July 14, 2001

RT0420

Computer Science 10 pages

Research Report

Incremental Computation of Complex Object Queries

Hiroaki Nakamura

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

Incremental Computation of Complex Object Queries

Hiroaki Nakamura
IBM Research, Tokyo Research Laboratory
1623-14 Shimotsuruma, Yamato-shi, Kanagawa-ken 242-8502 Japan
hnakamur@jp.ibm.com

ABSTRACT

The need for incremental algorithms for evaluating database queries is well known, but constructing algorithms that work on object-oriented databases (OODBs) has been difficult. The reason is that OODB query languages involve complex data types including composite objects and nested collections. As a result, existing algorithms have limitations in that the kinds of database updates are restricted, the operations found in many query languages are not supported, or the algorithms are too complex to be described precisely.

We present an incremental computation algorithm that can handle any kind of database updates, can accept any expressions in complex query languages such as OQL, and can be described precisely. By translating primitive values and records into collections, we can reduce all query expressions into ones composed of only one kind of operation, namely comprehension. This makes the problems with incremental computation less complicated and thus allows us to describe the algorithm precisely. Our incremental algorithm consists of two parts: one is to maintain the consistency in each comprehension occurrence and the other is to update the value of an entire expression. The algorithm is so flexible that we can use strict updates, lazy updates, and their combinations. By comparing the performance of applications built with our mechanism and that of equivalent hand written update programs, we show that our incremental algorithm can be implemented efficiently.

1. INTRODUCTION

Incremental computation has been studied in many fields [19]. The underlying idea is that we can improve the performance of a function by computing an output using the result of a previous computation whose input is slightly different from the current one. Incremental computation is also receiving attention in the database community [11]. It is normally cheaper to compute the changes to a query answer in response to changes to the database than to compute the entire answer from scratch.

Although many incremental algorithms have been proposed for relational databases (RDBs) [12], we cannot directly apply them to object-oriented databases. The reason is OODBs must be able to handle complex data structures including composite objects and nested collections while RDBs handle only flat tables. Queries of OODBs are also different from those of RDBs in that they are recursively composed of sub-queries so that they can traverse the complex data structures.

Since OODBs must support complex data types and operations, which can be composed recursively, incremental algorithms for OODB queries are inherently complicated. A straightforward implementation would produce infinite combinations of cases. Thus existing incremental algorithms suffer from the following deficiencies: (1) the kinds of database updates are restricted, (2) the operations found in many query languages are not supported, and (3) the algorithms are presented so informally that we cannot check their correctness or completeness.

In this paper we will present an incremental computation algorithm for OODB queries in which (1) any kinds of database updates can be handled, (2) any expressions in complex query languages such as ODMG OQL [4] can be accepted, and (3) the algorithm can be precisely represented. The contributions of this paper are: (1) the technique for translating various types of operations in OODB query languages into single operation, (2) the algorithm that computes differential elements for collection expressions correctly and efficiently, (3) the flexible incremental evaluation method in which we can use strict updates, lazy updates, and their combinations, and (4) the demonstration that the combination of the above can be implemented efficiently.

2. COMPREHENSION QUERY NOTATION

A clear distinction between a query language and a programming language is that the former must allow us to describe operations on collections concisely. A literature review [21] covers a wide range of languages for collection types. We use a query notation based on *comprehensions* [22, 3] as the basis of our algorithm. In particular we will employ the monoid comprehension calculus [6], in which multiple collection types and aggregate operations can be treated uniformly.

A set comprehension $\{x * x | x \in S \wedge x > 0\}$ describes the set of squares of all the positive numbers in a set S. The resulting set is obtained by evaluating $x * x$ for each $x \in S$ where $x > 0$ holds.

This notation leads us to a collection query language that takes the form:

$$\begin{array}{rcll} e & ::= & \ldots & \\ & | & \{e_1, \ldots, e_n\} & (n \geq 0) \\ & | & \{e | q_1, \ldots, q_n\} & (n \geq 0) \end{array}$$

in which e is an expression and q_i is a *qualifier*. A qualifier is either

- a *generator* of the form $v \leftarrow e'$, where v is a variable that ranges over the elements of e', or
- a *filter*, which is a boolean valued expression.

A comprehension is defined by the following equations:

$$\begin{array}{rl} \{e|\} = & \{e\} \\ \{e | v \leftarrow e', q_1, \ldots, q_n\} = & \text{if } e' = \{\} \\ & \quad \text{then } \{\} \\ & \quad \text{else } C(a_1) \cup \ldots \cup C(a_m) \\ & \qquad \text{where } C(v) = \{e | q_1, \ldots, q_n\}, \\ & \qquad\quad e' = \{a_1, \ldots, a_m\} \\ \{e | filter, q_1, \ldots, q_n\} = & \text{if } filter \\ & \quad \text{then } \{e | q_1, \ldots, q_n\} \\ & \quad \text{else } \{\} \end{array}$$

The following are examples of comprehension expressions:

$$\begin{array}{rcl} select(f, e) & = & \{x | x \leftarrow e, f(x)\} \\ project(f, e) & = & \{f(x) | x \leftarrow e\} \\ flatten(e) & = & \{x | s \leftarrow e, x \leftarrow s\} \end{array}$$

A comprehension with multiple generators is equivalent to a nested loop in an imperative programming language. Thus $flatten(\{\{1, 3\}, \{5\}\})$ iterates over $\{1, 3\}$ and $\{5\}$, which results in the set $\{1, 3, 5\}$.

