Demand-Driven Incremental Object Queries*

Yanhong A. Liu Jon Brandvein† Scott D. Stoller Bo Lin
Computer Science Department, Stony Brook University
{liu,jbrandve,stoller,bolin}@cs.stonybrook.edu

ABSTRACT

Object queries are essential in information seeking and decision making in vast areas of applications. However, a query may involve complex conditions on objects and sets, which can be arbitrarily nested and aliased. The objects and sets involved as well as the demand—the given parameter values of interest—can change arbitrarily. How to implement object queries efficiently under all possible updates, and furthermore to provide complexity guarantees?

This paper describes an automatic method. The method allows powerful queries to be written completely declaratively. It transforms demand as well as all objects and sets into relations. Most importantly, it defines invariants for not only the query results, but also all auxiliary values about the objects and sets involved, including those for propagating demand, and incrementally maintains all of them. Implementation and experiments with problems from a variety of application areas, including distributed algorithms and probabilistic queries, confirm the analyzed complexities, trade-offs, and significant improvements over prior work.

CCS Concepts

• **Information systems** → **Query optimization; Query languages;** • **Theory of computation** → **Invariants;** • **Software and its engineering** → **Object oriented languages; Source code generation;** *Compilers;*

Keywords

object queries; demand-driven incremental computation; program transformation; complexity guarantees

1. INTRODUCTION

Consider the following query. Given a special user, `celeb`, and a group, `group`, as parameter values of interest, the query returns the set of email addresses of all users who are in both the set of followers of `celeb` and `group`, and whose location satisfies condition `cond`:

```
// parameters: celeb, group
{user.email: user in celeb.followers, user in group,
             cond(user.loc)}
```

This query can help find and monitor, for example, voters in a political campaign, suspects in a criminal case, or targets of a planned advertisement. Similar queries can be about, for example, sellers of a special product, authorized personnel of a certain organization, or health-care providers for a particular illness, instead of followers of a special user.

In general, such queries are essential in information seeking and decision making, in everyday life, distributed computation, probabilistic inference, etc. The challenging problems are:

- A query can involve any number and combination of objects and sets, with complex conditions on them, and the objects and sets can be arbitrarily nested and aliased.
- The query can be repeatedly asked while the sets and objects involved starting from the given parameter values can change arbitrarily, and the parameter values can change arbitrarily too.

While such queries can be programmed manually at a low level to handle all the changes efficiently, it is much more desirable to be able to write the queries at a high level, and have efficient implementations generated automatically.

For the given example query, 9 different kinds of updates may affect the query result, possibly with many instances of each kind, scattered in different places in the rest of the program. It requires significant effort to write efficient incremental computation code that handles all the updates.

This paper describes an automatic method. It consists of three main new contributions:

1. It allows queries to be written completely declaratively, using flexible constraints on sets and objects that can be arbitrarily nested and aliased.

   Writing such queries arbitrarily could lead to constraints that are impossible to solve, including the famous Russell's paradox. We introduce a simple, natural condition to exclude such cases, while ensuring that normal queries can be written completely declaratively.

2. It handles changes to demand, i.e., the query parameter values of interest, uniformly as other changes, and defines invariants for not only the query results, but also

*This work was supported in part by NSF under grants CCF-1414078, CNS-1421893, IIS-1447549, CCF-1248184, CCF-0964196, and CCF-0613913; and ONR under grants N000141512208 and N000140910651.
†Jon Brandvein's new email is brandjon@google.com

PPDP '16, September 05-07, 2016, Edinburgh, United Kingdom

DOI: http://dx.doi.org/10.1145/2967973.2968610

all auxiliary values about the objects and sets involved, including those for propagating demand. This allows the overall method to provide precise complexity guarantees.

Providing complexity guarantees is extremely challenging due to arbitrary dynamic changes to demand and to all objects and sets that might be relevant. With all values captured by invariants, our method uses systematic maintenance of all invariants to support precise complexity calculation.

3. It generates standalone efficient incremental maintenance code that handles arbitrary changes to arbitrarily nested and aliased sets and objects without resorting to additional runtime support.

   This is done by transforming everything into flat relations, similar to prior work, but then generating complete, properly ordered maintenance code to tie the maintenance of all invariants together, with appropriate tests to ensure correct maintenance under nesting and aliasing.

We have developed IncOQ, a prototype implementation of the method, and used it to experiment with complex queries from a variety of applications, including those from the most relevant previous work [32, 55, 48, 56, 12] and from new applications in distributed algorithms [31, 29] and probabilistic queries [35, 36, 4]. Our evaluations consider all important factors: asymptotic time and space complexities, constant-factor optimizations, demand set size, query-update ratio, auxiliary indices, runtime overhead, demand propagation strategies, and transformation time and other characteristics.

There is a large amount of related work on object queries, incremental computation, and demand-driven computation, as discussed in Section 8. Previous works do not support fully declarative object queries with a simple well-defined semantics; they handle limited queries and updates or require sophisticated runtime support to handle demand and dynamic updates and are less efficient; and they do not provide complexity guarantees for such complex queries and updates. They also do not evaluate the wide variety of important factors as we do.

2. LANGUAGE AND PROBLEM DESCRIPTION

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

Object queries. Object queries are queries over objects. Precisely, an *object query* is a comprehension of the following form plus a set of *parameters*—variables whose values are bound before the query:

$$\begin{aligned} \mathit{query} &::= \texttt{\{}\mathit{result}\ \texttt{:}\ (\mathit{membership} \mid \mathit{condition})^{+}\texttt{\}} \\ \mathit{membership} &::= \mathit{variable}\ \texttt{in}\ \mathit{selector} \\ \mathit{selector} &::= \mathit{variable} \mid \mathit{selector}.\mathit{field} \\ \mathit{condition} &::= \mathit{expression} \\ \mathit{result} &::= \mathit{expression} \end{aligned}$$

where *expression* is any expression that is function of the values of *variables* and *selectors* in the expression.

For a query to be well-formed, we require that every variable in the query be *reachable* from a parameter, i.e., be a parameter or recursively be the left-side variable of a membership clause whose variable on the right side is reachable.

Objects, including set objects, have reference semantics. Object equality is reference equality. Given values of parameters, a query returns the set of values of the result expression for all combinations of values of variables that satisfy all membership and condition clauses. If an error occurs during the evaluation of a condition or the result expression, that combination of values is skipped. A formal semantics is given in the context of an object-oriented language that includes also constructs for distributed programming [30].

We can see that an object query may contain arbitrary object field selections and set membership constraints, where objects and sets can be arbitrarily nested and aliased. Our method just requires that each condition and the result expression be a function of the variables and selectors in the query. This gives us the freedom to decide when to evaluate a condition or the result expression.

The queries are similar to those defined by Rothamel and Liu [48], but there, membership and condition clauses are ordered, as $\mathit{membership}^{+}\mathit{condition}^{*}$, albeit superficially, and the requirements specified are not completely correct—due to an insufficient requirement on variables without using reachability and an overly strong requirement on conditions and the result expression being functions of only variables in the query.

Flexible constraints on sets and objects. Note that we allow all membership and condition clauses to be written in any order, so each clause is simply a constraint, and the entire query is completely declarative.

It is well known that arbitrary constraints could lead to queries that have no meaningful answers, including Russell's paradox:

> If `s` is the set `{x: not x in x}`, is `s` in `s`?
> If `s` is in `s`, then by definition of `s`, `s` should not be in `s`.
> If `s` is not in `s`, then by definition of `s`, `s` should be in `s`.

Well-known solutions, led by language SETL, require the use of a membership clause to enumerate a variable first, and later uses of the variable are simply tests. So, `{x: not x in x}` is not allowed.

However, this ordering forces queries to be less declarative. For example, the order of the four clauses in the following query in our language does not affect the semantics or efficiency.

```
{y-x: x in s, y in t, y > x, x < 5}
```

Consider the following three out of 24 (=4!) possible orders of the four clauses written in Python syntax, or other similar syntax in SETL, Haskell, etc. In Python, the second query below may be much faster than the first, whereas the third gives a runtime error. In Haskell, the third gives a collection of unevaluated subtractions instead of the results of the subtractions. In SETL, the last two are not legal, because all bindings of variables must go first.

```
{y-x for x in s for y in t if y > x if x < 5}
{y-x for x in s if x < 5 for y in t if y > x}
{y-x for x in s if y > x for y in t if x < 5}
```

Even database and logic languages typically have requirements on the order of conditions, e.g., in SQL, variables must be bound in the from clause before used in the where clause. In some logic languages that do relax such requirements, it is well known that non-stratified negation may arise, with no commonly agreed upon semantics, especially when encoding paradox queries like `{x: not x in x}` using recursive rules.

