an image-set expression, this gives us the corresponding costs for domains and for image-sets.

Function calls are handled by processing each function in topological order, so that each call's cost is known at the time it is seen. Recursive functions cannot be analyzed in this way, but all of our maintenance functions are non-recursive. The cost term for a function may use atomic domains and also image-set costs, so long as the only keys to the image set are its formal parameters; otherwise the image set cost must be widened to apply to any key. For instance, if a function has a parameter `x` and performs a nested iteration

```
def f(x):
    ...
    for y in E.out{x}:
        for z in E.out{y}:
    ...
```

then the cost $O(\texttt{E.out\{x\}} \times \texttt{E.out\{y\}})$ would have to be widened by replacing `E.out{y}` with $\texttt{E}_{\texttt{out}}$. At a function call `f(a)`, we then replace the formal parameter `x` with the actual argument variable `a` to obtain a cost $O(\texttt{E.out\{a\}} \times \texttt{E}_{\texttt{out}})$ for the call. If the actual argument is an expression instead of a variable, we widen once again to obtain a cost of $O(\texttt{E}_{\texttt{out}} \times \texttt{E}_{\texttt{out}})$. Using this system, we can associate every non-recursive function in the program with a cost, and every maintenance function with a cost that does not involve unknown costs.

Simplifying with types: The costs that are produced using the above method may be difficult for the user to read and understand because they can depend on the sizes of many different sets. We can simplify further by replacing domains representing set-valued variables with domains representing their types. This gives us very simple cost bounds like those in Tables 5.6 and 5.7. For a relation, generally this means replacing the domain of the relation with a product of its component's type domains, and replacing an image-set cost for the relation with the product of the domains of the unbound components. For instance, if the relation `E` above is a set of pairs of numbers, the costs would become $O(\texttt{number} \times \texttt{number})$ where `number` is the domain associated with the primitive type for numbers.

We have added a type inferencer to our implementation based on abstract interpretation. The type lattice is similar to the one described on page 2.6.10, but it also includes a type constructor for subtyping an existing type. This is useful for assigning meaningful names to types, for instance, declaring a `clock` to be the name of a particular subtype based on `number`. With type inference we can propagate type information that is provided by user annotations to other variables and expressions created by our transformation.

3.2 Experiments

In this section we investigate the effect of our method on the time and space performance of a program implementing the social network query (Example 1). Three benchmarks are performed. The first looks at the asymptotic tradeoff in running time caused by making the computation incremental, and at the asymptotic time and space benefits of demand filtering. The second benchmark examines how the overhead associated with demand filtering varies as the number of demanded parameter values changes. The final benchmark applies some of the optimizations of Section 2.6 and shows their constant-factor effects on running time and space usage. For benchmarks on other queries and programs, see Chapters 4 and 5.

All experiments were run using CPython 3.4.4, under 64-bit Windows 10 on an Intel Core i5-5250U CPU at 1.6 GHz[6] with 16 GB memory. No RAM exhaustion or disk paging was observed. Each measured execution of a program occurred in its own process, with the garbage collector disabled. Unless otherwise noted, all datapoints are the average of between 10 and 50 trials, using fewer than 50 only once the standard deviation is within 10% of the mean. All time measurements are CPU time in seconds. Additional space is

[6]Max 2.7 GHz under Intel Turbo.

Program	LOC	Time
`Original`	39	N/A
`Incremental`	237	1.02
`Filtered`	451	2.20
`Filtered (w/ type checks)`	494	2.27
`Filtered (OSQ strategy)`	397	1.97

Table 3.1: Lines of code and transformation time, for different programs implementing the running example query and its updates. The OSQ strategy variant is discussed in Chapter 4. All LOC measurements are without whitespace or comments. All transformation times do not include time for cost analysis.

measured as the size of the inserted global sets and maps, including in particular the stored query result and auxiliary maps, but not the space usage of objects and sets that are from the original program. The size of a set is the number of elements, and the size of an auxiliary map is the number of keys plus the size of the image set for each key. For all transformed programs, where possible, we avoid making a copy of the retrieved query result, and we avoid testing whether the queried values are in the demand set when this is already guaranteed by the test setup.

Transformed programs: We have written a small program containing functions for creating and updating the social network data and performing the query. Three different implementations are benchmarked: the original program (`original`), an incrementalized version with a demand set but no demand filtering (`incremental`), and an incrementalized version with demand filtering as well (`filtered`). Their transformation statistics are shown in Table 3.1. The demand strategy used for filtering is as in Example 11. The join orders are chosen by using our standard greedy heuristic, breaking ties by choosing the leftmost tied clause. No copy of the query result is needed, and any additions to the demand set are done during setup, so the transformed programs take constant time to perform the query.

Table 3.2 shows the generated asymptotic time bounds for all operations, in the general case. Our test data is generated to satisfy several requirements that enable us to apply optimizations and simplify these cost bounds.

- Each celebrity has a unique, unaliased `followers` field, so that the terms $|F_{\texttt{folin}}|$ and $|dF_{\texttt{folin}}|$ are at most 1.
- Each user has a unique `email` field value, so that the counting elimination and result set elimination optimizations both apply to `incremental` and

Operation	`incremental`	`filtered`
`make_user`	$O((\lvert U_{\texttt{out}}\rvert \times \lvert M_{\texttt{out}}\rvert)+$ $(\lvert U_{\texttt{out}}\rvert \times \lvert M_{\texttt{in}}\rvert \times \lvert F_{\texttt{fol}_{\texttt{in}}}\rvert))$	$O((\lvert U_{\texttt{out}}\rvert \times \lvert M_{\texttt{out}}\rvert)+$ $(\lvert U_{\texttt{out}}\rvert \times \lvert dM_{1\texttt{in}}\rvert \times \lvert dF_{\texttt{fol}_{\texttt{in}}}\rvert))$
`make_group`	$O(1)$	$O(1)$
`follow`	$O((\lvert F_{\texttt{fol}_{\texttt{in}}}\rvert \times \lvert U_{\texttt{out}}\rvert) + \lvert U_{\texttt{in}}\rvert)$	$O((\lvert dF_{\texttt{fol}_{\texttt{in}}}\rvert \times \lvert U_{\texttt{out}}\rvert) + \lvert U_{\texttt{in}}\rvert)$
`unfollow`	as above	as above
`join_group`	as above	as above
`leave_group`	as above	as above
`change_loc`	$O(\lvert F_{\texttt{fol}_{\texttt{in}}}\rvert \times \lvert M_{\texttt{in}}\rvert \times \lvert U_{\texttt{out}}\rvert)$	$O(\lvert dF_{\texttt{fol}_{\texttt{in}}}\rvert \times \lvert dM_{1\texttt{in}}\rvert \times \lvert U_{\texttt{out}}\rvert)$
`query`	$O(\lvert$`celeb.followers`$\rvert)$	$O(\lvert$`celeb.followers`$\rvert + \lvert$`group`$\rvert)$

Table 3.2: Running time cost bounds for each of the functions from the original social network program, after being transformed to contain incremental maintenance code with and without demand filtering. These bounds are automatically generated by our system, but we have taken the liberties of simplifying redundant terms and of remaining terms to be consistent with our convention; see Appendix B for the raw generated bounds. For `query`, the cost bound includes the time for demanding the parameter values; if they are already demanded then the cost is constant. For `make_user`, the costs are in actuality constant because the newly constructed user is not yet connected to any other values for the maintenance to iterate over. The cost bounds for `original` (not shown) are $O(1)$ for each operation besides `query`, which is $O(\lvert$`celeb.followers`$\rvert)$.

`filtered`.

- The demanded data is well-typed with respect to the safe typing in Example 20, so type check elimination is able to apply to `filtered` (but not to `incremental`).

Our benchmarks focus on two operations in particular: querying a pair of values for `celeb` and `group`, and updating a user's location. Table 3.3 shows the time costs for these two main operations, and the space costs for storing additional structures.

Asymptotic time and space: Our benchmark for measuring time performance is based on performing alternating queries and updates to a user's location, since both of these operations would be frequent in a real system, and since updates to nested objects are harder to support efficiently. Our test data is constructed to have up to 20,000 users and 1% as many groups as users. Each user follows 0.5% of all users, is in 5% of the groups, and is given one of 20 possible locations, one of which satisfies the query's condition clause. Three

Implementation	Query	Change location
`original`	$O(\|\texttt{celeb.followers}\|)$	$O(1)$
`incremental`	$O(1)$	$O(\|M\texttt{.in\{user\}}\| \times \|U_{\texttt{out}}\|)$
`filtered`	$O(1)$	$O(\|dM_1\texttt{.in\{user\}}\| \times \|U_{\texttt{out}}\|)$ or $O(1)$ if `user` is irrelevant
Implementation	**Additional space**	
`original`	0	
`incremental`	$O(\|U\| + \|F_{\texttt{fol}}\| + \|M\| + \|Q\|)$	
`filtered`	$O(\|U\| + \|dM_1\| + \|dM_2\| + \|Q\|)$	

Table 3.3: Time bounds for the benchmarked operations and space bounds for auxiliary structures used by the transformed programs, under the given assumptions about the input data. Q represents the query result. Note that `incremental` takes space proportional to the number of sets and celebrity user objects in the whole program.

of the users are chosen to be demanded celebrities, and one group is chosen to be a demanded group (i.e., the demand set holds three pairs, one for each celebrity with this group). All relationships are chosen uniformly randomly, so that the sizes of all image sets are easier to understand. This setup gives us the following growth rates for the cost terms in Table 3.3, where x is the number of users.[7]

Time term	Bound
$\|\texttt{celeb.followers}\|$	$O(x)$
$\|M\texttt{.in\{user\}}\|$	$O(x)$
$\|U_{\texttt{out}}\|$	$O(1)$
$\|dM_{1\texttt{in}}\|$	$O(1)$

Space term	Bound
$\|U\|$	$O(1)$
$\|F_{\texttt{fol}}\|$	$O(x)$
$\|M\|$	$O(x^2)$
$\|Q\|$	$O(x)$
$\|dM_1\|$	$O(x)$
$\|dM_2\|$	$O(x)$

The benchmark measures the time to perform 200,000 pairs of query and update operations, where the query operation runs the query on a randomly chosen pair in the demand set, and the update operation changes a random user's location. Space usage is measured after the query and update operations are done. This procedure is also repeated without the query operations in order to measure just the time cost of the updates and their associated maintenance;

[7] We can also expect $|Q|$ to obey a tighter bound of $O(1)$, since the increase in the number of followers of demanded celebrities is offset by the decreased likelihood of belonging to a demanded group. This also means that only a constant number of users are expected to be in the intersection of the two tag sets for `user`.

the cost of just the query operations is then obtained by subtracting from the results of the first run.

The results are shown in Figure 3.1 and confirm the above asymptotic bounds. In particular, they show that the transformation eliminates the linear cost of querying, and that demand filtering eliminates the linear cost of updating. Demand filtering also reduces the additional space usage by a linear factor. (Note that the total space usage of the program is still quadratic due to user data that is present in the original program.)

Effect of demand set size: Demand filtering works best when the number of demanded parameter values (and hence the number of relevant values that can be in tag sets and filters) is small compared to the whole dataset. To investigate this effect, we ran a modified version of the previous benchmark, where the data is always generated as for the rightmost datapoint of Figure 3.1 (i.e., with 20,000 users), but where the number of demanded users varies from 1 to 20,000. In addition, each of the random updates is to one of the users that both follows a demanded celebrity and is in the demanded group; otherwise, most of the maintenance for updates in `filtered` would terminate almost immediately. To better show the tradeoffs, we report two measurements for running time: the time needed to perform the queries and updates (as in the previous benchmark), and this time summed with the time for demanding the parameter values and performing the associated maintenance during the initial setup.

The results are shown in Figure 3.2. `filtered` runs relatively faster when few celebrities are demanded, but its advantage diminishes as the size of $dM_{1\texttt{in}}$ grows larger relative to $M_{\texttt{in}}$. If we are not concerned with the time needed to demand the parameter values, then `filtered` always performs better no matter how many users are demanded, because the extra work done to maintain $dF_{\texttt{loc}}$ is offset by the time saved by not performing type checks — an optimization that in general does not apply to `incremental`. To isolate this effect, we also ran a version of `filtered` that does not perform type check elimination; for this version there is a crossover point when about 70% of the users are demanded. On the other hand, when we consider the total time including time for demanding the parameter values, the crossover point is around 20–25%. For space usage, `filtered` is initially much better, but ends up taking even more space than `incremental` due to overlap in the tags, filters, and auxiliary maps.

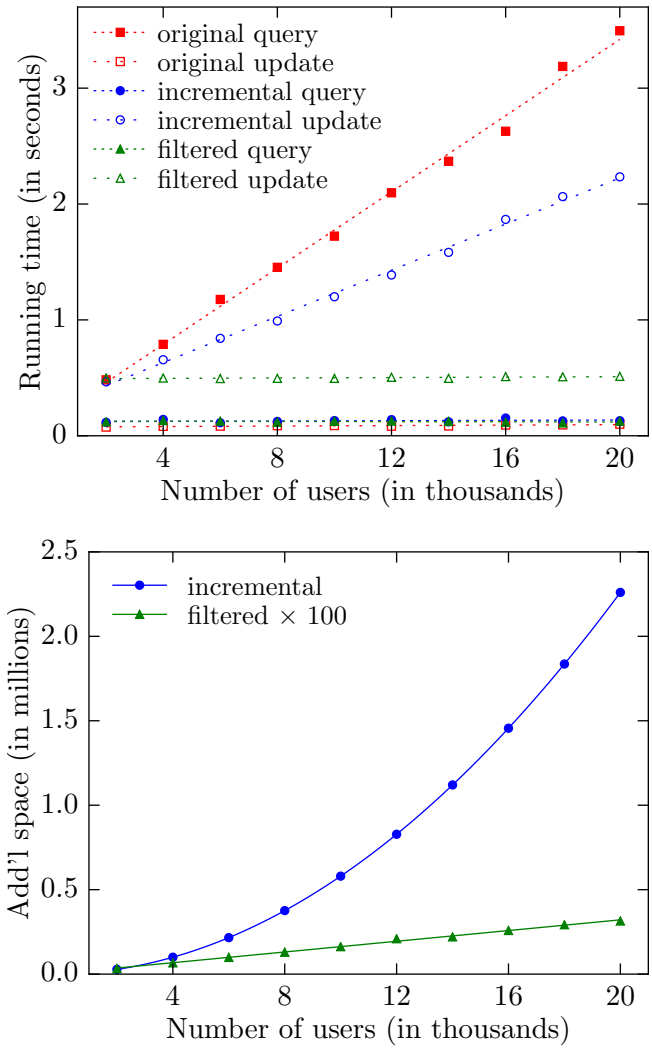


Figure 3.1: Growth in running time (top) and additional space usage (bottom). The flat lines in the top figure actually have a slight (~30 ms) increase from the leftmost datapoint to the rightmost one. This appears to be due to increased cache misses from doing the same number of random accesses over an increasingly broad set of possible users. Indeed, even `original` updates suffer from this effect, indicating that the cause is present in our test setup. We have run experiments to confirm that cache misses play a role in this increase.

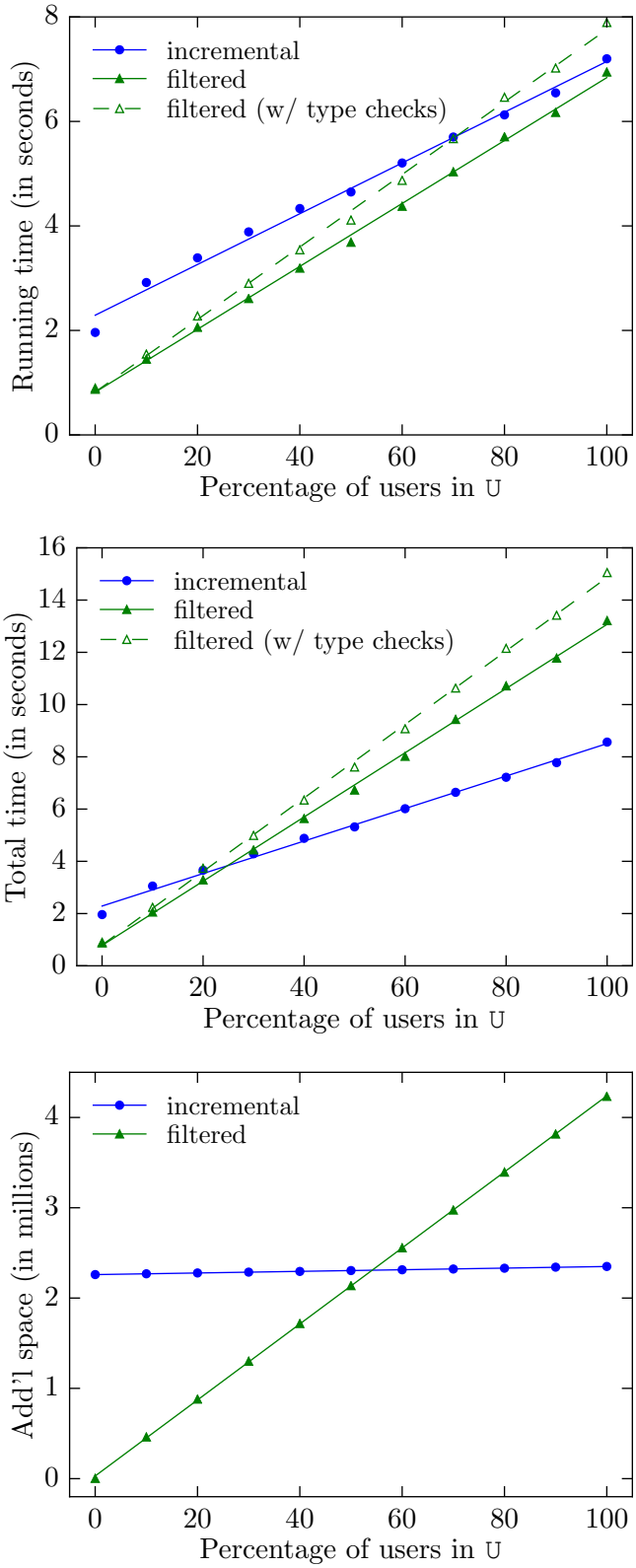


Figure 3.2: Effect of demand set size on (1) the running time for just queries and updates (top), (2) the running time of these operations plus time for demanding the parameter values (middle), and (3) additional space usage (bottom). The initial datapoint is at $x = 1$ user.

Program	LOC	Time
`Unoptimized`	601	2.66
`Inlining`	957	3.72
`Counting elim`	798	3.22
`Result set elim`	779	3.25
`Type check elim`	708	3.06
`Maint case elim`	411	N/A

Table 3.4: Transformation statistics for social network program at different levels of optimization. `Maint case elim` has no transformation time since it is a manually modification to `Type check elim`.

Constant-factor optimizations: In order to investigate the benefits of the optimizations of Section 2.6, we produced a series of increasingly optimized versions of the transformed program with demand filtering. 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, maintenance case elimination depends on inlining since different maintenance joins are omitted at different updates to M, and result set elimination depends on counting elimination to ensure the result sets are dead code. A more thorough examination of these optimizations was performed in [12]; compared with that work, we do not treat selecting a better set datatype as its own separate optimization for benchmarking purposes.

The benchmark varies how "dense" the data is, in the sense of users following other users. This affects how many changes need to be made to the stored result, and thus how much of the running time is affected by the part of the code that is optimized. The setup is similar to the rightmost datapoint from Figure 3.2, but the number of followers per user is made to vary inversely with x, so as to fix the total size of all follower sets at 2,000,000. For example, the followers per user are 1,000 and 100 at $x = 2,000$ and $x = 20,000$ respectively.

The improvement in running time is shown in Figure 3.3. Counting elimination has the largest effect of any single optimization, in part because it enables the use of a more efficient set datatype. Maintenance case elimination has no measurable effect on running time because the benchmarked update — changing a user's location field — is unchanged by getting rid of maintenance code for updates to M. The effect of these optimizations on code size is shown in Table 3.4; all but inlining reduce code size.