We can treat other types of collection such as *bag* in the same framework. In addition, aggregate operations such as *sum*, *and*, and *or* can also be captured by comprehensions. Each of the data types can be understood as a *monoid*, an algebraic structure that is a pair of an associative *merge* operator and the left and right *identity* of the operator. Table 1 shows the monoids we use in this paper. A monoid is *idempotent* when $\forall x. x \circ x = x$.

The definition of comprehension above can easily be extended so that it can handle other monoids by replacing

	$\mathcal{M}$	merge	identity	idempotent
collection	*set*	union	empty set	$\surd$
monoid	*bag*	additive union	empty bag	
primitive	*sum*	$+$	0	
monoid	*and*	$\wedge$	true	$\surd$
	or	$\vee$	false	$\surd$

Table 1: Monoid Examples

$\cup$ and $\{\}$ with their merge operators and identities. To distinguish the monoids we describe them with prefixes that specify collection types or aggregate operations. For example, $sum\{x | x \leftarrow bag\{1, 2\}\}$ computes the integer value 3. When no confusion results, we will use $\{\ldots\}$ for $set\{\ldots\}$.

Other examples are:

$$\begin{array}{rcl} length(e) & = & sum\{1 | x \leftarrow e\} \\ includes(e, a) & = & or\{a = x | x \leftarrow e\} \\ intersect(e_1, e_2) & = & \{x_1 | x_1 \leftarrow e_1, or\{x_2 = x_1 | x_2 \leftarrow e_2\}\} \end{array}$$

A comprehension language can capture most features of OODB query languages when it is equipped with primitive values and records in addition to collections. The language must also support operations on such data types. The language now takes the form:

$$\begin{array}{rcll} e & ::= & \ldots & \\ & | & c & \text{constant} \\ & | & v & \text{variable} \\ & | & e_1 + e_2 & \text{also for } *, =, <, \cdots \\ & | & <l_1 = e_1, \ldots, l_n = e_n> & \text{record construction} \\ & | & e.l & \text{record projection} \\ & | & \mathcal{M}\{e_1, \ldots, e_n\} & \text{monoid} \\ & | & \mathcal{M}\{e | q_1, \ldots, q_n\} & \text{monoid comprehension} \\ & & & \mathcal{M} \in \{set, bag, sum, and, or\} \end{array}$$

This can serve as an intermediate language for other query languages such as OQL, a standardized OODB query language. The query *"select the employees whose family incomes are higher than $50,000"* is represented in OQL as follows:

```
select e
from e in Employees
where sum(select m.salary
          from m in e.family) + e.salary > 50000
```

This OQL query is translated into the following comprehension expression:

$$\begin{array}{l} bag\{e | e \leftarrow Employees, \\ \qquad sum\{m.salary | m \leftarrow e.family\} + e.salary > 50000\} \end{array}$$

Most OQL expressions have a direct translation into comprehension expressions [6, 10]. Relational algebra is a special case of comprehensions in which the values are restricted to sets of records of primitives.

3. PROBLEMS WITH INCREMENTAL COMPUTATION

A comprehension language can serve as a foundation for incremental computation of expressions that involve collections. Assume we have

$$f(E) \quad = \quad \{x * x | x \leftarrow E\}$$

When the new elements ΔE are added to E, we can obtain the new value of $f(E_{new})$, where $E_{new} = E \cup \Delta E$, by combining the previous result $f(E)$ and the value of $f(\Delta E)$ as follows:

$$f(E_{new}) \quad = \quad f(E) \cup f(\Delta E)$$

This implies that we can reuse the value of $f(E)$ to compute $f(E_{new})$.

However, we cannot apply the idea to all the cases for two major reasons. One reason is that we have to handle primitives and records as well as collections, which prevents us from using the same idea for incremental computation.

The other reason is that we cannot handle deletion of elements with the same approach. For example, assume we have $E = \{-4, 1, 4\}$ and $\Delta E = \{4\}$. Then $f(E) = \{16, 1\}$ and $f(\Delta E) = \{16\}$, thus $f(E) - f(\Delta E) = \{1\}$. However $f(E_{new})$, where $E_{new} = E - \Delta E$, is $\{16, 1\}$, which does not satisfy $f(E_{new}) = f(E) - f(\Delta E)$. Even if we use a bag, which satisfies the equation above, computing $f(\Delta E)$ from scratch is often too expensive because the elements of $f(\Delta E)$ will not be used in $f(E_{new})$.

We will address the first problem in Section 4 and the second problem in Section 5.

4. TRANSLATION INTO COLLECTIONS

Primitive Values

When a primitive value a is changed into a' in an expression $g(a)$, we can obtain the new value $g(a')$ by first cancelling the effect caused by the old value a and then introducing the effect caused by the new value a'. For example, when $a = 2$, the expression $bag\{x * 2 | x \leftarrow bag\{1, a\}\}$ yields $bag\{2, 4\}$. When the value of a is changed to 3, the new value of the expression can be computed by first removing 4, which is the effect of the old value of a, and then adding 6, which is the effect of the new value. The result is $bag\{2, 6\}$.