Our requirement for queries to be well-formed removes the ordering requirement to allow queries to be completely declarative, while excluding abnormal cases. For example, `{x: not x in x}` is well-formed if `x` is a parameter, and not well-formed otherwise. Our transformation method determines a best implementation for satisfying all the constraints while efficiently handling all possible changes.

Note that our queries are not recursive, whereas queries in logic languages may be, but logic languages do not have arbitrarily nested and aliased sets and objects as our language has.

Updates to objects. Our method handles all possible updates to the values that the query depends on. Note that the result of an object query may depend on not only the values of query parameters, but also other values—specifically, all objects reachable by following the object fields and set elements used starting from the values of the parameters. There are three kinds of fundamental updates to the values that a query depends on, besides assignments to query parameters; our method decomposes all updates into combinations of these fundamental updates, and handles all combinations of these fundamental updates:

- `o.f = x`—assign value `x` to field `f` of object `o`
- `s.add(x)`—add element `x` to set `s`
- `s.del(x)`—delete element `x` from set `s`

Cost. Object queries are expensive if evaluated straightforwardly. Let `s1`, ..., `sk` be largest possible sets of values obtained from evaluating the right sides of membership clauses in a query, and $time(conditions)$ and $time(result)$ be the times for evaluation of the conditions and the result expression, respectively. Let `#s` denote the number of elements in set `s`. The cost of straightforward evaluation of the query is $O(\texttt{\#s1} \times \ldots \times \texttt{\#sk} \times (time(conditions) + time(result)))$.

The fundamental updates each take $O(1)$ time.

Running example. We use the example query in Section 1 as a running example. We indicate the parameters in comments for clarity; they are bound variables before the query, determined automatically.

The query is well-formed because all variables in the query—`celeb`, `group`, and `user`—are reachable from the parameters. The query satisfies the requirement about condition and result expressions if `cond(user.loc)` is a function of `user` and `user.loc`—for example, by testing `user.loc` against, say, `"NYC"` or some range of geographic coordinates. We assume that the requirement is satisfied and the test takes $O(1)$ time.

If the query is run straightforwardly, by iterating using each membership clause, it takes $O(\texttt{\#celeb.followers} \times \texttt{\#group})$ time; even if run optimally, it takes $O(\texttt{\#celeb.followers} + \texttt{\#group})$ time, because the sets used must at least be read and processed.

There are 9 possible kinds of fundamental updates that may affect the query result: 2 for change of demand by assigning to parameters `celeb` and `group`, 4 for adding to and deleting from sets `celeb.followers` and `group`, and 3 for assigning to fields `celeb.followers`, `user.loc`, and `user.email`.

Demand-driven incremental computation. Not only are object queries complex and expensive, they can also be repeated while the sets and objects involved change. Therefore, it is important to be able to compute the query results incrementally with respect to the changes. Furthermore, query parameters may take on any combination of values, and it may be impossible to determine the values statically. Therefore, it is essential to be able to incrementally maintain the query result for parameter values that are determined dynamically on demand.

We define a set of *demand parameters* of a query to be a subset of the parameters such that the query is still well-formed if only the demand parameters are given as parameters.

Example. For the running example, both `celeb` and `group` must be demand parameters for the query to be well-formed, but if a new clause, `celeb in group`, is added in the query, then the query is still well-formed if only `group` is taken as a demand parameter. ■

Note that fewer demand parameters mean more parameter values for which the query results may be maintained.

Example. For the running example, if `celeb in group` is added and `group` is the only demand parameter, then the query result may be maintained for all `celeb` values in `group` instead of just the given parameter value of `celeb`. ■

We define the *demand set*, `demand`, of the query to be a set of combinations of values of the demand parameters.

- The program can explicitly specify demand parameters, and add and delete elements to and from `demand` to control what query results are to be maintained, or specify a replacement strategy, such as the least-recently-used, to use when space is short.

- Alternatively, by default, our method considers all query parameters as demand parameters, adds the values of these parameters to `demand` when these values are first queried on, and deletes them when it can be determined conservatively that these values will never be queried on again.

Given queries, updates, and demands, the problem is to incrementally maintain the query results, at the updates, for all values of demand parameters in `demand`. Our method guarantees that each execution of a transformed query expression returns the same value as the original query expression.

We consider `demand` to be relatively small, compared with the set of all possible combinations of values of all query parameters.

3. METHOD AND NOTATION

Our method for transforming a single query has 3 phases.

Phase 1 transforms all object queries, demands, and updates into relational queries and updates.

It transforms demands, as well as objects and sets that can be arbitrarily nested and aliased, all into flat relations that are not aliased. This allows queries over arbitrarily nested and aliased objects and sets for any demand set to be incrementalized in a simpler and uniform way.

Phase 2 generates efficient implementations for relational queries and updates by exploiting constraints from the objects, updates, and demands.

The key idea is to define and incrementally maintain invariants for not only the query results, but also all auxiliary values about the objects and sets involved, including those for propagating demand. This allows systematic maintenance of invariants to take all the objects, demands, and updates into account in generating efficient implementations.

This phase then generates complete, properly ordered maintenance code to tie the maintenance of all invariants together, with appropriate tests to ensure correct maintenance under nesting and aliasing.

Phase 3 transforms the implementations on relations back to implementations on objects.

This makes best use of given objects and sets in the original program as well as maps for auxiliary values to minimize the time and space of the resulting program.

Overall algorithm. Our overall algorithm handles any number of queries, including nested queries and aggregate queries. Handling multiple queries, including nested queries, is done by repeated application of the method for handling a single query. As an optimization, to avoid transforming between the object and relational domains multiple times, our method does Phase 1 for all queries, then Phase 2 for all queries, and then Phase 3 for all queries.

For independent queries, i.e., queries whose results do not depend on the results of other queries, Phase 2 can be done for them in any order. For a dependent query, i.e., a query whose result depends on the results of other queries, including subqueries in a nested query, Phase 2 is done following the chain of dependencies—first for those other queries and then for the dependent query. Aggregate queries employ a library of rules specialized to the aggregate operations `count`, `max`, etc.

Details of support for nested queries, aggregate queries, and other features for ease of efficient queries can be found in [6].

Relational queries. Relational queries are queries over flat relations, i.e., sets of flat tuples. Precisely, a *relational query* has the same form as an object query except that in membership clauses, the left side can also be a tuple form, not just a variable, and the right side can only be a variable, not a field selection, i.e.,

$$\textit{membership} ::= (\textit{variable}^{+})\ \texttt{in}\ \textit{variable}$$

Relational queries are just SQL queries but expressed using a simpler syntax and where tuple components are referred to by position numbers instead of names.

Relational queries are also expensive if evaluated straightforwardly. Our method uses constraints from objects, demands, and updates to help optimize incremental relational queries.

Generated code using operations on relations. Our generated code for efficient relational queries uses the following operations on relations, besides usual operations on sets:

- `R add x` and `R del x`, *counted addition and deletion*, increments and decrements, respectively, the count for the number of times `x` has been added to but not deleted from `R`, and keeps `x` in `R` iff its count is at least 1.

  These correspond to a way of implementing bag element addition and deletion, contrasting standard set element addition and deletion, done using assignments `R = R+{x}` and `R = R-{x}`, respectively, where `+` and `-` denote set union and difference, respectively.

- `R.(j1,...,jh){(i1,...,ik)=(x1,...,xk)}`, *image set of* `(x1,...,xk)` *under* `R`*, mapping components* `(i1,...,ik)` *to components* `(j1,...,jh)`, is the set of values of components `j1`, ..., `jh` of tuples of `R` whose components `i1`, ..., `ik` have values `x1`, ..., `xk`, respectively.

  Example. If `R` is a relation of arity at least 4, i.e., a set of tuples of at least 4 components, then `R.(4,1){(1,3)=(x,y)}` is the set of values of components 4 and 1 of tuples in `R` whose components 1 and 3 equal the values of `x` and `y`, respectively. ■

  If `i1`, ..., `ik` is a prefix of the list of indices, we can omit `(i1,...,ik)=`; and if `j1`, ..., `jh` is the list of remaining indices, we can omit `.(j1,...,jh)`. If a tuple has only one component, we can omit `()`. If the tuples in `{}` are empty, we can omit `{()=()}`.

  Example. If `R` is a relation of arity 4, then `R.(3,4){(1,2)=(x,y)}` can be abbreviated as `R.(3,4){(x,y)}`, `R{(1,2)=(x,y)}`, and `R{(x,y)}`.