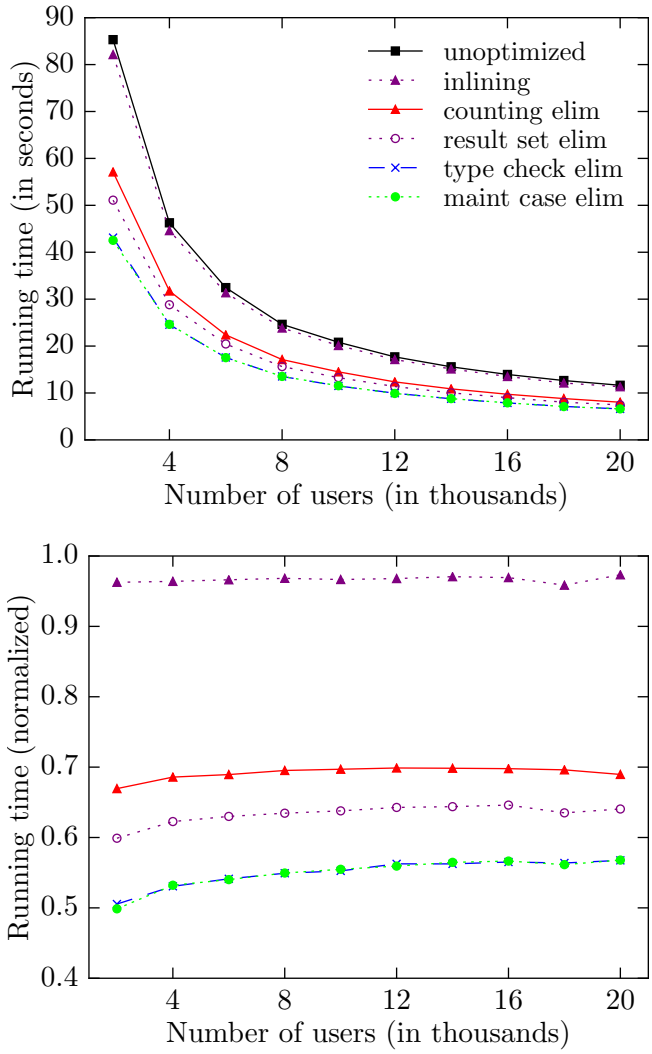


Figure 3.3: Running time for various levels of optimization of the transformed, demand-filtered program. Each optimization is applied cumulatively, so that `Maint case elim` is the most optimized. The second plot shows time normalized to the cost of `Unoptimized`. Time decreases for higher numbers of users because the data is sparser and therefore requires fewer changes to the stored query result.

Chapter 4

Comparing approaches

We have two goals in this chapter. The first is to put our approach in context with related work. The bulk of this discussion is contained in Section 4.1. We single out two particular systems for more detailed analysis, including benchmarks, in Sections 4.3 and 4.4: the Object Set Query (OSQ) method [74, 72] and the Java Query Language (JQL) [84, 85]. Both of these systems support incremental computation of high-level object queries similar to ours, but without cost guarantees or without handling all kinds of updates.

Our second goal is to examine alternative strategies for formulating invariants, and in doing so, to justify why we chose the particular strategy we did. This is explored in Section 4.2. One of the results of the discussion is that we arrive at the specific modifications to our demand strategy that would be needed to model the behavior of OSQ.

Our system can express any object-set comprehension query that OSQ and JQL can.[1] Moreover, there exist choices for the demand strategies (including possibly using the modified invariants that model OSQ) and for the join orders, such that the implementation generated by our method performs at least as well. Indeed, the benchmarks of Sections 4.3 and 4.4 show asymptotic improvements.

4.1 Related work

There is much work on incremental computation [69, 53] and materialized views [26]. We begin by describing systems for efficient incremental querying. We then discuss self-adjusting computation, which makes programs incremen-

[1] Treating JQL list comprehensions as sets, and not considering OSQ extension features like group-by and union, which we do not discuss.

tal even without queries. Following this, we discuss the relationship between our work and reactive programming, demand transformation, invariant transformation, efficient join computation, and other works. Our own related publications on this topic include [45, 12].

Query systems: There is much work on incrementalizing relational queries (e.g., in RDBMSs) and logic queries (e.g., in Datalog). Qian and Wiederhold [68] and Griffin et al. [23, 24] apply finite differencing to relational algebra to derive expressions for change propagation. Ceri and Widom [15] derive rules for incrementally maintaining an SQL view; the class of views is restricted so as to avoid the need to count duplicate derivations for tuples in the view result. For Datalog, Gupta et al. provide the Counting algorithm [27, 25] and Delete-and-Rederive (DRed) method [27] (adjusted into a Backward/Forward algorithm by Motik et al. [59]) to incrementally evaluate queries involving stratified negation, aggregation, and recursion. Liu et al. [49, 47] compile Datalog rules into an imperative program that performs a bottom-up evaluation using incremental addition updates. Unlike our approach, it materializes intermediate joins, which can improve join time at the expense of consuming asymptotically more space than the original query result; see the Join Decomposition approach in Section 4.2.

A central theme in most of these methods is that a relational join $R \bowtie S$ can be incrementally computed by algebraically distributing the join over the union of the old input value R and the change to this input ΔR.

$$(R \cup \Delta R) \bowtie S = (R \bowtie S) \cup (\Delta R \bowtie S)$$

The first join on the right-hand side is the old result of the query (which is already available), and the second join is the incremental change. In our work this change is computed by maintenance joins, but many other works for incremental queries perform the same underlying computation, including other methods for object queries like OSQ and JQL.

For certain kinds of queries, the delta expressions for the change to the query can themselves be recursively incrementalized. Koch et al. [36, 37] do this to achieve constant-time updates for a class of aggregate queries. Obtaining similar performance under our method would require a more sophisticated treatment of aggregates over comprehensions; we would have to consider both the inner and outer queries together instead of transforming them one at a time. The repeated incrementalization of delta expressions to obtain simpler delta expressions is reminiscent of how Paige's finite differencing can be used to derive Brigg's method for efficiently evaluating polynomials [66].

Armbrust et al. use incremental views to answer queries in a *scale-independent* way, meaning that the response latency remains constant-bounded as the size of both the input data and computing cluster increase. Such strict performance bounds preclude queries with non-constant bounded output size.

The problem of incremental maintenance becomes harder when we consider nested relational queries. The main issue is that updates to inner relations cannot be efficiently expressed; even small changes require deleting and reinserting the entire nested relation. This limitation applies even when the change is to a top-level relation, since inner relations may be created by subqueries over those relations. Lupei et al. [54] use a shredding transformation to extract the inner queries into a family of mostly flat queries, where the occurrences of inner queries are replaced by labels identifying the particular result relation. Incremental changes can then be made to a separate mapping from labels to subquery results. Kawaguchi et al. [33] extend the Counting algorithm to joins of nested relations. The occurrence of the inner relation is replaced in the outer relation by its id, and the occurrence of the inner relation in the join expression is replaced by a new top-level relation that incorporates this id as one of its attributes. This kind of flattening is similar in spirit to how we flatten nested sets and fields, but our method operates in a different problem domain as it also deals with aliasing, demand, and generating imperative code.

A variety of query systems exist for imperative programming languages. Liu et al. [52] give a system for incrementalizing set comprehensions using auxiliary maps to implement image-set expressions (see also tuple pattern matching [73]). Both our method and OSQ improve upon [52] by supporting objects, projections, self-joins, clause ordering, and demand. LINQ [55] provides an interface to querying relational and XML data from within the host programming language. i3QL [58] provides in-memory SQL-style queries with recursion (using DRed), optimized with algebraic rewriting, but it does not provide object queries or cost bounds.

Self-adjusting computation:[2] This technique for incremental computation works by maintaining a dynamic dependency graph of the computation [6, 2, 5]. It is a general technique — for instance, examples from these works highlight problems in computational geometry. Intuitively, it instruments the program's dataflow to log how intermediate and output values are produced during the initial scratch computation. Then, when the inputs are modified, a change propagation algorithm reexecutes the relevant pieces of code. This process can be demand-driven [31], and may associate names with intermedi-

[2]When used without memoization, it is called *Adaptive functional programming.*

ate results to better track and reuse them [29]. Although originally formulated for functional languages (with write-once mutable cells), it has been extended to the imperative domain as well [3, 28, 30], and a similar technique has been applied to invariant checking in Java [76]. Self-adjusting computation can be used to derive algorithms for the dynamic versions of problems, as an alternative to writing complicated specialized code.

However, bounding the cost for self-adjusting computation can be a difficult and ad hoc process, and the system's overhead can be large.[3] Obtaining the bound requires a detailed understanding of how the program responds to changes, as opposed to having a source-level reasoning technique. Asymptotically, it is known that the time overhead can never be worse than a scratch recomputation [5], although the space cost can be as large as the program's running time [4]. This has been somewhat mitigated with newer techniques, which enable tracing the computation at the level of abstract data types instead of memory cells [4], and bounding cost by the change to the computation trace [43]. Improvements have also been made to make it more programmer friendly, by using a simple annotation to mark changeable data instead of requiring an obfuscated monad syntax [18].

Nonetheless, it is not clear how to apply self-adjusting computation to the computation of queries like our object-set comprehensions; or whether doing so would yield good update maintenance times; or whether it would require auxiliary data proportional in size to the result of intermediate large joins. Our cost bounds are possible because of our reliance on invariants. The only auxiliary space needed in our method is to hold the result of an expression that is computed incrementally using an invariant. The maintenance time required for a comprehension is easily determined by looking at its clauses, without the need for program-wide or application-specific reasoning.

Reactive programming: Although reactive programming [9, 75] is sometimes specified using imperative dataflow rules [19], it can also make use of high-level declarative queries. Instead of having the programmer store the result of a computation in a variable and worry about manually keeping this value up-to-date, the programmer writes a query and lets the framework take care of reevaluating it when the inputs change. An incremental and demand-driven implementation has the additional advantage that the reevaluation is efficient, but usually the primary concern is consistency with the inputs. With minor modifications, IncOQ be made to facilitate reactive programming by having the queries return view objects over their results. Existing systems

[3] Around 2x for complex programs, and around 15-30x for basic list operations [5].

[42] for reactive programming with object queries may require the user to specify a dataflow pipeline with a definite order, e.g. using map and filter operations. This contrasts with our object-set comprehensions, where clauses are unordered but maintenance code may choose from many join orders.

Demand: Magic-set transformation (MST) [10, 11] is a well-known technique for transforming a logic program such that its bottom-up computation becomes demand-driven. Both MST and our method define demand in an intensional manner (i.e., using rules and invariants). Our tag sets are analogous to the magic sets themselves, with our demand set corresponding to the set of seed values. Our way of defining filters using tags can be characterized as a Sideways Information Passing strategy (SIP) [11] that intentionally drops demand precision in order to save space; see the Join Decomposition approach in Section 4.2 for what would happen otherwise. Besides MST there exist other demand transformations with strong performance guarantees [78, 79].

Invariant-based transformation: Transforming a program to maintain many interrelated invariants requires a systematic method (see [53] for an overview). For our purposes finite differencing [66] serves as that framework. This framework has also been used in database contexts to efficiently implement integrity constraints [65], and has inspired many of the aforementioned change-propagation approaches.[4] Finite differencing requires a library of rules describing how to maintain each query form for different kinds of updates. In our case, the rules are implicitly generated by the method for deriving maintenance joins for relational comprehensions. Finite differencing also informs the design of our method with regard to the order in which invariant transformations are performed. We discuss this aspect more in Filter vs. Incrementalize in Section 4.2.

Although our implementation includes a complete incrementalization subsystem, it would also be possible to have our system emit these rules for a dedicated transformation system (e.g., InvTS [46, 21]). A sophisticated transformation engine with stronger alias analysis and type analysis would help reduce the number of update occurrences that are transformed and the number of type checks that are inserted, thereby reducing code size and improving running time by a constant factor. The transformation could also be improved in the future to be sensitive to, and localized to, class scope [50].

[4] Loosely, we can think of maintaining invariants as a form of repairing [81] inconsistent database states, although there is only one correct way to repair a view by bringing it up-to-date with its base relations.

Joins: Our object-set comprehension queries, once flattened, are join queries. Their incremental maintenance code makes use of other, smaller join queries. It stands to reason that our method can benefit from any headway made in the general problem of efficient join computation, which is well-known to be difficult.

Recent work on natural join queries has focused on bounding the running time by considering the problem's geometric interpretations. Ngo et al. [64] developed an algorithm that is worst-case optimal, meaning that for a given input size it is no worse than the time required for a pathological input of that size. This upper bound is based on the best possible fractional edge cover of the query's hypergraph. Notably, there exist queries for which no project-join plan, including the nested clause orderings that we produce, can achieve the bound. The problem is not merely theoretical; it actually prevents us from achieving optimal worst-case complexity in the running example query (see the footnote in the Join Decomposition approach).

Subsequent work [63] considers a more fine-grained notion of optimality that uses the size of a minimum "certificate" for the problem instance. Their algorithm is based on exploring gaps in the output space, whose projections down to the input relations correspond to gaps between successive input tuples. They achieve optimal running time for β-acyclic queries, with the restriction that the input relations are all indexed by the same search criteria. This restriction was later removed [35] by generalizing certificates to be based on geometric gap boxes rather than index comparisons. This yielded a single framework for natural joins that achieves both the optimal worst-case bounds and the beyond-worst-case certificate-based results.

Better asymptotic terms for bounding join results means better cost bounds for code that iterates over join results, e.g., in maintenance code for queries with nested subqueries. Better algorithms for actually computing the join could mean faster maintenance code, although we would have to see how such strategies interact with demand.

Other works: For the functional programming paradigm, memoization is a basic precursor to incremental computation [56]. More powerful approaches look at better reuse of subproblems [67] and modifying functions to return and make use of intermediate and auxiliary data [44]. Cai et al. [13] provide a framework for incrementalizing the simply typed lambda calculus, similar to how finite differencing is used to transform imperative programs. Although it is static and general, it heavily relies on specialization in order to be of use to a particular application: It requires users to provide the primitives, their

type information, maintenance rules, and certain lemmas for extending the correctness proof.

Nakamura [60] gives a translation of complex values down to nested sets, and incrementalizes the sets in a uniform manner. However, no cost bounds are given, and it does not use invariants for propagating demand; our translation also results in flatter queries. Kemper et al. [34] describe caching of query functions on objects, but the general problem with memoizing such functions is that the results are invalidated or inapplicable when the nested values are updated, including when a collection is modified. Both [34] and another work [38] dealing with objects attempt to optimize using object-oriented features such as encapsulation and inheritance, whereas our object model is relatively relaxed in that it allows structural typing (indeed, duck typing) and does not assume that objects are related or organized by class.

4.2 Alternative strategies for incremental computation

In this section we discuss several alternative ways of designing and applying invariants for incremental computation. The first four techniques concern the demand strategy, and of those, three are useful for modeling the behavior of OSQ with respect to demand. After that, we examine an approach that computes more intermediate results, saving time at the expense of space. Finally, we justify why our Filter step operates on the maintenance joins produced by the Incrementalize step, rather than on the relational comprehension that is the input to the Incrementalize step.

Single Tag: Our default heuristic for demand strategies is to choose any maximum DAG satisfying the restrictions. This produces strategies such as the one in Example 11. The Single Tag strategy adds the constraint that all arcs labeled with a given variable have to originate from the same node. For the running example, this means we have to eliminate either $T_{\texttt{user}_1}$ or $T_{\texttt{user}_2}$, and the two filters $dF_{\texttt{loc}}$ and $dF_{\texttt{email}}$ would be changed to depend only on the the one that is not eliminated, e.g.,

$$dF_{\texttt{loc}} = \texttt{\{(user, loc) : user in } T_{\texttt{user}_1}\texttt{, (user, loc) in } F_{\texttt{loc}}\texttt{\}}$$
$$dF_{\texttt{email}} = \texttt{\{(user, email) : user in } T_{\texttt{user}_1}\texttt{, (user, email) in } F_{\texttt{email}}\texttt{\}}.$$

The benefit of this strategy is that less work is spent maintaining demand invariants. Indeed, in some cases the benefit can even be asymptotic. In

the case of the running example, the cost of adding a new pair to the demand set can be reduced from $O(|\texttt{celeb.followers}| + |\texttt{group}|)$ down to just $O(|\texttt{celeb.followers}|)$ or just $O(|\texttt{group}|)$, provided that we are also able to eliminate one of dM_1 or dM_2. Such an optimization would require replacing uses of the removed filter with uses of $T_{\texttt{celeb\_fol}}$ or $T_{\texttt{group}}$ respectively, and ensuring that no join order requires an inverse map over the filter.

Monotonic Tags: Monotonic relations are relations that can only grow, never shrink. For any relation R, define its monotonic version R^m to hold all the tuples that have ever been added to R. Note that R^m can be computed incrementally without using a counted set: Whenever an element is added to R, it is also added to R^m if not already present, and whenever an element is removed from R, no action is taken.

In the Monotonic Tags strategy, we replace U and all tags with their monotonic versions, and we modify the definition of each filter by replacing uses of tags with uses of their monotonic versions. For instance, $dF_{\texttt{loc}}$ becomes

$$dF_{\texttt{loc}}{}' = \texttt{\{(user, user\_loc) : user in } T_{\texttt{user}_1}{}^m\texttt{, user in } T_{\texttt{user}_2}{}^m\texttt{,}$$
$$\texttt{(user, user\_loc) in } F_{\texttt{loc}}\texttt{\}}$$

Under these definitions, filters are still subsets of their underlying relation, but values may be in tags and filters even if they are no longer accessible from demanded parameter values, so long as they satisfy a chain of "has-ever-been-in" relationships. It is still correct to use these modified filter definitions in a maintenance join, because they are supersets of the normal filters and subsets of the underlying relation. However, optimizations such as type check elimination may no longer be valid.

Performance wise, this strategy can potentially save a constant factor of time and space by not tracking counts for tags and the demand set. It also ensures that maintaining tags and filters for removal updates takes only constant time, as there is no demand propagation cost. At the same time, the total cost of demand propagation for additions is amortized to constant time, over all updates to any value that is demanded or will later become demanded. This is because the same value cannot be added to the same tag set twice. The downside, of course, is that this strategy "leaks" demand in the same sense that a program that never deallocates objects will leak memory. This drawback is best observed when the demanded values are small in number but are constantly being replaced by new values.

Shared Filters: Normally, a separate filter is created for each occurrence of a relation in the query's relational comprehension. Under this strategy, only a

single shared filter is created for each relation, regardless of how many times it occurs. The shared filter holds the union of the corresponding individual filters. Correctness follows by the same argument used above for the Monotonic Tags strategy: For each place that a shared filter is used in a maintenance join, it holds a superset of the filter that would have appeared there under the normal method, and a subset of the relation that it is filtering.

The rule for maintaining a shared filter is simple: Use a counted set, where the count associated with each element is how many of the underlying filters contain the element. Each addition or removal to an underlying filter triggers a counted addition or removal to the shared filter. In the case that the Monotonic Tags strategy above is also used, any removal from an underlying filter must be due to a removal from the relation being filtered, and therefore must also be accompanied by removals from all the other underlying filters. This makes it possible to perform counting elimination on the shared filter, and to eliminate all the underlying filters as dead code. Furthermore, in the special case where the underlying filters had significant overlap between them, this can lead to a constant-factor reduction in storage space.

In general, we expect the benefit of this reduced space usage, which only occurs in specific circumstances, to be offset by the increase in query maintenance time associated with reduced demand precision. It also makes Theorem 2 no longer hold, so type check elimination and other optimizations may not be possible.

Low-information Tags: One of our restrictions on the demand strategy avoids introducing new tags based on the first position of a clause over M or F_*. For the running example, these tags might include

$$T_{\texttt{celeb\_fol}_2} = \{\texttt{celeb\_fol : (celeb\_fol, user) in } dM_1\}$$
$$T_{\texttt{user}_3} = \{\texttt{user : (user, user\_loc) in } dF_{\texttt{loc}}\}$$

The intuition is that these tags only indicate whether a set is empty and whether an object has defined its field, respectively. Presumably such tags are less likely to be worth their maintenance overhead. It is trivial to configure our implementation to generate these tags nonetheless.

Join Decomposition: In our standard strategy and all of the alternative strategies considered above, demand information is only propagated via tags. This intentionally discards information that could help us determine more precisely what values are relevant to the query, in order to save space. Thanks to this sacrifice, the space usage for all tags and filters is amortized over the

space used by the data accessible from the demanded parameters values, at least when nested queries are not used.

We can instead consider an approach that retains all available demand information, by keeping the result of every join available. In this new approach, the query is decomposed into a left-join tree, with each intermediate result materialized and incrementally maintained. In general, the relational comprehension

$$\texttt{\{}result\ \texttt{:}\ clause_1\texttt{,}\ \ldots\texttt{,}\ clause_n\texttt{\}}$$

gets split into the invariants

$$\begin{aligned} J_1 &= \texttt{\{}jresult_1\ \texttt{:}\ clause_1\texttt{\}} \\ J_i &= \texttt{\{}jresult_i\ \texttt{:}\ jresult_{i-1}\ \texttt{in}\ J_{i-1}\texttt{,}\ clause_i\texttt{\}} \qquad \text{for } i = 2, \ldots, n \end{aligned}$$

where $jresult_n$ is just $result$, and $jresult_i$ for $1 \leq i < n$ is a tuple of all query variables that (1) appear somewhere in clauses $1, \ldots, i$, and (2) appear somewhere in clauses $i+1, \ldots, n$ or in the result. Each J_i is equivalent to the join of all clauses up to the i^{th} clause, projecting out variables that aren't needed for later joins or the final result. The final query answer is stored in J_n.

This strategy corresponds to the implementation technique used in [49, 47] for rules with more than two hypotheses (i.e., break them down into rules with only two hypotheses). It also corresponds to magic-set transformation with a full (maximal) SIP [11].

Here are the invariants produced for the running example, along with their

asymptotic storage costs.

