Query Unnesting in Object-Oriented Databases

Leonidas Fegaras
U. of Texas at Arlington

Various Approaches

Source-to-source transformations:

• unnesting SQL	(Kim, Ganski, Muralikrishna)
• magic sets	(Mumick & Pirahesh)

Evaluation techniques:

• query decorrelation	(Seshadri et al)
• memoization (caching)	(Hellerstein)

Algebraic approaches:

• algebraic equalities	(Cluet & Moerkotte)
• normalization	(Fegaras, Trinder, Wong, etc)

Our Approach

is purely algebraic.

Formalism: **monoid comprehensions**

(NF^2 + aggregation + quantification)

It treats nested collections, aggregation, and quantification in the same way.

Many forms of query nesting are removed by normalization; the rest are removed by a simple, compositional, algorithm.

Monoids

A *monoid* is an algebraic structure that captures many collection and aggregate types:

$$(\oplus,\ Z_\oplus)$$

The *merge* function $\oplus$ is associative with *zero* $Z_\oplus$:

$$x \oplus Z_\oplus = Z_\oplus \oplus x = x$$

A parametric type (e.g. set(a)) is associated with a free monoid that has a *unit* $U_\oplus$:

$$(\oplus, Z_\oplus, U_\oplus)$$

It is called a *collection monoid.*

Any other monoid is a *primitive monoid.*

Some Monoids

Collection monoids:

set(a):	($\cup$, $\{\}$, $\lambda x.\ \{x\}$)
bag(a):	($\uplus$, $\mathbf{\{\}}$, $\lambda x.\ \mathbf{\{}x\mathbf{\}}$)
list(a):	(++, [ ], $\lambda x.\ [x]$)

Primitive monoids:

integer:	(+, 0)
integer:	(*, 1)
boolean:	($\vee$, false)
boolean:	($\wedge$, true)

Monoid Comprehensions

A monoid comprehension takes the form:

$$\boxed{\oplus\{\ e \mid r_1, \ldots, r_n\ \}}$$

accumulator head qualifiers

where $\oplus$ is a monoid and each *qualifier* r_i is either:

- a *generator* $v \leftarrow u$, or
- a *filter* pred.

Examples

$\cup\{\ (a,b) \mid a \leftarrow x,\ b \leftarrow y\ \}$

```
res = {};
for each a in x do
    for each b in y do
        res = res ∪ {(a,b)};
return res;
```

$\cup\{\ (a,b) \mid a \leftarrow [1,2,3],\ b \leftarrow \{4,5\}\ \}$
$= \{\ (1,4), (1,5), (2,4), (2,5), (3,4), (3,5)\ \}$

$+\{\ a \mid a \leftarrow [1,2,3],\ a \geq 2\ \}$
$= 2+3 = 5$

Translating ODMG OQL

```
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;
```

$\cup\{\ hotel.price \mid hotel \leftarrow \cup\{\ h \mid c \leftarrow Cities,\ h \leftarrow c.hotels,\ c.name = \text{"Arlington"}\ \},\ \vee\{\ r.bed\_num = 3 \mid r \leftarrow hotel.rooms\ \}\ \}$

Normalization

Canonical form:

$$\oplus\{\ e \mid x_1 \leftarrow path_1, \ldots, x_n \leftarrow path_n, pred\ \}$$

(path is a cascade of projections: $X.A_1.A_2\ldots A_m$)

Two important normalization rules:

$$\oplus\{\ e \mid \boxed{\textcircled{1}}, x \leftarrow \otimes\{\ u \mid \boxed{\textcircled{2}}\ \}, \boxed{\textcircled{3}}\ \}$$
$$\rightarrow\ \oplus\{\ e \mid \boxed{\textcircled{1}}, \boxed{\textcircled{2}}, x \equiv u, \boxed{\textcircled{3}}\ \}$$

$$\oplus\{\ e \mid \boxed{\textcircled{1}}, \vee\{\ pred \mid \boxed{\textcircled{2}}\ \}, \boxed{\textcircled{3}}\ \}$$
$$\rightarrow\ \oplus\{\ e \mid \boxed{\textcircled{1}}, \boxed{\textcircled{2}}, pred, \boxed{\textcircled{3}}\ \}$$

Example

```
∪{  hotel.price
  |  hotel ← ∪{  h  |  c ← Cities,
                       h ← c.hotels,
                       c.name = "Arlington"  },
     ∨{  r.bed_num = 3  |  r ← hotel.rooms  } }

=  ∪{  h.price | c ← Cities,
                 h ← c.hotels,
                 r ← h.rooms,
                 c.name = "Arlington",
                 r.bed_num = 3 }
```

Unnesting OQL Queries

```
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;
```

```
select distinct h.price
 from c in Cities,
      h in c.hotels,
      r in h.rooms
 where c.name = “Arlington”
   and r.bed_num = 3;
```


Query Unnesting by Normalization

Normalization

- eliminates intermediate data structures;
- improves performance in many cases;
- has been shown to be effective in other domains (e.g. logic);
- allows the proof of useful theorems.

