Query Unnesting in Object-Oriented Databases

Leonidas Fegaras

Department of Computer Science and Engineering
The University of Texas at Arlington
416 Yates Street, P.O. Box 19015
Arlington, TX 76019-19015
email: *fegaras@cse.uta.edu*

Abstract

There is already a sizable body of proposals on OODB query optimization. One of the most challenging problems in this area is query unnesting, where the embedded query can take any form, including aggregation and universal quantification. Although there is already a number of proposed techniques for query unnesting, most of these techniques are applicable to only few cases. We believe that the lack of a general and simple solution to the query unnesting problem is due to the lack of a uniform algebra that treats all operations (including aggregation and quantification) in the same way.

This paper presents a new query unnesting algorithm that generalizes many unnesting techniques proposed recently in the literature. Our system is capable of removing *any* form of query nesting using a very simple and efficient algorithm. The simplicity of the system is due to the use of the monoid comprehension calculus as an intermediate form for OODB queries. The monoid comprehension calculus treats operations over multiple collection types, aggregates, and quantifiers in a similar way, resulting in a uniform way of unnesting queries, regardless of their type of nesting.

1 Introduction

There are many recent proposals on OODB query optimization that are focused on unnesting nested queries [9, 8, 6, 7, 20]. Query nesting appears more often in OODB queries than in relational queries, because OODB query languages allow complex expressions at any point in a query. Current OODB systems typically evaluate nested queries in a nested-loop fashion, which does not leave many opportunities for optimization. Most unnesting techniques for OODB queries are actually based on similar techniques for relational queries [16, 15, 18]. For all but the trivial nested queries, these techniques require the use of outer-joins, to prevent loss of data, and grouping, to accumulate the data and to remove the null values introduced by the outer-joins.

If considered in isolation, query unnesting itself does not result in performance improvement. Instead, it makes possible other optimizations, which would not be possible otherwise. More specifically, without unnesting, the only choice of evaluating nested queries is a naive nested-loop method: for each step of the outer query, all the steps of the inner query need to be executed. Query unnesting promotes all the operators of the inner query into the operators of the outer query. This operator mixing allows other optimization techniques to take place, such as the rearrangement of operators to minimize cost and the free movement of selection predicates between inner and outer operators, which enables operators to be more selective.

All early unnesting techniques were actually source-to-source transformations over SQL code, mostly due to the lack of a group-by operator in the relational algebra to express grouping. The absence of a formal theory in a form of an algebra to express these transformations resulted in a number of bugs (such as the infamous count bug [15]) that were eventually detected and corrected. Since OODB queries are far more complex than relational queries, it is more crucial to express the unnesting transformations in a formal algebra that will allow us to prove the soundness and completeness of these transformations. The first work with that goal in mind was by Cluet and Moerkotte [9, 8], which covered many cases of nesting, including nested aggregate queries, and validated all the transformations. Their work was extended by Claussen et al [7] to include universal quantification.

Our work extends previous work in two ways. First, our unnesting algorithm is not only sound, but it is also complete. That is, our system is capable of removing any form of nesting. Second, our unnesting algorithm is easier to understand and implement than earlier work, mostly due to the use of the monoid comprehension calculus [13] as an intermediate form for OODB queries. The monoid comprehension calculus treats operations over multiple collection types, aggregates, and quantifiers in a similar way, resulting in a uniform way of unnesting queries, regardless of their type of nesting. In fact, many forms of nested queries can be unnested by a simple normalization algorithm for monoid comprehensions [12]. All other forms require the introduction of outer-joins and grouping. Our algorithm unnests the queries that cannot be normalized by the normalization algorithm using two rewrite rules only.

Queries in our framework are translated into monoid comprehensions, which serve as an intermediate form, and then are translated into a version of the nested relational algebra that supports aggregation, quantification, outer-joins, and outer-unnestings. Query unnesting is performed during the translation of monoid comprehensions into algebraic forms. We decided to use both a calculus and an algebra as intermediate forms because the calculus closely resembles current OODB languages and is easy to normalize, while the algebra is lower-level and can be directly translated into the execution algorithms supported by database systems.

Finally, we report on an implementation of our ideas using a powerful optimizer specification framework, called OPTGEN [10]. Since the merit of query unnesting cannot be judged in isolation from other optimizations, we combined unnesting with other optimization techniques, such as materialization of path expressions into joins [1], performing selections as early as possible, rearranging join orders, choosing access paths, assigning evaluation algorithms to operators, etc. Our preliminary results suggest that these optimization techniques improve performance considerably when combined with query unnesting.

1.1 Motivating Examples

All the examples given in this paper are expressed in ODMG OQL [4]. Our unnesting algorithm, though, can be applied to any OODB language that resembles OQL and can be easily adapted to handle object-relational and relational languages.

As an example of how OODB queries are translated into algebraic forms, consider the following OQL query:

select distinct struct(E: e.name, C: c.name)
from e **in** Employees, c **in** e.children

which is translated as follows in the monoid comprehension calculus:

QUERY A: $\cup\{\ \langle\ \mathsf{E} = \mathsf{e.name},\ \mathsf{C} = \mathsf{c.name}\ \rangle\ \mid\ \mathsf{e} \leftarrow \mathsf{Employees},\ \mathsf{c} \leftarrow \mathsf{e.children}\ \}$

This form constructs one tuple for every employee e and for every child c of the employee e. All these tuples are lifted to singleton sets and the sets are merged using $\cup$. The operation $\cup$ is called the *comprehension accumulator* because it is used to accumulate the values in the head of the comprehension (the tuples $\langle$ E = e.name, C = c.name $\rangle$).

Figure 1.A presents the algebraic form of QUERY A. Algebraic forms are displayed as trees in which the tree leaves are sets of objects (class extents) and the output of the tree root is the output of the algebraic form. The functionality of an algebraic form can be better understood if we use a stream-based interpretation in which a stream of tuples flows from the leaves to the root of the tree. Under this interpretation, the algebraic form in Figure 1.A generates a stream of tuples of type set($\langle$ e: Employee $\rangle$) from the extent Employees. Variable e ranges over employees (i.e. it is of type Employee). The unnest operator, $\mu_{\mathsf{e.children}}$, accepts the stream of tuples of type set($\langle$ e: Employee $\rangle$) and constructs a stream of tuples of type set($\langle$ e: Employee, c: Person $\rangle$), connecting each employee with one of his/her children. The reduce operator, $\Delta^{\cup/<\mathsf{E=e.name,C=c.name}>}$, at the top of this algebraic form, is a generalization of the relational operator, project: it evaluates the expression $\langle$ E=e.name, C=c.name $\rangle$ for every input element and constructs a set from these tuples using $\cup$ (i.e. the tuples are lifted to singleton sets and the sets are merged together using set union). The reduce operator can also be used to compute aggregations (if we use an aggregation function, such as $+$, as an accumulator instead of $\cup$), and universal or existential quantifications (if we use $\wedge$ or $\vee$ instead of $\cup$).

As an example of how OODB queries are unnested in our model, consider the following nested OQL query:

select distinct struct(D: d, E: (**select distinct** e
from e **in** Employees
where e.dno = d.dno))
from d **in** Departments

This query has a straightforward translation in the monoid comprehension calculus:

QUERY B: $\cup\{\ \langle\ \mathsf{D{=}d},\ \mathsf{E}{=}\cup\{\ \mathsf{e}\ \mid\ \mathsf{e} \leftarrow \mathsf{Employees},\ \mathsf{e.dno} = \mathsf{d.dno}\ \}\ \rangle\ \mid\ \mathsf{d} \leftarrow \mathsf{Departments}\ \}$

The nesting inherent in this query can be avoided by using an outer-join combined with grouping [18], as it is shown in Figure 1.B. The nest operator, $\Gamma_{\mathsf{d}}^{\cup/\mathsf{e}}$, combines the functionality of the nested-relational operator, nest, with the functionality of reduce. It groups the input by the range variable d, constructing a set of all e's that are associated with the value of d. This set becomes the value of the range variable m. That is, this nest operator reads a stream of tuples of type set($\langle$ d: Department, e: Employee $\rangle$) and generates a stream of tuples of type:

set($\langle$ d: Department, m: set($\langle$ e: Employee $\rangle$) $\rangle$)

Like the reduce operator, the nest operator can perform aggregation after grouping instead of constructing a set.

The join before the nesting in Figure 1.B is a left outer-join: if there are no employees or the predicate e.dno=d.dno is false for all employees in a department d, then the result of the join is d associated with a null value, i.e. the value of the range variable e becomes null. The nest operator converts this null value into the empty set (the zero element of $\cup$, if $\cup$ is handled as a monoid). That is, the outer-join introduces nulls and the nest operator converts nulls into zeros.

Another interesting example is the expression $A \subseteq B$, which is equivalent to $\forall a \in A : \exists b \in B : a = b$. It can be expressed in our calculus as follows:

QUERY C: $\wedge\{\ \vee\{\ \mathsf{true}\ \mid\ \mathsf{b} \leftarrow \mathsf{B},\ \mathsf{a{=}b}\ \}\ \mid\ \mathsf{a} \leftarrow \mathsf{A}\ \}$

The inner comprehension, which captures the existential quantification, checks if there is at least one $b \in B$ with $a = b$ (it can also be written as $\vee\{\,\mathsf{a{=}b} \mid \mathsf{b} \leftarrow \mathsf{B}\,\}$). If there is at least one, the result of the inner comprehension is true, since all the true values are merged together using the disjunction operator, $\vee$; otherwise it is false, which is the zero element of $\vee$. The outer comprehension captures the universal quantification since it merges the results of the inner comprehension using the conjunction operation, $\wedge$. If A is empty, it returns true, which is the zero element of $\wedge$.