This observation leads us to represent a primitive value as a set whose only element is the primitive value. A change of the value is realized in two steps: (1) remove the old value from the set and (2) add the new value to the set. Given the value representation as a set S, the first step is realized by:

$$S_1 \quad = \quad S - \{e_{old}\}$$

The second step is

$$S_2 \quad = \quad S_1 \cup \{e_{new}\}$$

We also have to modify the operations on primitive values so that they take and produce sets. The modified operator that corresponds to $'+'$ has to be:

$$add(X_1, X_2) = \begin{cases} \{\} & \text{if } X_1 = \{\} \\ \{\} & \text{if } X_2 = \{\} \\ \{x_1 + x_2\} & \text{otherwise} \\ \quad \text{where } \{x_1\} = X_1, & \\ \quad\quad\quad\quad \{x_2\} = X_2 & \end{cases}$$

This can be represented by the following comprehension:

$$\{x_1 + x_2 | x_1 \leftarrow X_1, x_2 \leftarrow X_2\}$$

Note that since X_1 and X_2 are singletons or empty sets, the result has one element at most. The first example in this section is expressed as follows:

$$bag\{\{x_1 * x_2 | x_1 \leftarrow x, x_2 \leftarrow \{2\}\} | x \leftarrow bag\{\{1\}, \{a\}\}\}$$

This yields $bag\{\{2\}, \{4\}\}$ when $a = 2$, which becomes $bag\{\{2\}, \{\}\}$ in the intermediate state, and finally becomes $bag\{\{2\}, \{6\}\}$ when $a = 3$

Records

We can also encode records into collections. The basic idea is to represent a record as a set of pairs, each of which consists of a field name and its value[1]. For example, the record $< name = ''John'', age = 32 >$ can be represented by $r = \{(name, \{''John''\}), (age, \{32\})\}$. The value of age can be obtained by a comprehension as follows:

$$\{x | (l, a) \leftarrow r, l = age, x \leftarrow a\}$$

Figure 1 shows the entire encoding algorithm in terms of the function $\mathcal{E}[\![.]\!]$. Although the sets for representing primitive values and records are nothing special in that they obey all the axioms of sets, we denote them by set_p and set_r so that the inverse function $\mathcal{D}[\![.]\!]$ in Figure 2 can uniquely decode values.

Booleans

For the translated language in Figure 1, since the booleans are now represented by {true} and {false}, the definition of a filter should be changed as follows:

$$\{e | filter, q_1, \ldots, q_n\} \;=\; \text{if } filter = \{true\} \text{ then } \{e | q_1, \ldots, q_n\} \text{ else } \{\}$$

[1]A pair (a,b) can be represented by a set {{a},{a,b}} (Kuratowski's ordered pair). This makes pairs unnecessary, but we use them for readability.

$$
\begin{array}{rcll}
\mathcal{E}[\![c]\!] & = & set_p\{c\} & \\
\mathcal{E}[\![v]\!] & = & v & \\
\mathcal{E}[\![e_1 + e_2]\!] & = & set_p\{x_1 + x_2 | x_1 \leftarrow \mathcal{E}[\![e_1]\!], x_2 \leftarrow \mathcal{E}[\![e_2]\!]\} & \text{also for } *, =, <, \cdots \\
\mathcal{E}[\![< l_1 = e_1, \ldots, l_n = e_n >]\!] & = & set_r\{(l_1, \mathcal{E}[\![e_1]\!]), \ldots, (l_n, \mathcal{E}[\![e_n]\!])\} & \\
\mathcal{E}[\![e.l]\!] & = & \mathcal{E}[\![\mathcal{M}\{x | (l', a) \leftarrow \mathcal{E}[\![e]\!], l = l', x \leftarrow a\}]\!] & \mathcal{M} \text{ is the type of } a \\
\mathcal{E}[\![\mathcal{M}\{e_1, \ldots, e_n\}]\!] & = & \begin{cases} \mathcal{M}\{\mathcal{E}[\![e_1]\!], \ldots, \mathcal{E}[\![e_n]\!]\} & \text{for collection types} \\ set_p\{\mathcal{M}\{\mathcal{E}[\![e_1]\!], \ldots, \mathcal{E}[\![e_n]\!]\}\} & \text{for aggregate operations} \end{cases} & \\
\mathcal{E}[\![\mathcal{M}\{e | q_1, \ldots, q_n\}]\!] & = & \begin{cases} \mathcal{M}\{e | \mathcal{Q}[\![q_1]\!], \ldots, \mathcal{Q}[\![q_n]\!]\} & \text{for collection types} \\ set_p\{\mathcal{M}\{e | \mathcal{Q}[\![q_1]\!], \ldots, \mathcal{Q}[\![q_n]\!]\}\} & \text{for aggregate operations} \end{cases} & \\
\\
\mathcal{Q}[\![v \leftarrow e]\!] & = & v \leftarrow \mathcal{E}[\![e]\!] & \\
\mathcal{Q}[\![e]\!] & = & \mathcal{E}[\![e]\!] &
\end{array}
$$