But ...

Normalization cannot unnest all forms:

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

In comprehension form:

$\cup\{$ < D = d, E = $\cup\{$ e | e $\leftarrow$ Employees, e.dno = d.dno $\}$ >
| d $\leftarrow$ Departments $\}$

Is Query Unnesting Useful?

Query unnesting promotes the operators of the inner queries into the operators of the outer query.
It allows:

- more choices of evaluating the inner operators;
- rearrangement (sorting) of all operators in one level;
- free movement of predicates between inner and outer queries.

Lessons from Relational Databases

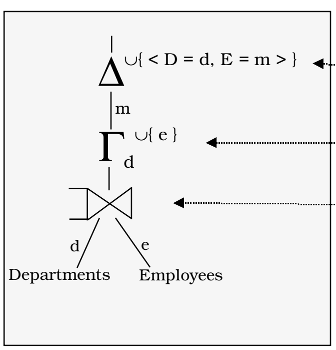
select distinct d.name
from Departments d
where 20 > (**select count**(e.ssn)
from Employees e
where d.dno = e.dno);

⇩

select distinct d.dname
from (Departments d **left-outerjoin** Employees e
where d.dno = e.dno)
group by d.dno
having 20 > **count**(e.ssn);

A Need for an Algebra

$\cup\{ < D = d,\ E = \cup\{ e \mid e \leftarrow \text{Employees},\ e.dno = d.dno \} > \mid d \leftarrow \text{Departments} \}$



Reduce by ∪: form a set of tuples

Nest by d and form a set of e's

Left outer-join

Why both Algebra and Calculus?

The calculus
- is higher-level and uniform;
- has a solid theoretical basis;
- closely resembles OODB languages;
- is easy to normalize.

The algebra
- is lower-level;
- can be directly translated into physical algorithms;
- is a better basis for query unnesting.

Monoid Algebra

$$\sigma_p(R) = \cup\{\, r \mid r \leftarrow R,\ p(r) \,\}$$

$$R \bowtie_p S = \cup\{\, (r,s) \mid r \leftarrow R,\ s \leftarrow S,\ p(r,s) \,\}$$

$$\mu_p^{path}(R) = \cup\{\, (r,s) \mid r \leftarrow R,\ s \leftarrow path(r),\ p(r,s) \,\}$$

$$\Delta_p^{\oplus/e}(R) = \oplus\{\, e(r) \mid r \leftarrow R,\ p(r) \,\}$$

$$\Gamma_p^{\oplus/e/f}(R) = \cup\{\, (\, f(r), \oplus\{\, e(s) \mid s \leftarrow R,\ f(r)=f(s), p(s) \,\}\,) \mid r \leftarrow R \,\}$$

Other operators:

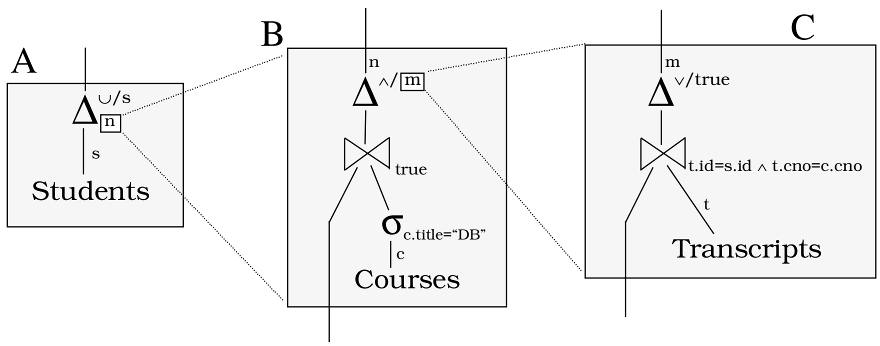
$R =\!\bowtie_p S$ left outer-join

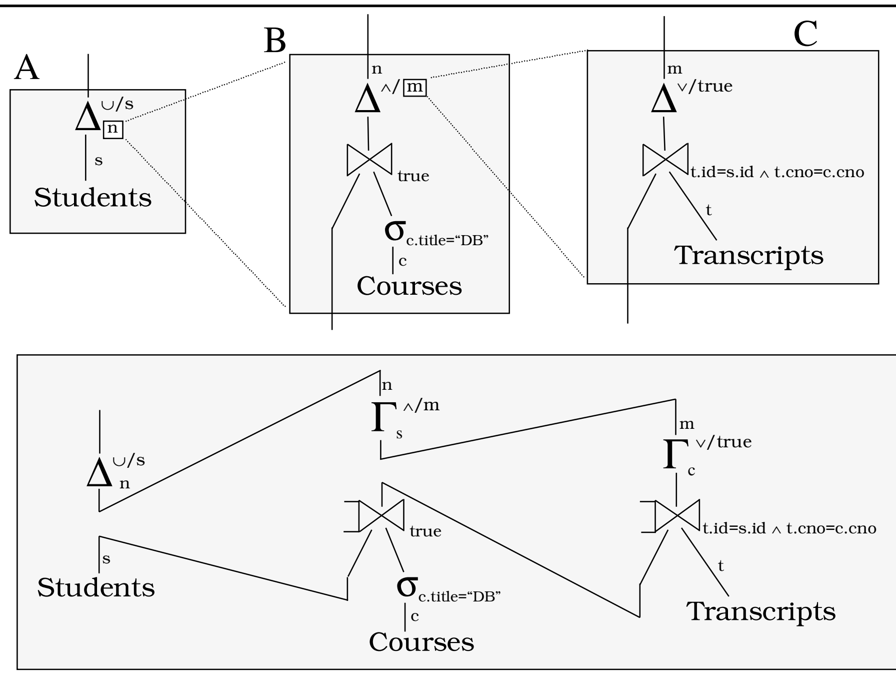
$=\!\mu_p^{path}(R)$ outer-unnest

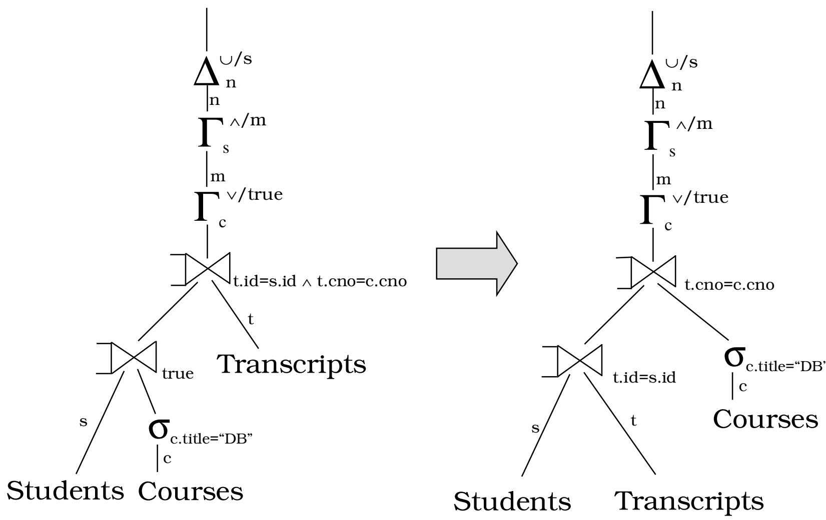
Example of Query Unnesting

Find all students who have taken all DB courses:

$$\cup\{\, s \mid s \leftarrow \text{Students},\ \ \wedge\{\, \vee\{\, t.cno = c.cno \mid t \leftarrow \text{Transcript},\ t.id = s.id \,\} \mid c \leftarrow \text{Courses},\ c.title = \text{“DB”} \,\}\,\}$$







Translating Calculus to Algebra

Query unnesting is done during the translation from calculus to algebra. The translation

- is simple & compositional;
- requires 9 rules only (2 for unnesting);
- is linear to the query size;
- is sound and complete

    (for NF^2 + aggregation + quantification).

It is the first query unnesting algorithm proven to be complete.

What about bag and list comprehensions?

Implementation and Performance

A prototype OQL optimizer already in existence at UTA:

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

The unnesting algorithm has been tested:

- on random ODL schemas;
- on random nested queries (in comprehension form) containing 2 to 29 inner queries.

The only optimization allowed was pushing predicates down the query tree (but not from outer to inner queries).

Result: the scaled cost improvement gained by query unnesting is exponential to the number of inner queries!

Conclusion

I have presented:

- a calculus based on comprehensions;
- a normalization algorithm that unnests many forms of nested comprehensions;
- a lower-level algebra that reflects many DBMS physical algorithms;
- a translation algorithm from calculus to algebra that unnests the remaining forms of query nesting.