The algebraic form of QUERY C after unnesting is shown in Figure 1.C. During the outer-join, if there is no a with b=a, then a will be matched with a null value. In that case, the nest operator will convert the null value into a false value, which is the zero element of $\vee$. The reduction at the root tests whether all m's are true. If there is at least one false m, then there is at least one a in A that has no equal in B.

A more challenging example is the following double-nested OQL query:

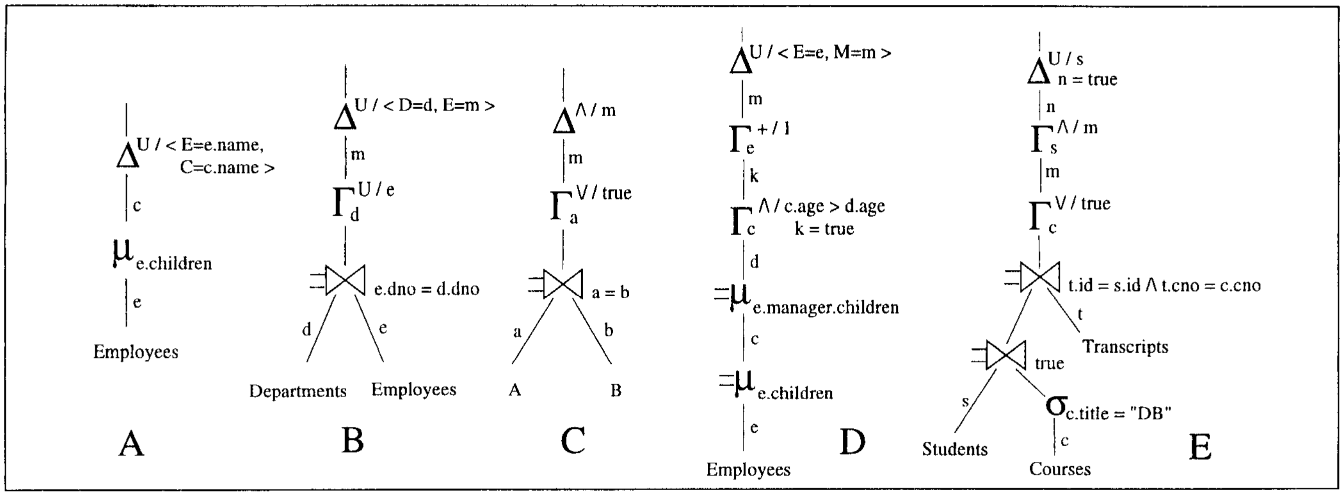


Figure 1: The Algebraic Form of Some OODB Queries

```
select distinct struct( E: e,
        M: count( select distinct c
                   from c in e.children
                   where for all d in e.manager.children:
                          c.age > d.age ) )
  from e in Employees
```

which has the following translation in the monoid comprehension calculus:

QUERY D:

```
∪{ ⟨ E=e, M=+{ 1 | c ← e.children,
                   ∧{ c.age > d.age
                      | d ← e.manager.children } } ⟩
  | e ← Employees }
```

The comprehension with the accumulator '+' counts the children of the employee e because, for each child, it returns 1 and all these 1's are added together when merged by the accumulator '+'. The algebraic form of QUERY D is shown in Figure 1.D. Here, instead of an outer-join we are using an outer-unnesting: the $=\!\!\!\mu_{\text{e.children}}$ operator introduces a new range variable, c, whose value is the unnesting of the set e.children. If this set is empty, then c becomes null, i.e. the employee e is padded with a null value. Since we have a double-nested query, we need to use two nested unnest-nest pairs. The top nest operator groups the input stream by each employee e, generating the number of children of e. The second nest operator groups the input stream by each child c and, for each group, it evaluates the predicate c.age > d.age and extends the output stream with an attribute k bound to the conjunction of all the predicates (indicated by the ∧ accumulator). Note that every operator in our algebra can be assigned a predicate (such as the predicate k=true in the second nest operation) to restrict the input data.

The following query finds the students who have taken all database courses [7]:

```
select distinct s
 from s in Student
where for all c in select c
                   from c in Courses
                  where c.title = "DB":
      exists t in Transcript: (t.id=s.id and t.cno=c.cno)
```

Its comprehension form uses a predicate that has the same pattern as that of $A \subseteq B$:

QUERY E:

```
∪{ s | s ← Student,
       ∧{ ∨{ true | t ← Transcript, t.id=s.id, t.cno=c.cno }
          | c ← Courses, c.title = "DB" } }
```

The algebraic form of this query is shown in Figure 1.E. This query can be evaluated more efficiently if we use the associativity property of outer-joins. In that case, the resulting outer-joins would both be assigned equality predicates, thus making them more efficient. Optimizations like these justify query unnesting.

1.2 The Unnesting Algorithm

Figure 2 shows how our unnesting algorithm unnests QUERY E. Every comprehension is first translated into an algebraic form consisting of regular joins, selections, unnests, and reductions; the latter being the root of the algebraic form. We will see that this translation is straightforward. The outermost comprehension has one output (the result of the reduction) and no input streams. For example, the dashed box A in Figure 2 represents the outer comprehension in QUERY E. The shaded box on the reduction represents a nested query. The algebraic form of an inner comprehension (i.e. a comprehension inside another comprehension) has one input stream and one output value: the input stream is the same stream of tuples as that of the operation in which this form is embedded to and the output is the result of this form. For example, the dashed box B in Figure 2 represents the universally quantified comprehension (with accumulator ∧): the input of this box is the same input stream as that of the reduction in box A (namely the stream of employees) since box B is embedded in the predicate of the reduction. The output value of the box B, n, is used in the predicate of the reduction in the box A. Similarly, the existential quantification (with accumulator ∨) is translated into the box C.

Our unnesting algorithm is very simple: for each box that corresponds to a nested query (i.e. Boxes B and C), it converts reductions into nests, joins into outer-joins, and unnests into outer-unnests. At the same time, it embeds

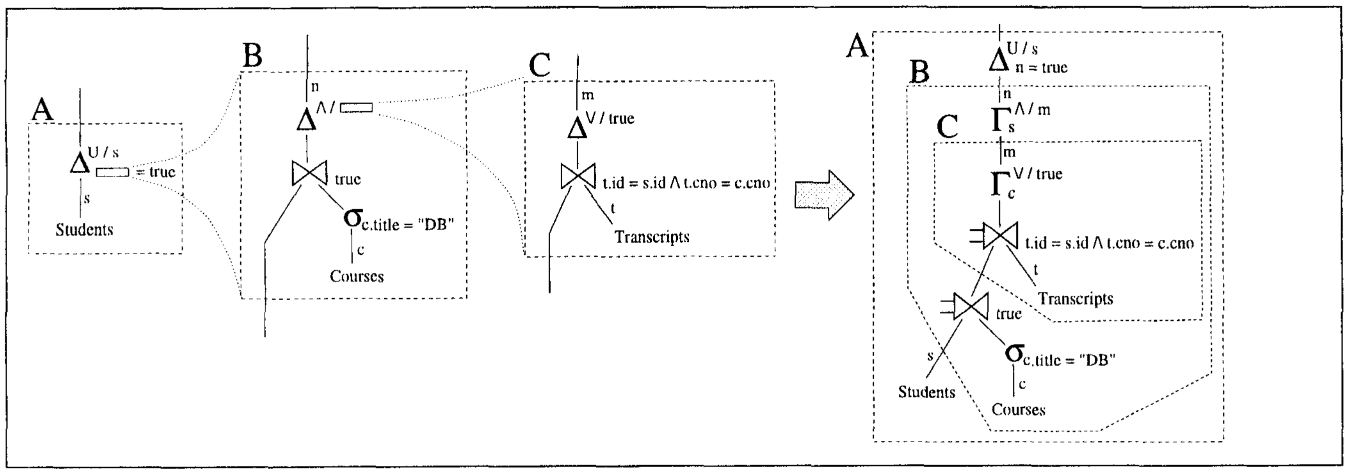


Figure 2: Unnesting QUERY E

the resulting boxes at the points immediately before they are used. For example, box C will be embedded before the reduction in box C and the output value of box C will be used as the result of the innermost form. Similarly, box B will be embedded immediately before its output value is used in box A.

There is a very simple explanation why this algorithm is correct (a formal proof is given in the extended version of this paper [11]): a nested query is represented as a box, say box C, that consumes the same input stream as that of the embedding operation and computes a value m which is used in the embedding query (box B). If we want to splice this box onto the stream of the embedding query we need to guarantee two things. First, box C should not block the input stream by removing tuples from the stream. This is achieved by converting joins into outer-joins and unnests into outer-unnests. Second, instead of returning one value, namely the value m of box C, we need to extend the stream with the new value, m. This can be done by converting the reduction of box C into a nest. At the same time, the nest operator will convert null values to zeros so that the stream that you get from the output of the spliced box C will be exactly the same as it was before the splice. There are some very important details that we omitted here but we will present later when we describe the unnesting algorithm in detail. The most important one is which nulls to convert to zeros each time: if we convert the null c's (i.e. the courses) to false in the second nest operation ($\Gamma_c^{\vee/m}$) in the resulting unnested form in Figure 2, it will be too soon; this nest should convert null t's to false and the first nest should convert null c's to true. This is indicated by an extra parameter to the nest operator which is not shown here.