Figure 1: Encoder

$$
\begin{array}{rcll}
\mathcal{D}[\![set_p\{c\}]\!] & = & c & \\
\mathcal{D}[\![set_r\{(l_1, v_1), \ldots, (l_n, v_n)\}]\!] & = & < l_1 = \mathcal{D}[\![v_1]\!], \ldots, l_n = \mathcal{D}[\![v_n]\!] > & \\
\mathcal{D}[\![\mathcal{M}\{v_1, \ldots, v_n\}]\!] & = & \mathcal{M}\{\mathcal{D}[\![v_1]\!], \ldots, \mathcal{D}[\![v_n]\!]\} & (\mathcal{M} \neq set_p \text{ or } set_r)
\end{array}
$$

Figure 2: Decoder

If we replace $\{false\}$ with $\{\}$, the definition becomes:

$$
\begin{array}{rcl}
\{e | filter, q_1, \ldots, q_n\} & = & \text{if } filter = \{\} \\
& & \quad \text{then } \{\} \\
& & \quad \text{else } \{e | q_1, \ldots, q_n\} \\
& = & \text{if } filter = \{\} \\
& & \quad \text{then } \{\} \\
& & \quad \text{else } C(a) \\
& & \qquad \text{where } C(v) = \{e | q_1, \ldots, q_n\}, \\
& & \qquad\qquad \{a\} = filter \\
& = & \{e | v \leftarrow filter, q_1, \ldots, q_n\}
\end{array}
$$

In this way we can implement filters as generators, which makes the equation for filters unnecessary in the definition of comprehensions. Note that the iteration variable v (or the value of a) is never used in q_i.

5. INCREMENTAL COMPUTATION OF COMPREHENSION INSTANCES

Comprehension Instances

We will introduce the notion of a *comprehension instance* for separating an expression from its value. A comprehension instance C is an occurrence of a comprehension expression in an environment, an association of values with variables. So far we have not clearly distinguished between an expression and the value it denotes. However, since expressions can have free variables, their values vary depending on the environments in which they are evaluated. We describe the value of C as $C.value$ and the environment of C as $C.env$.

Let C be a comprehension instance for $\{e | v \leftarrow e', q_1, \ldots, q_n\}$, E be that for e', and S_i be that for $\{e | q_1, \ldots, q_n\}$. Then the following equations hold:

$$
\begin{array}{rcl}
C.value & = & \text{if } E.value = \{\} \\
& & \quad \text{then } \{\} \\
& & \quad \text{else } S_1.value \cup \ldots \cup S_m.value \\
E.env & = & C.env \\
S_i.env & = & C.env \cup \{(v, a_i)\} \\
& & \quad \text{where } E.value = \{a_1, \ldots, a_m\}
\end{array}
$$

Therefore we can keep the value of $C.value$ up to date by modifying it in response to the changes in $E.value$ and $S_i.value$.

Bag Comprehensions

We first look into the method for computing a bag comprehension incrementally. The discussion can be generalized to any comprehensions whose underlying monoids are anti-idempotent. As we have mentioned in Section 3, the incremental computation of a bag comprehension is easier than that of a set comprehension because a bag allows elements to be duplicated but a set does not.

The problem in maintaining a bag comprehension is when we remove a_k from $E.value$, creating a comprehension instance S_k for a_k to compute

$$C.value_{new} \quad = \quad C.value - S_k.value$$

is often too expensive because S_k is discarded just after $S_k.value$ is used.

We solve this problem by retaining the mapping between a_i and S_i, and reusing S_i when a_i is removed. For that purpose we use an auxiliary data structure *C.map* such that

$$C.map \;=\; bag\{(a, S \text{ such that } (v,a) \in S.env) | a \leftarrow E.value\}$$

Then *C.value* can be obtained as

$$C.value \;=\; bag\{x | (a,S) \leftarrow C.map, x \leftarrow S.value\}$$

When a_k is removed from *E.value*, the elements to be removed from *C.value* can be obtained as

$$\bar{\Delta}(C.value) \;=\; lookup(C.map, a_k).value$$

where $lookup(m,x)$ selects from m one element (a,b) such that $x = a$, and gives b. By defining *C.value* and its differential elements $\bar{\Delta}(C.value)$ in terms of *C.map*, we can avoid computing S_k multiple times.

We also maintain the following relationships so that we can propagate changes among comprehension instances and can ensure the consistency of entire expressions.

$$\begin{aligned} E &\in C.components \\ S_i &\in C.components \\ C &\in E.dependents \\ C &\in S_i.dependents \end{aligned}$$

Set Comprehensions

For a set comprehension, we also use the auxiliary data structure *C.map* so that we can avoid unnecessary computation in deleting elements. Since the value of a set comprehension is a set, *C.value* is obtained as

$$C.value \;=\; set\{x | (a,S) \leftarrow C.map, x \leftarrow S.value\}$$

Unlike a bag comprehension, a set comprehension requires us to manage the input elements that yield the same value. In the example in Section 3, since both -4 and 4 yielded the same value 16, we could not exclude 16 from the result when deleting the input element 4. However, if we delete both -4 and 4, we have to exclude 16 from the result.