  `R.(4){(2)=(x)}` can be abbreviated as `R.4{2=x}`—the set of values of component 4 of tuples in `R` whose component 2 equals the value of `x`.

  `R.4{()=()}` can be abbreviated as `R.4`—the set of values of component 4 of the tuples in `R`. ■

- When `R` is a binary relation, `inv_R`, *inverse relation of* `R`, is the set of pairs in `R` with the first and second components switched, i.e., `inv_R = {(y,x): (x,y) in R}`.

  Thus, `inv_R{x} = R.1{2=x}` whereas `R{x} = R.2{1=x}`.

We use a map to implement the mapping from values of components `i1`, ..., `ik` to values of components `j1`, ..., `jh` for `R`. This uses a nested structure, like a trie, with one level for each component `i1`, ..., `ik`; hashing is used to implement the sets at each level. This allows the image set operation to take expected $O($`#image`$)$ time, where `image` is the resulting image set, to enumerate the elements, or $O(1)$ time, to return just a reference to the resulting set. The map is updated in expected $O(1)$ time for each element addition to and deletion from `R`. The space taken by the map is $O($`#R`$)$.

Generated code using operations on objects. Our method generates code in a conventional object-oriented programming language that supports sets, maps, and tuples and where all values are objects. Besides set element addition and deletion and object field assignment, operations in the following table are used. Sets and maps are empty when first created. Tuples are of constant length and are immutable. Our method adds membership tests to guard the given updates—`x not in s` to guard `s.add(x)`, and `x in s` to guard `s.del(x)`—when the test results cannot be determined statically, so that updates do not propagate unnecessarily.

`s.cadd(x)`	add element `x` to counted set `s`
`s.cdel(x)`	delete element `x` from counted set `s`
`x in s`	return whether `x` is an element of set `s`
`m.add(x,y)`	add `y` to the image set of key `x` under map `m`
`m.del(x,y)`	delete `y` from the image set of key `x` under map `m`
`m.cadd(x,y)`	add `y` to the counted image set of key `x` under map `m`
`m.cdel(x,y)`	delete `y` from the counted image set of key `x` under map `m`
`m.keys()`	return the set of keys in map `m`
`m.get(x)`	return the value of key `x` under map `m`
`(x1,...,xk)`	create a tuple with components `x1`, ..., `xk`
`x isset`	return whether `x` is a set
`x hasfield y`	return whether `x` has field `y`

Each of these operations takes expected $O(1)$ time.

We use standard statements for assignment (`v = e`), sequencing (`stmt1 stmt2`), branching (`if b: stmt`), and looping (`for v in s: stmt`). We abbreviate assignment `v = v op e`, where `op` is any binary operation, as `v op= e`. We assume that all bound variables in the program are renamed so they are distinct.

4. PHASE 1: TRANSFORM INTO RELATIONAL QUERIES AND UPDATES

Phase 1 transforms each object query and its demand parameters into a relational query, and transforms updates as well. Transformations of queries and updates are as in a prior work [48]; only the simple addition of demand is new, but it will be used substantially in Phase 2 to define auxiliary relations to contain only objects that are reachable from values in the demand set.

Transform object queries into relational queries. We use the following relations. For each field `f`, relation `field_f` relates an object with the value of the field `f` of the object; that is, (o,x) $\in$ `field_f` $\iff$ x = o.f. Relation `member` relates each set with each member of the set; that is, (s,x) $\in$ `member` $\iff$ x $\in$ s.

To transform an object query into a relational query, the following two rules are applied repeatedly until they do not apply:

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

So, for example, a sequence of field selections are transformed by applying the first rule repeatedly from left to right.

Example. In the running example, this yields the following, where `e`, `fs`, and `l` are fresh variables:

```
// parameters: celeb, group
{e: (user,e) in field_email,
    (celeb,fs) in field_followers, (fs,user) in member,
    (group,user) in member, (user,l) in field_loc,
    cond(l)}
```

■

Add demands to relational queries. We transform the demands into an additional relational query constraint, by adding a membership clause, `(dp1,...,dpj) in demand`, in the query, constraining the values of demand parameters `dp1,...,dpj` to be in `demand`.

Example. For the running example, with demand parameters `celeb` and `group`, this adds `(celeb, group) in demand`, yielding:

```
// parameters: celeb, group
{e: (celeb,group) in demand, (user,e) in field_email,
    (celeb,fs) in field_followers, (fs,user) in member,
    (group,user) in member, (user,l) in field_loc,
    cond(l)}
```

■

Transform updates to objects into updates to relations. Updates to objects, including set objects, are transformed into updates to the field relations and member relation.

- `o.f = x` is transformed into the two updates below, or only the second update if `o.f` had no value before:

```
field_f -= {(o,o.f)}
field_f += {(o,x)}
```

- `s.add(x)` is transformed into `member += {(s,x)}`.
- `s.del(x)` is transformed into `member -= {(s,x)}`.

Example. For the running example, the transformed updates are additions to and deletions from `member`, `field_email`, `field_followers`, and `field_loc`. ■

5. PHASE 2: INCREMENTALIZE UNDER UPDATES WITH FILTERING BY DEMANDS

Phase 2 incrementalizes the relational query from Phase 1 with respect to updates and demands. We present it in two steps to show how to minimize both the running time and space usage: Step 2-INC generates efficient incremental maintenance with respect to the updates, and Step 2-FIL extends Step 2-INC to generate efficient incremental maintenance filtered with demands. Both steps provide precise cost guarantees.

5.1 Generate incremental maintenance

Step 2-INC first stores the query result in a fresh relation `r`, with components for parameters `p1`, ..., `pk` and the result of the query, and maintains an invariant of the form

`r = {(p1,...,pk,` *result*`): (`*membership* | *condition*`)`$^+$`}`

Thus, `r{(p1,...,pk)}` equals the query result, and the query is replaced with the constant-time retrieval `r{(p1,...,pk)}`, a reference to the result set.

The result set is updated as the values it depends on are updated. Where necessary, a copy of the result set is made and used, at cost linear in the size of the set. Techniques exist for determining where copying is necessary [11, 14].

Example. For the running example, the following invariant is maintained:

```
r = {(celeb,group,e)
    : (celeb,group) in demand, (user,e) in field_email,
      (celeb,fs) in field_followers, (fs,user) in member,
      (group,user) in member, (user,l) in field_loc,
      cond(l)}
```

and the query is replaced with `r{(celeb,group)}`. ■

Step 2-INC then generates efficient incremental maintenance code for updates that correspond to each membership clause in the right side of the invariant, for example, for updates `field_loc += {(user,l)}` and `field_loc -= {(user,l)}` that correspond to the last clause. The key challenge is to find an optimal order of accessing variables through relations in the other clauses to arrive at needed updates to the query results, corresponding to the well-known join order problem.

The basic ideas for generating incremental maintenance code are as in prior work, e.g., [41, 32, 48]. The main new ideas here to address the key challenge are (1) formulate the problem as an optimal growing edge cover problem, with the cost for each growth step captured precisely and symbolically, and (2) use the constraint from demands, together with constraints from objects and updates as in a prior work [48], as good heuristics for solving the problem, despite its worse-case factorial time, as follows.

(1) Create a query graph: take variables as vertices, and clauses as directed edges (directed hyperedges in the

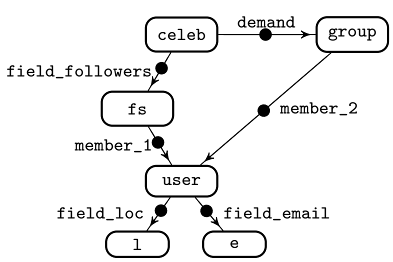
case of `demand`) labeled with the relation used in the clause, i.e., take clause `(x,y) in R` as directed edge `(x,y)` labeled `R`, and take clause `(dp1,...,dpj) in demand` as directed hyperedge `(dp1,...,dpj)` labeled `demand`; different occurrences of the same relation are suffixed with an integer indicating their order of occurrence.

- Repeatedly select an edge to access its source and target vertices (its different subsequences of vertices in the case of a hyperedge) until all edges are covered.
- Analyze the cost of selecting each next edge, `(x,y)` labeled `R`, to access its source and target vertices (with `x` and `y` denoting two different subsequences of vertices in the case of a hyperedge, and with the corresponding sequences of indices used in place of 1 and 2 below):
  – $O(1)$ if both `x` and `y` have been accessed already,
  – $O($`#R.2{1=x}`$)$ if `x` has been accessed but `y` has not,
  – $O($`#R.1{2=y}`$)$ if `y` has been accessed but `x` has not,
  – $O($`#R`$)$ if neither `x` nor `y` have been accessed before.
- For `k` clauses, there are $O($`k`$!)$ possible orders to consider in the worse case.