$J_1 = U$ $(O(|U|))$

$J_2 =$ `{(celeb, group, celeb_fol) :` $(O(|U|))$
`(celeb, group) in` J_1`,`
`(celeb, celeb_fol) in` $F_{\texttt{fol}}$`}`

$J_3 =$ `{(celeb, group, user) :` $(O(|U| * |M_{\texttt{out}}|))$
`(celeb, group, celeb_fol) in` J_2`,`
`(celeb_fol, user) in` M`}`

$J_4 =$ `{(celeb, group, user) :` $(O(\#triangles))$
`(celeb, group, user) in` J_3`,`
`(group, user) in` M`}`

$J_5 =$ `{(celeb, group, user, user_loc) :` $(O(\#triangles))$
`(celeb, group, user) in` J_4`,`
`(user, user_loc) in` $F_{\texttt{loc}}$`}`

$J_6 =$ `{(celeb, group, user_loc, user_email) :` $(O(\#triangles))$
`(celeb, group, user, user_loc) in` J_5`,`
`(user, user_email) in` $F_{\texttt{email}}$`}`

$J_7 =$ `{(celeb, group, user_email) :` $(O(\#result))$
`(celeb, group, user_loc, user_email) in` J_6`,`
`user_loc = "NYC"}`

Here, *#triangles* means the number of celebrity-group-user triples where the celebrity and group form a demanded pair and where the user is in the intersection of the celebrity's set of followers and the group set.

Under these invariants, the time cost of maintenance for an update to a user's location is nearly optimal. The maintenance code for J_5 does a retrieval from an auxiliary map over J_4 to obtain precisely all pairs in U that form a triangle with the updated user, while the maintenance for J_6 and J_7 is constant-time for each such triangle. Compare this to our normal demand strategy, where we conservatively estimate whether the updated user is relevant to the query, and where we possibly find many `(celeb, group)` pairs that do not form triangles with the updated user.

On the other hand, the decomposition approach can take much more space. Invariants J_3 through J_6 can each be larger than both the space used by the original program and the space needed to hold the final query result (i.e. J_7). J_3 in particular has a tight worst-case cost bound of $\Theta(n^2)$ (where $n = |U| = |M|$), even though J_4 through J_7 are bounded by $O(n^{3/2})$.[5]

[5]Any triangle join $R(A, B) \bowtie S(B, C) \bowtie T(C, A)$ has output size at most $O(n^{3/2})$ ($n = |R| = |S| = |T|$), by the Loomis-Whitney inequality and AGM bound; see [64].

Filter vs. Incrementalize: Recall that the Incrementalize step creates an invariant for a relational comprehension, and then inserts maintenance joins that will later be rewritten by the Filter step. A natural question to ask is: Why not filter the relational comprehension itself, and then use it as the basis of the invariant instead of the original relational comprehension? That way, the inserted maintenance joins would not require any additional filtering. We considered this possibility, and it turns out that although the generated code would be correct, our method produces smaller code size and makes it easier to find good cost bounds. To explain why, we must first understand how the order of invariant transformation is determined by finite differencing.

Finite differencing iteratively transforms the program to preserve one invariant at a time. Let Q_1 and Q_2 be two comprehensions, and let us write $Q_1 \leq Q_2$ if we are required to transform the program to maintain Q_1's invariant before we may transform the program to maintain Q_2's invariant. There are two reasons why we may have the constraint $Q_1 \leq Q_2$:[6]

(1) Q_2 directly depends on the value of Q_1. The maintenance rule for Q_2 will insert code at updates to Q_1, but before the rule for Q_2 can apply, we must have first inserted the maintenance code for Q_1.

(2) Q_1's maintenance rule introduces new code that contains occurrences of Q_2. Transforming Q_2 afterwards ensures that its invariant holds at all program points where it is used, including inside the maintenance code for Q_1.

Under our normal method, where we filter maintenance joins, the second constraint tells us that tags and filters must be transformed after the comprehension that produced those maintenance joins, so that the contents of the tags and filters are up-to-date where they are needed.[7] In contrast, under the alternative approach that filters the relational comprehension, the comprehension being incrementalized now directly depends on the filters (and indirectly on the tags), and so the first constraint implies that the query must be incrementalized after its tags and filters.

Now let's illustrate the code that is produced using both methods. We will use a code outline of the inserted code, similar to what we did in Section 2.4 but without showing costs for each line. We'll use an update to U because

[6]See the definition of a "differentiable chain" in [66].

[7]Even though only filters are directly used by filtered maintenance joins, the tags must still be up-to-date at that point in code, or else the filters would be out of sync with the original relations.

this update causes the most demand propagation and therefore shows the most pronounced difference. Under the standard method we get the following.

```
    add to U
```
after incrementalizing the query:
```
    add to U
    maint query for U
```
after filtering:
```
    add to U
    maint query for U, using tags & filters
```
after incrementalizing tags and filters, in dependency order:
```
    add to U
    maint T_group for U
      maint dM_2 for T_group
        maint T_user2 for dM_2
          maint dF_email for T_user2
          maint dF_loc for T_user2
    maint T_celeb for U
      maint dF_fol for T_celeb
        maint T_celeb_fol for dF_fol
          maint dM_1 for T_celeb_fol
            maint T_user1 for dM_1
              maint dF_email for T_user1
              maint dF_loc for T_user1
(A) maint query for U, using tags & filters
```

Note that there is only one maintenance join created for the query (marked A), and therefore only one place where we have any join order choices to make. Regardless of the choice of this join order, the overall cost is $O(|\texttt{celeb.followers}|+|\texttt{group}|)$.

Now here is the maintenance outline for the alternative approach, where we are incrementalizing the filtered comprehension that depends on $dF_{\texttt{fol}}$, dM_1, etc., instead of $F_{\texttt{fol}}$, M, etc.

```
    add to U
```
after incrementalizing tags and filters, in dependency order:

```
    add to U
    maint T_group for U
      maint dM_2 for T_group
        maint T_user2 for dM_2
          maint dF_email for T_user2
          maint dF_loc for T_user2
    maint T_celeb for U
      maint dF_fol for T_celeb
        maint T_celeb_fol for dF_fol
          maint dM_1 for T_celeb_fol
            maint T_user1 for dM_1
              maint dF_email for T_user1
              maint dF_loc for T_user1
```

after incrementalizing the filtered query:

```
    add to U
(A) maint filtered query for U
    maint T_group for U
      maint dM_2 for T_group
(B)     maint filtered query for dM_2
        maint T_user2 for dM_2
          maint dF_email for T_user2
(C)         maint filtered query for dF_email
          maint dF_loc for T_user2
(D)         maint filtered query for dF_loc
    maint T_celeb for U
      maint dF_fol for T_celeb
(E)     maint filtered query for dF_fol
        maint T_celeb_fol for dF_fol
          maint dM_1 for T_celeb_fol
(F)         maint filtered query for dM_1
            maint T_user1 for dM_1
              maint dF_email for T_user1
(G)             maint filtered query for dF_email
              maint dF_loc for T_user1
(H)             maint filtered query for dF_loc
```

There are now eight different pieces of maintenance for the filtered query (marked A–H), each of which expands to a dozen or so lines of maintenance code. The maintenance for the tags and filters has not changed, so it is clear that this approach leads to a significantly greater code size for the generated program.

Moreover, each of the eight pieces of query maintenance requires a choice of a join order, and it is harder to choose these orders in a way that delivers an overall asymptotic cost that is as good as the normal method. In particular, the addition to U indirectly incurs maintenance of the query for an addition to $dF_{\texttt{loc}}$ via $T_{\texttt{user}_2}$ (line D). Given that this line runs once for each user that has just been added to $T_{\texttt{user}_2}$, its cumulative contribution to the cost can be as great as $\Theta(|\texttt{group}| \times |dM_{1\texttt{in}}| \times |U_{\texttt{out}}|)$. This pathological case is realized when a newly demanded group contains many users that are not already in $T_{\texttt{user}_2}$ but that each folow many demanded celebrities.

A simple workaround would be to use the opposite join order for line D: When a new pair is added to $dF_{\texttt{loc}}$ due to the user gaining tag $T_{\texttt{user}_2}$, maintain the query using a lookup in $dM_{2\texttt{in}}$, which is guaranteed to return just the one newly demanded group. But notice that we have the symmetric problem on line (H). Thus, the only way for the alternative method to match the overall asymptotic cost bounds of the normal method is to use different join orders for different occurrences of the same kind of maintenance code. Conversely, the alternative method can never produce a cost bound that is better than what the normal method can produce, because any join order that we may use for line (A) in this code is also available for the same maintenance join appearing in the normal method's code.

4.3 Comparison to Object-Set Query (OSQ)

Of all the techniques for computing object queries efficiently, the Object-Set Query (OSQ) method [74, 72] bears the most similarity to our own. In fact, our method grew out of an attempt to characterize OSQ's behavior using invariants. Both methods compute queries in an incremental and demand-driven way, and both handle complex nesting and aliasing by reducing the query down to a relational form. The two main differences are that OSQ is not able to use demand strategies that are as precise and flexible as ours, and OSQ's behavior is not defined by invariants. This in turn means that it is not able to provide cost guarantees.

We first discuss the conceptual differences between the two systems with regard to demand and invariants in particular. We then show benchmarks demonstrating how our method can obtain better time performance, including by an asymptotic factor. For these experiments we have ported OSQ's implementation to Python 3.4. The first two benchmarks are reimplementations of OSQ experiments that showed the asymptotic advantage of incremental computation over straightforward programs; we achieve this same asymptotic

speedup and also a modest constant factor improvement over OSQ. Along the way, we show how a simplification to the second benchmark's query enables much better performance from both systems for the case where few parameter values are demanded. The final benchmark uses our running example to demonstrate a case where OSQ's demand strategy is asymptotically worse. Transformation statistics for programs created by our method are given in Table 4.1.

Demand strategy: OSQ's equivalent of our demand strategy is a forest over the query variables, rooted at the parameters. Edges go from variables representing sets and objects to variables representing their elements and field retrievals, respectively. The fact that the graph is a forest as opposed to a DAG means that demand propagates by only one path from a parameter variable, instead of by an intersection of paths; see the Single Tag strategy.

Once a value becomes tracked for demand purposes, it stays tracked, even if it later becomes inaccessible from the demanded parameter values. Maintenance code for it will continue to execute at its updates (with no effect besides consuming resources), and it will continue to propagate demand to its elements or field values, until the value is garbage collected; see the Monotonic Tags strategy.

At most one auxiliary map is used for each relation in the flattened comprehension, even if it has multiple occurrences. The auxiliary map contains all map entries that are needed for each occurrence; see the Shared Filters strategy. A consequence of this is that Theorem 2 does not necessarily hold for OSQ, and so type check elimination and other optimizations may not be safe.

Use of invariants: OSQ organizes the inserted maintenance code using ad hoc rules. That is, the relative order in which it runs code for maintaining the query, code for maintaining auxiliary maps, and code for propagating demand, is specified explicitly as part of the method. Because these details need to be considered when reasoning about the method's correctness and the correctness of any extensions, they place a constraint on how sophisticated the design can reasonably be. In addition, to understand what values will contribute toward the cost of incremental maintenance, we need to understand the operational behavior of the generated code.

This is most apparent in OSQ's handling of demand propagation. While it would be possible to extend OSQ to follow a demand strategy similar to ours (i.e., without behaving like the Single Tag, Monotonic Tags, and Shared Filters

strategies), this is a non-trivial change that would require care to implement and verify. In contrast, since our method reduces the problem of modeling demand down to the problem of computing multiple relational comprehensions, no additional thought is needed once the invariants are decided. Even the order for running different kinds of maintenance code follows naturally from the invariant dependencies and rules of finite differencing. We can easily adopt a new demand strategy by changing invariants, without having to develop and justify new ad hoc rules. In fact, for any query handled by OSQ (without its extensions), there exists a suitable choice of the demand strategy and join orders such that, with the three OSQ demand-strategy modifications above, we can simulate OSQ's asymptotic performance.

Invariants also allow us to reason about the values that are tracked by auxiliary data structures, and therefore about the time and space costs of incremental computation, without requiring an operational understanding of generated code beyond the choice of join orders. In particular, all tags and filters are constrained to hold only values that are relevant for their respective variables. OSQ's lack of invariants leads to uncertainty about the correctness of using the method on nested queries (see Appendix A), and of using static optimizations like type check elimination.

Other differences: We address a number of details that were not explored in OSQ, including propagation of demand to nested queries, aggregates nested inside comprehensions, and the semantics of code that can raise expressions and have type errors. We add extensions for negated membership clauses and more expressive patterns in queries, as well as many optimizations. Our handling of relational comprehensions with self-joins avoids using asymptotically significant amounts of temporary space, by not materializing the union of maintenance joins.

Wifi query: The first benchmark is a simple selection/projection query [72].

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

For our system, we have added annotations to the program indicating that the demanded data is well-typed (so type checks can be eliminated) and that counting elimination is safe to perform (which is true so long as `ssid` is a key for `ap` objects).

The benchmark starts by creating a single `wifi` object and demanding it before the timing loop (this also gets OSQ's startup penalty out of the way). It then measures the time to do x many loop iterations, where each iteration

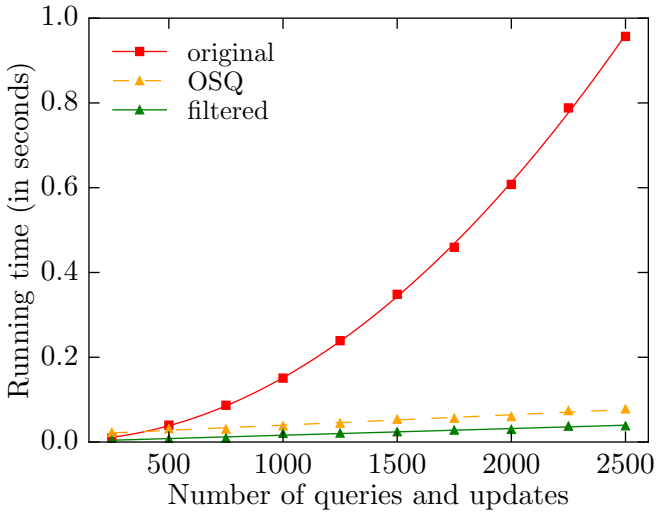


Figure 4.1: Wifi query. Each datapoint is the average of exactly 20 trials, as was done in the original experiment.

consists of creating a new `ap` object, adding it to `wifi.scan`, and performing the query. All `ap` objects satisfy the signal strength condition. No copy of the query result is created.

Analytically, we expect incremental approaches to take constant time for each query and update operation, while straightforward evaluation would cost $O(|\texttt{wifi.scan}|)$ per query operation. This translates to an overall cost that is quadratic for `original` and linear for `osq` and `filtered`. Figure 4.1 confirms these predictions and is consistent with the original results for this benchmark (Figure 4.4 of [72]).

Authorization query: The second benchmark is an authorization query [74, 72], loosely based on a query found in the Django web framework.

```
{p.name : u in users, g in u.groups, p in g.perms,
          u.id == uid, g.active}
```

We have added an annotation to the program, indicating that the demanded data is well-typed. To match OSQ's behavior on this query, the `uid` parameter is not made a demand parameter. Furthermore, our system recognizes that the `users` set acts as a relation in our benchmark program, so this set is not flattened. This means that there are no demand parameters and the demand clause can be eliminated; demand filtering starts with the membership over `users` instead of a membership over U.

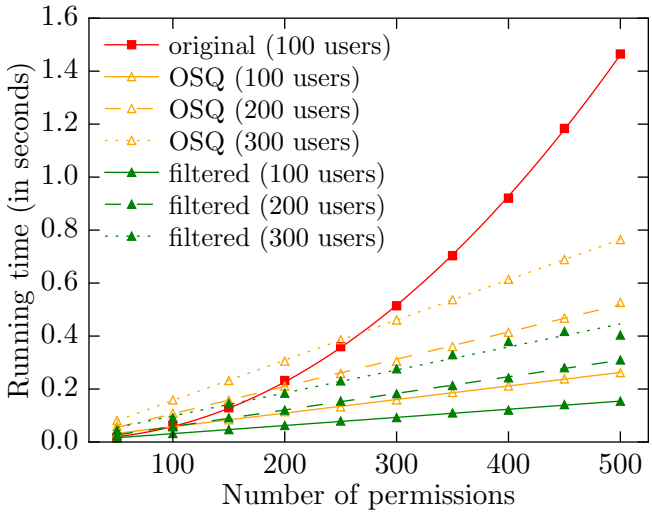


Figure 4.2: Authorization query. Each datapoint is the average of exactly 50 trials, as was done in the original experiment. We were unable to reproduce the same running times for the original program and for the OSQ system as were reported in Figure 5 of [74], but the asymptotic behavior is the same.

The benchmark initializes the `users` set with 100, 200, or 300 users, all of which share a single group (having the `active` flag set), and creates x many permissions. As before, it demands the `users` set (for OSQ's benefit; for our method it's always demanded) before starting the timing loop. It then measures the total time for alternately adding a permission to the group and performing a query over a random user id, until all permissions have been added. No copy of the query result is created. Since the number of users and groups is fixed, the asymptotic analysis for query and update operations is identical to that of the wifi benchmark. Figure 4.2 confirms this, and is qualitatively the same as the original results (Figure 5 of [74]), while showing a constant factor improvement for our system.

A curious property of this query is that it passes in the entire `users` set as a parameter, even though only information about a single user is being requested. This causes both our method and the OSQ method to track all users in the `users` set as relevant, and likewise for the groups and permissions of these users. Each permission addition update causes maintenance code to iterate over all users, regardless of how many users are actually passed as parameter values to the quey.

We can address this problem by simplifying the query so that it takes in a single user rather than a set of users and a user id.

```
{p.name : g in user.groups, p in g.perms, g.active}
```

To show the difference between the simplified and original queries, we run a modified version of the benchmark where the number of users and permissions is fixed at 300 and 500 respectively, while the number of users that can be randomly selected for querying ranges from 1 to 300. These users are demanded prior to the timing loop. We have used our method to generate, for each query, an incremental program without demand filtering. This program acts as a baseline for comparison since its performance is not affected by how many users are demanded.

Figure 4.3 shows the resulting times for both methods running on both queries, as a fraction of the time taken by the corresponding baseline program. We can see that for the original query, neither implementation is able to take advantage of having fewer demanded users. Meanwhile, for the simplified query, both implementations are more than ten times faster when there are very few demanded users compared to when all users are demanded. (Optimizations like type check elimination allow the demand-filtered version to remain better than the baseline program even when all users are demanded.) This example underscores the importance of writing queries that are straightforward and that avoid touching unneeded data, in order to get the most out of automatic systems like ours and OSQ.

Demand strategy precision: The final benchmark uses our running example to demonstrate a case where OSQ is asymptotically worse than our system due to differences in demand strategy. Unfortunately, due to a bug in OSQ, we could not initially run it on this example. A subsequent fix allowed us to run it, but the performance results were inconsistent, with outliers 3–15x higher than the highest point in the trend line, although the output was correct. We therefore instead chose to use a program generated by our method as a stand-in. This program is incremental and uses demand-filtering, but with the Single Tag strategy, so that no $T_{\texttt{user}_2}$ tag is generated. The performance of the actual OSQ system, although inconsistent, was observed to be no better than our stand-in, even asymptotically.

The benchmark procedure is similar to the one used for Figure 3.1, except that every user is demanded (instead of only a small fixed number of users), and each user belongs to exactly 5 groups (instead of a proportion of all the groups). Under this setup, on average, each group will have 500 users, since there are 100 times as many users as groups. This means that every user will be in $T_{\texttt{user}_1}$, but only about 500 users will be in $T_{\texttt{user}_2}$, if it exists.

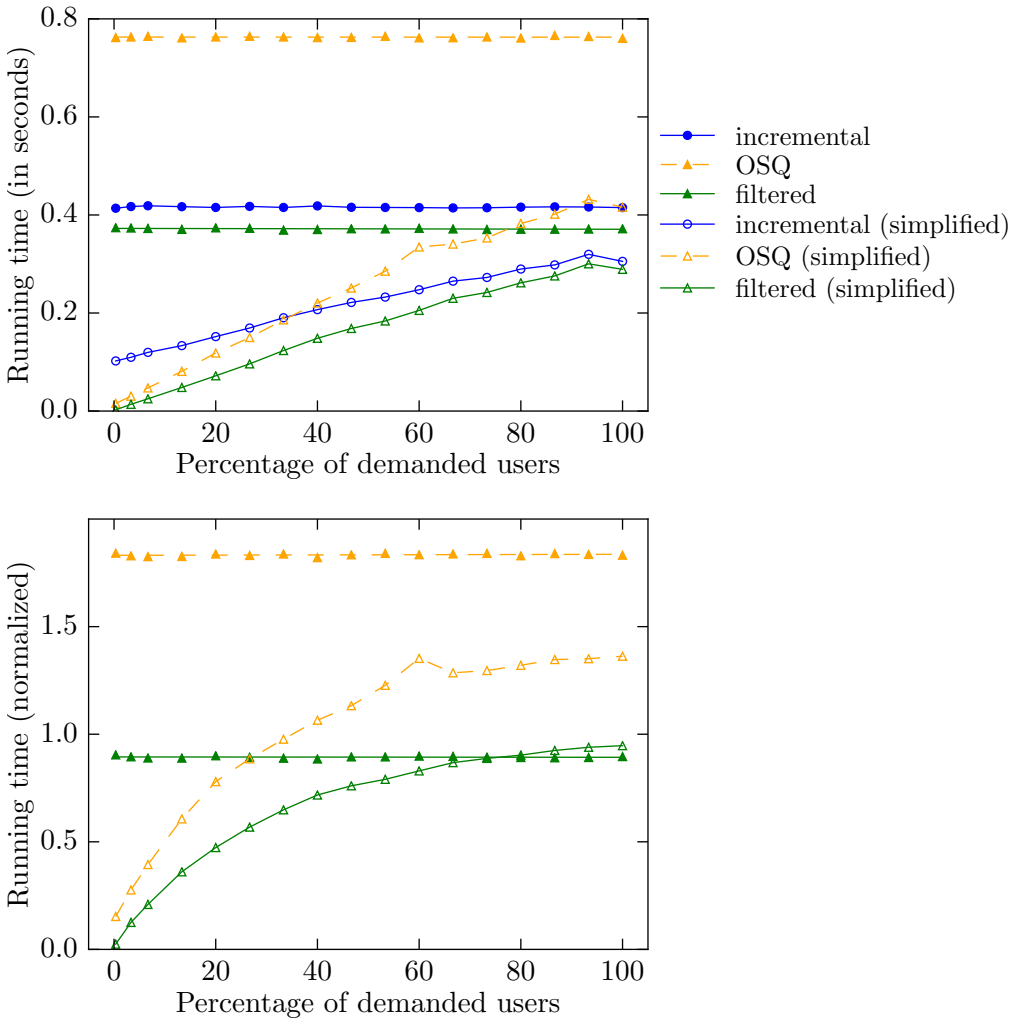


Figure 4.3: Performance of OSQ and our method, on both the original authorization query and its simplified version. In the second figure, each series is normalized relative to how long the baseline incremental program takes for its respective query. Each datapoint is the average of exactly 50 trials.