To handle those situations correctly, we use another auxiliary data structure *C.counts* such that

$$C.counts \;=\; set\{(x, count(C.map, x)) | x \leftarrow C.value\}$$

The function $count(m,x)$ is defined as follows:

$$count(m,x) = sum\{1 | (a,S) \leftarrow m, y \leftarrow S.value, x = y\}$$

C.counts maintains the mapping between an element in the result *C.value* and the number of duplicates that yielded the element.

When a_k is removed from *E.value*, the elements to be removed from C.value can be obtained as

$$\begin{aligned} \bar{\Delta}(C.value) \;=\; set\{x | \, & x \leftarrow lookup(C.map, a_k).value, \\ & lookup(C.counts, x) = \\ & count(lookup(C.map, a_k).value, x)\} \end{aligned}$$

The expression checks if an element to be removed satisfies the condition that the number of duplicates in *C.value* is same as the number of duplicates in *S.value*. The technique here can be generalized to any comprehensions whose underlying monoids are idempotent.

6. UPDATE PROPAGATIONS

Strict Updates

Given the incremental evaluator $update(c)$ for a comprehension instance c, we can update the value of an entire expression by propagating changes among comprehension instances. When the value of a comprehension instance c is changed, the value of another comprehension instance c' that depends on c should also be updated. Figure 3 shows the algorithm that directly implements this idea. The algorithm updates the value of a comprehension instance and propagates the update along the edges of the dependency graph. This algorithm is strict in that the algorithm updates the value of an entire expression whenever the value of a piece of the expression is changed. Since comprehension instances that are not involved in the change are not evaluated, this algorithm is incremental.

Lazy Updates

Although $StrictUpdate(c)$ correctly updates the values of expressions, it has two problems. One is that because a single comprehension instance can depend on multiple other instances, it may be evaluated more than once for a single change. This makes the update propagation inefficient. The other problem is that the algorithm always computes the latest value even if the value will not be used. When the value is overwritten, the previous value becomes useless.

The two problems can be solved by delaying updates until the value is actually required. To avoid traversing the entire graph when the value is required, we use an algorithm that consists of two phases (Figure 4). The first phase propagates the invalid marker along the dependency graph. The second phase makes all of the components of a comprehension instance valid, then evaluates the instance itself. The first phase visits an edge of the dependency graph at most once, and the second phase computes the new value of an instance at most once. Thus the algorithm does not do the same work multiple times. This algorithm computes the latest value only when the second phase is invoked. The idea behind this algorithm is presented as the Adaptive Propagator in Feiler and Tichy [7].

Hybrid Updates

Unfortunately, the lazy update algorithm makes the response time worse when collections are dominant in an expression. The second phase updates all the invalid comprehensions at once. If we have many unprocessed elements in collections, this phase takes long time, which lessens the effect of incremental computation.

```
StrictUpdate($c$: comprehension instance) {
  update($c$);
  for each $c' \in c.dependents$
    StrictUpdate($c'$);
}
```

Figure 3: Strict Update

```
LazyUpdatePhase1($c$: comprehension instance) {
  if $c.valid = true$ {
    $c.valid := false$;
    for each $c' \in c.dependents$
      LazyUpdatePhase1($c'$);
  }
}

LazyUpdatePhase2($c$: comprehension instance) {
  for each $c' \in c.components$
    if $c'.valid = false$ {
      LazyUpdatePhase2($c'$);
      $c'.valid := true$;
    }
  update(c);
}
```

Figure 4: Lazy Update

One solution is to delay changes of primitive values only; addition and deletion of elements are processed immediately. Since the effect of updates in primitive values is always overwritten when the value is changed, it is clear that delaying computation of primitive values is effective to improve the performance. The resulting algorithm is a combination of the strict updates and the lazy updates shown in Figure 5. For primitive values, the lazy algorithm is applied; otherwise the strict algorithm is used. Note that the hybrid update algorithm is not always the best solution because it has the same problem as the strict updates.

7. IMPLEMENTATION

We implemented an experimental system using Cincom VisualWorks 3.1, a Smalltalk development environment. We use OQL instead of a comprehension language as a surface language. The system takes OQL queries as an input and translates them into byte codes that will generate the abstract syntax trees (ASTs) whose nodes are comprehensions. When a query, which is now a compiled Smalltalk method, is invoked, the generated AST constructs comprehension instances, each of which holds its initial value as well as dependency relationships.

Maintenance of dependencies between objects, change notifications, and automatic updates are well represented by the Observer pattern [8] in an object oriented system. VisualWorks provides a framework for the Observer pattern, namely the dependency mechanism, but we implemented our own mechanism because we wanted to separate change notifications from value updates. When they are synchronous, the computation is strict. When they are asynchronous, the

```
HybridUpdatePhase1($c$: comprehension instance) {
  if $c.type = set_p$ {
    LazyUpdatePhase1($c$);
  } else {
    update(c);
    for each $c' \in c.dependents$
      HybridUpdatePhase1($c'$);
  }
}

HybridUpdatePhase2($c$: comprehension instance) {
  LazyUpdatePhase2($c$);
}
```

Figure 5: Hybrid Update

computation is lazy.