(2) Use the heuristic of selecting minimum-cost edges first to drastically reduce the search space, exploiting $O(1)$ or small cost edges based on constraints from objects, updates, and demand:
  – `(o,x) in field_f` when `o` has been accessed, for any f,
  – `(x,y) in R` corresponding to the update being handled,
  – `(dp1,...,dpj) in demand`.

  The first two are $O(1)$, and the last one is relatively small.

Example. For the running example, the relational query has the following query graph:



For updates `field_loc += {(user,l)}` and `field_loc -= {(user,l)}`, out of 5! possible orders, only two are considered following the heuristics in (2) above, shown in Table 1.

edges followed	cost factors
`(user,e)` labeled `field_email`	1
`(group,user)` labeled `member_2`	`#inv_member{user}`
`(celeb,group)` labeled `demand`	`#demand.1{2=group}`
`(celeb,fs)` labeled `field_followers`	1
`(fs,user)` labeled `member_1`	1
`(user,e)` labeled `field_email`	1
`(fs,user)` labeled `member_1`	`#inv_member{user}`
`(celeb,fs)` labeled `field_followers`	`#inv_field_followers{fs}`
`(celeb,group)` labeled `demand`	`#demand.2{1=celeb}`
`(group,user)` labeled `member_2`	1

Table 1: Growing edge covers for update to `field_loc`

If no other constraints are given, the first order has the minimum cost. If it is given that there is only one `celeb` in `inv_field_followers{fs}` and only one `group` for each `celeb` in `demand`, then the second order has the minimum cost. ■

Afterward, code generation uses the following basic algorithm:

1. Each ordering obtained gives an order of generated incremental maintenance clauses—`for` clauses for unbound variables and `if` clauses for bound variables, as in [32, 48].
2. A condition clause can be inserted anywhere after all variables used in it are bound, but least expensive clauses are inserted earliest.
3. A final statement updates the query result relation `r`.

Example. For the running example, the first order gives the following incremental maintenance code for update `field_loc += {(user,l)}`. Comments show asymptotic costs.

```
if cond(l):
  for e in field_email{user}:         // 1
    for group in inv_member{user}:    // #inv_member{user}
      for celeb in demand.1{2=group}: // #demand.1{2=group}
        for fs in field_followers{celeb}:  // 1
          if (fs,user) in member:          // 1
            r add (celeb,group,e)
```

Because `inv_member` is used, it is also maintained incrementally. For `member += {(group,user)}`, the following incremental maintenance is generated:

```
inv_member += {(user,group)}
```

If `inv_field_loc` is used in incremental maintenance of any query, then, for `field_loc += {(user,l)}`, the following incremental maintenance is also generated:

```
inv_field_loc += {(l,user)}
```

■

Note that, for addition to `demand`, which binds the query parameters to a new combination of values, incremental maintenance considers matching values of all other clauses, which corresponds to computing from scratch, as can be expected. For updates to `field_f` and `member`, incremental maintenance considers only matching values for the remaining clauses led to by the added or deleted tuple, and thus can be much more efficient when `demand` is small.

5.2 Generate filtered incremental maintenance

Step 2-FIL uses `demand` to more significantly reduce running times and space usage, by filtering the auxiliary relations to contain only tuples that are "reachable" from values in `demand`. Intuitively, this is done by "tagging" objects following the graph edges starting from objects in `demand`.

Formally, this is the key idea of defining and maintaining invariants, for not only the query results, but also all auxiliary values about the objects and sets involved, including those for propagating demand.

Example. For the running example, in the generated code in Section 5.1, the for-loop that binds `group` iterates over the set of all groups that the updated `user` is in, but we can filter that set to contain only the groups in `demand`. Also, the generated code need not be run at all if the updated `user` is not in the follower set of any user in `demand` or any group in `demand`. ■

Define invariants for filtered maintenance. The algorithm for defining invariants for capturing demand has two steps:

1. For each variable `var` (such as `group` in the above example) whose value is retrieved through an image set

operation, or is a variable (such as `user` in the above example) in the clause for the update being handled, find a maximal directed acyclic subgraph G(`var`), of the query graph, that reaches the vertex for `var` starting from vertices for the variables in `demand`.

2. Define a *tag set* for each vertex, and a *filtered relation* for each edge, in the subgraph G(`var`), mutually recursively:
   - The *tag set for a vertex* `v` is the intersection of objects projected from the filtered relations for the incoming edges `e1`, ..., `ek` of `v`:

     `tag_v = {v: (u1,v) in fil_e1, ..., (uk,v) in fil_ek}`

   - The *filtered relation for an edge* `e` is the relation filtered using objects in the tag set for the source vertex of `e`:

     `fil_e = {(u,v): (u,v) in e, u in tag_u}`

     We define filtered relations also for inverse member and inverse fields used to retrieve values in tag sets.
   - Initially, *the tag set for a vertex for a demand parameter* `dpi`, `i = 1`, ..., `j`, is projected from `demand`:

     `tag_dpi = {dpi: (dp1,...,dpj) in demand}`

   These definitions form the additional invariants that capture the demand precisely.

Incremental maintenance of these invariants automatically realizes incremental propagation of demand.

To obtain filtered maintenance, in the incremental maintenance code, replace uses of the original relations with the corresponding filtered relations, and add tests to check that the objects in the updates are in the corresponding tag sets.

Example. Continue the running example. For the use of `inv_member{user}` to retrieve the value of `group`, a single-vertex subgraph, containing only the vertex for `group`, is found.

Then, the following tag set and filtered relation are defined for `group`, and the original use of `inv_member` is replaced with `fil_inv_member_2`; suffix `_2` distinguishes this occurrence of `inv_member` from other occurrences.

```
tag_group  =  {group: (celeb,group) in demand}
fil_inv_member_2  =  {(user,group)
                     : (user,group) in inv_member,
                       group in tag_group}
```

These auxiliary values are then incrementally maintained, just as for any query. This yields the following maintenance code for `demand += {(celeb,group)}`, following the dependency chain:

```
// maintain tag_group for demand += {(celeb,group)}
tag_group add group
// maintain fil_inv_member_2 for tag_group add group
for user in member{group}:
  fil_inv_member_2 += {(user,group)}
```

and the following maintenance for `inv_member += {(user,group)}`, i.e., for `member += {(group,user)}`:

```
if group in tag_group:
  fil_inv_member_2 += {(user,group)}
```

Similarly, tag sets and filtered relations are defined and maintained for `user`, and the original maintenance code is preceded with the test `user in tag_user`. Test to check that `l` in the update `field_loc += {(user,l)}` is in `tag_l` is not needed, because it is implied by `user in tag_user`.

We obtain the following filtered incremental maintenance code; the two comments indicate the changes from the maintenance code in Section 5.1.

```
if user in tag_user:                           // added
  if cond(l):
    for e in field_email{user}:
      for group in fil_inv_member_2{user}: // use filtered
        for celeb in demand.1{2=group}:
          for fs in field_followers{celeb}:
            if (fs,user) in member:
              r add (celeb,group,e)
```
■

This filtering based on demand can reduce both time and auxiliary space significantly, because `demand` is small compared to the entire data set.

- In the cost formulas for time, sizes of the relations used are replaced with sizes of the corresponding filtered relations. The resulting formulas are used to determine an optimal order of retrieving matching values.
- The auxiliary space used is also reduced, to be proportional to only elements of relations reachable from parameter values in `demand`, instead of all objects.

Example. Filtering reduces the cost of maintenance for update `field_loc += {(user,l)}` from O(`#inv_member{user}` $\times$ `#demand.1{2=group}`) to O(`#fil_inv_member_2{user}` $\times$ `#demand.1{2=group}`), which is $O(1)$ when `#demand` is $O(1)$, and reduces the auxiliary space from number of members of all groups to only members of the groups in `demand`. Relation `inv_member` is not used in incremental maintenance—it is dead and eliminated. ■

Note that filtered relations for member and fields, unlike for inverse member and inverse fields, could be omitted to save the space for storing them; the original member and field relations, together with the tag sets for the source vertices, can be used instead. This does not affect the asymptotic time complexities.

Organize all maintenance. To organize all maintenance for a query, to ensure that all invariants hold, even under arbitrary nesting and aliasing, four rules are used to properly order the maintenance blocks and add appropriate tests.

1. Maintenance of the query result and of all auxiliary values can be placed before or after the update to a relation.

   This is because the maintenance does not use the relation being updated but the element being added or deleted.

   We place the maintenance after an addition update and before a deletion update, for overall simpler ordering.