The rest of this paper is organized as follows. Section 2 summarizes our earlier work on the monoid comprehension calculus. A more formal treatment is presented elsewhere [13, 12]. In addition, this section describes a simple normalization algorithm that unnests most simple forms of nesting that do not require outer-joins, outer-unnests, or grouping. Section 3 describes a version of the nested-relational algebra that supports aggregation, quantification, and the handling of null values (using outer-joins and outer-unnests). The semantics of these operations is given in terms of the monoid calculus. The real contribution of this paper is given in Section 4. This section presents the transformation rules for unnesting OODB queries by translating terms in our monoid calculus into terms in the nested-relational algebra. Section 5 presents a simplification rule to improve the forms derived from our unnesting algorithm. Section 7 compares our approach to related work. Finally, Section 6 reports on an implementation of our unnesting algorithm and the experience gained from it.

$$\sigma \vdash v : \sigma(v) \qquad \text{(T1)}$$

$$\frac{\sigma \vdash e : \langle A_1 : t_1, \ldots, A_n : t_n\rangle}{\sigma \vdash e.A_i : t_i} \qquad \text{(T2)}$$

$$\frac{\sigma \vdash e_1 : \text{bool},\ \sigma \vdash e_2 : \text{t},\ \sigma \vdash e_3 : \text{t}}{\sigma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : t} \qquad \text{(T3)}$$

$$\frac{\sigma \vdash e_2 : set(t_2),\ \sigma[t_2/v] \vdash \oplus\{\, e_1 \,|\, \bar{r}\,\} : t_1}{\sigma \vdash \oplus\{\, e_1 \,|\, v \leftarrow e_2, \bar{r}\,\} : t_1} \qquad \text{(T4)}$$

$$\frac{\sigma \vdash e_2 : \text{bool},\ \sigma \vdash \oplus\{\, e_1 \,|\, \bar{r}\,\} : \text{t}}{\sigma \vdash \oplus\{\, e_1 \,|\, e_2, \bar{r}\,\} : t} \qquad \text{(T5)}$$

$$\frac{\sigma[t_1/v] \vdash e : t_2}{\sigma \vdash \lambda v^{t_1}.e : t_1 \rightarrow t_2} \qquad \text{(T6)}$$

$$\frac{\sigma \vdash e_1 : t_1 \rightarrow t_2,\ \sigma \vdash e_2 : t_1}{\sigma \vdash e_1(e_2) : t_2} \qquad \text{(T7)}$$

$$\frac{\oplus \not\equiv \cup,\ \sigma \vdash e : \oplus_T}{\sigma \vdash \oplus\{\, e \,|\, \} : \oplus_T} \qquad \text{(T8)}$$

$$\frac{\sigma \vdash e : t}{\sigma \vdash \cup\{\, e \,|\, \} : set(t)} \qquad \text{(T9)}$$

Figure 3: Typing Rules of the Monoid Calculus

2 Background: The Monoid Comprehension Calculus

This section summarizes our earlier work on the monoid calculus. A more formal treatment is presented elsewhere [13, 12].

The monoid calculus is based on the concept of *monoids* from abstract algebra. A monoid of type T is a pair $(\oplus, Z_\oplus)$, where $\oplus$ is an associative function of type $T \times T \rightarrow T$ (i.e. a binary function that takes two values T and returns a value T), called the *accumulator* or the *merge* function of this monoid, and $Z_\oplus$ of type T, called the *zero* element of the monoid, is the left and right identity of $\oplus$. That is, the zero element satisfies $Z_\oplus \oplus x = x \oplus Z_\oplus = x$ for every x. Since the accumulator function uniquely identifies a monoid, we will often use the accumulator name as the monoid name. In addition, we will denote the type, T, of a monoid $\oplus$ by $\oplus_T$. Examples of monoids include $(\cup, \{\,\})$ for sets, $(+, 0)$, $(*, 1)$, and $(max, 0)$ for integers, and $(\vee, \text{false})$ and $(\wedge, \text{true})$ for booleans. All but the set monoid are called *primitive monoids* because they construct values of a primitive type, such as integers. The set monoid is called a *collection monoid* and requires the additional definition of a *unit function*, which, along with $\cup$ and $\{\,\}$, allows us to construct set values. For sets, the unit function is $\lambda x.\{x\}$, that is, it takes a value x as input and constructs the singleton set $\{x\}$ as output. Consequently, any set can be constructed by merging singleton values.

All the monoids used in this paper are commutative, i.e. they satisfy $x \oplus y = y \oplus x$ for every x and y. In addition, some of them ($\cup$, $\wedge$, $\vee$, and max) are idempotent, i.e. they satisfy $x \oplus x = x$ for every x. To make our analysis simpler, we will use only one collection monoid, namely the set monoid. There are other collection monoids that are not idempotent (such as the bag monoid) or they are neither commutative nor idempotent (such as the list monoid). We leave it as future work to extend our unnesting algorithm to capture these collection monoids (it is not obvious what the semantics of outer-join for lists and bags should be).

A *monoid comprehension* over the monoid $\oplus$ takes the form $\oplus\{\, e \mid \bar{r} \,\}$. Expression e is called the *head* of the comprehension. Each term r_i in the term sequence $\bar{r} = r_1, \ldots, r_n$, for $n \geq 0$, is called a *qualifier*, and is either a *generator* of the form $v \leftarrow e'$, where v is a *range variable* and e' is an expression (the generator domain) that constructs a set, or a *filter* p, where p is a predicate. We will use the shorthand $\{\, e \mid \bar{r} \,\}$ to denote the set comprehension $\cup\{\, e \mid \bar{r} \,\}$.

A monoid comprehension is defined by the following reduction rules:

$$
\begin{array}{rcllr}
\{\, e \mid \,\} & \rightarrow & \{e\} & & \text{(D1)} \\
\oplus\{\, e \mid \,\} & \rightarrow & e & \text{for } \oplus \neq \cup & \text{(D2)} \\
\oplus\{\, e \mid \text{false}, \bar{r} \,\} & \rightarrow & Z_\oplus & & \text{(D3)} \\
\oplus\{\, e \mid \text{true}, \bar{r} \,\} & \rightarrow & \oplus\{\, e \mid \bar{r} \,\} & & \text{(D4)} \\
\oplus\{\, e \mid v \leftarrow \{\,\}, \bar{r} \,\} & \rightarrow & Z_\oplus & & \text{(D5)} \\
\oplus\{\, e \mid v \leftarrow \{e'\}, \bar{r} \,\} & \rightarrow & \text{let } v = e' \text{ in } \oplus\{\, e \mid \bar{r} \,\} & & \text{(D6)} \\
\oplus\{\, e \mid v \leftarrow (e_1 \cup e_2), \bar{r} \,\} & \rightarrow & (\oplus\{\, e \mid v \leftarrow e_1, \bar{r} \,\}) & & \text{(D7)} \\
 & & \oplus\ (\oplus\{\, e \mid v \leftarrow e_2, \mathcal{I}, \bar{r} \,\}) & &
\end{array}
$$

where $\mathcal{I} = v \notin e_2 = \wedge\{\, w \neq v \mid w \leftarrow e_2 \,\}$ for idempotent $\oplus$, $\mathcal{I} = \text{true}$ otherwise. Rules (D3) and (D4) reduce a comprehension in which the leftmost qualifier is a filter, while Rules (D5-D7) reduce a comprehension in which the leftmost qualifier is a generator. The let-statement in (D6) binds v to e' and uses this binding in every free occurrence of v in $\oplus\{\, e \mid \bar{r} \,\}$. The case for a non-idempotent $\oplus$ (such as $+$ or $*$) is necessary in Rule (D7) to avoid semantic inconsistencies, such as:

$$
\begin{aligned}
1 &= +\{\, a \mid a \leftarrow \{1\} \,\} = +\{\, a \mid a \leftarrow \{1\} \cup \{1\} \,\} \\
&= (+\{\, a \mid a \leftarrow \{1\} \,\}) + (+\{\, a \mid a \leftarrow \{1\} \,\}) = 2
\end{aligned}
$$

Here $+$ was treated as idempotent monoid, which led us into inconsistencies.

The only purpose of the above rules is to give semantics to monoid comprehensions, not to suggest in any way how they are implemented. Monoid comprehensions can be effectively translated into efficient physical algorithms, as we will show in this paper.

Our calculus has additional primitive types, such as strings, and additional operations, such as tuple construction, tuple projection, function abstraction and application, if-then-else expression, etc. Furthermore, every type domain is extended with the null value, NULL. The only operations we support for nulls are creating them and testing whether a value is null.

A complete formal definition of the calculus is presented elsewhere [13]. Here, though, we present the typing rules of some important terms in our calculus (Figure 3). The name v in Figure 3 indicates a variable name, names starting with A are attribute names, names starting with e represent terms in our calculus, and names starting with t represent types. The notation $\sigma \vdash e : t$ indicates that the term e is assigned the type t under the substitution σ. If a type equation is a fraction, the numerator is the premise while the denominator is the consequence. The substitution list σ binds variable names to types ($\sigma(v)$ returns the binding of v in σ and $\sigma[t/v]$ extends σ with the binding from v to t). The λ-variable, v in Equation T6 is annotated by its type, t_1. This is not necessary for type inference systems, since this type can be inferred.