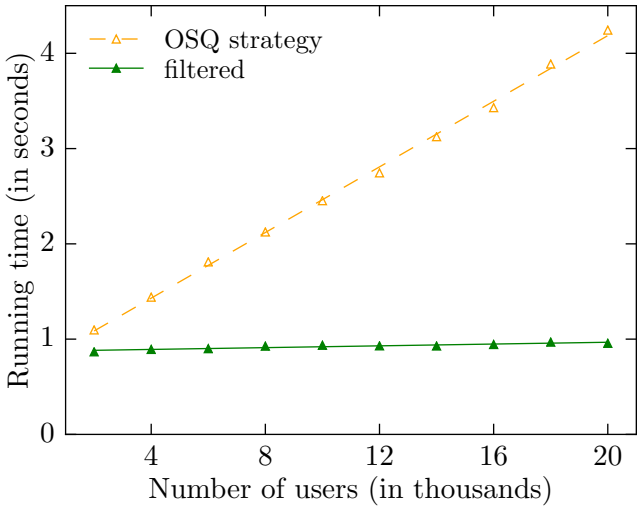


Figure 4.4: Standard demand invariants compared to an OSQ-style demand strategy using the Single Tag strategy.

The maintenance code for updating a user's location begins with a membership test over $dF_{\texttt{loc}}$, which succeeds only if the user is in all applicable tag sets. If it does succeed, then the subsequent code iterates over all celebrities that the user follows. For the program using the Single Tag strategy, this test will always succeed, so the overall cost grows proportionally with the number of users in the system. In contrast, for the program using our standard demand strategy, the increased cost is canceled out by the proportionally decreasing probability that an updated user is in the second tag set. Thus, the cost is constant instead of linear. This is confirmed in Figure 4.4.

4.4 Comparison to the Java Query Language (JQL)

The Java Query Language [84, 85] is an extension to Java that adds a declarative comprehension expression. This expression is a bit more limited than our object-set comprehensions: It does not support projection or pattern matching, and it does not allow multiple levels of nested sets and objects when the query is to be computed incrementally. (It does however support sequences, which we do not handle.) Maintenance for removal updates is performed by iterating over the entire result set and deleting entries that depended on the removed element.

The JQL system provides three strategies for computing a query: batch computation, caching (i.e., incremental computation), and a hybrid approach that switches between the two when the query/update ratio crosses a threshold. In all cases, JQL relies on hash joins, which require rescanning the entire contents of a relation each time it is joined.[8] This contrasts with our approach, which uses incrementally maintained auxiliary maps, implemented as hash maps in Python.

In order to empirically compare our system's performance with JQL, we reimplemented their cache incrementalization benchmark (Section 4.3 of [85]), which examines the benefit of incremental computation as the query-update ratio changes. Expressed in our notation, the three tested queries are

(1)
```
{a : a in attends, a.course == COMP101}
```

(2)
```
{(a, s) : a in attends, s in students,
          a.course == COMP101, a.student == s}
```

(3)
```
{(a, s, c) : a in attends, s in students, c in courses,
             a.course == COMP101, a.student == s, a.course == c}
```

We used JQL version 0.3.3, along with Java 1.6.0_24, AspectJ 1.7.2, and ANTLR 3.0.1. We did not disable the Java garbage collector, as Java does not provide an API for doing so; nonetheless, when we separately ran our system with the Python garbage collector enabled, we observed little effect on the running time, with an average variation of about 7%.

The implementations we compare are JQL with no caching (`JQL no caching`) and with always caching (`JQL always caching`), and the three standard strategies we used in our running example benchmark (Chapter 3): `original`, `incremental`, and `filtered`. For `original` we used the most straightforward translation of the queries to Python code, which preserves the order of the join conditions even though they could be optimized by running the condition clauses sooner (i.e., pushing selections inside joins). For `incremental` and `filtered`, the system recognizes the three top-level sets as relations that do not need flattening, so the only parameter tracked by the U set is `COMP101`, which is given a constant value anyway. We add an annotation that the demanded data is well-typed so that type checks can be eliminated from `filtered`. Our system automatically recognizes that counting elimination and result set elimination also apply.

[8] However, later work has focused on giving JQL more sophisticated query planning and allowing it to cache intermediate results [62, 61].

Program	Orig. LOC	# queries	# updates	Inc. LOC	Inc. Time	Filt. LOC	Filt. Time
`Wifi`	30	1	6	173	0.83	310	1.62
`Django`	36	1	8	279	1.44	506	3.02
`Django, simplified`	32	1	6	217	1.17	438	2.48
`JQL 1`	41	1	3	73	0.33	104	0.47
`JQL 2`	43	1	5	124	0.55	190	0.84
`JQL 3`	47	1	6	164	0.70	249	1.11

Table 4.1: Transformation statistics for programs used in this chapter. OSQ and JQL implementations are not included; OSQ dynamically generates its code, while JQL's compilation adds a few dozen lines to invoke its library with the query's parsed structure. The OSQ benchmark on the social network program is listed in Table 3.1.

The benchmark initializes the three sets with 1,000 objects of their respective types, where each object in `attends` relates a random student to a random course. The total time to perform 5,000 operations is measured, where each operation is either (1) executing the query that is being benchmarked, or else (2) a pair of updates to `attends` that first adds a newly constructed random object and then removes a random object. Our method's automated cost analysis indicates that these operations take constant time for `incremental` and `filtered`. This procedure is repeated for different query-update ratios.

The results in Figure 4.5 confirm our intuition that the incremental approaches should become more efficient as the query-update ratio increases, while the non-incremental approaches should become more expensive. `filtered` has no advantage over `incremental` since all values are demanded, but its overhead is mitigated by the savings of type check elimination. As we proceed from Query 1 to Query 3, `incremental` and `filtered` are essentially unchanged, while `original` and the JQL strategies take noticeably more time.

The extent to which our incremental implementations ran faster than JQL with caching warranted further investigation. We therefore performed a new benchmark that varies the size of the sets while holding the query-update ratio at about 0.5. The results are shown in Figure 4.6. As predicted by cost analysis, the size of the sets does not affect the time for our incremental strategies. The other approaches are asymptotically worse, consistent with JQL's hash joins taking an increasing amount of time to build their indices at query or update operations.

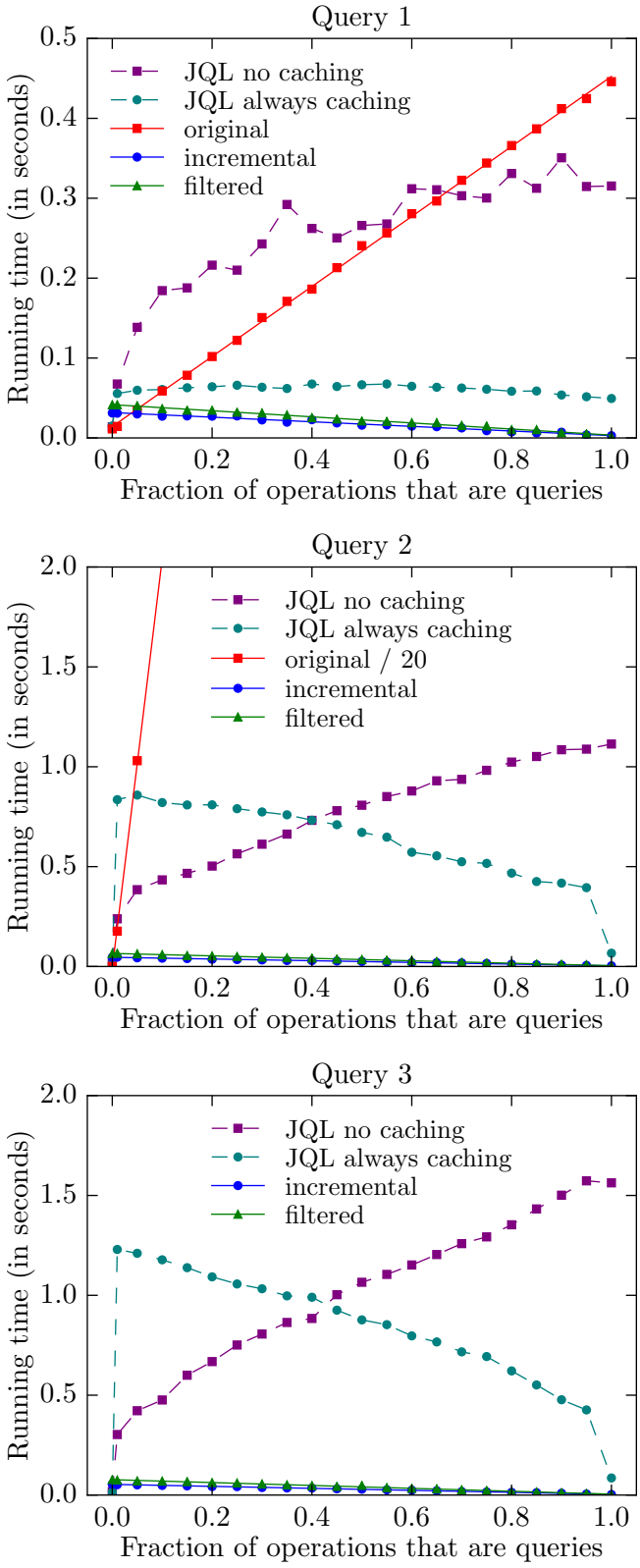


Figure 4.5: JQL cache incrementalization benchmark (c.f. Figure 4 of [85]). Each datapoint is the average of exactly 50 trials, as was done in the original experiment. `original` quickly times out (> 60s) for queries 2 and 3.

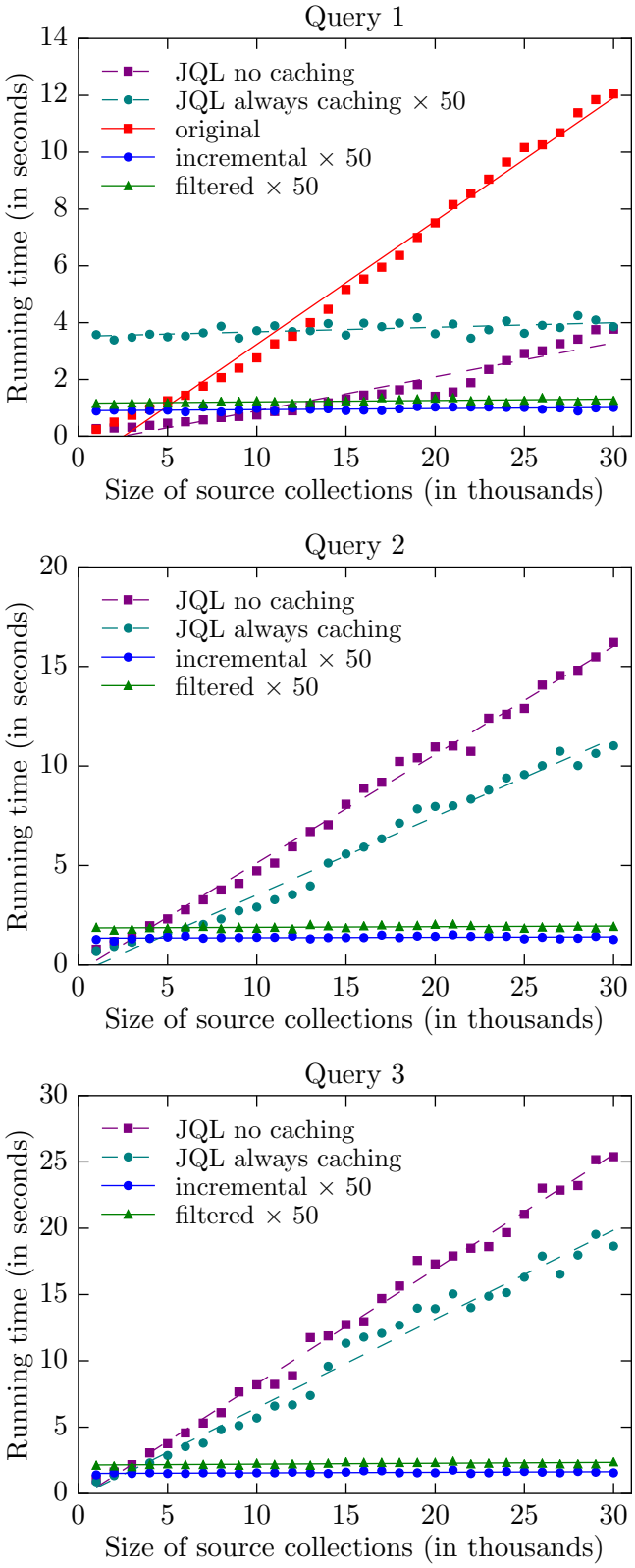


Figure 4.6: JQL scalability. The first two plots use exactly 50 trials per datapoint, and the third uses exactly 15 trials per datapoint. `original` times out (> 60s) immediately in the second and third queries. A linear fit is drawn in each case, though the exact asymptotic growth is inconclusive. Note the scaling in the legends.

Chapter 5

Applications

In this chapter we gauge the effectiveness of our method on applications that model real-world systems. The applications are drawn from diverse domains including access control, a student information system, approximate probabilistic inference, and distributed algorithms. The time performance of the programs generated with our system asymptotically improve over the straightforward implementations. For programs like Core RBAC and Lamport's distributed mutual exclusion, our system can match the asymptotic performance of previous implementations obtained by using handwritten transformation rules. This is significant because hand-written rules require effort on the part of the programmer and can be erroneous. The transformation statistics for these applications help give an idea of the effort that would be involved in producing this code without a systematic method.

5.1 Role-Based Access Control (RBAC)

The term Role-Based Access Control describes a system where the capabilities of users are determined by what roles they are assigned. We first discussed this application in Chapter 1 to illustrate various implementation strategies for a simple query, and then again in Chapter 2 as an example of a relational comprehension. ANSI RBAC [7] is a standard defining specific query and update operations. We consider two out of its four components: Core RBAC and RBAC with Static Separation of Duties. Transformation statistics for the programs in this section are given in Table 5.1.

Core RBAC defines the basic model: Users have roles, roles have permissions (pairs of operations and objects), and users may create sessions that utilize a subset of the user's roles. The `CheckAccess` query takes in a session

and a permission and determines whether the session has any activated role that could authorize the permission:

```
{r : r in ROLES, (session, r) in SR, ((operation, object), r) in PR}.
```

With some suitable preprocessing to treat the set parameters as relations, this query can be transformed as a relational comprehension.

Previous work has developed an executable Python implementation of the Core RBAC specification [48].[1] This implementation was designed specifically to be concise and use high-level queries without regard for performance. Subsequent work generated, analyzed, and benchmarked efficient implementations that were produced using handwritten incrementalization rules [52].

We have used our system to generate similar implementations automatically. Four implementations were produced: two that incrementalize all 16 queries with and without demand filtering (`incremental (all)`, `filtered (all)`), and two that incrementalize just `CheckAccess` and two helpers involved in creating and deleting sessions, with and without demand filtering (`incremental (CA)`, `filtered (CA)`). The ones without demand filtering are analogous to implementations that were generated in [52]. The statistics for all four versions are shown in Table 5.1.

In the generation of all four programs, we are able to skip the Flatten step because the queries contain no object field retrievals or nested sets, and because the queries and updates do not use aliasing. We are also able to skip the Restrict step because, after the preprocessing described in [52], every parameter appears on the left-hand side of a membership. For the two demand-filtered versions, in the case of the `CheckAccess` query, we have specifically directed our system to use a demand set that holds values of `object`, and to use a demand strategy that filters `PR` first. This better models how we may want to only track authorization information for certain objects, and also means that the filtered image-set lookup d`SR.out{session}` returns just those roles that grant a permission involving one of the demanded objects. The remaining queries are demand-filtered with no demand set, using our system's default approach of a demand strategy that works left-to-right on the query's memberships.

We have performed an experiment that is similar to the one used in Figure 6 from [52], which measures the time to perform `CheckAccess` queries and to create and delete sessions, as the total number of roles increases.[2] The

[1] `http://www.cs.stonybrook.edu/~liu/rbac/coreRBAC.py`

[2] We were unable to reproduce their exact quantitative results due to discrepancies in the benchmark setup, even when incrementalizing all queries like they did; our asymptotic results are still the same.

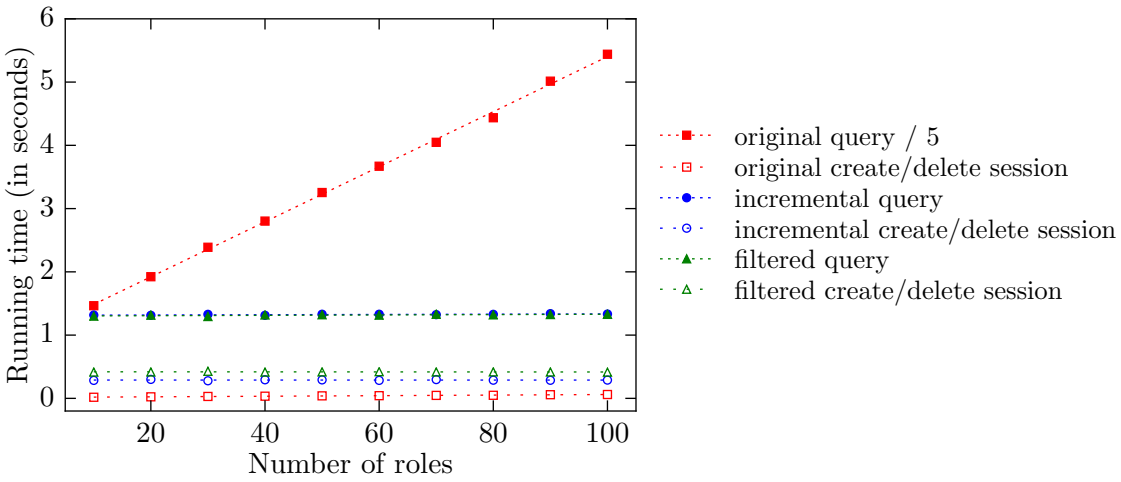


Figure 5.1: Total time for performing `CheckAccess` and session creation and deletion, varying the total number of roles, but holding the number of active roles per session and permissions per role fixed.

benchmark performs 1,000 repeats of an operation pattern consisting of: (1) creating a session with 10 active roles and 10 permissions per role, (2) performing 1,000 `CheckAccess` queries over the session and a random permission, and (3) deleting the session. For demand-filtered `CheckAccess`, all the objects that will be queried are entered into the demand set before the timing loop. As in Figure 3.1, time for creating and deleting the session is measured by performing a run with no queries, and time for querying is obtained by subtracting this from the total time.

Analytically, we know that both the session-modifying operations and the query operation take linear time in the total number of roles for the original implementation, while they both take constant time in the total number of roles for all transformed implementations. This is confirmed for `incremental (CA)` and `filtered (CA)` by Figure 5.1. Results for `incremental (all)` and `filtered (all)` are not shown, but are similar with just a larger constant factor. Even though querying takes longer than updating for the transformed implementations (due to the sheer volume of queries performed), it is still beneficial when compared to the cost of the original program.