2. For an element addition update, the maintenance of the query result is placed after the maintenance of the auxiliary values used.

   This is because the element being added may join with elements to be added to filtered auxiliary values to yield elements to be added to the query result.

3. For an element deletion update, the maintenance of the query result is placed before the maintenance of the auxiliary values used.

   This is because the element being deleted may join with elements to be deleted from filtered auxiliary values to yield elements to be deleted from the query result.

4. If a relation `R` occurs in multiple membership clauses (same `field_f` or `member`) in the query, we augment the clause for each occurrence of `R` before the one for which we are generating a maintenance block, with a test to exclude the element being added or deleted.

This ensures that an update that may affect a query result through multiple membership clauses, under nesting or aliasing, has the correct overall effect—that is, contributes to the maintenance of the query result as a single update—after combining separate maintenance blocks for updates corresponding to each of the clauses.

An example is given in Appendix A of [26].

Complexity guarantees. Overall, our method provides a precise asymptotic complexity bound on the running time of the generated code as follows: for nested loops, multiply the cost associated with each `for` clause, i.e., the number of elements to iterate over, and the sum of the cost of evaluating other conditions and the result expression; for concatenated blocks, sum the cost of each block.

The space complexity for the auxiliary values is bounded asymptotically by the size of the given data, because tag sets are subsets of the given sets and objects, and filtered relations are subsets of the given set membership and object field relations. More precisely, this is bounded by only the number of elements of relations reachable from parameter values in `demand`, as opposed to the size of all given data.

6. PHASE 3: TRANSFORM BACK TO IMPLEMENTATIONS ON OBJECTS

Phase 3 transforms operations on relations in all maintenance code and query result retrieval back to operations on objects. Transformations for `field_f`, `member`, and query results are as in a prior work [48]; transformations for `demand`, tag sets, and filtered relations are new.

In each case, the operation is transformed to best use given object fields and sets in the original program as well as auxiliary maps for efficiency, and to add appropriate tests to ensure correctness in any context. Detailed transformations are given in Appendix B of [26]; we show the results here through the running example.

Example. For the running example, this yields, from the filtered maintenance code in Section 5.2, the following maintenance code for `field_loc += {(user,l)}`, i.e., for update `user.loc = l` where `user.loc` had no value before:

```
if user in tag_user:   // line 1 of maint code in Sec 5.2
  if cond(l):          // line 2
    if user hasfield email:          // line 3
      e = user.email                 //
      if user in fil_inv_member_2.keys():        // line 4
        for group in fil_inv_member_2.get(user): //
          if group in demand21.keys():         // line 5
            for celeb in demand21.get(group):  //
              if celeb hasfield followers:  // line 6
                fs = celeb.followers        //
                if fs isset:            // line 7
                  if user in fs:        //
                    r.cadd((celeb,group), e)  // line 8
```

and maintenance of auxiliary values at updates to `demand` and `member` as in Section 5.2, except for using `cadd` for `add`.

The retrieval of the query result is transformed into `r.get((celeb,group))`. ■

Constant-factor optimizations. Unnecessary use of counted sets can be eliminated, as discussed below, and type information and static analysis [13] can help remove the tests that use `hasfield`, `isset`, and `keys`, to give constant-factor improvements.

A main constant-factor optimization is counting elimination. Result relations in general need to be implemented using counted sets, i.e., having a count of how many times each element is added to the set. Doing this for all maintained results, including all auxiliary values, is an obvious overhead. We remove counts in several main cases:

- When all possible updates to a set are additions.
- When there are deletions, but there is only one possible combination of values of variables in the query for each element in the result set, based on two conditions:

  (1) each variable in the query is a parameter or a variable in the *result* expression, or has its value determined uniquely by such a variable by following `field_f` edges from the variable; and

  (2) the *result* expression returns different values for different combinations of values of variables in it, e.g., it is a variable or a tuple of variables.
- When it is any filtered relation.

7. EXPERIMENTAL EVALUATION

We implemented our method in a prototype system, IncOQ, available at http://github.com/IncOQ/, and performed a large number of experiments to evaluate the method and confirm the analyzed complexities, advantages, and trade-offs.

Our system takes programs written in Python as input, transforms queries and updates in the program according to our method, and outputs the resulting program. For queries in the input program, the system interprets all clauses of the form `for x in s` and `if x in s` in the queries as membership clauses, and the rest as condition clauses.

As part of the evaluation, we also experimented with two best available previous implementations of incremental object queries: Java Query Language (JQL) [55, 56] and Object-Set Queries (OSQ) [48], including porting OSQ from Python 2.5 to Python 3.4 for better comparison.

Except for experiments using JQL, all experiments were run using Python 3.4.2, under 64-bit Windows 8.1 on an Intel Core i5 4200M CPU at 2.50 GHz with 8GB memory. Each measured execution of a program occurred in its own process, with the garbage collector disabled. All reported times are CPU time, in seconds, as the mean of repeated runs until the standard deviation is less than 10% of the mean, with at least 10 runs. Additional space is the size of the additional sets and maps used; the size of a set is the number of elements, and the size of a map is the number of keys plus the size of the image set of each key.

All JQL experiments used JQL version 0.3.3, with Java 1.6.0_45, AspectJ 1.7.2, and ANTLR 3.0.1 and were run on the same machine as running Python. The CPU times reported are the mean of 50 repeated runs, as in [56].

7.1 Asymptotic performance, demand, and constant factors—via the running example query

We use the running example query to evaluate asymptotic performance, the effect of demand set size, and constant-factor optimizations, because the analyzed complexities and generated code have already been discussed in detail. Similar results were observed for other queries too.

Asymptotic time and space performance. We describe experiments using uniformly distributed random data because they are simpler and easier to understand, even tiny differences such as described in Footnote 1. We also experimented with Zipf distribution, well known for data studied in physical and social sciences, and observed similar asymptotic performance improvement.

Table 2 summarizes the analyzed time complexities for the running example query and update to a user location, for the realistic case of one user in `inv_field_followers{fs}` for each `fs` (i.e., each `followers` set `fs` belongs to one user) and a constant number of groups in `demand`. The update times for the incremental program and incremental program with filtering (called "filtered program" for brevity) correspond to the second order in Table 1, because the condition for it to have the minimum cost holds for the realistic case. The implemented heuristic chooses this order automatically.

program	query	update
original (Section 2)	$O($`#celeb.followers` $+$ `#group`$)$	$O(1)$
incremental (Section 5.1)	$O(1)$	$O($`#inv_member{user}` $\times$ `#demand.2{1=celeb}`$)$
filtered (Section 5.2)	$O(1)$	$O($`#fil_inv_member_1{user}` $\times$ `#demand.2{1=celeb}`$)$

Table 2: Analyzed time complexities.

The analyzed additional space is $O($`#inv_member`$)$ for the incremental program and $O($`#fil_inv_member_1`$)$ for the filtered program.

We use test data with up to 20,000 users and 1% as many groups. Each user follows 0.5% of all users, is in 5% of the groups, and is at one of 20 locations, one of which satisfies the location condition in the query. There are 3 special users and 1 group (i.e., 3 pairs, where each user is paired with the same group) in `demand`, where each special user has 0.5% of all users as followers (more followers only make the original query take linearly longer to run, not making the updates or other queries take longer). Thus, in the time complexity formula, `#celeb.followers`, `#group`, and `#inv_member{user}` are linear in the number of users, and `#fil_inv_member_1{user}` and `#demand.2{1=celeb}` are constants; in the space complexity formula, `#inv_member` is quadratic in the number of users, and `#fil_inv_member` is linear.

Figure 1-top shows the measured query and update times for 200,000 queries and 200,000 updates, respectively, obtained by measuring (1) the total time of 200,000 repeats of a query-update pair, querying on a random user-group pair in `demand` and updating the location of a random user, and (2) the total time without the queries, and subtracting (2) from (1). Figure 1-bottom shows the additional space measured at the end of (1). It confirms the analyzed complexities:

- The query time grows linearly with the number of users for the original program, and remains constant for the incremental and filtered programs.
- The update time is constant for the original program, grows linearly for the incremental program, and is constant albeit a larger constant for the filtered program.[1]
- The additional space is quadratic in the number of users for the incremental program, and linear for the filtered.

[1]There is actually a tiny increase for the original and filtered. It (and a tiny portion of the increase for the incremental) is due to increased cache misses in randomly accessing increasingly more users for location updates. We confirmed this by running experiments to measure cache misses.