When restricted to sets, monoid comprehensions are equivalent to set monad comprehensions [2], which capture precisely the nested relational algebra [13]. Most OQL expressions have a direct translation into the monoid calculus [13]. For example, the OQL query

```
select distinct hotel.price
 from hotel in ( select h
                 from c in Cities, h in c.hotels
                 where c.name = "Arlington" )
where exists r in hotel.rooms: r.bed_num = 3
  and hotel.name in ( select t.name
                      from s in States, t in s.attractions
                      where s.name = "Texas" );
```

is translated into the following comprehension:

```
{ hotel.price | hotel ← { h | c ← Cities, h ← c.hotels,
                              c.name="Arlington" },
              ∨{ r.bed_num=3 | r ← hotel.rooms },
              ∨{ e=hotel.name
                 | e ← { t.name | s ← States, t ← s.attractions,
                                  s.name="Texas" } } }
```

We use the following convention to represent variable bindings in a comprehension:

$$
\oplus\{\, e \mid \bar{r},\ x \equiv u,\ \bar{s} \,\} \longrightarrow \oplus\{\, e[u/x] \mid \bar{r},\ \bar{s}[u/x] \,\} \qquad \text{(D8)}
$$

$$
\begin{array}{rcll r}
(\lambda v.e_1)\,e_2 & \longrightarrow & e_1[e_2/v] & \text{beta reduction} & \text{(N1)} \\
\langle A_1 = e_1, \ldots, A_n = e_n \rangle.A_i & \longrightarrow & e_i & & \text{(N2)} \\
\oplus\{\, e \mid \overline{q},\ v \leftarrow (\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3),\ \overline{s} \,\} & \longrightarrow & (\oplus\{\, e \mid \overline{q},\ e_1,\ v \leftarrow e_2,\ \overline{s} \,\}) \oplus (\oplus\{\, e \mid \overline{q},\ \neg e_1,\ v \leftarrow e_3,\ \overline{s} \,\}) & & \text{(N3)} \\
\oplus\{\, e \mid \overline{q},\ v \leftarrow \{\,\},\ \overline{s} \,\} & \longrightarrow & Z_{\oplus} & & \text{(N4)} \\
\oplus\{\, e \mid \overline{q},\ v \leftarrow \{e'\},\ \overline{s} \,\} & \longrightarrow & \oplus\{\, e \mid \overline{q},\ v \equiv e',\ \overline{s} \,\} & & \text{(N5)} \\
\oplus\{\, e \mid \overline{q},\ v \leftarrow (e_1 \cup e_2),\ \overline{s} \,\} & \longrightarrow & \left\{ \begin{array}{l} (\oplus\{\, e \mid \overline{q},\ v \leftarrow e_1,\ \overline{s} \,\}) \oplus (\oplus\{\, e \mid \overline{q},\ v \leftarrow e_2,\ \overline{s} \,\}) \\ (\oplus\{\, e \mid \overline{q},\ v \leftarrow e_1,\ \overline{s} \,\}) \oplus (\oplus\{\, e \mid \overline{q},\ v \leftarrow e_2,\ v \notin e_1,\ \overline{s} \,\}) \end{array} \right. & \begin{array}{r} \text{for idempotent } \oplus \\ \text{otherwise} \end{array} & \text{(N6)} \\
\oplus\{\, e \mid \overline{q},\ v \leftarrow \{\, e' \mid \overline{r} \,\},\ \overline{s} \,\} & \longrightarrow & \oplus\{\, e \mid \overline{q},\ \overline{r},\ v \equiv e',\ \overline{s} \,\} & & \text{(N7)} \\
\oplus\{\, e \mid \overline{q},\ \vee\{\, pred \mid \overline{r} \,\},\ \overline{s} \,\} & \longrightarrow & \oplus\{\, e \mid \overline{q},\ \overline{r},\ pred,\ \overline{s} \,\} & \text{for idempotent } \oplus & \text{(N8)} \\
\oplus\{\, \oplus\{\, e \mid \overline{r} \,\} \mid \overline{s} \,\} & \longrightarrow & \oplus\{\, e \mid \overline{s},\ \overline{r} \,\} & \text{for } \oplus \neq \cup & \text{(N9)}
\end{array}
$$

Figure 4: The Normalization Algorithm

where $e[u/x]$ is the expression e with u substituted for all the free occurrences of x (i.e. $e[u/x]$ is equivalent to **let** x = u **in** e). In addition, as a syntactic sugar, we allow irrefutable patterns in place of lambda variables, range variables, and variables in bindings. Patterns like these can be compiled away using standard pattern decomposition techniques [19]. For example, $\{\, x + y \mid (x,(y,z)) \leftarrow A,\ z = 3 \,\}$ is equivalent to $\{\, a.\text{fst} + a.\text{snd.fst} \mid a \leftarrow A,\ a.\text{snd.snd} = 3 \,\}$, where fst/snd retrieves the first/second element of a pair. Another example is $\lambda(x,(y,z)).x + y + z$, which is a function that takes three parameters and returns their sum. It is equivalent to $\lambda a.\ a.\text{fst} + a.\text{snd.fst} + a.\text{snd.snd}$.

The monoid calculus can be put into a canonical form by an efficient rewrite algorithm, called the *normalization algorithm*. The evaluation of these canonical forms generally produces fewer intermediate data structures than the initial unnormalized programs. Moreover, the normalization algorithm improves program performance in many cases (as we will prove below). It generalizes many optimization techniques already used in relational algebra, such as fusing two selections into one selection.

Figure 4 gives the normalization rules. The soundness of the normalization rules can be easily proved using the definition of the monoid comprehension. Rule (N7) flattens a comprehension that contains a generator whose domain is another comprehension (it may require variable renaming to avoid name conflicts). Rule (N8) unnests an existential quantification.

For example, the previous OQL query is normalized into:

```
{ h.price | c ← Cities, h ← c.hotels, r ← h.rooms, s ← States,
            t ← s.attractions, c.name="Arlington",
            r.bed_num=3, s.name="Texas", t.name=h.name }
```

by applying Rule (N7) to unnest the two inner set comprehensions and Rule (N8) to unnest the two existential quantifications.

All generator domains can be normalized by the normalization algorithm into paths (i.e. sequences of projections of the form $x.A_1.A_2 \ldots A_n$, for $n \geq 0$, where x is a range variable or an extent, and A_i are attributes). This can be proved by induction over the structure of the domain of a generator; the rules in Figure 4 normalize all possible forms of generator domains other than paths.

Our normalization algorithm unnests all type N and J nested queries [16] (using Rules (N7) and (N8) respectively). The important question, though, is whether normalization always improves performance. Unfortunately, this is not always the case. Consider for example the term

$$\{\, (v,v) \mid v \leftarrow \{\, E \mid w \leftarrow X \,\} \,\}$$

where E is a very costly query. This term is normalized into

$$\{\, (E,E) \mid w \leftarrow X \,\}$$

that is, it repeats the computation of E twice. In this case, the normalized form is worse than the original term. Cases like these occur frequently in lazy functional languages [19]. In those languages, function application is evaluated using beta reduction (Rule (N1)), which, if it is implemented naively as term substitution, it may repeat computations (if v appears more than once in e_1). To avoid situations like these, the evaluators of these languages use graph reduction techniques [19] in which all occurrences of v in e_1 share the same term by pointing to the same memory address, thus forming an acyclic graph. When this term is reduced to a value, the term is replaced by this value in the graph, thus avoiding the need to compute this value twice. If we apply this technique to our normalization algorithm, the normalized form $\{\, (E,E) \mid w \leftarrow X \,\}$ will not repeat the evaluation of E; instead it will use two pointers to the same term E.