Filtering does not improve performance because all objects are demanded. We have therefore run another version of this benchmark where we vary the number of demanded objects from 1 to 1,000, and fix the number of roles and the number of permissions per role at 100. Each random query oper-

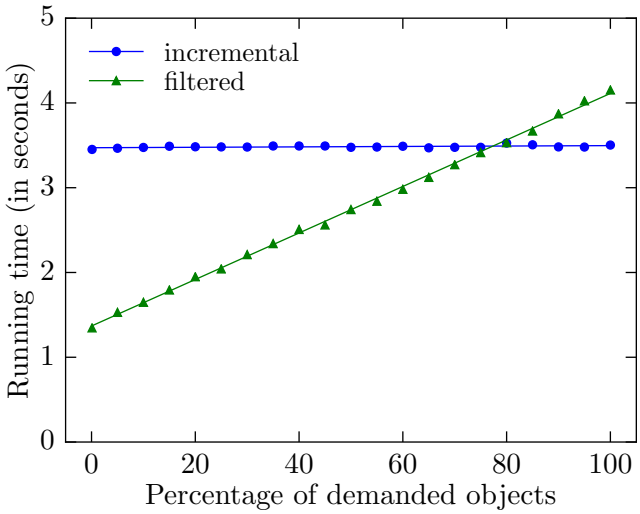


Figure 5.2: Total time for `incremental (CA)` and `filtered (CA)` to perform `CheckAccess` operations and to create and destroy sessions. First datapoint at $x = 1$.

ation is constrained to be over a demanded object. Figure 5.2 shows that, as we would expect, `filtered`'s advantage is strongest when few objects are demanded, with the break-even point being when around 80% of the objects are demanded.

The second RBAC component we deal with provides static separation-of-duty (SSD) constraints. Each constraint consists of a set of roles and a cardinality limit, and requires that every user have fewer than the limit many roles from its set. The constraints are static in the sense that they apply to user-role assignments rather than session-role assignments.

We created a simple test program to simulate just this query and its updates. Following [48], we model constraints with two sets: `SsdNR` holds pairs of a constraint set name and a role in the set, and `SsdNC` holds pairs of a constraint set name and its cardinality limit. The query can naturally be expressed by using aggregate and comprehension subqueries:

```
{(u, name) : u in USERS, (name, c) in SsdNC,
     count({r : (u, r) in UR, (name, r) in SsdNR}) >= c}
```

It returns the set of all violations of the constraints. With appropriate optimization, this query can be treated as a relational comprehension just like the previous query.

Program	LOC	Time
`RBAC, Original`	141	N/A
`RBAC, Incremental (CheckAccess)`	353	1.69
`RBAC, Filtered (CheckAccess)`	697	3.42
`RBAC, Incremental (all)`	1410	6.84
`RBAC, Filtered (all)`	3554	29.95
`Constrained RBAC, Original`	34	N/A
`Constrained RBAC, Auxmap`	67	0.47
`Constrained RBAC, Incremental`	313	1.27
`Constrained RBAC, Filtered`	526	2.19

Table 5.1: Transformation statistics for RBAC programs.

When we compared the straightforward implementation (`original`) with our standard incremental approach without demand filtering (`incremental`), we found that `original` scales exceptionally poorly due to its inefficient handling of joins as nested loops. We therefore built a middle-of-the-road approach (`auxiliary maps`) that computes the query result at the time the result is requested, just like `original`, but does so using the same strategy used to compute maintenance joins — i.e., the clauses are ordered according to the join order heuristic, and then turned into code that retrieves from incrementally maintained auxiliary maps.

The benchmark alternately grants roles to users and performs the query, until all users have all roles. It creates n many users and constraints. There are just 5 total roles. Each constraint includes all roles in its set and has a cardinality limit of 3. Cost analysis is as follows:

Implementation	Querying	Adding a role
`original`	$O(\lvert\texttt{USERS}\rvert \times \lvert\texttt{SsdNC}\rvert \times \lvert\texttt{UR}\rvert \times \lvert\texttt{SsdNR}\rvert)$	$O(1)$
`auxiliary maps`	$O(\lvert\texttt{USERS}\rvert \times \lvert\texttt{SsdNC}\rvert \times \lvert\texttt{UR}_{\texttt{out}}\rvert)$	$O(1)$
`incremental`	$O(1)$	$O(\lvert\texttt{SsdNR}_{\texttt{in}}\rvert)$

Using the facts that the number of roles is fixed, the number of users and constraints is n, and the number of queries and updates is $O(n)$, we obtain total cost bounds of $O(n^5)$, $O(n^3)$, and $O(n^2)$ for `original`, `auxiliary maps`, and `incremental` respectively. These predictions are confirmed precisely in Figure 5.3. A demand filtered version was also benchmarked and observed to be a constant factor higher than `incremental`.

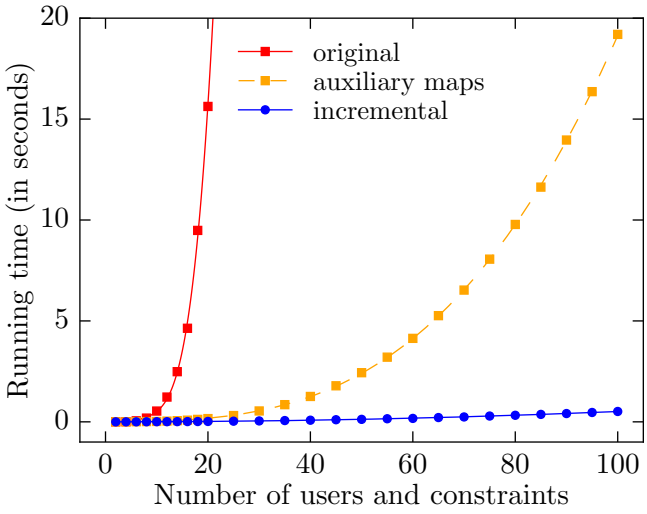


Figure 5.3: Time for three implementations of Constrained RBAC.

5.2 Student information management system

The OSQ method has been applied to a number of high-level queries for managing a graduate student database [74]. We have run our system on the same queries in order to compare the size of the generated code under both methods. Except for the New Students query, these queries are not executed or timed. In fact, we observed that a few of the queries violate our safety requirement by including negated memberships (which our implementation does not yet support) or calls to functions that access their arguments' field values. This means that the generated code does not handle all possible updates, but can still work if these updates do not occur during incremental computation.[3]

Table 5.2 shows the transformation time and generated code size under our system for the 10 queries that appear in [74]. We transform each query by placing it in a file by itself and applying our system, both with and without demand filtering enabled. The resulting code includes functions containing maintenance code for all kinds of updates, except those kinds that arise from a violation of the safety requirement. Although these generated functions are never called in the output program, their size is a good metric for comparing against OSQ's generated code because they serve the same purpose as OSQ's dynamically generated maintenance callback functions.

Comparing to OSQ's reported results (Table 3 of [74]) across the average

[3]The OSQ method would not be able to handle these updates either.

Program	Update kinds	Inc. Time	Inc. Lines	Inc. AST nodes	Filt. Time	Filt. Lines	Filt. AST nodes
`Current Students`	12	2.55	509	5049	5.05	851	7440
`New Students`	12	2.55	509	5067	5.08	851	7458
`TAs and Instructors`	18	4.19	652	6123	11.59	1194	9894
`New TA Emails`	10	1.45	422	3217	3.25	754	5496
`TA Waitlist`	10	1.69	480	3815	3.97	878	6555
`Good TAs`	8	1.06	278	2394	2.20	516	4041
`Qual Exam Results`	18	2.95	680	6091	5.98	1069	8840
`Advisors by Student`	14	2.72	576	5507	5.48	945	8088
`Advisor Overdue`	12	2.55	508	4953	5.06	850	7336
`Prelim Exam Overdue`	10	1.38	350	3037	2.89	638	5024

Table 5.2: Transformation statistics for the 10 student information system queries appearing in [74]. Update kinds is computed by taking the number of unique relations each query depends on besides the demand set, and multiplying by two since each relation can have additions and removals; the resulting counts approximate but do not equal the counts of updates listed in [74]. Lines and number of AST nodes is for the transformed output. Dead code elimination for unused maintenance functions was skipped since it would remove much of the generated code.

of these 10 queries, without filtering, we generate about the same number of lines of code and 30% more AST nodes. With filtering its about 70% more lines of code and twice as many AST nodes. Our transformation time is larger by an order of magnitude, but that is only a few seconds for these queries, and the cost is paid at compile time whereas OSQ may pay it as part of a startup phase at runtime. The larger code size can be explained in part by the inclusion of type checks, the computation of a more powerful demand strategy (see Chapter 4), and our implementation's lack of intraprocedural analysis for eliminating unnecessary temporary variables and tuple packing/unpacking operations.

We have also reproduced an OSQ benchmark of the New Students query (Figure 6 of [74]), which reads as follows.

```
{s : s in students, p in s.programs,
     s.joined == sem or (p.start != None and p.start == sem)}
```

This benchmark was performed by embedding the query in an executable program as normal, unlike the above transformations that were performed on unexecutable skeleton programs. The `students` set is optimized as a relation since our test program never reassigns this variable. The benchmark varies

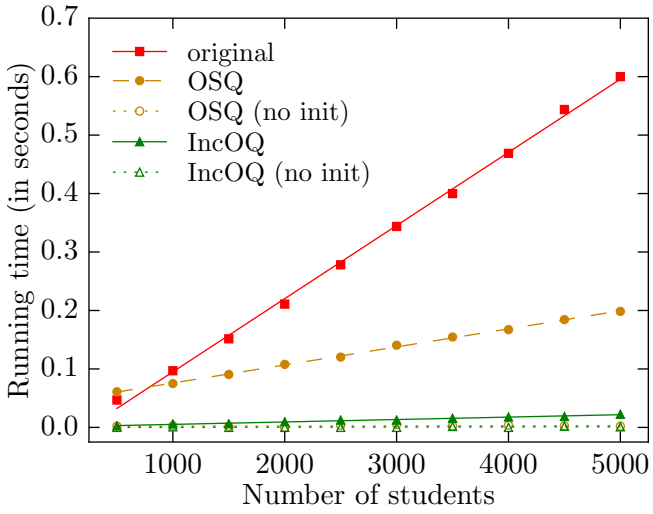


Figure 5.4: Time to perform the New Students query for OSQ and our system, with and without including time for demanding the initial query result.

the number of students, and measures the total time to perform 100 iterations of updating the `start` field of a program belonging to a random student, interleaved with performing the query.

The results are shown in Figure 5.4, both with and without including the time needed to demand the parameter values the first time that the query is run. Both OSQ and our system take linear time for that first query, but all subsequent queries are constant time. In addition, our system is faster by a constant factor.

5.3 Approximate probabilistic inference

We have applied our system to three queries for approximate probabilistic inference using Markov chain Monte Carlo (MCMC) sampling [8, 57]. Transformation statistics for each are shown in Table 5.3. We describe the first example from [8], Birthday Collision, at length since it is well-known and gives the intuition for how sampling may be used on other examples.

The birthday paradox states that when a room contains more than 23 people, there is a greater than 50% chance that at least two people share a birthday, assuming that there are exactly 365 days in the year and birthdays

Program	Orig. LOC	# queries	# updates	Inc. LOC	Inc. Time	Filt. LOC	Filt. Time
`Birthday`	22	4	5	254	1.09	376	1.80
`Publications`	16	2	4	63	0.28	100	0.42
`Citations`	22	2	7	152	0.81	318	1.61

Table 5.3: Transformation statistics for probabilistic queries.

are uniformly distributed. We can express this in our language as a query,

```
count({day : day in calendar,                                    ,
            count({person : person in room,
                            person.bday == day}) > threshold})
```

where parameters `calendar` and `room` are, respectively, the domain of all days of the year (the integers 1 to 365) and a set of objects with a `bday` field. The query returns the number of days that greater than `threshold` many people share as their birthday.

We have written a simple program embedding this query and some basic updates to add and remove people to the room. The program can be called as a library from a driver program to facilitate a sampling based simulation. The straightforward implementation performs the query in time proportional to the number of people in the room (since the number of calendar days is fixed), but when incrementalized, both querying and updating is constant-time. Both the `calendar` and `room` sets are optimized as relations, and since there are no image-set expressions over M or F_f relations, no demand filtering is necessary.

As an aside, the strategy used to compute this query incrementally is particularly intuitive and easy to explain to a non-expert. The straightforward approach is equivalent to asking every person in the room if they have a birthday on Januart 1st, then repeating for January 2nd and all other days in the year. The incremental approach is equivalent to keeping a calendar where every day holds the number of people whose birthday is on that day. When a person enters or leaves the room, the count for their birthday is incremented or decremented, respectively. Whenever the count for a particular day crosses the threshold in either direction, the count of all days satisfying the threshold is itself incremented or decremented. It's easy to see that this requires only a few operations and is independent of the number of people in the room and the number of days in the year.

The driver program we have written estimates the probability of a birthday collision by first populating the room with a certain number of people, and then alternately replacing one person in the room at a time and sampling

Implementation	People	Threshold	Samples	Avg. Prob.	Spread	Avg. Time
`original`	23	1	5,000	0.490	0.076	4.87
`incremental`	23	1	100,000	0.510	0.014	1.54
`original`	1,000	8	500	0.418	0.742	16.20
`incremental`	1,000	8	100,000	0.536	0.070	1.52

Table 5.4: The average probability returned, the spread (difference) between the largest and smallest probabilities returned, and the average running time, over 10 trials each for four different configurations of the birthday paradox simulation.

the result of the query. Table 5.4 shows our results for both the original and incremental implementations. For the original implementation we are forced to use fewer samples due to its asymptotically worse performance. This results in a larger spread in the estimated probabilities across different runs of the program. The effect is more severe when the number of people in the room is larger. In the case where 1,000 people are in the room, the incremental program can handle 200 times as many samples in one tenth of the running time. That extra sampling reduces the difference between the highest and lowest probabilities returned across ten trials by an order of magnitude.

We have transformed and benchmarked two other queries for this kind of sampling-based simulation, used in the Publication Citation Matching example from [8].

```
count({p : p in pubs, p.author == "Einstein"})
count({c : c in citations, c.source.author == "Einstein",
           c.target.author == "Galileo"})
```

The benchmarks confirm that as publications and citations are added and removed, the incremental maintenance is constant time per update, whereas the original query time is linear in the total number of publications or citations.

5.4 Distributed algorithms

Distributed algorithms are especially difficult to design and implement correctly. Recent work has focused on expressing such algorithms in a new succint language called DistAlgo [51]. The language uses high-level queries such as comprehensions and quantifiers in order to avoid specifying unnecessary detail. We have applied our system to the DistAlgo programs listed in Table 5.5, obtaining asymptotic improvement in several cases. We describe the La mutex example in detail because it is easy to understand, it has multiple implementations to run our system on, and it is the running example of [51].

Program	Description
La mutex	Lamport's distributed mutual exclusion [39]
RA mutex	Ricart-Agrawala's distributed mutual exclusion [71]
RA token*	Ricart-Agrawala's token-based mutual exclusion [70]
SK token*	Suzuki-Kasami's token-based mutual exclusion [77]
CR leader	Chang-Rober's leader election [17]
HS leader	Hirschberg-Sinclair's leader election [32]
2P commit	Two-phase commit [22]
DS crash*	Dolev-Strong's consensus under crash failures [20]
La Paxos	Lamport's Paxos for distributed consensus [40, 41]
CL Paxos	Castro-Liskov's Paxos under Byzantine failures [14]

Program	n	`original`	`incremental`
CL Paxos	*procs*	$O(n^4)$ to $O(n^5)$	$O(n^3)$
RA mutex	*procs*	$O(n^3)$ to $O(n^4)$	$O(n^2)$
RA token	*procs*	$O(n^2)$ to $O(n^3)$	$O(n^2)$
RA token	*rounds*	$O(n^2)$ to $O(n^3)$	$O(n)$

Table 5.5: Transformed DistAlgo examples (top), and empirical asymptotic costs of examples besides La mutex for which an improvement was observed (bottom); all other examples had performance that was essentially unaffected by transformation. The terms *procs* and *rounds* are defined in the same way as they are for La mutex. The observed costs for `original` tended to fit between polynomials over the range of n we tried. The examples marked with asterisks required flattening and demand filtering, while the rest could be handled as relational queries with no demand filtering.

In all cases, our system takes as input a module generated by the DistAlgo compiler that contains functions wrapping the queries and updates of the original program, with quantification expressions turned into uses of aggregates. The characteristics of the input and transformed programs are given in Table 5.9. To benchmark the performance of distributed programs, we instrumented the code by adding a centralized controller process. This controller is responsible for measuring wall-clock time, collecting the total CPU time consumed by the individual processes, and ensuring that these times are not affected by unnecessarily creating and destroying processes while the benchmark is ongoing. CPU time may be larger than wall-clock time since our benchmark system is multicore.

Under Lamport's distributed mutual exclusion algorithm, when a process wants to enter the critical section it sends a request to all of its peers and collects their acknowledgements. The condition for entering the critical section is

that 1) this process must be next in line, ordered by logical clock, and 2) this process has received an acknowledgement of its request from each other process. DistAlgo will reevaluate this condition on every new incoming message by running a query. The asymptotic performance of the system is described in terms of the number of processes in the system (*procs*) and the number of requests per process (*rounds*).

We have transformed and benchmarked three different DistAlgo programs that implement Lamport's algorithm. The first (`lamutex orig`) corresponds closely to the original statement of the algorithm; it uses imperative operations to manage a queue of pending requests. The second (`lamutex spec`) is a more declarative program that queries over the received messages instead of explicitly constructing a queue. These queries are complex and nested, in part because they are translated from quantifier expressions. The third (`lamutex spec lam`) is similar to `lamutex spec`, but uses acknowledgement messages that contain the sender's current logical clock value instead of the clock corresponding to the request they are acknowledging. This makes the query do a little more work to correlate acknowledgements with requests, but the program is more consistent with the original description of the algorithm than `lamutex spec` is.

The demand sets for these queries generally involve the process's own id, which is a constant, and the clock value corresponding to the current request it is trying to make. Since the process can only have one outstanding request at a time, we generated the implementations such that the demand sets hold at most one entry, removing the previous entry when the first query under the new request is made. This avoids maintaining unneeded results for previous requests that have already completed.

We have run our automated cost analysis on the transformed examples and also performed a detailed analysis for `lamutex orig`, shown in Tables 5.6 and 5.7. The benchmarked CPU and wall-clock times for each implementation as the number of processes and rounds vary are shown in Figures 5.5, 5.6, and 5.7. The empirically observed asymptotic growth rates for these figures are summarized in Table 5.8. In each case the incremental program performs at least as well as the original program. Note that all implementations are lower-bounded by the number of messages exchanged, which is $\Theta(procs^2 \times rounds)$.

Table 5.6 shows the cost analysis for `lamutex orig`. The frequency is the number of times the operation occurs in total across all processes. For demanding, this is the number of times a request is made, while for all other operations it is proportional to the number of messages exchanged. The total cost contributed by each operation is the product of its frequency and its per-operation cost; the overall cost is the maximum of the total costs for each

Operation	Frequency	Per-operation cost	Total cost
Querying	$O(procs^2 \times rounds)$	$O(procs^2 \times rounds)$	$O(procs^4 \times rounds^2)$
Update queue	$O(procs^2 \times rounds)$	$O(1)$	$O(procs^2 \times rounds)$
Rec. Ack.	$O(procs^2 \times rounds)$	$O(1)$	$O(procs^2 \times rounds)$

Operation	Frequency	Per-operation cost	Total cost
Demanding	$O(procs \times rounds)$	$O(procs \times rounds)$	$O(procs^2 \times rounds^2)$
Querying	$O(procs^2 \times rounds)$	$O(1)$	$O(procs^2 \times rounds)$
Update queue	$O(procs^2 \times rounds)$	$O(1)$	$O(procs^2 \times rounds)$
Rec. Ack.	$O(procs^2 \times rounds)$	$O(1)$	$O(procs^2 \times rounds)$

Table 5.6: Cost analysis for `lamutex orig` for `original` (top) and `incremental` (bottom). When the original program performs the query, it does a nested iteration over each process and then each received acknowledgement, where the number of acknowledgements is $O(procs \times rounds)$. When the incremental program demands the clock value corresponding to a new request, it does a similar iteration, except that the set of received acknowledgements is indexed by process id so it only takes $O(rounds)$ time. All cost bounds are analytically tight in the worst case.

operation. For `original`, the per-operation costs were obtained by manual analysis, while for `incremental` it was our implementation's automatic analysis. This analysis relies on type information for the inputs to query and update operations; the fact that the demand set size is constant-bounded; and the knowledge that the number of possible logical clock values on any one host is proportional to the number of received messages, which is also proportional to the product of *procs* and *rounds*.

Although the total cost predictions of Table 5.6 are tight in the worst case, Figure 5.5 shows better performance in practice. We found that the discrepancy can be explained in part by the fact that the DistAlgo program arranges the queries using a short-circuiting `and` operator. Depending on the result of the first query, this sometimes prevents the second, more expensive query from running. When we removed short-circuiting and reran the benchmark, we found that `original`'s running time became about cubic in the number of processes and quadratic in the number of rounds.

When varying the number of rounds, `incremental` initially appeared to remain linear even after eliminating short-circuiting, but quadratic behavior became apparent when we ran a separate benchmark that increased the number of rounds up to 100,000. Without the cost analysis we may very well have missed this asymptotic behavior; thus this example demonstrates the need to use both empirical benchmarks and analytic bounds. In addition, quadratic

Operation	`lamutex spec`	`lamutex spec lam`
Demanding	$O(procs^2 \times rounds)$	$O(procs^2 \times rounds)$
Rec. Request	$O(1)$	$O(1)$
Rec. Release	$O(1)$	$O(1)$
Rec. Ack.	$O(1)$	$O(1)$

Table 5.7: Automatically generated asymptotic cost bounds for individual operations of the `incremental` implementations for `lamutex spec` and `lamutex spec lam`. Operation frequencies are as in Table 5.6, so the total cost bound for both implementations is $O(procs^3 \times rounds^2)$. As before, this analysis uses knowledge of the types of the input, the fact that the demand set size is $O(1)$, and the fact that the number of possible clock values is $O(procs \times rounds)$.

behavior is visible much sooner in a version of the transformed program where we do not cap the size of the demand sets.