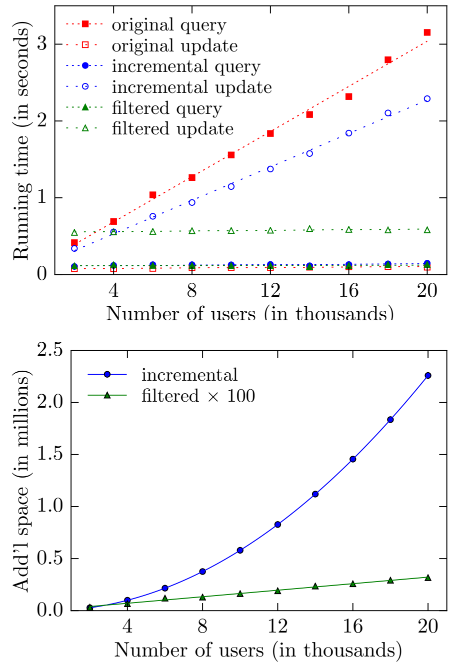


Figure 1: Running time and additional space for varying input size.

Effect of demand set size. We also evaluated the effect of demand set size on performance, as described in Appendix C of [26]. This helped confirm that filtering using demand provides significant gains when the demand set is small relative to the entire domain, but not otherwise.

Constant-factor optimizations. We further measured the benefit of eliminating unnecessary use of counted sets and of optimizations enabled by alias and type analysis, as described in [5]. We found that the former is generally significant, while the latter is relatively small.

7.2 Effect of query-update ratio, auxiliary indices, and other implementation factors — via JQL queries

We evaluated how query-update ratio affects the performance of our generated programs, using all three query benchmarks tested for incrementalization of object queries implemented in JQL [56].[2] Their three queries, in our syntax, are

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

We discovered, unexpectedly, that the JQL implementation, even though in Java, is significantly slower than our generated Python programs, and even asymptotically slower for join queries.

[2]JQL evaluation [56] also studies performance benefit on Robocode, a Java application. Our work studies queries in Python applications in Section 7.4.

We use the same setup as described in [56] for JQL experiments: 1000 objects in each source collection, performing 5000 operations, each being either a query or a random addition and removal of an element of `attends`.

Figure 2 shows the running times of the benchmark for the second query as the ratio of queries over updates increases: (1) the time of original Python program increases, similar to JQL with no caching, and (2) the time of incremental and filtered Python programs decrease, similar to JQL with always caching. The crossover point depends on the query and the implementation, but the incremental Python program outperforms all other programs significantly—it outperforms filtered here because there is only one fixed (reference) value for each query parameter (`attends`, etc.) in `demand`, and all objects in `attends` are in the tag set for `attends`, making the maintenance of tag sets and filtered relations unnecessary overhead.

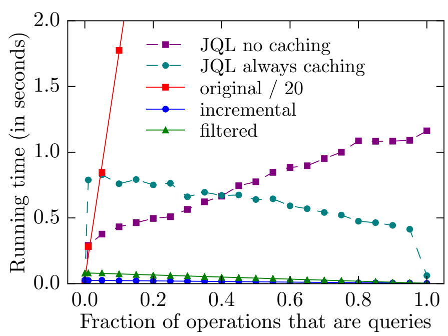


Figure 2: Running times in comparison with JQL on JQL benchmark 2 for varying query-update ratio.

Figure 2 also shows, unexpectedly, that (1) the incremental and filtered Python programs are faster than JQL with incremental caching, even though Python is much slower than Java, and (2) these Python programs appear even asymptotically faster. There is also a larger time advantage of our generated Python programs over JQL with caching for the third query, and smaller for the first. By examining the implementation of JQL, we believe that (1) the runtime overhead of AspectJ used to implement JQL contributed to a constant-factor slowdown, consistent with runtime overhead of dynamic methods in general, such as OSQ discussed in Section 7.3, and (2) JQL does an inefficient join with each additional source collection, yielding asymptotic slowdowns compared to our method of using efficient auxiliary indices. To confirm the asymptotic time difference, we modified the benchmarks to not vary the query-update ratio, but to increase the size of the source collections.

Figure 3 shows the running times of the modified benchmark for the second query, for collections of sizes 2,000 to 20,000, for equal numbers of queries and updates. It shows a continued increase by JQL, both with no caching and with always caching, whereas running times of both our generated programs stay constant. For the third query, the increase by JQL is even larger, whereas the times of both our programs again stay constant. For the first query, JQL programs are about 2-5 times slower than ours. We also confirmed that enabling garbage collection has little effect on the running times of our generated programs, with only an average variation of about 7%.

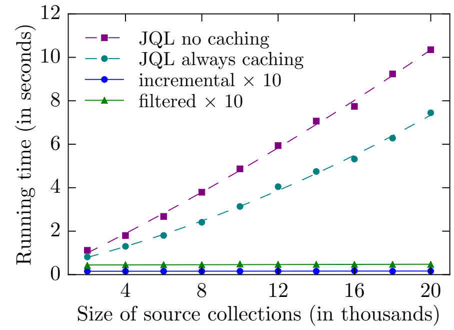


Figure 3: Running times in comparison with JQL on JQL benchmark 2 for varying input size (times of the original program was too large and thus omitted).

7.3 Runtime overhead and demand propagation strategies — via OSQ queries

We evaluated savings of runtime overhead by our method compared to a dynamic method, by comparing with OSQ for incremental object queries in Python [48]. OSQ generates similar code for maintaining the query result but uses dynamic assignment of obligations to objects to track demand and invoke the query result maintenance code, instead of using tag sets and filtered relations. We found that such a dynamic method is not only less efficient, but also error-prone, and hard to understand and optimize. The method in this paper can improve over OSQ asymptotically.

We implemented all the benchmarks of OSQ in [48] and compared the running times of our filtered programs and OSQ. We observed that our method produces the same asymptotic speedups as OSQ, but our filtered programs are consistently faster. For example, for the Django authorization query below, our filtered program is over 20% faster; other improvements measured are even larger, to over 50%.

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

More importantly, we determined that OSQ has a less refined strategy than our method for propagating demand for queries that involve intersection, and that strategy can yield asymptotically worse performance. For example, for our running example, OSQ uses only the first membership clause to filter using demand, not both clauses as our method does. However, we could not demonstrate the performance difference using the OSQ implementation on our running example, due to a bug we discovered in OSQ; we also found OSQ difficult to understand and improve. Instead, because our invariant-based method and implementation allow us to easily define and implement different strategies for filtering using demand, we implemented the OSQ strategy by defining it using invariants, and generated the corresponding filtered programs using our system.

We use the same setup as for Figure 1 except with each user in 5 groups and with all users in `demand`. Using our demand propagation strategy, only users in the single group in `demand` are in the tag set for `user`, and so maintenance is needed only for updates to these users, which total 500 on average (1% as many groups as users, with 5 groups per user). Using OSQ's strategy, all users following any user in `demand` are in the tag set for `user`, and so maintenance is run for updates to all such users.

Figure 4 shows the running times of our filtered program in comparison with the filtered program that uses OSQ's demand propagation strategy. It confirms that our method improves over OSQ asymptotically: the running time increases linearly for OSQ, but is constant for our generated program.

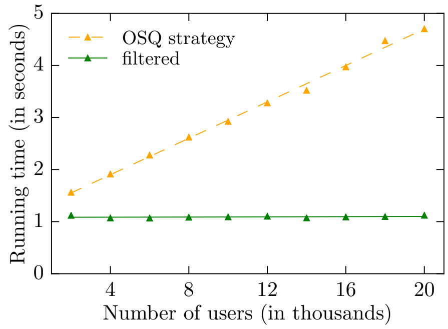


Figure 4: Running times in comparison with using OSQ's demand propagation strategy.

7.4 Transformation time and other measures — via access control, distributed algorithms, and probabilistic queries

We also applied our system to other examples, including the `CheckAccess` query and all 16 queries in Core RBAC [32], the `SSD` and `DSD` constraints in Constrained RBAC [12], and the most difficult queries in a set of distributed algorithms [31] and in approximate probabilistic inference [4, 52]. We summarize the most interesting results here.

Our filtered programs for RBAC automatically allow RBAC to be run with multiple instances, where arbitrary data can be passed around dynamically among the instances, and all operations of all instances are run incrementally. This was not possible previously [32]. The overhead of the filtered program when there is only one instance is small compared to our incremental program and the incremental program from [32], and in fact the tag sets and filtered relations help reduce incremental maintenance time. For example, for the `CheckAccess` query in Core RBAC, the time of incremental maintenance for creating and deleting sessions is reduced by about 67%.