Even though the normalization algorithm unnests many forms of nested queries, there still some forms of queries that cannot be unnested that way. The following query contains three examples of such forms:

```
{ ⟨ E=e, M={ c | c ← e.children,
                 ∧{ c.age > d.age
                    | d ← e.manager.children } } ⟩
  | e ← Employees,
    e.salary>max{ m.salary | m ← Managers,
                             e.age>m.age } }
```

The inner set comprehension, which appears in the head of the outer set comprehension, cannot be unnested by the nor-

$$X \bowtie_p Y = \{\, (v,w) \mid v \leftarrow X,\ w \leftarrow Y,\ p(v,w) \,\} \tag{O1}$$

$$\sigma_p(X) = \{\, v \mid v \leftarrow X,\ p(v) \,\} \tag{O2}$$

$$\mu_p^{path}(X) = \{\, (v,w) \mid v \leftarrow X,\ w \leftarrow path(v),\ p(v,w) \,\} \tag{O3}$$

$$\Delta_p^{\oplus/e}(X) = \oplus\{\, e(v) \mid v \leftarrow X,\ p(v) \,\} \tag{O4}$$

$$\begin{aligned} X \mathbin{=\!\!\bowtie}_p Y = \{\, (v,w) \mid v \leftarrow X,\ w \leftarrow & \textbf{if } \wedge\{\, \neg p(v,w') \mid v \neq \text{NULL},\ w' \leftarrow Y \,\} \\ & \textbf{then } \{\text{NULL}\} \\ & \textbf{else } \{\, w' \mid w' \leftarrow Y,\ p(v,w') \,\} \,\} \end{aligned} \tag{O5}$$

$$\begin{aligned} =\!\!\!\mu_p^{path}(X) = \{\, (v,w) \mid v \leftarrow X,\ w \leftarrow & \textbf{if } \wedge\{\, \neg p(v,w') \mid v \neq \text{NULL},\ w' \leftarrow path(v) \,\} \\ & \textbf{then } \{\text{NULL}\} \\ & \textbf{else } \{\, w' \mid w' \leftarrow path(v),\ p(v,w') \,\} \,\} \end{aligned} \tag{O6}$$

$$\Gamma_{p/g}^{\oplus/e/f}(X) = \{\, (f(v), \oplus\{\, e(w) \mid w \leftarrow X,\ g(w) \neq \text{NULL},\ f(v) = f(w),\ p(w) \,\}) \mid v \leftarrow X \,\} \tag{O7}$$

Figure 5: The Semantics of the Algebraic Operators

Rules (C5) through (C7) apply to inner comprehensions and are similar to Rules (C2) through (C4) with the only difference that reductions become nests, joins become left outer-joins, and unnests become outer-unnests. The notation $w \backslash u$ indicates all the variables in w that do not appear in u. These are the attributes to group by (u are the attributes to convert into zeros when they are nulls). Rules (C8) and (C9) perform the actual unnesting. They do exactly what we have done in Figure 2 when we composed boxes: here the boxes are actually the results of the translation of the outer and inner comprehensions. Rule (C8) unnests a nested comprehension in the predicate p. It is applied as early as possible, that is, immediately when the generators $\bar{s}$ do not affect the inner comprehension (i.e. when the free variables of the inner comprehensions do not depend on the generator variables in $\bar{s}$). Rule (C9) unnests a nested comprehension in the head of a comprehension. This unnesting is performed when all the generators of the outer comprehension have been reduced.

For example, QUERY D is compiled as follows:

$$\begin{aligned} &\llbracket \{\, \langle E = e,\ M = +\{\, 1 \mid c \leftarrow e.\text{children}, \\ &\qquad\qquad \wedge\{\, c.\text{age} > \text{d.age} \\ &\qquad\qquad\quad \mid d \leftarrow e.\text{manager.children},\ \text{true} \,\} \,\} \rangle \\ &\quad \mid e \leftarrow \text{Employees},\ \text{true} \,\} \rrbracket_{()}^{()}\ \{()\} \end{aligned}$$

$$\begin{aligned} &= \llbracket \{\, \langle E = e,\ M = +\{\, 1 \mid c \leftarrow e.\text{children}, \\ &\qquad\qquad \wedge\{\, c.\text{age} > \text{d.age} \\ &\qquad\qquad\quad \mid d \leftarrow e.\text{manager.children},\ \text{true} \,\} \,\} \rangle \\ &\quad \mid \text{true} \,\} \rrbracket_{e}^{()}\ \text{Employees} \end{aligned}$$

from (C1), if we ignore the selection over Employees since it has a true predicate.

$$\begin{aligned} &= \llbracket \{\, \langle E = e,\ M = m \rangle \mid \text{true} \,\} \rrbracket_{(\text{e,m})}^{()} \\ &\quad (\llbracket +\{\, 1 \mid c \leftarrow e.\text{children}, \\ &\qquad\qquad \wedge\{\, c.\text{age} > \text{d.age} \mid \text{d} \leftarrow \text{e.manager.children},\ \text{true} \,\} \,\} \rrbracket_{\text{e}}^{\text{e}} \\ &\quad\ \text{Employees}) \end{aligned}$$

from (C9) to handle the inner + comprehension.

$$\begin{aligned} &= \llbracket \{\, \langle E = e,\ M = m \rangle \mid \text{true} \,\} \rrbracket_{(\text{e,m})}^{()} \\ &\quad (\llbracket +\{\, 1 \mid \wedge\{\, c.\text{age} > \text{d.age} \mid \text{d} \leftarrow \text{e.manager.children},\ \text{true} \,\} \,\} \rrbracket_{(\text{e,c})}^{\text{e}} \\ &\qquad (=\!\!\!\mu_{\lambda(\text{e,c}).\text{true}}^{\lambda\text{e. e.children}}(\text{Employees}))) \end{aligned}$$

from (C7) to translate the e.children into an unnest.

$$\begin{aligned} &= \llbracket \{\, \langle E = e,\ M = m \rangle \mid \text{true} \,\} \rrbracket_{(\text{e,m})}^{()} \\ &\quad (\llbracket +\{\, 1 \mid k \,\} \rrbracket_{((e,c),k)}^{e} && \text{from (C8)} \\ &\qquad (\llbracket \wedge\{\, c.\text{age} > \text{d.age} \mid \text{d} \leftarrow \text{e.manager.children},\ \text{true} \,\} \rrbracket_{(\text{e,c})}^{(\text{e,c})} \\ &\qquad\quad (=\!\!\!\mu_{\lambda(e,c).\text{true}}^{\lambda\text{e. e.children}}(\text{Employees})))) \end{aligned}$$

$$\begin{aligned} &= \llbracket \{\, \langle E = e,\ M = m \rangle \mid \text{true} \,\} \rrbracket_{(\text{e,m})}^{()} \\ &\quad (\llbracket +\{\, 1 \mid k \,\} \rrbracket_{((e,c),k)}^{e} && \text{from (C7)} \\ &\qquad (\llbracket \wedge\{\, c.\text{age} > \text{d.age} \mid \text{true} \,\} \rrbracket_{((\text{e,c}),\text{d})}^{(\text{e,c})} \\ &\qquad\quad (=\!\!\!\mu_{\lambda((e,c),d).\text{true}}^{\lambda(e,c).\text{ e.manager.children}}(=\!\!\!\mu_{\lambda(e,c).\text{true}}^{\lambda\text{e. e.children}}(\text{Employees}))))) \end{aligned}$$

$$\begin{aligned} &= \llbracket \{\, \langle E = e,\ M = m \rangle \mid \text{true} \,\} \rrbracket_{(\text{e,m})}^{()} \\ &\quad (\llbracket +\{\, 1 \mid k \,\} \rrbracket_{((e,c),k)}^{e} && \text{from (C5)} \\ &\qquad (\Gamma_{\lambda((e,c),d).\text{true}/\lambda((\text{e,c}),\text{d}).\text{d}}^{\wedge/\lambda((e,c),d).\text{ c.age}>\text{d.age}/\lambda((\text{e,c}),\text{d}).(\text{e,c})} \\ &\qquad\quad (=\!\!\!\mu_{\lambda((e,c),d).\text{true}}^{\lambda(e,c).\text{ e.manager.children}}(=\!\!\!\mu_{\lambda(e,c).\text{true}}^{\lambda\text{e. e.children}}(\text{Employees}))))) \end{aligned}$$

$$\frac{\sigma \vdash X : set(t_1),\ \sigma \vdash Y : set(t_2),\ \sigma \vdash p : t_1 \times t_2 \rightarrow \text{bool}}{\sigma \vdash X \bowtie_p Y : set(t_1 \times t_2)}$$

$$\frac{\sigma \vdash X : set(t),\ \sigma \vdash p : t \rightarrow \text{bool}}{\sigma \vdash \sigma_p(X) : set(t)}$$

$$\frac{\sigma \vdash X : set(t_1),\ \sigma \vdash path : t_1 \rightarrow set(t_2),\ \sigma \vdash p : t_1 \times t_2 \rightarrow \text{bool}}{\sigma \vdash \mu_p^{path}(X) : set(t_1 \times t_2)}$$

$$\frac{\sigma \vdash X : set(t_1),\ \sigma \vdash e : t_1 \rightarrow \oplus_T,\ \sigma \vdash p : t_1 \rightarrow \text{bool}}{\sigma \vdash \Delta_p^{\oplus/e}(X) : \oplus_T}$$

$$\frac{\begin{array}{l}\sigma \vdash X : set(t_1),\ \sigma \vdash e : t_1 \rightarrow \oplus_T,\ \sigma \vdash f : t_1 \rightarrow t_2 \\ \sigma \vdash p : t_1 \rightarrow \text{bool},\ \sigma \vdash \text{g} : \text{t}_1 \rightarrow \text{t}_3\end{array}}{\sigma \vdash \Gamma_{p/g}^{\oplus/e/f}(X) : set(t_1 \times \oplus_T)}$$

Figure 6: The Typing Rules of the Algebraic Operators

malization algorithm because the computed set must be embedded in the result of every iteration of the outer set comprehension. Similarly, the universal quantification (the $\wedge$-comprehension) and the aggregation (the *max*-comprehension) cannot be unnested by the normalization algorithm. These cases (which are types A and JA nested queries [16]) require the use of outer-joins and grouping and they will be covered in detail in the rest of this paper.