It is possible to generate a version of the program whose total cost is the optimal $O(procs^2 \times rounds)$ by (1) demanding the new request's clock value as soon as the request is made (rather than at the time the first query is performed for that request), and (2) optimizing away maintenance code for adding to the demand set, using the knowledge that no received acknowledgement for that request can possibly exist at the instant the request is made. The latter can itself be inferred by using the general properties of logical clocks in DistAlgo programs.

The automated analysis and benchmarks for the other two variants of Lamport's mutual exclusion are given in Table 5.7 and Figures 5.6 and 5.7. The improvement is more pronounced for these versions than `lamutex orig` because of how the queries are written: They are higher-level and more declarative, but much more expensive to compute straightforwardly. As before, the incremental versions of these programs were observed to be quadratic in the number of rounds, when that number grows into the thousands.

Varying	Procs		Rounds	
Implementation	`original`	`incremental`	`original`	`incremental`
`lamutex orig`	$O(n^2)$	$O(n^2)$	$O(n)$ to $O(n^2)$*	$O(n)$*
w/o short circuiting	$O(n^3)$	$O(n^2)$	$O(n^2)$	$O(n)$*
`lamutex spec`	$O(n^3)$ to $O(n^4)$	$O(n^2)$	$O(n^2)$ to $O(n^3)$	$O(n)$*
`lamutex spec lam`	$O(n^2)$ to $O(n^3)$	$O(n^2)$	$O(n^2)$ to $O(n^3)$	$O(n)$*

Table 5.8: Empirically observed asymptotic costs for each Lamport mutex implementation, as per Figures 5.5, 5.6, and 5.7. The cost for `lamutex orig` without short-circuiting is also given. Costs are approximate and not necessarily worst-case bounds. In the case of the asterisked costs, further experimentation where we increased the number of rounds into the thousands (not shown) demonstrated quadratic behavior.

Program	Orig. LOC	# queries	# updates	Trans. LOC	Trans. Time
`clpaxos`	167	34	15	1670	15.70
`crleader`	28	4	2	92	0.42
`dscrash`	54	7	9	414	2.25
`hsleader`	67	10	8	253	1.39
`lamutex_orig`	55	6	9	391	1.81
`lamutex_spec`	45	8	3	554	2.66
`lamutex_spec_lam`	45	7	5	466	2.16
`lapaxos`	141	26	15	1267	9.97
`ramutex`	54	11	6	826	4.39
`ratoken`	85	13	10	1829	13.78
`sktoken`	109	12	15	1543	10.52
`tpcommit`	84	21	8	1180	7.45

Table 5.9: Transformation statistics for DistAlgo programs. Orig. LOC is for the file that DistAlgo generates as input to our system; it contains functions encapsulating queries and updates.

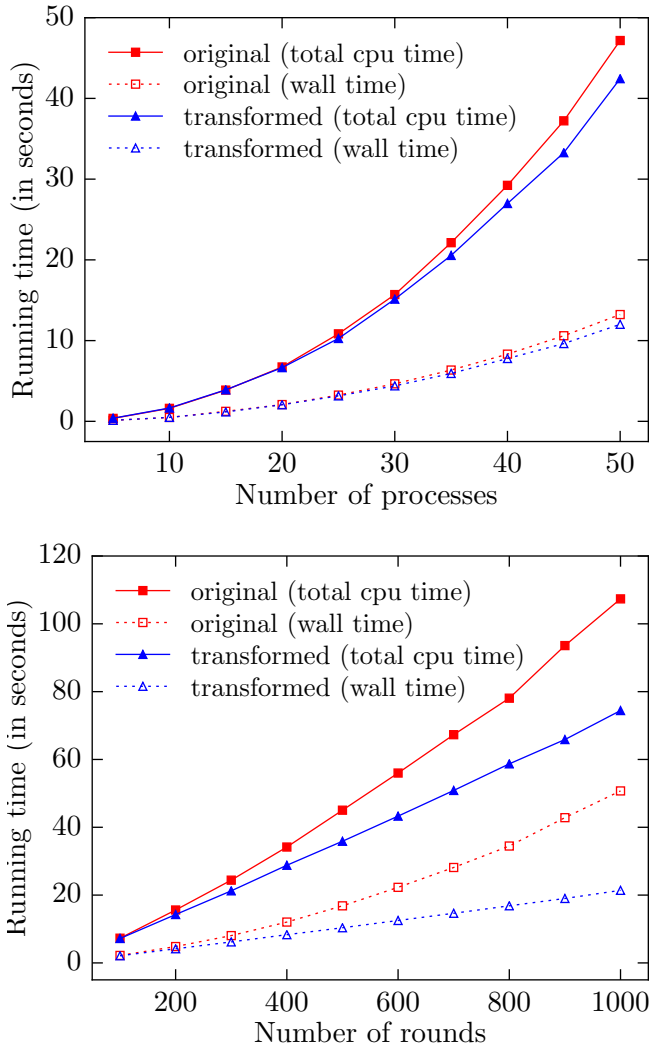


Figure 5.5: Lamport's mutual exclusion algorithm using an imperative message queue (`lamutex orig`), as the algorithm was originally described. We first vary the number of processes while holding the number of rounds at 5 (top). We then vary the number of rounds while the number of processes is held at 5 (bottom). Each datapoint is the average of exactly 5 trials.

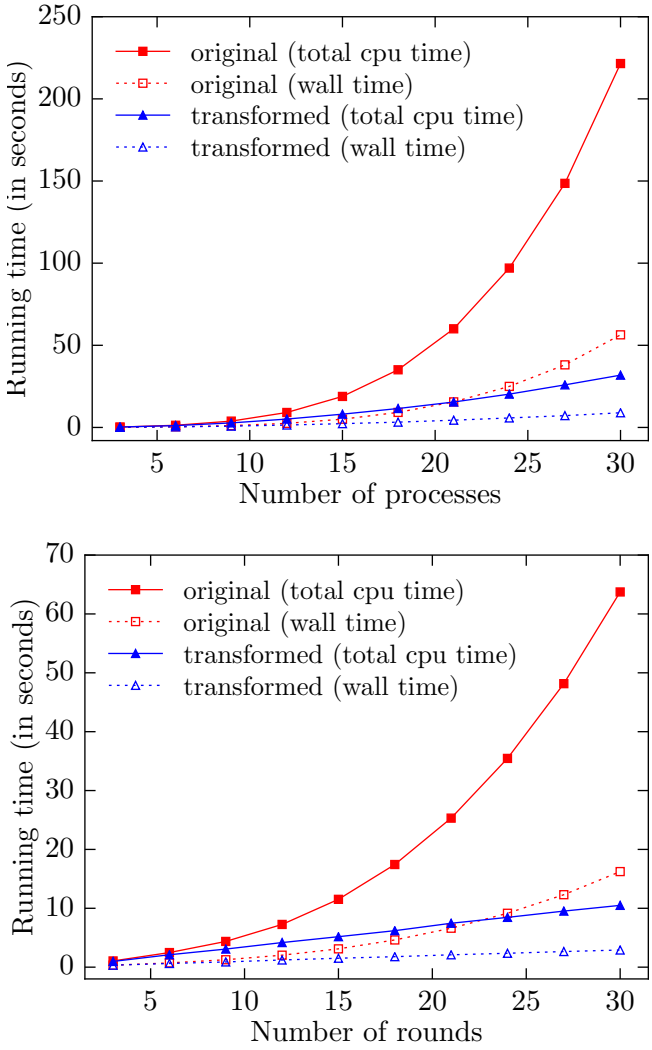


Figure 5.6: Lamport's algorithm using high-level queries over sets of received messages (`lamutex spec`). The experiment setup is similar to Figure 5.5 except the x-axis range is different, and the number of processes or rounds is held constant at 10. Each datapoint is the average of exactly 5 trials.

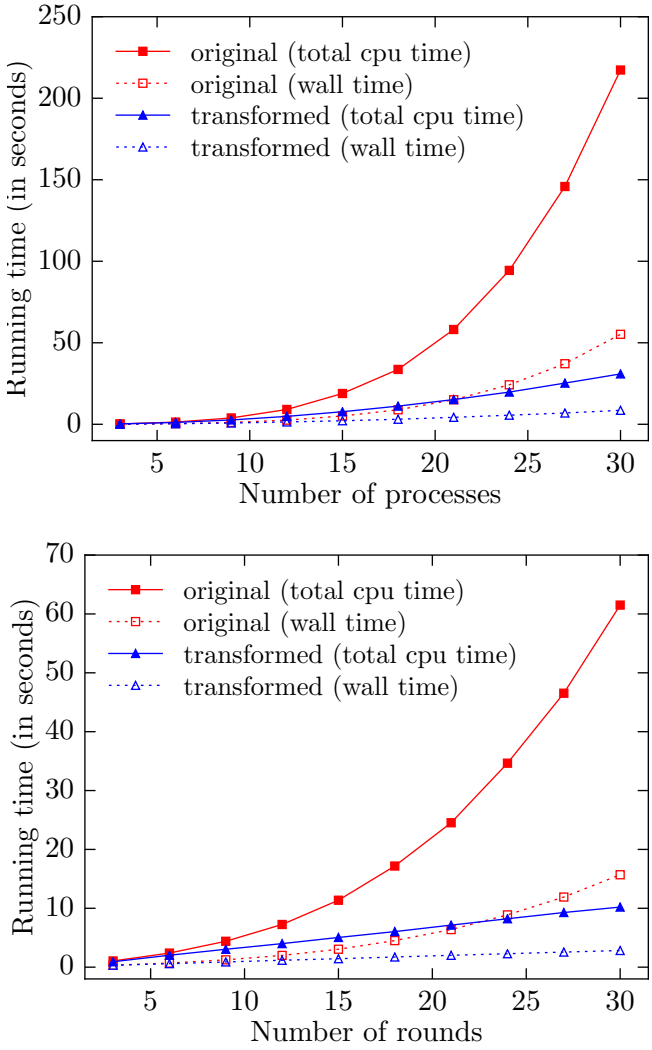


Figure 5.7: Lamport's algorithm using high-level queries where acknowledgement messages do not directly refer to the request they acknowledge (`lamutex spec lam`); this version is more consistent with Lamport's original description. Setup is as in Figure 5.6. Each datapoint is the average of exactly 5 trials.

Chapter 6

Conclusion

We have presented our method for transforming high-level object queries into efficient implementations with cost bounds. There are three key advantages of our method. First, our generated implementations are efficient because they are incremental and demand-driven. The maintenance code itself makes use of incrementally computed auxiliary indices (shown to be beneficial by themselves in Figure 5.3), leading to faster running time than JQL (Figure 4.6). The use of demand leads to much lower time and space costs by restricting the stored result and indices to the relevant portion of the data, as is shown by our experiments with our running example (Figure 3.2), the OSQ method (Figures 4.3 and 4.4), and Core RBAC (Figure 5.2).

Second, there is the expressiveness of our queries. We handle arbitrary nesting of objects and sets; arbitrary aliasing among expressions in the query and expressions appearing in updates; and a number of other features including nested queries, aggregates, maps, negated memberships, and pattern matching. Our method produces incremental code for every kind of update, the only caveat being that the query must satisfy a safety requirement that prevents it from hiding the ways in which it depends on updates. We have shown that our method applies to, and improves the asymptotic performance of, examples used to benchmark previous systems (OSQ in Section 4.3 and JQL in Section 4.4), and applications in distributed algorithms, access control, approximate probabilistic inference, and student information management (Chapter 5).

Third, our method is systematic and based on transformations that preserve invariants. This gives us the ability to reason about precisely what values will be stored in and retrieved from auxiliary data structures. This in turn allows us to give precise cost bounds on running time and space usage, and to perform optimizations such as counting elimination and type-check

elimination that benefit from strong guarantees about the behavior of maintenance code (Chapter 2). Furthermore, we are free to make different high-level choices when determining the invariants and join orders, leading to different cost trade-offs in the generated code without sacrificing our confidence in its correctness (Section 4.2).

As discussed in Chapter 4, ours is the first method to satisfy all three of the above criteria simultaneously. Methods like JQL do not allow such expressive queries where nested objects may be updated [85], while methods like OSQ are not invariant-based [74, 72]. Both systems require a substantial runtime environment[1] and do not provide precise cost bounds. Other techniques used in relational databases or logic programs do not handle objects and nested sets and do not generate imperative code, while techniques based on memoization and change propagation, including self-adjusting computation [6, 2, 5], do not apply to queries but rather to specific computations.

A recurring theme in the progression of programming languages is to abstract more details away from the programmer and offload more responsibility onto the compiler. Methods like ours are an important and natural step in this direction. It should be clear from the discussion in Example 1, and from the complexity and size of the code generated, that manually writing code to handle all possible updates correctly and efficiently would be very difficult and tedious, even without considering demand. A programmer faced with this task can either simplify their implementation strategy, possibly sacrificing efficiency, or they can spend precious developer resources producing and maintaining a sophisticated implementation. This investment would be wasted if the query were later changed.

Our aim is to remove obstacles to the adoption of high-level queries in everyday programming. By supporting object queries, we ensure that there is no impedance mismatch between the query language and the queried data. By generating efficient implementations, we hope to encourage the programmer to rely on an automated and provably correct method so they can spend their time elsewhere. Our method can be just another compilation step in the build workflow.

Future work for our method can focus on more ways of generating, maintaining, and using invariants. We are already aware of several improvements that would be beneficial to our distributed algorithm applications. These include eliminating dead maintenance code based on high-level reasoning to determine when the query is unsatisfiable, adding support for efficient range

[1] Although OSQ may also be used to generate static code, by pairing it with a code transformation system [72].

queries [82, 83], and incorporating methods for rewriting quantifiers using aggregates [51]. We may also investigate generalizations of our demand strategies, for instance, using more filters than are currently generated in order to define more precise indices to help with query maintenance. Examples of some other alternative invariants have already been discussed in Section 4.2. In general, there are many possible invariants that could be considered, and further study is needed to determine which ones are most applicable in which situations.

One of the advantages of having an automated invariant-based transformation system is that we are able to easily generate different implementations with different cost tradeoffs, e.g., between time and space, or between query result maintenance time and demand propagation time for different kinds of updates. We would like to study ways of automatically selecting, or helping the developer to select, the optimal implementation for his or her use case. This can make use of a combination of cost analysis, analysis of the input data, and systematic benchmarking of the various implementations.

Other future work may focus on making the system more practical for real programmers, e.g., in the Python ecosystem. While the applications we have studied solve real problems, they are idealized in the sense that they are written using declarative queries, they provide a clean API at the module level, and they respect our restrictions on allowed forms of updates. In the real world, programmers may write imperative code because they are not familiar with, or fear the performance of, declarative queries; they may only wish to apply specialized systems such as ours to a small piece of code, say, in a certain class or a region of code marked by annotations; and they may use difficult-to-analyze features that update the queried values in non-straightforward ways.

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[2] Umut A. Acar. Self-adjusting computation: (an overview). In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '09, pages 1–6, New York, NY, USA, 2009. ACM.

[3] Umut A. Acar, Amal Ahmed, and Matthias Blume. Imperative self-adjusting computation. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 309–322, New York, NY, USA, 2008. ACM.

[4] Umut A. Acar, Guy Blelloch, Ruy Ley-Wild, Kanat Tangwongsan, and Duru Turkoglu. Traceable data types for self-adjusting computation. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 483–496, New York, NY, USA, 2010. ACM.

[5] Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation. *ACM Trans. Program. Lang. Syst.*, 32(1):3:1–3:53, November 2009.

[6] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. *ACM Trans. Program. Lang. Syst.*, 28(6):990–1034, November 2006.

[7] American National Standards Institute, Inc. *Role-Based Access Control*, February 2004. INCITS 359-2004.

[8] Stuart Russel's Group at UC Berkeley. Bayesian logic (BLOG) examples. `https://sites.google.com/site/bloginference/example-usage`. Last accessed on 2016-04-25.

[9] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34, August 2013.

[10] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '86, pages 1–15, New York, NY, USA, 1986. ACM.

[11] C. Beeri and R. Ramakrishnan. On the power of magic. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '87, pages 269–284, New York, NY, USA, 1987. ACM.

[12] Jon Brandvein and Yanhong A. Liu. Removing runtime overhead for optimized object queries. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM 2016, pages 73–84, New York, NY, USA, 2016. ACM.

[13] Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. A theory of changes for higher-order languages: Incrementalizing λ-calculi by static differentiation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 145–155, New York, NY, USA, 2014. ACM.

[14] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[15] 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.

[16] Ashok K Chandra and Philip M Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 77–90. ACM, 1977.

[17] Ernest Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM*, 22(5):281–283, May 1979.

[18] Yan Chen, Joshua Dunfield, and Umut A. Acar. Type-directed automatic incrementalization. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 299–310, New York, NY, USA, 2012. ACM.

[19] Camil Demetrescu, Irene Finocchi, and Andrea Ribichini. Reactive imperative programming with dataflow constraints. *ACM Trans. Program. Lang. Syst.*, 37(1):3:1–3:53, November 2014.

[20] Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

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

[22] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, 1978. Springer-Verlag.

[23] Timothy Griffin and Leonid Libkin. Incremental maintenance of views with duplicates. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 328–339, New York, NY, USA, 1995. ACM.

[24] Timothy Griffin, Leonid Libkin, and Howard Trickey. An improved algorithm for the incremental recomputation of active relational expressions. *IEEE Trans. on Knowl. and Data Eng.*, 9(3):508–511, May 1997.

[25] Ashish Gupta, Dinesh Katiyar, and Inderpal Singh Mumick. Counting solutions to the view maintenance problem. In *In Workshop on Deductive Databases, JICSLP*, 1992.

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

[27] 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.

[28] Matthew A. Hammer, Umut A. Acar, and Yan Chen. Ceal: A c-based language for self-adjusting computation. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 25–37, New York, NY, USA, 2009. ACM.

[29] Matthew A. Hammer, Joshua Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael Hicks, and David Van Horn. Incremental computation with names. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 748–766, New York, NY, USA, 2015. ACM.

[30] Matthew A. Hammer, Georg Neis, Yan Chen, and Umut A. Acar. Self-adjusting stack machines. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 753–772, New York, NY, USA, 2011. ACM.

[31] Matthew A. Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey S. Foster. Adapton: Composable, demand-driven incremental computation. Technical report, University of Maryland, July 2013. CS-TR-5027.

[32] Daniel S. Hirschberg and James Bartlett Sinclair. Decentralized extrema-finding in circular configurations of processors. *Communications of the ACM*, 23(11):627–628, 1980.

[33] Akira Kawaguchi, Daniel Lieuwen, Inderpal Mumick, and Kenneth Ross. Implementing incremental view maintenance in nested data models. In *Database Programming Languages*, pages 202–221. Springer, 1997.

[34] Alfons Kemper, Christoph Kilger, and Guido Moerkotte. Function materialization in object bases. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, SIGMOD '91, pages 258–267, New York, NY, USA, 1991. ACM.

[35] Mahmoud Abo Khamis, Hung Q. Ngo, Christopher Ré, and Atri Rudra. Joins via geometric resolutions: Worst-case and beyond. *CoRR*, abs/1404.0703, 2014.

[36] Christoph Koch. Incremental query evaluation in a ring of databases. In *Proceedings of the Twenty-ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '10, pages 87–98, New York, NY, USA, 2010. ACM.

[37] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. Dbtoaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB J.*, 23(2):253–278, 2014.

[38] Harumi A. Kuno and Elke A. Rundensteiner. Incremental maintenance of materialized object-oriented views in multiview: Strategies and performance evaluation. *IEEE Trans. on Knowl. and Data Eng.*, 10(5):768–792, September 1998.

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

[40] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[41] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[42] Stefan Lehmann, Tim Felgentreff, Jens Lincke, Patrick Rein, and Robert Hirschfeld. Reactive object queries: Consistent views in object-oriented languages. In *Companion Proceedings of the 15th International Conference on Modularity*, MODULARITY Companion 2016, pages 23–28, New York, NY, USA, 2016. ACM.

[43] Ruy Ley-Wild, Umut A. Acar, and Matthew Fluet. A cost semantics for self-adjusting computation. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 186–199, New York, NY, USA, 2009. ACM.

[44] Yanhong A. Liu. Efficiency by incrementalization: An introduction. *Higher-Order and Symbolic Computation*, 13(4):289–313, 2000.

[45] 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.

[46] 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.

[47] Yanhong A. Liu and Scott D. Stoller. From datalog rules to efficient programs with time and space guarantees. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming*, PPDP '03, pages 172–183, New York, NY, USA, 2003. ACM.

[48] Yanhong A. Liu and Scott D. Stoller. Role-based access control: A simplified specification. Technical report, 2005.

[49] Yanhong A. Liu and Scott D. Stoller. From datalog rules to efficient programs with time and space guarantees. *ACM Trans. Program. Lang. Syst.*, 31(6):21:1–21:38, August 2009.