Our filtered programs for queries in distributed algorithms are larger and somewhat slower than those generated using a previous system [31], mostly due to the generality of the new implementation, compared to the specialized, tediously manually written incrementalization rules used in [31]; we have not focused on reducing code size, because it does not affect the running time as much. For example, for Lamport's distributed mutual exclusion, the filtered program is 480 lines compared to 124 before, and it uses more CPU time than the incrementalized program from [31] by about 45% for 50 processes and less for more processes, but it still gains asymptotically over the original program.

Our incremental and filtered programs for queries in probabilistic inference are all constant time under every possible update that may be used in Markov chain Monte Carlo (MCMC) sampling [36], whereas the original queries take linear or quadratic time in the domain size.

Table 3 contains a list of programs for which we have done extensive evaluations to confirm analyzed performance and trade-offs. The original programs are small, at most 174 lines, but the incremental and filtered programs grow significantly, up to 2645 lines. Manually writing incremental and filtered programs is challenging, not only because of their size, but much more because of complex interactions among all demands and updates, typically scattered in many places. Every program where 1 to 8 queries are incrementalized is transformed in about a half minute or less, except that transforming Core RBAC and two distributed algorithms with 16 to 22 queries each takes longer but less than two and a half minutes; this is consistent with the transformation time being linear in program size.

8. RELATED WORK AND CONCLUSION

There is much previous work on incremental computation [44, 25], including demand-driven incremental computation, for lower-level programs, high-level relational queries, and object queries.

There are numerous methods for incremental computation in lower-level languages: memo functions [34], caching in functional programs [42], static incrementalization of recursive functions and loops [24], memoization and dynamic dependence graphs for functional and imperative programs [1], function caching in object-oriented programs [50], and making dependence-graph-based approaches demand-driven and composable [19], among others. These methods do not handle high-level queries under all updates that affect the query results

There has been significant work on incremental computation of high-level queries over sets and relations, in set languages, relational databases, and logic languages, e.g., [17]: using finite differencing rules to incrementalize set expressions [41], deriving production rules for incremental view maintenance [8], incremental computation of relational algebra expressions [43, 16], incremental view maintenance for recursive views [18], Datalog queries with complexity guarantees [27], demand-driven computation using magic sets (MST) [3], demand transformation (DT) with complexity guarantees [54], scale-independent relational queries [2], implementation in newer languages [37], and incremental view maintenance for nested queries [33], among others. These methods do not handle arbitrarily nested and aliased sets and objects.

Filtering using demand is similar to the idea underlying MST [3] and DT [54]. The difference, besides handling arbitrary sets and objects, is that our tag sets and filtered relations are asymptotically no larger than the given data, whereas MST and DT may use asymptotically larger space. For example, MST and DT may need $O(n_1 \times n_2)$ space to store the join of two relations of sizes $O(n_1)$ and $O(n_2)$, whereas our method requires at most $O(n_1 + n_2)$ space. The trade-off is that this stored join may help reduce the asymptotic query time more than our method does.

Incremental computation of object queries has been studied in object databases and object-oriented programming languages [49]. We discuss closest related work below. Note that our method allows powerful object queries to be written completely declaratively, contrasting previous work, as discussed in Section 2.

Caching the results of query functions in object databases, e.g., [21], allows the results to be reused when queries on the same keys are encountered again. The problem with caching for queries over sets of objects is that, unless all objects and sets used are taken as keys for cache lookup, which is not feasible in general, the cached results may become invalid

name	description	original (LOC)	query (count)	update (count)	inc. (LOC)	inc. trans. time (s)	filtered (LOC)	fil. trans. time (s)
Running	running example query	40	1	9	251	10.12	491	21.73
JQLbench1	JQL benchmark for query 1 [56]	41	1	3	101	2.02	136	3.50
JQLbench2	JQL benchmark for query 2 [56]	43	1	5	143	4.45	204	7.23
JQLbench3	JQL benchmark for query 3 [56]	46	1	6	162	5.97	249	10.41
WiFi	WiFi access point query [47]	28	1	6	217	10.95	369	19.16
Auth	Django authorization query [48]	34	1	8	232	18.72	468	33.92
Access	`CheckAccess` query in Core RBAC [32]	139	3	13	322	6.50	546	14.00
CoreRBAC	all queries in Core RBAC [32]	139	16	21	1074	39.83	2645	141.31
SSD	SSD constraint in Constrained RBAC [12]	29	3	8	295	7.20	348	8.89
La mutex	Lamport's distributed mutual exclusion [23]	44	6	6	403	9.31	480	11.50
RA mutex	Ricart-Agrawala's distributed mutual exclusion [45]	55	8	4	683	18.03	804	23.11
RA token	Ricart-Agrawala's token-based mutual exclusion [46]	67	8	8	N/A	N/A	967	31.30
SK token	Suzuki-Kasami's token-based mutual exclusion [53]	72	6	11	N/A	N/A	760	22.75
CR leader	Chang-Robert's leader election [9]	30	2	1	81	1.34	114	1.69
HS leader	Hirschberg-Sinclair's leader election [20]	51	2	2	135	1.84	170	2.23
2P commit	Two-phase commit [15]	82	16	6	1137	32.20	1345	43.11
DS crash	Dolev-Strong's consensus under crash failures [10]	52	6	8	N/A	N/A	626	17.53
CL Paxos	Castro-Liskov's Paxos under Byzantine failures [7]	174	22	14	1517	64.97	1624	74.47
Birthday	Birthday Collision [52]	14	4	4	173	4.77	201	5.55
Cite author	Publication Citation Matching author query [52]	13	2	3	59	1.47	87	1.92
Cite exist	Publication Citation Matching existential query [52]	19	2	3	65	1.81	93	2.12

For `CheckAccess` in Core RBAC, the original LOC is for the entire Core RBAC, and only queries needed for the `CheckAccess` function are incrementalized.
For distributed algorithms, the original LOC is for queries and updates after quantifications are transformed into aggregate queries and nested queries using the method in [31] and its interface with our system. Two of the algorithms are not listed here due to incomplete interface in transforming witness for existential quantifications. The N/A entries for three of the algorithms are due to incomplete interface for transforming set values sent in messages; this does not affect generation of filtered programs because demand-driven queries can be computed after the complete set value is received.
For probabilistic queries, the original LOC is for queries and updates after transforming quantifications into aggregate and nested queries.

Table 3: Program and transformation statistics: size of original, incremental, and filtered programs in lines of code (LOC); number of queries incrementalized and of updates to those queries; and transformation time to generate the incremental and filtered programs, respectively.

and must be discarded when any of the values not taken as keys are updated. In contrast, our method maintains the cached results incrementally as values used in the queries are updated.

Incremental view maintenance using update propagation, e.g., [22, 38], dynamically tracks and propagates changes. A particularly clean method [38] was implemented using Smalltalk: it translates all values into collections, including scalar values into singleton sets, and object values into sets of attribute name-value pairs, and handles all collections and updates uniformly; it gives no cost analysis, although some other work does [22]. However, these methods do not define invariants for propagating demand and change or for creating indices, making the formulation of those complex mechanisms ad hoc. These dynamic methods also do not generate incremental maintenance code, leading to the unnecessary overhead and hard-to-predict behavior characteristic of dynamic methods.

Liu et al. [28] studies incrementalization for object-oriented programs. It transforms expensive queries and updates by applying manually written incrementalization rules from a library of rules. How to automatically generate the incremental maintenance code needed in such rules was left open; it was later studied for relational queries, in implementing core RBAC [32], but that method does not handle nested and aliased sets and objects. Our method translates nested and aliased sets and objects into flat field and member relations as in [48], not into nested sets with attribute name-value pairs for objects as in [38], because the former allows simpler and more efficient implementations, essentially as in column databases [51].

Rothamel and Liu [48] generates code for incremental maintenance of query results for arbitrary objects and sets, but tracks demand and maintains indices using dynamic mechanisms, and propagates change by dynamically assigning and invoking maintenance code as obligations on objects. The complex dynamic mechanisms made it exceedingly difficult to understand the method, let alone predict its performance and improve it. Our method of defining everything using invariants allowed us to generate complete code, provide precise complexity guarantees, and develop a refined demand mechanism that gives asymptotic improvement over [48], as shown in Section 7.3.

JQL extends Java with object queries [55, 56]. The class of queries and updates it incrementalizes is very restricted: only queries over source sets of objects with tests against fields of these objects, and only updates to the source sets and fields of the element objects. Implemented using AspectJ, the method used for join queries requires iterating over entire sets, not just changes determined using incrementally maintained indices as in our method, and thus yields asymptotically slower programs than our method, as shown in Section 7.2. A recent work builds on JQL by adding more sophisticated query planning and by caching intermediate results using various strategies [40, 39]; it shows constant-factor improvements but has the same lack of auxiliary indices that leads to asymptotic slowdowns.