3 The Nested Relational Algebra

Before we describe the unnesting algorithm, we need to define the nested-relational algebraic operators more formally. Figure 5 defines the algebraic operators in terms of the monoid calculus and Figure 6 gives the typing rules of these operators (The last two rules in Figure 6 are defined for a primitive monoid $\oplus$; there are similar rules for a set monoid). We decided to use pairs of values instead of a stream of values to pass values between operators. For example, in the stream-based approach, the join operator concatenates each pair of qualified tuples from the two input streams into a new tuple and makes a stream from these new tuples. In our formal definition, though, a join between X and Y (see Equation (O1) in Figure 5) accepts any value v from X and any value w of Y and generates a set of all qualified pairs (v, w). (v and w can be nested pairs, such as $(x, (y, z))$.) This approach gives a compositional way of defining operators that does not depend on the structure of input. Instead, in the stream-based view, the free variables of the join predicate depend on the structure of the input streams and do not have a valid meaning if they applied to different streams. Even though streams are convenient forms for explaining examples (as we did in the introduction), they are not appropriate for giving semantics and proving theorems. Of course, in the actual implementation of the algebraic operators we can always use streams, as it is done for real physical algorithms.

Equations (O1) through (O4) are straightforward. The outer-join in Equation (O5) is a little bit different than the join in Equation (O1): the domain of the second generator (the generator of w) is always non-empty; if Y is empty or there are no elements that can be joined with v (this is tested using the universal quantification), then the domain is the singleton value {NULL}, i.e. w becomes null; otherwise each qualified element w of Y is joined with v. The outer-unnest operation in Equation (O6) works in the same way as the outer-join operator.

In Equation (O7), the nest operator uses the group-by function f: if two values v and w from a set X are equal under f (i.e. when $f(v) = f(w)$), their images under e (i.e. $e(v)$ and $e(w)$) are grouped together in the same group. After a group is formed, it is reduced by the accumulator $\oplus$ and a pair of the group-by value along with the result of the reduction of this group is returned. Function g indicates which nulls to convert into zeros (i.e. into $Z_\oplus$). For example,

$$\Gamma_{\lambda(d,e).\text{true}/\lambda(d,e).e}^{\cup/\lambda(d,e).e/\lambda(d,e).d}(X)$$
$$= \{\, (d', \{\, e \mid (d,e) \leftarrow X,\ e \neq \text{NULL},\ d' = d \,\}) \mid (d', e') \leftarrow X \,\}$$

in Figure 1.B, groups the input (which consists of pairs (d, e) of a department d and an employee e) by d and converts the null e's into empty sets. The result of the nesting is a set of pairs, where each pair associates a department with a set of employees.

4 The Query Unnesting Algorithm

The query unnesting algorithm is given in Figure 7. We assume that all comprehensions in a query have been put into the canonical form $\oplus\{\, e \mid v_1 \leftarrow path_1, \ldots, v_n \leftarrow path_n, pred \,\}$ before this algorithm is applied. That is, all generator domains have been reduced to paths and all predicates have been collected to the right of the comprehension into *pred* by anding them together (*pred* is set to true if no predicate exists). The translation of a monoid comprehension $\oplus\{\, e \mid \overline{r} \,\}$ is accomplished by using $[\![\oplus\{\, e \mid \overline{r} \,\}]\!]_w^u\ E$. The comprehension $\oplus\{\, e \mid \overline{r} \,\}$ is translated by compiling the qualifiers in $\overline{r}$ from left to right using the term E as a seed that grows at each step. That is, the term E is the algebraic tree derived at this point of compilation. The variables in w are all the variables encountered so far during the translation and u are the variables that need to be converted to zeros during nesting if they are nulls. When $u = ()$ (i.e. when we have no variables in u), this indicates that we are compiling an outermost comprehension (not a nested one). Rules (C1) through (C4) compile outermost comprehensions while Rules (C5) through (C7) compile inner comprehensions. Rules (C8) and (C9) do the actual unnesting (here u can be of any value, including (), and $\otimes$ is not necessarily the same monoid as $\oplus$).

Rule (C1) is the first step of the unnesting algorithm: the comprehension must be the outermost comprehension; thus, the first generator must be over an extent X. In that case, the seed becomes a selection over X. The notation $p[v]$ specifies the part of the predicate p that refers to v exclusively. The rest of the predicate is denoted by $p[\overline{v}]$ and satisfies $p[v] \wedge p[\overline{v}] = p$. This is used for pushing predicates to the appropriate operators. Rule (C2) is the last rule to be performed after all generators have been compiled. Rule (C3) converts a generator over an extent into a join. Here we split the predicate p into three parts: into $p[v]$ that refers to v exclusively, into $p[(w, v)]$ that refers to both w and v, and $p[\overline{(w, v)}]$ for the rest of the predicate. Rule (C4) compiles generators with path domains into unnests.

$$\llbracket \oplus\{\, e \mid v \leftarrow X,\, \overline{r},\, p \,\} \rrbracket^{()}_{()} \, \{()\} \;=\; \llbracket \oplus\{\, e \mid \overline{r},\, p[\overline{v}] \,\} \rrbracket^{()}_{v} \, (\sigma_{\lambda v.p[v]}(X)) \tag{C1}$$

$$\llbracket \oplus\{\, e \mid p \,\} \rrbracket^{()}_{w} \, E \;=\; \Delta^{\oplus/\lambda w.e}_{\lambda w.p}(E) \tag{C2}$$

$$\llbracket \oplus\{\, e \mid v \leftarrow X,\, \overline{r},\, p \,\} \rrbracket^{()}_{w} \, E \;=\; \llbracket \oplus\{\, e \mid \overline{r},\, p[\overline{(w,v)}] \,\} \rrbracket^{()}_{(w,v)} \, (E \bowtie_{\lambda(w,v).p[(w,v)]} (\sigma_{\lambda v.p[v]}(X))) \tag{C3}$$

$$\llbracket \oplus\{\, e \mid v \leftarrow path,\, \overline{r},\, p \,\} \rrbracket^{()}_{w} \, E \;=\; \llbracket \oplus\{\, e \mid \overline{r},\, p[\overline{v}] \,\} \rrbracket^{()}_{(w,v)} \, (\mu^{\lambda w.path}_{\lambda(w,v).p[v]}(E)) \tag{C4}$$

$$\llbracket \oplus\{\, e \mid p \,\} \rrbracket^{u}_{w} \, E \;=\; \Gamma^{\oplus/\lambda w.e/\lambda w.u}_{\lambda w.p/\lambda w.w\setminus u}(E) \tag{C5}$$

$$\llbracket \oplus\{\, e \mid v \leftarrow X,\, \overline{r},\, p \,\} \rrbracket^{u}_{w} \, E \;=\; \llbracket \oplus\{\, e \mid \overline{r},\, p[\overline{(w,v)}] \,\} \rrbracket^{u}_{(w,v)} \, (E \mathbin{=\!\!\bowtie}_{\lambda(w,v).p[(w,v)]} (\sigma_{\lambda v.p[v]}(X))) \tag{C6}$$

$$\llbracket \oplus\{\, e \mid v \leftarrow path,\, \overline{r},\, p \,\} \rrbracket^{u}_{w} \, E \;=\; \llbracket \oplus\{\, e \mid \overline{r},\, p[\overline{v}] \,\} \rrbracket^{u}_{(w,v)} \, (=\!\!\!\mu^{\lambda w.path}_{\lambda(w,v).p[v]}(E)) \tag{C7}$$

$$\llbracket \oplus\{\, e_1 \mid \overline{s},\, p(\otimes\{\, e_2 \mid \overline{r} \,\}) \,\} \rrbracket^{u}_{w} \, E \;=\; \llbracket \oplus\{\, e_1 \mid \overline{s},\, p(v) \,\} \rrbracket^{u}_{(w,v)} \, (\llbracket \otimes\{\, e_2 \mid \overline{r} \,\} \rrbracket^{w}_{w} \, E) \tag{C8}$$

if $\otimes\{\, e_2 \mid \overline{r} \,\}$ does not depend on the $\overline{s}$ generators

$$\llbracket \oplus\{\, f(\otimes\{\, e \mid \overline{r} \,\}) \mid p \,\} \rrbracket^{u}_{w} \, E \;=\; \llbracket \oplus\{\, f(v) \mid p \,\} \rrbracket^{u}_{(w,v)} \, (\llbracket \otimes\{\, e \mid \overline{r} \,\} \rrbracket^{w}_{w} \, E) \tag{C9}$$

Figure 7: Translating and Unnesting Comprehensions ($u \neq ()$ in Rules C5-C7)

$$\begin{aligned}
&= \llbracket \{\, \langle E = e,\ M = m \rangle \mid \text{true} \,\} \rrbracket^{()}_{(\text{e,m})} \\
&\quad (\Gamma^{+/\lambda((e,c),k).1/\lambda((e,c),k).e}_{\lambda((e,c),k).k/\lambda((e,c),k).e} && \text{from (C5)} \\
&\quad\; (\Gamma^{\wedge/\lambda((e,c),d).\ \text{c.age}>\text{d.age}/\lambda((e,c),d).(e,c)}_{\lambda((e,c),d).\text{true}/\lambda((e,c),d).d} \\
&\quad\;\; (=\!\!\!\mu^{\lambda(e,c).\ \text{e.manager.children}}_{\lambda((e,c),d).\text{true}} (=\!\!\!\mu^{\lambda e.\ \text{e.children}}_{\lambda(e,c).\text{true}} (\text{Employees})))))
\\
&= \Delta^{\cup/\lambda(e,m).\langle E=e,\ M=m \rangle}_{\lambda(e,m).\text{true}} \\
&\quad (\Gamma^{+/\lambda((e,c),k).1/\lambda((e,c),k).e}_{\lambda((e,c),k).k/\lambda((e,c),k).e} && \text{from (C2)} \\
&\quad\; (\Gamma^{\wedge/\lambda((e,c),d).\ \text{c.age}>\text{d.age}/\lambda((e,c),d).(e,c)}_{\lambda((e,c),d).\text{true}/\lambda((e,c),d).d} \\
&\quad\;\; (=\!\!\!\mu^{\lambda(e,c).\ \text{e.manager.children}}_{\lambda((e,c),d).\text{true}} (=\!\!\!\mu^{\lambda e.\ \text{e.children}}_{\lambda(e,c).\text{true}} (\text{Employees})))))
\end{aligned}$$