[50] Yanhong A. Liu, Scott D. Stoller, Michael Gorbovitski, Tom Rothamel, and Yanni Ellen Liu. Incrementalization across object abstraction. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 473–486, New York, NY, USA, 2005. ACM.

[51] 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.

[52] 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.

[53] Yanhong Annie Liu. *Systematic Program Design: From Clarity to Efficiency*. Cambridge University Press, New York, NY, USA, 2013.

[54] Daniel Lupei, Christoph Koch, and Val Tannen. Incremental view maintenance for nested-relational databases. *CoRR*, abs/1412.4320, 2014.

[55] Erik Meijer, Brian Beckman, and Gavin Bierman. Linq: Reconciling object, relations and xml in the .net framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 706–706, New York, NY, USA, 2006. ACM.

[56] Donald Michie. Memo functions and machine learning. *Nature*, 218(5136):19–22, 1968.

[57] Brian Milch and Stuart Russell. Extending bayesian networks to the open-universe case. *Heuristics, Probability and Causality: A Tribute to Judea Pearl. College Publications*, 2010.

[58] Ralf Mitschke, Sebastian Erdweg, Mirko Köhler, Mira Mezini, and Guido Salvaneschi. i3ql: Language-integrated live data views. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 417–432. ACM, 2014.

[59] Boris Motik, Yavor Nenov, Robert Edgar Felix Piro, and Ian Horrocks. Incremental update of datalog materialisation: the backward/forward algorithm. In *AAAI*, pages 1560–1568, 2015.

[60] Hiroaki Nakamura. Incremental computation of complex object queries. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 156–165, New York, NY, USA, 2001. ACM.

[61] Venkata Krishna Suhas Nerella, Sanjay Madria, and Thomas Weigert. Efficient caching and incrementalization of object queries on collections in programming codes. In *Proceedings of the 2014 IEEE 38th Annual Computer Software and Applications Conference*, COMPSAC '14, pages 229–238, Washington, DC, USA, 2014. IEEE Computer Society.

[62] Venkata Krishna Suhas Nerella, Swetha Surapaneni, Sanjay K. Madria, and Thomas Weigert. Exploring optimization and caching for efficient collection operations. *Automated Software Engineering*, pages 1–38, 2013.

[63] Hung Q. Ngo, Dung T. Nguyen, Christopher Re, and Atri Rudra. Beyond worst-case analysis for joins with minesweeper. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '14, pages 234–245, New York, NY, USA, 2014. ACM.

[64] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms: [extended abstract]. In *Proceedings of the 31st Symposium on Principles of Database Systems*, PODS '12, pages 37–48, New York, NY, USA, 2012. ACM.

[65] Robert Paige. Applications of finite differencing to database integrity control and query/transaction optimization. In HervÃľ Gallaire, Jack Minker, and JeanMarie Nicolas, editors, *Advances in Data Base Theory*, pages 171–209. Springer US, 1984.

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

[67] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 315–328, New York, NY, USA, 1989. ACM.

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

[69] G. Ramalingam and Thomas Reps. A categorized bibliography on incremental computation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 502–510, New York, NY, USA, 1993. ACM.

[70] G. Ricart and A. K. Agrawala. Author's response to 'On Mutual Exclusion in Computer Networks' by Carvalho and Roucairol. Commun. ACM, 26(2):147–148, 1983.

[71] Glenn Ricart and Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24(1):9–17, January 1981.

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

[73] Tom Rothamel and Yanhong A. Liu. Efficient implementation of tuple pattern based retrieval. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '07, pages 81–90, New York, NY, USA, 2007. ACM.

[74] 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.

[75] Guido Salvaneschi and Mira Mezini. Reactive behavior in object-oriented applications: An analysis and a research roadmap. In *Proceedings of the 12th Annual International Conference on Aspect-oriented Software Development*, AOSD '13, pages 37–48, New York, NY, USA, 2013. ACM.

[76] Ajeet Shankar and Rastislav Bodik. Ditto: automatic incrementalization of data structure invariant checks (in java). *ACM SIGPLAN Notices*, 42(6):310–319, 2007.

[77] Ichiro Suzuki and Tadao Kasami. A distributed mutual exclusion algorithm. *ACM Transactions on Computer Systems (TOCS)*, 3(4):344–349, 1985.

[78] K. Tuncay Tekle and Yanhong A. Liu. Precise complexity analysis for efficient datalog queries. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, PPDP '10, pages 35–44, New York, NY, USA, 2010. ACM.

[79] K. Tuncay Tekle and Yanhong A. Liu. More efficient datalog queries: Subsumptive tabling beats magic sets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 661–672, New York, NY, USA, 2011. ACM.

[80] Daniel C Wang, Andrew W Appel, Jeffrey L Korn, and Christopher S Serra. The zephyr abstract syntax description language. In *DSL*, volume 97, pages 17–17, 1997.

[81] Jef Wijsen. Database repairing using updates. *ACM Trans. Database Syst.*, 30(3):722–768, September 2005.

[82] Dan E. Willard. Applications of range query theory to relational data base join and selection operations. *J. Comput. Syst. Sci.*, 52(1):157–169, February 1996.

[83] Dan E Willard. An algorithm for handling many relational calculus queries efficiently. *Journal of Computer and System Sciences*, 65(2):295–331, 2002.

[84] 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.

[85] 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.

Appendix A

Demand for nested queries

To make the discussion on page 55 concrete, we give an example of nested relational comprehensions where naively updating the inner query's demand set at-will leads to incorrect code. This example should serve to convince the reader of the importance of using an invariant-based approach, and to justify why we manage demand for inner queries using a comprehension rather than an arbitrarily updated relation.

The OSQ method claims that nested queries can be handled by supplying the result of the inner query as a parameter to the outer query [74], but it does not address the issue of how to manage the inner query's demand. We show that simply iterating the overall method, in innermost-query-first order and treating each query as if it were top-level, is not sufficient.

The example we will consider is a very simple query that does not need flattening, just strengthening and restricting. Its only variable is a parameter. For brevity we omit the syntax for singleton tuples.

```
{x : x in {x : x in R}}
```

That is, the query simply returns the set containing the given value of `x` if it is in R, and the empty set otherwise. For simplicity we will skip the part of the Strengthen step that modifies the result expressions and inserts image-set expressions, because the semantics are the same without them; instead we will just turn `x` into a local variable. After the Restrict step, if we were to create demand sets for both the inner and outer query, we would end up with,

```
{x : x in U2, x in {y : y in U1, y in R}},
```

where we have renamed the inner uses of `x` to `y` to emphasize that they live in a distinct scope. Incrementalizing the inner comprehension and calling the

inner and outer comprehensions Q1 and Q2 respectively, we have the invariants,

$$\texttt{Q1} = \texttt{\{y : y in U1, y in R\}}$$
$$\texttt{Q2} = \texttt{\{x : x in U2, x in Q1\}}.$$

Suppose that R contains a single element a, and U1 and U2 are empty, so that Q1 and Q2 are also empty. The first time we perform the outer query where a is the parameter value of x, we will have to add it to U2 and maintian Q2.

```
U2.add(a)
x = a                    # maintain Q2 for U2
if x in Q1:              # using Q1
    Q2.add_c(x)
retrieve result of Q2
```

But since we are reading Q1's value inside the maintenance for Q2, we need to first add to U1 or else Q1 will be empty. In the absence of a systematic method telling us how and when to adjust U1, we will naively assume that we can just add a to U1 immediately before the use of Q1. This in turn causes us to insert maintenance for Q1 and Q2.[1]

```
U2.add(a)
x = a
U1.add(x)
y = x                    # maintain Q1 for U1
if y in R:
    Q1.add_c(y)
    x' = y               # maintain Q2 for Q1
    if x' in U2:         # use fresh instantiation x' for x
        Q2.add_c(x')
if x in Q1:
    Q2.add_c(x)
retrieve result of Q2
```

Now we have a problem, because after this code has run, a will be present in the counted set Q2 with a count of 2, when it should have count 1.

Under our system of using an invariant for the nested query's demand clause, instead of using a demand set U1 we should have a demand comprehension. There is only one preceding clause, so we end up with the following

[1]Properly speaking, by the time we are incrementalizing Q2, the program should have already been transformed to maintain Q1 at updates to U1 and R. By inserting a new update to U1 and additional code to maintain Q1, we are not strictly following finite differencing.

query after the Restrict step.

```
{x : x in U2, x in {y : y in {z : z in U2}, y in R}}
```

Here `z` is another renamining of `x` to again indicate a new scope. Since the demand comprehension is semantically the same as `U2` itself, we can replace it with `U2`, obtaining

$$\texttt{Q1} = \texttt{\{y : y in U2, y in R\}}$$
$$\texttt{Q2} = \texttt{\{x : x in U2, x in Q1\}}.$$

The correct code is obtained by simply incrementalizing `Q1` and then `Q2`.

```
U2.add(a)
x = a                   # maintain Q2 for U2
if x in Q1:
    Q2.add_c(x)
y = a                   # maintain Q1 for U2
if y in R:
    Q1.add_c(y)
    x = y               # maintain Q2 for Q1
    if x in U2:
        Q2.add_c(x)
retrieve result of Q2
```

Appendix B

Generated code

We give an example of the code generated by our implementation, for the social network query. First, the input program. The beginning contains annotations instructing the system to translate nested sets and objects using M and F_f, and telling the system that counting elimination and type check elimination are safe to apply. In general, the safety of these optimizations follows from any annotation or inference that tells the system that users have unique emails and that the demanded values for `celeb` and `group` are well-typed at all times.

```
# Social network example.

from incoq.runtime import *

CONFIG(
    obj_domain = 'true',
)

SYMCONFIG('Q',
    count_elim_safe_override = 'true',
    well_typed_data = 'true',
)

def make_user(email, loc):
    u = Obj()
    u.followers = Set()
    u.email = email
    u.loc = loc
    return u

def make_group():
    g = Set()
    return g

def follow(u, c):
    assert u not in c.followers
    c.followers.add(u)

def unfollow(u, c):
```

```
        assert u in c.followers
        c.followers.remove(u)

    def join_group(u, g):
        assert u not in g
        g.add(u)

    def leave_group(u, g):
        assert u in g
        g.remove(u)

    def change_loc(u, loc):
        del u.loc
        u.loc = loc

    def do_query(celeb, group):
        return QUERY('Q', {user.email for user in celeb.followers if user in group
                                        if user.loc == 'NYC'})
```