In conclusion, by establishing invariants for not only query results but also auxiliary values for tracking demand and propagating change, our method can generate complete implementations of demand-driven incremental queries with precise complexity guarantees. Additional details and extensions can be found in [6]. Future work includes more efficient range queries, optimal selection of specialized incremental maintenance when appropriate, improved static analysis for reducing constant factors, parallel computation for independent maintenance of query results, and asynchronous maintenance of auxiliary indices.

Acknowledgment. We thank Xuetian Weng for help making the JQL implementation work and for experiments using the Zipf distribution.

9. REFERENCES

[1] U. A. Acar. Self-adjusting computation: (an overview). In *Proc. of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 1–6, 2009.

[2] M. Armbrust, E. Liang, T. Kraska, A. Fox, M. J. Franklin, and D. A. Patterson. Generalized scale independence through incremental precomputation. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 625–636, 2013.

[3] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proc. of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 1–16, 1986.

[4] BayesianLogic. The BLOG Programming Language examples. https://github.com/BayesianLogic/blog/tree/master/example. Last accessed May 21, 2016.

[5] J. Brandvein and Y. A. Liu. Removing runtime overhead for optimized object queries. In *Proc. of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 73–84, 2016.

[6] J. G. Brandvein. *Demand-Driven Incremental Computation of Object Queries*. PhD thesis, Computer Science Department, Stony Brook University, May 2016. http://www3.cs.stonybrook.edu/~jbrandve/thesis.pdf.

[7] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20:398–461, 2002.

[8] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proc. of the 17th International Conference on Very Large Data Bases*, pages 577–589. Morgan Kaufman, 1991.

[9] E. J. H. Chang and R. Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 22(5):281–283, 1979.

[10] D. Dolev and H. R. Strong. Authenticated algorithms for Byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

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

[12] M. Gorbovitski, Y. A. Liu, S. D. Stoller, and T. Rothamel. Composing transformations for instrumentation and optimization. In *Proc. of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 53–62, 2012.

[13] M. Gorbovitski, Y. A. Liu, S. D. Stoller, T. Rothamel, and T. Tekle. Alias analysis for optimization of dynamic languages. In *Proc. of the 6th Symposium on Dynamic Languages*, pages 27–42. ACM Press, 2010.

[14] D. Goyal and R. Paige. A new solution to the hidden copy problem. In *Proc. of the 5th International Symposium on Static Analysis*, pages 327–348. Springer, 1998.

[15] J. Gray. Notes on Data Base Operating Systems. In *Advanced Course: Operating Systems*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481, 1978.

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

[17] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. In *Materialized Views: Techniques, Implementations, and Applications*, pages 145–157. MIT Press, 1999.

[18] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 157–166, 1993.

[19] M. A. Hammer, Y. P. Khoo, M. Hicks, and J. S. Foster. Adapton: Composable, demand-driven incremental computation. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[20] D. S. Hirschberg and J. B. Sinclair. Decentralized extrema-finding in circular configurations of processors. *Communications of the ACM*, 23(11):627–628, 1980.

[21] A. Kemper, C. Kilger, and G. Moerkotte. Function materialization in object bases. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 258–267, 1991.

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

[23] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, 1978.

[24] Y. A. Liu. Efficiency by incrementalization: An introduction. *Higher-Order and Symbolic Computation*, 13(4):289–313, 2000.

[25] Y. A. Liu. *Systematic Program Design: From Clarity To Efficiency*. Cambridge University Press, 2013.

[26] Y. A. Liu, J. Brandvein, S. D. Stoller, and B. Lin. Demand-driven incremental object queries. *Computing Research Repository*, arXiv:1511.04583 [cs.PL], Nov. 2015 (Revised July 2016). http://arxiv.org/abs/1511.04583.

[27] Y. A. Liu and S. D. Stoller. From Datalog rules to efficient programs with time and space guarantees. *ACM Transactions on Programming Languages and Systems*, 31(6):1–38, 2009.

[28] Y. A. Liu, S. D. Stoller, M. Gorbovitski, T. Rothamel, and Y. E. Liu. Incrementalization across object

abstraction. In *Proc. of the 20th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 473–486, 2005.

[29] Y. A. Liu, S. D. Stoller, and B. Lin. High-level executable specifications of distributed algorithms. In *Proc. of the 14th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 95–110. Springer, 2012.

[30] Y. A. Liu, S. D. Stoller, and B. Lin. From clarity to efficiency for distributed algorithms. *Computing Research Repository*, arXiv:1412.8461 [cs.PL], Dec. 2014 (Revised January 2016). http://arxiv.org/abs/1412.8461.

[31] Y. A. Liu, S. D. Stoller, B. Lin, and M. Gorbovitski. From clarity to efficiency for distributed algorithms. In *Proc. of the 27th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 395–410, 2012.

[32] Y. A. Liu, C. Wang, M. Gorbovitski, T. Rothamel, Y. Cheng, Y. Zhao, and J. Zhang. Core role-based access control: Efficient implementations by transformations. In *Proc. of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 112–120, 2006.

[33] D. Lupei, C. Koch, and V. Tannen. Incremental View Maintenance for Nested-Relational Databases. *Computing Research Repository*, arXiv:1412.4320 [cs.DB], Dec. 2014. http://arxiv.org/abs/1412.4320.

[34] D. Michie. “Memo” functions and machine learning. *Nature*, 218:19–22, 1968.

[35] B. Milch, B. Marthi, S. Russell, D. Sontag, D. L. Ong, and A. Kolobov. BLOG: Probabilistic Models with Unknown Objects. In *Introduction to Statistical Relational Learning*, pages 373–399. MIT Press, 2007.

[36] B. Milch and S. Russell. Extending Bayesian networks to the open-universe case. In *Heuristics, Probability and Causality: A Tribute to Judea Pearl. College Publications*. College Publication, 2010.

[37] R. Mitschke, S. Erdweg, M. Köhler, M. Mezini, and G. Salvaneschi. i3ql: Language-integrated live data views. In *Proc. of the ACM International Conference on Object-Oriented Programming Systems Languages and Applications*, pages 417–432, 2014.

[38] H. Nakamura. Incremental computation of complex object queries. In *Proc. of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 156–165, 2001.

[39] V. Nerella, S. Madria, and T. Weigert. Efficient caching and incrementalization of object queries on collections in programming codes. In *Proc. of the IEEE 38th Annual Computer Software and Applications Conference*, pages 229–238, July 2014.

[40] V. K. S. Nerella, S. Surapaneni, S. K. Madria, and T. Weigert. Exploring optimization and caching for efficient collection operations. *Automated Software Engineering*, pages 3–40, 2014.

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

[42] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 315–328, 1989.

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

[44] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages*, pages 502–510, 1993.

[45] G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 1981.

[46] G. Ricart and A. K. Agrawala. Author's response to 'On Mutual Exclusion in Computer Networks' by Carvalho and Roucairol. *Communications of the ACM*, 26(2):147–148, 1983.

[47] T. Rothamel. *Automatic Incrementalization of Queries in Object-Oriented Programs*. PhD thesis, Computer Science Department, Stony Brook University, 2008.

[48] T. Rothamel and Y. A. Liu. Generating incremental implementations of object-set queries. In *Proc. of the 7th International Conference on Generative Programming and Component Engineering*, pages 55–66. ACM Press, 2008.

[49] G. Salvaneschi and M. Mezini. Reactive behavior in object-oriented applications: an analysis and a research roadmap. In *Proc. of the 12th Annual International Conference on Aspect-Oriented Software Development*, pages 37–48. ACM Press, 2013.

[50] A. Shankar and R. Bodík. DITTO: Automatic incrementalization of data structure invariant checks (in Java). In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 310–319, 2007.

[51] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: a column-oriented DBMS. In *Proc. of the 31st International Conference on Very Large Data Bases*, pages 553–564. VLDB Endowment, 2005.

[52] Stuart Russel's Group at UC Berkeley. Bayesian logic (BLOG) examples. https://sites.google.com/site/bloginference/example-usage. Last accessed May 21, 2016.

[53] I. Suzuki and T. Kasami. A distributed mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 3(4):344–349, 1985.

[54] K. T. Tekle and Y. A. Liu. More efficient Datalog queries: Subsumptive tabling beats magic sets. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 661–672, 2011.

[55] D. Willis, D. J. Pearce, and J. Noble. Efficient object querying for java. In *Proc. of the 20th European Conference on Object-Oriented Programming*, pages 28–49. Springer, 2006.

[56] D. Willis, D. J. Pearce, and J. Noble. Caching and incrementalisation in the Java query language. In *Proc. of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, pages 1–18, 2008.