Although we have only one kind of operation, which is comprehensions, we implemented them in multiple ways so that we can take advantage of the features of underlying data and operations. For example, since a set that represents a constant never changes, we can implement it in terms of a constant itself. We made constants behave as if they were sets by adding a method whose meaning is to access a set element but whose implementation is to return the constant object itself. Another example is the implementation of a primitive value. Since a set that represents a primitive value contains one element at most, we can implement it as an object whose only instance variable is to hold a value. This object is much lighter than general sets, which can hold any number of objects. We can also make a set for a record efficient by adding to the object indexes that make access to field values fast.

We also optimized our implementation by taking OQL-specific features into account. For example, the **group by** clause, which converts a bag into a set of bags, can be implemented efficiently using the knowledge that any two converted bags are disjoint.

8. EVALUATION

Evaluation Method

We evaluated the performance of our system by comparing (a) applications built on our incremental computation mechanism and (b) equivalent programs hand written in Smalltalk. A hand written Smalltalk program updates its results using procedural logic. Note that the hand written programs were implemented in a straightforward way; they were not optimized. To investigate the effect of lazy updates, we executed applications built on our mechanism in two modes: (a1) using only strict updates, and (a2) using both strict and lazy updates (hybrid updates). Since in our applications, collections are dominant, we excluded lazy only updates.

All the applications share the same underlying database. The data model is shown as a UML class diagram in Figure 6. In the data model, Order, Sale, and CashReceipt are primary transactions while OrderLineItem, SaleLineItem,

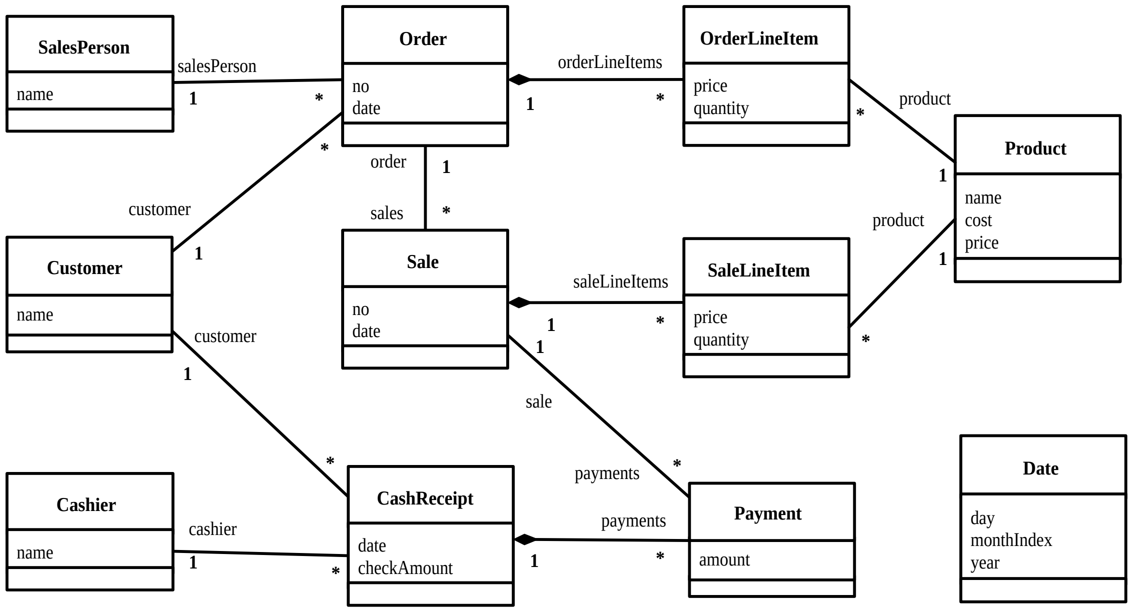

Figure 6: Benchmark Data Model

and Payment are transactions subsequent to Order, Sale and CashReceipt. Transaction objects are being inserted into the database while the benchmark is running but the other objects are fixed. We insert Orders, Sales, and CashReceipts in the same ratio 1:1:1. For each primary transaction, five subsequent transactions are generated. We used the OQL queries shown in Figure 7.

Note that although the only change to the underlying database is the addition of elements, the queries cause other types of change by means of the **sum** operator, the **group by** clause, and the binary operators on primitive values.

Results

The performance evaluation was conducted on a 300 MHz Pentium-II PC with 128MB memory under Windows NT 4.0. Table 2 shows the ratio of the processing time of our incremental framework to that of the hand written Smalltalk programs. The Order Statistics and Unpaid Sales queries achieve performance comparable to hand written programs. However, the General Ledger application is slow compared with the other queries. The reason is that the hand written General Ledger program uses few expensive operations such as table searches or collection iterations. Lazy updates are effective only in the Unpaid Sales application because it uses a complex condition in the **where** clause.

Figures 8, 9, and 10 indicate how the ratio changes with the amount of data inserted. The ratio is almost stable in the General Ledger and Order Statistics queries. For the Unpaid Sales application, the performance of our framework becomes closer to that of the hand written program as the amount of data increases. This is because VisualWorks does not handle large collections efficiently and thus the performance of the hand written program is dominated by the collection operations, which our framework also depends on.

9. RELATED WORK

Database View Maintenance

The algorithms for computing queries incrementally are known as incremental view maintenance techniques. Techniques for relational models have been studied extensively [11]. Recently incremental algorithms for nested collections have been attracting attention. Examples are: Gluche et al. [9], Baekgaard and Mark [2], Kawaguchi et al. [16], and Liu et al. [18]. However they addressed problems that arise only when elements are added or deleted; modification of record attributes were out of their scope.