Next, the generated code. Continuation lines are indicated with >>. The first part of the output file is a header describing the generated invariants. Then there are declarations for the introduced variables, and the maintenance functions, which take up the bulk of the file. Finally there is the original code with calls to maintenance functions inserted around updates. Each function besides the auxiliary map maintenance functions are annotated with an automatically generated time cost bound, which can be further simplified as we have in Table 3.2.

```
    # Q : celeb, group -> {(celeb, group, user_email) for (celeb, group) in REL(_U_Q
>>  ) for (celeb, celeb_followers) in F(followers) for (celeb_followers, user) in M(
>>  ) for (group, user) in M() for (user, user_loc) in F(loc) if (user_loc == 'NYC')
>>   for (user, user_email) in F(email)}
    # Q_T_celeb : {(celeb,) for (celeb, group) in REL(_U_Q)}
    # Q_T_group : {(group,) for (celeb, group) in REL(_U_Q)}
    # Q_d_F_followers : {(celeb, celeb_followers) for (celeb,) in REL(R_Q_T_celeb) f
>>  or (celeb, celeb_followers) in F(followers)}
    # Q_T_celeb_followers : {(celeb_followers,) for (celeb, celeb_followers) in REL(
>>  R_Q_d_F_followers)}
    # Q_d_M_1 : {(celeb_followers, user) for (celeb_followers,) in REL(R_Q_T_celeb_f
>>  ollowers) for (celeb_followers, user) in M()}
    # Q_T_user_1 : {(user,) for (celeb_followers, user) in REL(R_Q_d_M_1)}
    # Q_d_M_2 : {(group, user) for (group,) in REL(R_Q_T_group) for (group, user) in
>>   M()}
    # Q_T_user_2 : {(user,) for (group, user) in REL(R_Q_d_M_2)}
    # Q_d_F_loc : {(user, user_loc) for (user,) in REL(R_Q_T_user_1) for (user,) in
>>  REL(R_Q_T_user_2) for (user, user_loc) in F(loc)}
    # Q_d_F_email : {(user, user_email) for (user,) in REL(R_Q_T_user_1) for (user,)
>>   in REL(R_Q_T_user_2) for (user, user_email) in F(email)}
    from incoq.runtime import *
    _U_Q = Set()
    R_Q_T_celeb = CSet()
    R_Q_T_group = CSet()
    R_Q_d_F_followers = Set()
    R_Q_T_celeb_followers = CSet()
    R_Q_d_M_1 = Set()
```

```
R_Q_T_user_1 = CSet()
R_Q_d_M_2 = Set()
R_Q_T_user_2 = CSet()
R_Q_d_F_loc = Set()
R_Q_d_F_email = Set()
_U_Q_bu = Map()
R_Q_d_F_followers_ub = Map()
_U_Q_ub = Map()
R_Q_d_M_1_ub = Map()
R_Q_bbu = Map()
def _maint__U_Q_bu_for__U_Q_add(_elem):
    (_elem_v1, _elem_v2) = _elem
    _v54_key = _elem_v1
    _v54_value = _elem_v2
    if (_v54_key not in _U_Q_bu):
        _v55 = Set()
        _U_Q_bu[_v54_key] = _v55
    _U_Q_bu[_v54_key].add(_v54_value)

def _maint__U_Q_ub_for__U_Q_add(_elem):
    (_elem_v1, _elem_v2) = _elem
    _v57_key = _elem_v2
    _v57_value = _elem_v1
    if (_v57_key not in _U_Q_ub):
        _v58 = Set()
        _U_Q_ub[_v57_key] = _v58
    _U_Q_ub[_v57_key].add(_v57_value)

def _maint_R_Q_d_F_followers_ub_for_R_Q_d_F_followers_add(_elem):
    (_elem_v1, _elem_v2) = _elem
    _v60_key = _elem_v2
    _v60_value = _elem_v1
    if (_v60_key not in R_Q_d_F_followers_ub):
        _v61 = Set()
        R_Q_d_F_followers_ub[_v60_key] = _v61
    R_Q_d_F_followers_ub[_v60_key].add(_v60_value)

def _maint_R_Q_d_F_followers_ub_for_R_Q_d_F_followers_remove(_elem):
    (_elem_v1, _elem_v2) = _elem
    _v62_key = _elem_v2
    _v62_value = _elem_v1
    R_Q_d_F_followers_ub[_v62_key].remove(_v62_value)
    if (len(R_Q_d_F_followers_ub[_v62_key]) == 0):
        del R_Q_d_F_followers_ub[_v62_key]

def _maint_R_Q_d_M_1_ub_for_R_Q_d_M_1_add(_elem):
    (_elem_v1, _elem_v2) = _elem
    _v63_key = _elem_v2
    _v63_value = _elem_v1
    if (_v63_key not in R_Q_d_M_1_ub):
        _v64 = Set()
        R_Q_d_M_1_ub[_v63_key] = _v64
    R_Q_d_M_1_ub[_v63_key].add(_v63_value)

def _maint_R_Q_d_M_1_ub_for_R_Q_d_M_1_remove(_elem):
    (_elem_v1, _elem_v2) = _elem
    _v65_key = _elem_v2
    _v65_value = _elem_v1
    R_Q_d_M_1_ub[_v65_key].remove(_v65_value)
    if (len(R_Q_d_M_1_ub[_v65_key]) == 0):
```

```
            del R_Q_d_M_1_ub[_v65_key]

    def _maint_R_Q_bbu_for_R_Q_add(_elem):
        (_elem_v1, _elem_v2, _elem_v3) = _elem
        _v66_key = (_elem_v1, _elem_v2)
        _v66_value = _elem_v3
        if (_v66_key not in R_Q_bbu):
            _v67 = Set()
            R_Q_bbu[_v66_key] = _v67
        R_Q_bbu[_v66_key].add(_v66_value)

    def _maint_R_Q_bbu_for_R_Q_remove(_elem):
        (_elem_v1, _elem_v2, _elem_v3) = _elem
        _v68_key = (_elem_v1, _elem_v2)
        _v68_value = _elem_v3
        R_Q_bbu[_v68_key].remove(_v68_value)
        if (len(R_Q_bbu[_v68_key]) == 0):
            del R_Q_bbu[_v68_key]

    def _maint_R_Q_d_F_email_for_R_Q_T_user_1_add(_elem):
        # Cost: O(1)
        (_v48_user,) = _elem
        if ((_v48_user,) in R_Q_T_user_2):
            _v48_user_email = _v48_user.email
            _v48_result = (_v48_user, _v48_user_email)
            R_Q_d_F_email.add(_v48_result)

    def _maint_R_Q_d_F_email_for_R_Q_T_user_1_remove(_elem):
        # Cost: O(1)
        (_v49_user,) = _elem
        if ((_v49_user,) in R_Q_T_user_2):
            _v49_user_email = _v49_user.email
            _v49_result = (_v49_user, _v49_user_email)
            R_Q_d_F_email.remove(_v49_result)

    def _maint_R_Q_d_F_email_for_R_Q_T_user_2_add(_elem):
        # Cost: O(1)
        (_v50_user,) = _elem
        if ((_v50_user,) in R_Q_T_user_1):
            _v50_user_email = _v50_user.email
            _v50_result = (_v50_user, _v50_user_email)
            R_Q_d_F_email.add(_v50_result)

    def _maint_R_Q_d_F_email_for_R_Q_T_user_2_remove(_elem):
        # Cost: O(1)
        (_v51_user,) = _elem
        if ((_v51_user,) in R_Q_T_user_1):
            _v51_user_email = _v51_user.email
            _v51_result = (_v51_user, _v51_user_email)
            R_Q_d_F_email.remove(_v51_result)

    def _maint_R_Q_d_F_email_for__F_email_add(_elem):
        # Cost: O(1)
        (_v52_user, _v52_user_email) = _elem
        if ((_v52_user,) in R_Q_T_user_1):
            if ((_v52_user,) in R_Q_T_user_2):
                _v52_result = (_v52_user, _v52_user_email)
                R_Q_d_F_email.add(_v52_result)

    def _maint_R_Q_d_F_loc_for_R_Q_T_user_1_add(_elem):
```

```
        # Cost: O(1)
        (_v42_user,) = _elem
        if ((_v42_user,) in R_Q_T_user_2):
            _v42_user_loc = _v42_user.loc
            _v42_result = (_v42_user, _v42_user_loc)
            R_Q_d_F_loc.add(_v42_result)

    def _maint_R_Q_d_F_loc_for_R_Q_T_user_1_remove(_elem):
        # Cost: O(1)
        (_v43_user,) = _elem
        if ((_v43_user,) in R_Q_T_user_2):
            _v43_user_loc = _v43_user.loc
            _v43_result = (_v43_user, _v43_user_loc)
            R_Q_d_F_loc.remove(_v43_result)

    def _maint_R_Q_d_F_loc_for_R_Q_T_user_2_add(_elem):
        # Cost: O(1)
        (_v44_user,) = _elem
        if ((_v44_user,) in R_Q_T_user_1):
            _v44_user_loc = _v44_user.loc
            _v44_result = (_v44_user, _v44_user_loc)
            R_Q_d_F_loc.add(_v44_result)

    def _maint_R_Q_d_F_loc_for_R_Q_T_user_2_remove(_elem):
        # Cost: O(1)
        (_v45_user,) = _elem
        if ((_v45_user,) in R_Q_T_user_1):
            _v45_user_loc = _v45_user.loc
            _v45_result = (_v45_user, _v45_user_loc)
            R_Q_d_F_loc.remove(_v45_result)

    def _maint_R_Q_d_F_loc_for__F_loc_add(_elem):
        # Cost: O(1)
        (_v46_user, _v46_user_loc) = _elem
        if ((_v46_user,) in R_Q_T_user_1):
            if ((_v46_user,) in R_Q_T_user_2):
                _v46_result = (_v46_user, _v46_user_loc)
                R_Q_d_F_loc.add(_v46_result)

    def _maint_R_Q_d_F_loc_for__F_loc_remove(_elem):
        # Cost: O(1)
        (_v47_user, _v47_user_loc) = _elem
        if ((_v47_user,) in R_Q_T_user_1):
            if ((_v47_user,) in R_Q_T_user_2):
                _v47_result = (_v47_user, _v47_user_loc)
                R_Q_d_F_loc.remove(_v47_result)

    def _maint_R_Q_T_user_2_for_R_Q_d_M_2_add(_elem):
        # Cost: O(1)
        (_v40_group, _v40_user) = _elem
        _v40_result = (_v40_user,)
        if (_v40_result not in R_Q_T_user_2):
            R_Q_T_user_2.add(_v40_result)
            _maint_R_Q_d_F_email_for_R_Q_T_user_2_add(_v40_result)
            _maint_R_Q_d_F_loc_for_R_Q_T_user_2_add(_v40_result)
        else:
            R_Q_T_user_2.inccount(_v40_result)

    def _maint_R_Q_T_user_2_for_R_Q_d_M_2_remove(_elem):
        # Cost: O(1)
```

```
        (_v41_group, _v41_user) = _elem
        _v41_result = (_v41_user,)
        if (R_Q_T_user_2.getcount(_v41_result) == 1):
            _maint_R_Q_d_F_loc_for_R_Q_T_user_2_remove(_v41_result)
            _maint_R_Q_d_F_email_for_R_Q_T_user_2_remove(_v41_result)
            R_Q_T_user_2.remove(_v41_result)
        else:
            R_Q_T_user_2.deccount(_v41_result)

    def _maint_R_Q_d_M_2_for_R_Q_T_group_add(_elem):
        # Cost: O(_v36_group)
        (_v36_group,) = _elem
        for _v36_user in _v36_group:
            _v36_result = (_v36_group, _v36_user)
            R_Q_d_M_2.add(_v36_result)
            _maint_R_Q_T_user_2_for_R_Q_d_M_2_add(_v36_result)

    def _maint_R_Q_d_M_2_for_R_Q_T_group_remove(_elem):
        # Cost: O(_v37_group)
        (_v37_group,) = _elem
        for _v37_user in _v37_group:
            _v37_result = (_v37_group, _v37_user)
            _maint_R_Q_T_user_2_for_R_Q_d_M_2_remove(_v37_result)
            R_Q_d_M_2.remove(_v37_result)

    def _maint_R_Q_d_M_2_for__M_add(_elem):
        # Cost: O(1)
        (_v38_group, _v38_user) = _elem
        if ((_v38_group,) in R_Q_T_group):
            _v38_result = (_v38_group, _v38_user)
            R_Q_d_M_2.add(_v38_result)
            _maint_R_Q_T_user_2_for_R_Q_d_M_2_add(_v38_result)

    def _maint_R_Q_d_M_2_for__M_remove(_elem):
        # Cost: O(1)
        (_v39_group, _v39_user) = _elem
        if ((_v39_group,) in R_Q_T_group):
            _v39_result = (_v39_group, _v39_user)
            _maint_R_Q_T_user_2_for_R_Q_d_M_2_remove(_v39_result)
            R_Q_d_M_2.remove(_v39_result)

    def _maint_R_Q_T_user_1_for_R_Q_d_M_1_add(_elem):
        # Cost: O(1)
        (_v34_celeb_followers, _v34_user) = _elem
        _v34_result = (_v34_user,)
        if (_v34_result not in R_Q_T_user_1):
            R_Q_T_user_1.add(_v34_result)
            _maint_R_Q_d_F_email_for_R_Q_T_user_1_add(_v34_result)
            _maint_R_Q_d_F_loc_for_R_Q_T_user_1_add(_v34_result)
        else:
            R_Q_T_user_1.inccount(_v34_result)

    def _maint_R_Q_T_user_1_for_R_Q_d_M_1_remove(_elem):
        # Cost: O(1)
        (_v35_celeb_followers, _v35_user) = _elem
        _v35_result = (_v35_user,)
        if (R_Q_T_user_1.getcount(_v35_result) == 1):
            _maint_R_Q_d_F_loc_for_R_Q_T_user_1_remove(_v35_result)
            _maint_R_Q_d_F_email_for_R_Q_T_user_1_remove(_v35_result)
            R_Q_T_user_1.remove(_v35_result)
```

```
        else:
            R_Q_T_user_1.deccount(_v35_result)

    def _maint_R_Q_d_M_1_for_R_Q_T_celeb_followers_add(_elem):
        # Cost: O(_v30_celeb_followers)
        (_v30_celeb_followers,) = _elem
        for _v30_user in _v30_celeb_followers:
            _v30_result = (_v30_celeb_followers, _v30_user)
            R_Q_d_M_1.add(_v30_result)
            _maint_R_Q_d_M_1_ub_for_R_Q_d_M_1_add(_v30_result)
            _maint_R_Q_T_user_1_for_R_Q_d_M_1_add(_v30_result)

    def _maint_R_Q_d_M_1_for_R_Q_T_celeb_followers_remove(_elem):
        # Cost: O(_v31_celeb_followers)
        (_v31_celeb_followers,) = _elem
        for _v31_user in _v31_celeb_followers:
            _v31_result = (_v31_celeb_followers, _v31_user)
            _maint_R_Q_T_user_1_for_R_Q_d_M_1_remove(_v31_result)
            _maint_R_Q_d_M_1_ub_for_R_Q_d_M_1_remove(_v31_result)
            R_Q_d_M_1.remove(_v31_result)

    def _maint_R_Q_d_M_1_for__M_add(_elem):
        # Cost: O(1)
        (_v32_celeb_followers, _v32_user) = _elem
        if ((_v32_celeb_followers,) in R_Q_T_celeb_followers):
            _v32_result = (_v32_celeb_followers, _v32_user)
            R_Q_d_M_1.add(_v32_result)
            _maint_R_Q_d_M_1_ub_for_R_Q_d_M_1_add(_v32_result)
            _maint_R_Q_T_user_1_for_R_Q_d_M_1_add(_v32_result)

    def _maint_R_Q_d_M_1_for__M_remove(_elem):
        # Cost: O(1)
        (_v33_celeb_followers, _v33_user) = _elem
        if ((_v33_celeb_followers,) in R_Q_T_celeb_followers):
            _v33_result = (_v33_celeb_followers, _v33_user)
            _maint_R_Q_T_user_1_for_R_Q_d_M_1_remove(_v33_result)
            _maint_R_Q_d_M_1_ub_for_R_Q_d_M_1_remove(_v33_result)
            R_Q_d_M_1.remove(_v33_result)

    def _maint_R_Q_T_celeb_followers_for_R_Q_d_F_followers_add(_elem):
        # Cost: O(_v30_celeb_followers)
        (_v28_celeb, _v28_celeb_followers) = _elem
        _v28_result = (_v28_celeb_followers,)
        if (_v28_result not in R_Q_T_celeb_followers):
            R_Q_T_celeb_followers.add(_v28_result)
            _maint_R_Q_d_M_1_for_R_Q_T_celeb_followers_add(_v28_result)
        else:
            R_Q_T_celeb_followers.inccount(_v28_result)

    def _maint_R_Q_T_celeb_followers_for_R_Q_d_F_followers_remove(_elem):
        # Cost: O(_v31_celeb_followers)
        (_v29_celeb, _v29_celeb_followers) = _elem
        _v29_result = (_v29_celeb_followers,)
        if (R_Q_T_celeb_followers.getcount(_v29_result) == 1):
            _maint_R_Q_d_M_1_for_R_Q_T_celeb_followers_remove(_v29_result)
            R_Q_T_celeb_followers.remove(_v29_result)
        else:
            R_Q_T_celeb_followers.deccount(_v29_result)

    def _maint_R_Q_d_F_followers_for_R_Q_T_celeb_add(_elem):
```

```
        # Cost: O(_v30_celeb_followers)
        (_v24_celeb,) = _elem
        _v24_celeb_followers = _v24_celeb.followers
        _v24_result = (_v24_celeb, _v24_celeb_followers)
        R_Q_d_F_followers.add(_v24_result)
        _maint_R_Q_d_F_followers_ub_for_R_Q_d_F_followers_add(_v24_result)
        _maint_R_Q_T_celeb_followers_for_R_Q_d_F_followers_add(_v24_result)

    def _maint_R_Q_d_F_followers_for_R_Q_T_celeb_remove(_elem):
        # Cost: O(_v31_celeb_followers)
        (_v25_celeb,) = _elem
        _v25_celeb_followers = _v25_celeb.followers
        _v25_result = (_v25_celeb, _v25_celeb_followers)
        _maint_R_Q_T_celeb_followers_for_R_Q_d_F_followers_remove(_v25_result)
        _maint_R_Q_d_F_followers_ub_for_R_Q_d_F_followers_remove(_v25_result)
        R_Q_d_F_followers.remove(_v25_result)

    def _maint_R_Q_d_F_followers_for__F_followers_add(_elem):
        # Cost: O(_v30_celeb_followers)
        (_v26_celeb, _v26_celeb_followers) = _elem
        if ((_v26_celeb,) in R_Q_T_celeb):
            _v26_result = (_v26_celeb, _v26_celeb_followers)
            R_Q_d_F_followers.add(_v26_result)
            _maint_R_Q_d_F_followers_ub_for_R_Q_d_F_followers_add(_v26_result)
            _maint_R_Q_T_celeb_followers_for_R_Q_d_F_followers_add(_v26_result)

    def _maint_R_Q_T_group_for__U_Q_add(_elem):
        # Cost: O(_v36_group)
        (_v22_celeb, _v22_group) = _elem
        _v22_result = (_v22_group,)
        if (_v22_result not in R_Q_T_group):
            R_Q_T_group.add(_v22_result)
            _maint_R_Q_d_M_2_for_R_Q_T_group_add(_v22_result)
        else:
            R_Q_T_group.inccount(_v22_result)

    def _maint_R_Q_T_celeb_for__U_Q_add(_elem):
        # Cost: O(_v30_celeb_followers)
        (_v20_celeb, _v20_group) = _elem
        _v20_result = (_v20_celeb,)
        if (_v20_result not in R_Q_T_celeb):
            R_Q_T_celeb.add(_v20_result)
            _maint_R_Q_d_F_followers_for_R_Q_T_celeb_add(_v20_result)
        else:
            R_Q_T_celeb.inccount(_v20_result)

    def _maint_R_Q_for__U_Q_add(_elem):
        # Cost: O(_v10_celeb_followers)
        (_v10_celeb, _v10_group) = _elem
        if ((_v10_celeb, _v10_group) in _U_Q):
            _v10_celeb_followers = _v10_celeb.followers
            for _v10_user in _v10_celeb_followers:
                if (_v10_user in _v10_group):
                    _v10_user_loc = _v10_user.loc
                    if (_v10_user_loc == 'NYC'):
                        _v10_user_email = _v10_user.email
                        _v10_result = (_v10_celeb, _v10_group, _v10_user_email)
                        _maint_R_Q_bbu_for_R_Q_add(_v10_result)

    def _maint_R_Q_for__F_followers_add(_elem):
```

```
        # Cost: O((_U_Q_bu * _v12_celeb_followers))
        (_v12_celeb, _v12_celeb_followers) = _elem
        if ((_v12_celeb, _v12_celeb_followers) in R_Q_d_F_followers):
            for _v12_group in (_U_Q_bu[_v12_celeb] if (_v12_celeb in _U_Q_bu) else (
>>  )):
                for _v12_user in _v12_celeb_followers:
                    if (_v12_user in _v12_group):
                        _v12_user_loc = _v12_user.loc
                        if (_v12_user_loc == 'NYC'):
                            _v12_user_email = _v12_user.email
                            _v12_result = (_v12_celeb, _v12_group, _v12_user_email)
                            _maint_R_Q_bbu_for_R_Q_add(_v12_result)

    def _maint_R_Q_for__M_add(_elem):
        # Cost: O(((_U_Q_bu * R_Q_d_F_followers_ub) + _U_Q_ub))
        (_v14_celeb_followers, _v14_user) = _elem
        if ((_v14_celeb_followers, _v14_user) in R_Q_d_M_1):
            _v14_user_loc = _v14_user.loc
            if (_v14_user_loc == 'NYC'):
                _v14_user_email = _v14_user.email
                for _v14_celeb in (R_Q_d_F_followers_ub[_v14_celeb_followers] if (_v
>>  14_celeb_followers in R_Q_d_F_followers_ub) else ()):
                    for _v14_group in (_U_Q_bu[_v14_celeb] if (_v14_celeb in _U_Q_bu
>>  ) else ()):
                        if (_v14_user in _v14_group):
                            if ((_v14_group, _v14_user) != _elem):
                                _v14_result = (_v14_celeb, _v14_group, _v14_user_ema
>>  il)
                                _maint_R_Q_bbu_for_R_Q_add(_v14_result)
        (_v14_group, _v14_user) = _elem
        if ((_v14_group, _v14_user) in R_Q_d_M_2):
            _v14_user_loc = _v14_user.loc
            if (_v14_user_loc == 'NYC'):
                _v14_user_email = _v14_user.email
                for _v14_celeb in (_U_Q_ub[_v14_group] if (_v14_group in _U_Q_ub) el
>>  se ()):
                    _v14_celeb_followers = _v14_celeb.followers
                    if (_v14_user in _v14_celeb_followers):
                        _v14_result = (_v14_celeb, _v14_group, _v14_user_email)
                        _maint_R_Q_bbu_for_R_Q_add(_v14_result)

    def _maint_R_Q_for__M_remove(_elem):
        # Cost: O(((_U_Q_bu * R_Q_d_F_followers_ub) + _U_Q_ub))
        (_v15_celeb_followers, _v15_user) = _elem
        if ((_v15_celeb_followers, _v15_user) in R_Q_d_M_1):
            _v15_user_loc = _v15_user.loc
            if (_v15_user_loc == 'NYC'):
                _v15_user_email = _v15_user.email
                for _v15_celeb in (R_Q_d_F_followers_ub[_v15_celeb_followers] if (_v
>>  15_celeb_followers in R_Q_d_F_followers_ub) else ()):
                    for _v15_group in (_U_Q_bu[_v15_celeb] if (_v15_celeb in _U_Q_bu
>>  ) else ()):
                        if (_v15_user in _v15_group):
                            if ((_v15_group, _v15_user) != _elem):
                                _v15_result = (_v15_celeb, _v15_group, _v15_user_ema
>>  il)
                                _maint_R_Q_bbu_for_R_Q_remove(_v15_result)
        (_v15_group, _v15_user) = _elem
        if ((_v15_group, _v15_user) in R_Q_d_M_2):
            _v15_user_loc = _v15_user.loc
```

```
            if (_v15_user_loc == 'NYC'):
                _v15_user_email = _v15_user.email
                for _v15_celeb in (_U_Q_ub[_v15_group] if (_v15_group in _U_Q_ub) el
>>  se ()):
                    _v15_celeb_followers = _v15_celeb.followers
                    if (_v15_user in _v15_celeb_followers):
                        _v15_result = (_v15_celeb, _v15_group, _v15_user_email)
                        _maint_R_Q_bbu_for_R_Q_remove(_v15_result)

    def _maint_R_Q_for__F_loc_add(_elem):
        # Cost: O((_U_Q_bu * R_Q_d_F_followers_ub * R_Q_d_M_1_ub))
        (_v16_user, _v16_user_loc) = _elem
        if ((_v16_user, _v16_user_loc) in R_Q_d_F_loc):
            if (_v16_user_loc == 'NYC'):
                _v16_user_email = _v16_user.email
                for _v16_celeb_followers in (R_Q_d_M_1_ub[_v16_user] if (_v16_user i
>>  n R_Q_d_M_1_ub) else ()):
                    for _v16_celeb in (R_Q_d_F_followers_ub[_v16_celeb_followers] if
>>   (_v16_celeb_followers in R_Q_d_F_followers_ub) else ()):
                        for _v16_group in (_U_Q_bu[_v16_celeb] if (_v16_celeb in _U_
>>  Q_bu) else ()):
                            if (_v16_user in _v16_group):
                                _v16_result = (_v16_celeb, _v16_group, _v16_user_ema
>>  il)
                                _maint_R_Q_bbu_for_R_Q_add(_v16_result)

    def _maint_R_Q_for__F_loc_remove(_elem):
        # Cost: O((_U_Q_bu * R_Q_d_F_followers_ub * R_Q_d_M_1_ub))
        (_v17_user, _v17_user_loc) = _elem
        if ((_v17_user, _v17_user_loc) in R_Q_d_F_loc):
            if (_v17_user_loc == 'NYC'):
                _v17_user_email = _v17_user.email
                for _v17_celeb_followers in (R_Q_d_M_1_ub[_v17_user] if (_v17_user i
>>  n R_Q_d_M_1_ub) else ()):
                    for _v17_celeb in (R_Q_d_F_followers_ub[_v17_celeb_followers] if
>>   (_v17_celeb_followers in R_Q_d_F_followers_ub) else ()):
                        for _v17_group in (_U_Q_bu[_v17_celeb] if (_v17_celeb in _U_
>>  Q_bu) else ()):
                            if (_v17_user in _v17_group):
                                _v17_result = (_v17_celeb, _v17_group, _v17_user_ema
>>  il)
                                _maint_R_Q_bbu_for_R_Q_remove(_v17_result)

    def _maint_R_Q_for__F_email_add(_elem):
        # Cost: O((_U_Q_bu * R_Q_d_F_followers_ub * R_Q_d_M_1_ub))
        (_v18_user, _v18_user_email) = _elem
        if ((_v18_user, _v18_user_email) in R_Q_d_F_email):
            _v18_user_loc = _v18_user.loc
            if (_v18_user_loc == 'NYC'):
                for _v18_celeb_followers in (R_Q_d_M_1_ub[_v18_user] if (_v18_user i
>>  n R_Q_d_M_1_ub) else ()):
                    for _v18_celeb in (R_Q_d_F_followers_ub[_v18_celeb_followers] if
>>   (_v18_celeb_followers in R_Q_d_F_followers_ub) else ()):
                        for _v18_group in (_U_Q_bu[_v18_celeb] if (_v18_celeb in _U_
>>  Q_bu) else ()):
                            if (_v18_user in _v18_group):
                                _v18_result = (_v18_celeb, _v18_group, _v18_user_ema
>>  il)
                                _maint_R_Q_bbu_for_R_Q_add(_v18_result)
```

```
    def _demand_Q(_elem):
        # Cost: O((_v36_group + _v30_celeb_followers + _v10_celeb_followers))
        if (_elem not in _U_Q):
            _U_Q.add(_elem)
            _maint__U_Q_bu_for__U_Q_add(_elem)
            _maint__U_Q_ub_for__U_Q_add(_elem)
            _maint_R_Q_T_group_for__U_Q_add(_elem)
            _maint_R_Q_T_celeb_for__U_Q_add(_elem)
            _maint_R_Q_for__U_Q_add(_elem)

    def make_user(email, loc):
        # Cost: O((_v30_celeb_followers + (_U_Q_bu * _v12_celeb_followers) + (_U_Q_b
>>  u * R_Q_d_F_followers_ub * R_Q_d_M_1_ub)))
        u = Obj()
        _v1 = (u, Set())
        index(_v1, 0).followers = index(_v1, 1)
        _maint_R_Q_d_F_followers_for__F_followers_add(_v1)
        _maint_R_Q_for__F_followers_add(_v1)
        _v2 = (u, email)
        index(_v2, 0).email = index(_v2, 1)
        _maint_R_Q_d_F_email_for__F_email_add(_v2)
        _maint_R_Q_for__F_email_add(_v2)
        _v3 = (u, loc)
        index(_v3, 0).loc = index(_v3, 1)
        _maint_R_Q_d_F_loc_for__F_loc_add(_v3)
        _maint_R_Q_for__F_loc_add(_v3)
        return u

    def make_group():
        # Cost: O(1)
        g = Set()
        return g

    def follow(u, c):
        # Cost: O(((_U_Q_bu * R_Q_d_F_followers_ub) + _U_Q_ub))
        assert (u not in c.followers)
        _v4 = (c.followers, u)
        index(_v4, 0).add(index(_v4, 1))
        _maint_R_Q_d_M_2_for__M_add(_v4)
        _maint_R_Q_d_M_1_for__M_add(_v4)
        _maint_R_Q_for__M_add(_v4)

    def unfollow(u, c):
        # Cost: O(((_U_Q_bu * R_Q_d_F_followers_ub) + _U_Q_ub))
        assert (u in c.followers)
        _v5 = (c.followers, u)
        _maint_R_Q_for__M_remove(_v5)
        _maint_R_Q_d_M_1_for__M_remove(_v5)
        _maint_R_Q_d_M_2_for__M_remove(_v5)
        index(_v5, 0).remove(index(_v5, 1))

    def join_group(u, g):
        # Cost: O(((_U_Q_bu * R_Q_d_F_followers_ub) + _U_Q_ub))
        assert (u not in g)
        _v6 = (g, u)
        index(_v6, 0).add(index(_v6, 1))
        _maint_R_Q_d_M_2_for__M_add(_v6)
        _maint_R_Q_d_M_1_for__M_add(_v6)
        _maint_R_Q_for__M_add(_v6)
```

```
    def leave_group(u, g):
        # Cost: O(((_U_Q_bu * R_Q_d_F_followers_ub) + _U_Q_ub))
        assert (u in g)
        _v7 = (g, u)
        _maint_R_Q_for__M_remove(_v7)
        _maint_R_Q_d_M_1_for__M_remove(_v7)
        _maint_R_Q_d_M_2_for__M_remove(_v7)
        index(_v7, 0).remove(index(_v7, 1))

    def change_loc(u, loc):
        # Cost: O((_U_Q_bu * R_Q_d_F_followers_ub * R_Q_d_M_1_ub))
        _v8 = (u, u.loc)
        _maint_R_Q_for__F_loc_remove(_v8)
        _maint_R_Q_d_F_loc_for__F_loc_remove(_v8)
        del index(_v8, 0).loc
        _v9 = (u, loc)
        index(_v9, 0).loc = index(_v9, 1)
        _maint_R_Q_d_F_loc_for__F_loc_add(_v9)
        _maint_R_Q_for__F_loc_add(_v9)

    def do_query(celeb, group):
        # Cost: O((_v36_group + _v30_celeb_followers + _v10_celeb_followers))
        return ((_demand_Q((celeb, group)) or True) and (R_Q_bbu[(celeb, group)] if
>>  ((celeb, group) in R_Q_bbu) else Set()))
```