We can easily prove that the unnesting algorithm is complete:

Theorem 1 *The rules in Figure 7 unnest all nested comprehensions.*

Proof: After normalization, the only places where we can find nested queries are the comprehension predicate and the head of the comprehension. These cases are handled by Rules (C8) and (C9) respectively. Even though Rule (C8) has a precondition, it will be eventually applied to unnest any nested query in a predicate. (In the worst case, it will be applied when all generators of the outer comprehension have beed compiled by the other rules.) □

The soundness of our unnesting algorithm is a consequence of the following theorem:

Theorem 2 *The rules in Figure 7 are meaning preserving. That is:*

$$\llbracket \oplus\{\, e \mid \overline{r} \,\} \rrbracket^{()}_{()} \, \{()\} \;=\; \oplus\{\, e \mid \overline{r} \,\} \tag{TH1}$$

The proof of this theorem is given in the extended version of this paper [11]. Here we give an example of the validity of this theorem. If we apply the rules in Figure 7, QUERY B becomes:

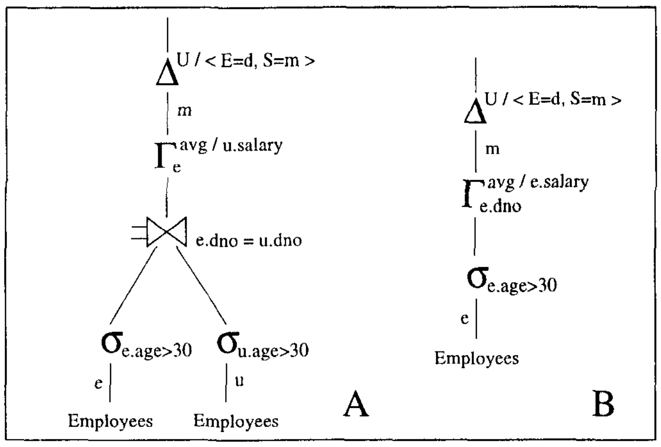
$$\begin{aligned}
&\llbracket \{\, \langle D = d,\ M = \{\, e \mid e \leftarrow \text{Employees},\ \text{e.dno} = \text{d.dno} \,\} \rangle \\
&\quad \mid d \leftarrow \text{Department} \,\} \rrbracket^{()}_{()} \, \{()\} \\
&= \Delta^{\cup/\lambda(d,m).\langle D=d,\ M=m \rangle}_{\lambda(d,m).\text{true}} (\Gamma^{\cup/\lambda(d,e).e/\lambda(d,e).d}_{\lambda(d,e).\text{true}/\lambda(\text{d,e}).\text{e}} (\mathcal{G}))
\end{aligned}$$

where $\mathcal{G}$ = Department $\mathbin{=\!\!\bowtie}_{\lambda(\text{d,e}).\ \text{e.dno}=\text{d.dno}}$ Employees. If we use the operator definitions in Figure 5 and normalize the resulting comprehensions, we get:

$$\begin{aligned}
&= \{\, \langle D = d,\ M = m \rangle \mid (d,m) \leftarrow (\Gamma^{\cup/\lambda(d,e).e/\lambda(d,e).d}_{\lambda(d,e).\text{true}/\lambda(\text{d,e}).\text{e}} (\mathcal{G})) \,\} \\
&= \{\, \langle D = d,\ M = m \rangle \\
&\quad \mid (d,m) \leftarrow \{\, (d, \{\, e' \mid (d',e') \leftarrow (\mathcal{G}),\ e' \neq \text{NULL},\ d = d' \,\}) \\
&\qquad\qquad \mid (d,e) \leftarrow (\mathcal{G}) \,\} \,\} \\
&= \{\, \langle D = d,\ M = \{\, e' \mid (d',e') \leftarrow (\mathcal{G}),\ e' \neq \text{NULL},\ d = d' \,\} \rangle \\
&\quad \mid (d,e) \leftarrow (\mathcal{G}) \,\} \\
&= \{\, \langle D = d,\ M = \{\, e' \mid (d',e') \leftarrow \{\, (d,e) \mid d \leftarrow \text{Department}, \\
&\qquad\qquad e \leftarrow F(d) \,\},\ e' \neq \text{NULL},\ d = d' \,\} \rangle \\
&\quad \mid (d,e) \leftarrow \{\, (d,e) \mid d \leftarrow \text{Department},\ \text{e} \leftarrow \text{F(d)} \,\} \,\}
\end{aligned}$$

where

$$\begin{aligned}
F(d) \;=\;& \textbf{if}\ \wedge\{\, \text{e.dno} \neq \text{d.dno} \mid \text{d} \neq \text{NULL},\ \text{e} \leftarrow \text{Employees} \,\} \\
& \textbf{then}\ \{\text{NULL}\} \\
& \textbf{else}\ \{\, e \mid e \leftarrow \text{Employees},\ \text{e.dno} = \text{d.dno} \,\}
\end{aligned}$$



$\Delta^{U / < E=d, S=m >}$
m
$\Gamma^{avg / u.salary}_{e}$
e.dno = u.dno
$\sigma_{e.age>30}$
$\sigma_{u.age>30}$
e
u
Employees
Employees
A
$\Delta^{U / < E=d, S=m >}$
m
$\Gamma^{avg / e.salary}_{e.dno}$
$\sigma_{e.age>30}$
e
Employees
B

Figure 8: Simplification of an Algebraic Form

Therefore, after *normalization* we get:

$$
\begin{aligned}
&= \{\, \langle D = d,\ M = \{\, e' \mid d' \leftarrow \text{Department},\ e' \leftarrow \text{F}(d'),\\
&\qquad\qquad e' \neq \text{NULL},\ d = d' \,\}\rangle\\
&\quad \mid d \leftarrow \text{Department},\ e \leftarrow \text{F(d)} \,\}\\
&= \{\, \langle D = d,\ M = \{\, e' \mid d' \leftarrow \text{Department},\\
&\qquad e' \leftarrow \{\, e \mid e \leftarrow \text{Employees},\ \text{e.dno} = \text{d.dno} \,\},\ \text{d} = \text{d}' \,\}\rangle\\
&\quad \mid d \leftarrow \text{Department},\ e \leftarrow \text{F(d)} \,\}\\
&= \{\, \langle D = d,\ M = \{\, e \mid d' \leftarrow \text{Department},\ e \leftarrow \text{Employees},\\
&\qquad\qquad e.\text{dno} = \text{d.dno},\ \text{d} = \text{d}' \,\}\rangle\\
&\quad \mid d \leftarrow \text{Department} \,\}
\end{aligned}
$$

In the last term we removed the generator $e \leftarrow F(d)$ because for every $Y \neq \emptyset$, we have $\{\, f(x) \mid x \leftarrow X,\ y \leftarrow Y,\ \overline{r(x)} \,\} = \{\, f(x) \mid x \leftarrow X,\ \overline{r(x)} \,\}$, since Y does not contribute to the result. Finally, since $d = d'$, we can safely remove the generator $d' \leftarrow$ Department from the inner comprehension. The final form is:

$$
\begin{aligned}
&\{\, \langle D = d,\ M = \{\, e \mid e \leftarrow \text{Employees},\ \text{e.dno} = \text{d.dno} \,\}\rangle\\
&\mid d \leftarrow \text{Department} \,\}
\end{aligned}
$$

which is the original comprehension.

5 Simplifications

There is a large class of nested queries that can be improved further after unnesting. Consider for example the following query that, for each department, it finds the average salary of all the employees in the department older than 30:

```
select distinct e.dno, avg(e.salary)
 from Employees e
where e.age>30
group by e.dno
```

Even though this query does not seem to be nested at a first glance, its translation to the monoid calculus is in fact nested:

$$
\begin{aligned}
&\{\, \langle E = e.\text{dno},\ \text{S} = avg\{\, \text{u.salary} \mid \text{u} \leftarrow \text{Employees},\\
&\qquad\qquad u.\text{age} > 30,\ \text{e.dno} = \text{u.dno} \,\}\rangle\\
&\mid e \leftarrow \text{Employees},\ \text{e.age} > 30 \,\}
\end{aligned}
$$

Our unnesting algorithm generates the algebraic form in Figure 8.A, but we would prefer to have the form in Figure 8.B, which is more efficient. This simplification can be easily accomplished with the help of the following rule:

$$\Gamma^{f(b)}_{a}(g(a) \mathrel{=\!\!\bowtie}_{a.M=b.M} g(b)) \quad\rightarrow\quad \Gamma^{f(a)}_{a.M}(g(a))$$

where a/b are range variables in $g(a)/g(b)$.