Kuno and Rundensteiner [17] considered modification of attribute values, but this can be treated independently of addition or deletion of elements because in their language collections and the other types do not interact. Fegaras [5] suggested that by extending the monoid calculus with object identities, we can handle the modification of attributes as well as addition and deletion of collection elements in a single framework. However, he left constructing an actual algorithm to future research.

To our knowledge, Ali et al. [1] is the only work that presented an incremental algorithm that can handle modifications, additions, and deletions in a complex object-oriented query language like OQL. However, although their algorithm is very complicated, they did not formally show the rationale behind the algorithm, which prevents us from checking the correctness or completeness of their algorithm.

Application	Hand Written	Incremental Framework			
		Strict Updates		Strict & Lazy Updates	
	Time (T_b)	Time (T_{a1})	Ratio (T_{a1}/T_b)	Time (T_{a2})	Ratio (T_{a2}/T_b)
General Ledger	0.65 s	5.14 s	7.9	4.94 s	7.6
Order Statistics	3.56 s	7.62 s	2.2	7.67 s	2.2
Unpaid Sales	10.81 s	17.28 s	1.6	13.79 s	1.3

(Number of primary transactions = 8K)

Table 2: Performance Comparison

Query: General Ledger

```
struct(
accountsReceivable:
        (sum(select i.price * i.quantity
             from s in sales, i in s.saleLineItems)
        - sum(select r.checkAmount
              from r in cashReceipts)),
cash:
        sum(select r.checkAmount
            from r in cashReceipts),
costOfGoodsSold:
        sum(select i.product.cost * i.quantity
            from s in sales, i in s.saleLineItems))
```

Query: Statistics on Product Orders

```
select struct(
       product: product.name,
       month: month,
       total: sum(select p.i.quantity
                  from p in partition))
from order in orders, i in order.orderLineItems
group by product: i.product,
         month: order.date.monthIndex
```

Query: Unpaid Sales

```
select struct(
       name: s.order.customer.name,
       no: s.no)
from s in sales
where (sum(select i.price * i.quantity
           from i in s.saleLineItems)
       - sum(select p.amount
             from p in s.payments)) > 0
```

Figure 7: Benchmark Queries

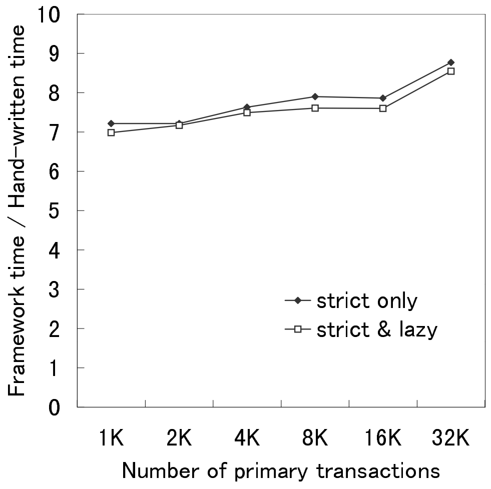

Figure 8: Performance Comparison (General Ledger)

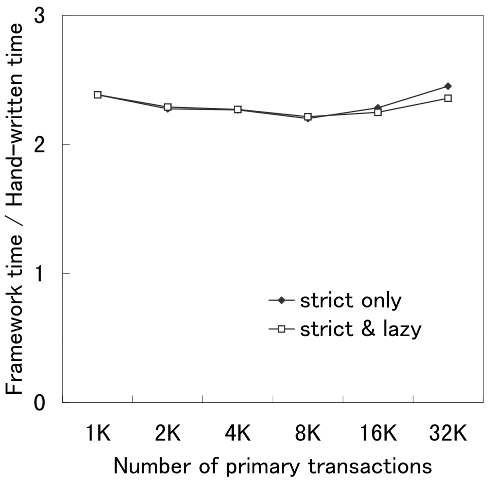

Figure 9: Performance Comparison (Order Statistics)

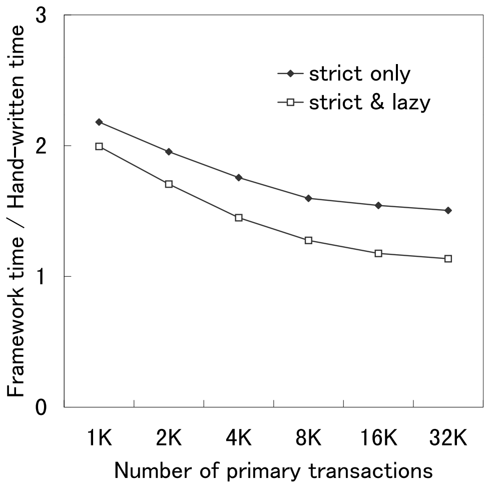

Figure 10: Performance Comparison (Unpaid Sales)

Incremental Attribute Evaluation

An attribute grammar supplies its accompanying semantics in terms of the equations that define attribute values associated with the grammar symbols. By dynamically maintaining the equations in response to changes to the attribute values, we can preserve the semantic correctness of a program in the language. This idea was explored extensively in the context of language-aware programming environments [20].