6 Building the Optimizer

We have already built a prototype OQL optimizer based on the unnesting algorithm described in this paper. It is described in detail in [10]. Our OQL optimizer is expressed in a very powerful optimizer specification language, called OPTL, and is implemented in a flexible optimization framework, called OPTGEN, which extends our earlier work on optimizer generators [14].

OPTL is a language for specifying query optimizers that captures a large portion of the optimizer specification information in a declarative manner. It extends C++ with a number of term manipulation constructs and with a rule language for specifying query transformations. OPTGEN is a C++ preprocessor that maps OPTL specification into executable code (C++ code).

Our OQL optimizer is only 825 lines of OPTL code (the produced C++ code is 4733 lines), from which 30 lines are for normalization of comprehensions, 34 lines for normalization of predicates (using DeMorgan's laws), 88 lines for query unnesting using the algorithm described in this paper, 42 lines for materialization of path expressions into joins [1], 48 lines for various algebraic optimizations (including permutation of joins), and 126 lines for translating algebraic forms into physical plans. The rest of the optimizer code is C++ support functions. Currently our OQL optimizer produces physical plans that are evaluated in memory, but we are planning to connect it to the SHORE object management system [3].

The source code and the manual of OPTGEN, and the OQL optimizer are available at:

`http://www-cse.uta.edu/~fegaras/optimizer/`

7 Related Work

Monad comprehensions were first introduced by Wadler [23] as a generalization of list comprehensions. Monoid comprehensions are related to monad comprehensions but they are considerably more expressive. In particular, monoid comprehensions can mix inputs from different collection types and may return output of a different type. This is not possible for monad comprehensions, since they restrict the inputs and the output of a comprehension to be of the same type. Monad comprehensions were first proposed as a convenient database language by Trinder [22, 21] who also presented many algebraic transformations over these forms as well as methods for converting comprehensions into joins. The monad comprehension syntax was also adopted by Buneman et al [2] as an alternative syntax to monoid homomorphisms.

Our normalization algorithm is influenced by Wong's work on normalization of monad comprehensions [24, 25]. He presented some very powerful rules for flattening nested comprehensions into canonical comprehension forms whose generators are over simple paths. These canonical forms are equivalent to our canonical forms for monoid comprehensions. His work though does not address query unesting for complex queries (in which the embedded query is part of the predicate or the comprehension head), which cannot be unnested without using outer-joins and grouping.

Our query unnesting algorithm is influenced by the work of Cluet and Moerkotte [9, 8], which covered many cases of nesting in OODBs, including nested aggregate queries, and, more importantly, validated all the transformations. Our work proposes a rewriting system for complete unnesting, while their work considers algebraic equalities for some forms of unnesting. Another promising work on query unnesting that has the same goals as ours is that of Cherniack and Zdonik [5]. In contrast to our approach, they used an automatic theorem prover to prove the soundness of their unnesting rewrite rules. The use of a theorem prover is highly desirable for extensible systems, since it makes the query optimizer very flexible. In particular, with the help of a theorem prover, an optimizer does not require validation in a form of a formal proof each time a new algebraic operator or a new rewrite rule is introduced. We are planning to experiment along this direction in the near future.

The work of Lin and Ozsoyoglu [17] addresses nested queries in a different way. For each nested query, a method is created and the inner query is replaced by a call to this method. This is a nice approach if the goal is to translate a language that allows query nesting to a language or algebra that does not allow it. But of course this is not a solution to the query unnesting problem since it replaces this problem with a more difficult problem: namely, the optimization of queries with embedded method calls.

8 Conclusion

We have presented a new query unnesting algorithm that removes any form of nesting in a very expressive calculus that supports nested sets, aggregation, and universal and existential quantification. This algorithm is compositional, that is, the translation of an embedded query does not depend on the context in which it is embedded; instead, each query is translated independently, and all translations are composed to form the final unnested query. This property enabled us to prove the soundness and completeness of the algorithm. Our unnesting algorithm is efficient since it takes time linear to the size of the query. It is also very easy to implement and can be easily adapted to handle object-relational and relational queries by removing some functionality (namely the unnesting and outer-unnesting operators) and handling more syntactic sugar. Our preliminary results suggest that various OODB optimization techniques improve performance considerably when combined with query unnesting.

As a future work, we are planning to extend this algorithm to support lists, bags, and vectors. There is a fundamental difficulty that prevents an easy solution: grouping alone is not capable of reconstructing the input stream after it is extended by outer-joins and outer-unnests because all these collection types are not idempotent and, therefore, it is not obvious how to extract the correct number of duplicates from the extended stream. Another goal is to quantify the performance improvement gained by query unnesting by testing various nested queries. These results would be highly sensitive to the other optimization techniques supported by the optimizer.

Acknowledgements: The author is grateful to David Levine for helpful comments on the paper. This work is supported in part by the National Science Foundation under grant IRI-9509955.

References

[1] J. Blakeley, W. McKenna, and G. Graefe. Experiences Building the Open OODB Query Optimizer. *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Washington, D.C.*, pp 287–296, May 1993.

[2] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension Syntax. *SIGMOD Record*, 23(1):87–96, March 1994.

[3] M. Carey, D. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White, and M. Zwilling. Shoring Up Persistent Applications. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota*, pp 383–394, May 1994.

[4] R. Cattell. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, 1994.

[5] M. Cherniack and S. Zdonik. Changing the Rules: Transformations for Rule-Based Optimizers. *ACM SIGMOD International Conference on Management of Data, Seattle, Washington*, June 1998.

[6] M. Cherniack, S. Zdonik, J. Lee, and K. Kim. Composing Rules the COKO-KOLA Way. Brown University, March 1997.

[7] J. Claussen, A. Kemper, G. Moerkotte, and K. Peithner. Optimizing Queries with Universal Quantification in Object-Oriented and Object-Relational Databases. In *Proceedings of the 23th VLDB Conference, Athens, Greece*, pp 286–295, September 1997.

[8] S. Cluet and G. Moerkotte. Efficient Evaluation of Aggregates on Bulk Types. Technical report, Aachen University of Technology, October 1995. Technical Report 95-05.

[9] S. Cluet and G. Moerkotte. Nested Queries in Object Bases. In *Fifth International Workshop on Database Programming Languages, Gubbio, Italy*, September 1995.

[10] L. Fegaras. An Experimental Optimizer for OQL. University of Texas at Arlington Technical Report TR-CSE-97-007. Available at `http://www-cse.uta.edu/~fegaras/oqlopt.ps.gz`, May 1997.

[11] L. Fegaras. Query Unnesting in Object-Oriented Databases (extended version). Available at `http://www-cse.uta.edu/~fegaras/sigmod98.ps`, January 1998.

[12] L. Fegaras and D. Maier. An Algebraic Framework for Physical OODB Design. In *Fifth International Workshop on Database Programming Languages, Gubbio, Italy*, September 1995.

[13] L. Fegaras and D. Maier. Towards an Effective Calculus for Object Query Languages. *ACM SIGMOD International Conference on Management of Data, San Jose, California*, pp 47–58, May 1995.

[14] L. Fegaras, D. Maier, and T. Sheard. Specifying Rule-based Query Optimizers in a Reflective Framework. *Deductive and Object-Oriented Databases, Phoenix, Arizona*, pp 146–168, December 1993. Springer-Verlag, LNCS 461.

[15] R. Ganski and H. Wong. Optimization of Nested SQL Queries Revisited. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, San Francisco, California*, pp 23–33, May 1987.

[16] W. Kim. On Optimizing an SQL-like Nested Query. *ACM Transactions on Database Systems*, 7(3):443–469, September 1982.

[17] J. Lin and M. Ozsoyoglu. Processing OODB Queries by O-Algebra. In *International Conference on Information and Knowledge Management (CIKM), Rockville, Maryland*, November 1996.

[18] M. Muralikrishna. Improved Unnesting Algorithms for Join Aggregate SQL Queries. In *Proc. Int'l. Conf. on Very Large Data Bases*, page 91, Vancouver, BC, Canada, August 1992.

[19] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, Inc., 1987.

[20] H. Steenhagen, P. Apers, and H. Blanken. Optimization of Nested Queries in a Complex Object Model. In M. Jarke, J. Bubenko, and K. Jeffery, editors, *Advances in Database Technology – EDBT '94*, pp 337–350. Springer-Verlag, 1994. LNCS 779.

[21] P. Trinder. Comprehensions: A Query Notation for DBPLs. In *Proceedings of the Third International Workshop on Database Programming Languages: Bulk Types and Persistent Data, Nafplion, Greece*, pp 55–68. Morgan Kaufmann Publishers, Inc., August 1991.

[22] P. Trinder and P. Wadler. Improving List Comprehension Database Queries. In *in Proceedings of TENCON'89, Bombay, India*, pp 186–192, November 1989.

[23] P. Wadler. Comprehending Monads. *Proceedings of the ACM Symposium on Lisp and Functional Programming, Nice, France*, pp 61–78, June 1990.

[24] L. Wong. Normal Forms and Conservative Properties for Query Languages over Collection Types. *Proceedings of the 12th ACM Symposium on Principles of Database Systems, Washington, DC*, pp 26–36, May 1993.

[25] L. Wong. *Querying Nested Collections*. PhD thesis, Univerity of Pennsylvania, March 1994. Also appeared as a Univerity of Pennsylvania technical report IRCS Report 94-09.