Hudson [14] presented an algorithm in which attribute evaluation can be deferred until the values are actually needed. He then applied the lazy evaluation algorithm to an object-oriented language [15]. Zanden et al. [25] also gave both a strict and a lazy incremental algorithms for manipulating data structures with pointer variables. Our work is actually an extension of their approaches, but applied to a language equipped with collections as a first-class data type.

Yellin and Strom [24] designed a functional language that can be evaluated incrementally. The language can handle bags as well as records, and thus their approach is similar to ours. One difference is that their notion of change is restricted to addition and deletion of bag elements, while our framework accepts any types of data updates. Another difference is that they attached independent incremental procedures to 15 operators, which makes analyzing the entire algorithm very hard, while our algorithm has to handle only one operator.

10. CONCLUDING REMARKS

We have presented an incremental query evaluation algorithm that can handle any kinds of database updates and can accept any expressions in complex query languages by translating diverse OODB queries into uniform comprehension expressions. Since we have to consider only one operation, the problems with incremental computation of OODB queries are manageable. We have also shown that the algorithm can be implemented efficiently. It achieves performance comparable to hand written update programs.

Since our work is essentially an application of incremental attribute evaluation techniques to collection languages, we can use the ideas developed in those two areas to improve our algorithm. For example, the algorithm shown in this paper does not use any sophisticated techniques in managing dependencies, but we can achieve faster incremental computation by handling non-local dependencies directly [13]. Another technique we can use is the meaning-preserving transformation between two collection expressions [6, 23]. Although their transformation rules are designed to increase the performance of queries in a non-incremental setting, we believe we can use them to improve the performance of our incremental algorithm.

11. ACKNOWLEDGEMENTS

The author would like to thank Ralph E. Johnson for his helpful comments and advice.

12. REFERENCES

[1] M. A. Ali, A. A. A. Fernandes, and N. W. Paton. Incremental maintenance of materialized OQL views. In *International Workshop on Data Warehousing and OLAP*, 2000.

[2] L. Baekgaard and L. Mark. Incremental computation of nested relational query expressions. *ACM Transactions on Database Systems*, 20(2), 1995.

[3] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1), 1994.

[4] R. G. G. Cattell and D. K. Barry, editors. *The Object Database Standard: ODMG 3.0*. Morgan Kaufmann Publishers, Inc., 2000.

[5] L. Fegaras. Optimizing queries with object updates. *Journal of Intelligent Information Systems*, 12, 1999.

[6] L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. *ACM Transactions on Database Systems*, December 2000.

[7] P. H. Feiler and W. F. Tichy. Propagator: A family of patterns. In *Proceedings of Technology of Object-Oriented Languages and Systems*, 1997.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[9] D. Gluche, T. Grust, C. Mainberger, and M. H. Scholl. Incremental updates for materialized views with user-defined functions. In *Proceedings of the 5th International Conference on Deductive and Object Oriented Databases (DOOD'97)*, 1997.

[10] T. Grust and M. H. Scholl. Translating OQL into monoid comprehensions - stuck with nested loops ? Technical Report 1430-3558, Universität Konstanz, 1996.

[11] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques and applications. *IEEE Quarterly Bulletin on Data Engineering; Special Issue on Materialized Views and Data Warehousing*, 18(2), 1995.

[12] A. Gupta and I. S. Mumick, editors. *Materialized Views Techniques, Implementations, and Applications*. MIT Press, 1999.

[13] R. Hoover. Dynamically bypassing copy rule chains in attribute grammars. In *Proceedings of Symposium on Principles of Programming Languages*, 1986.

[14] S. E. Hudson. Incremental attribute evaluation: A flexible algorithm for lazy updates. *ACM Transactions on Programming Languages and Systems*, 13(3), 1991.

[15] S. E. Hudson. A system for efficient and flexible one-way constraint evaluation in C++. Technical Report 93-15, College of Computing, Georgia Institute of Technology, 1997.

[16] A. Kawaguchi, D. F. Lieuwen, I. S. Mumick, and K. A. Ross. Implementing incremental view maintenance in nested data models. In *Proceedings of the International Workshop on Database Programming Languages*, 1997.

[17] 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), 1998.

[18] J. Liu, M. Vincent, and M. Mohania. Incremental evaluation of nest and unnest operators in nested relations. In *Proceedings of 1999 CODAS Conference*, 1999.

[19] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *Proceedings of the Symposium on Principles of Programming Languages*, 1993.

[20] T. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, 1988.

[21] V. Tannen. Tutorial: Languages for collection types. In *Proceedings of Symposium on Principles of Database Systems*, 1994.

[22] P. Trinder. a query notation for DBPLs. In *Proceedings of the Third International Workshop on Database Programming Languages*, 1991.

[23] L. Wong. Normal forms and conservative extension properties for query languages over collection types. *Journal of Computer and System Sciences*, 52(3), 1996.

[24] D. M. Yellin and R. E. Strom. INC: A language for incremental computations. *ACM Transactions on Programming Languages and Systems*, 13(2), 1991.

[25] B. T. V. Zanden, B. Myers, D. Giuse, and P. Szekely. Integrating pointer variables into one-way constraint models. *ACM Transactions on Computer Human Interaction*, June, 1994.