An Effective Framework for Processing Object-Oriented Database Languages

Leonidas Fegaras
U. of Texas at Arlington

Relational Database Systems

Many reasons for the commercial success:

- they offer good performance to many business applications;
- they offer data independence;
- they provide an easy-to-use, declarative, query language;
- they have a solid theoretical basis;
- they employ sophisticated query processing and optimization techniques.

The Gap Between Theory & Practice

Most commercial relational query languages are based on the relational calculus.

However in some respects they go beyond the formal model. They support:

- aggregate operators,
- sort orders,
- grouping,
- update capabilities.

New Applications

Relational DBs cannot effectively model many new applications:

- multimedia,
- scientific databases,
- CAD,
- CASE,
- GIS,
- data warehousing and OLAP,
- office automation.

New Requirements

New DB languages must be able to handle:

- type extensibility;
- multiple collections types (eg. sets, lists, trees, arrays);
- nesting of type constructors;
- large objects (eg. text, sound, image);
- unstructured data;
- temporal & spatial data;
- encapsulation and methods;
- active rules;
- object identity.

New Proposals for DB Languages

Object-Relational databases:

- UniSQL,
- Postgress/Illustra,
- SQL3.

Object-Oriented databases:

- O2,
- GemStone,
- ObjectStore,
- ODMG'93 OQL.

Deductive Databases, Persistent Languages, Toolkits.

Why Do We Need a Formal Calculus?

A formal calculus:

- facilitates equational reasoning;
- provides a theory for proving query transformations correct;
- imposes language uniformity;
- avoids language inconsistencies.

functional languages	$\longleftrightarrow$	lambda calculus
relational databases	$\longleftrightarrow$	relational calculus
object-oriented databases	$\longleftrightarrow$	?

What is an Effective Calculus?

Several aspects:

- coverage,
- ease of manipulation,
- ease of evaluation,
- uniformity.

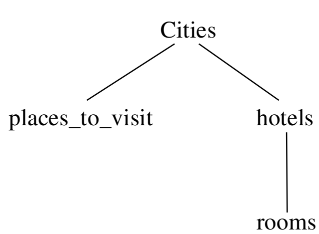
Rest of the Talk

- Monoids,
- monoid comprehensions,
- unnesting comprehensions,
- nested relational algebra,
- unnesting nested queries,
- implementation,
- current research work,
- future research plans.

Case Study: ODMG OQL

class city = < name: *string*,
hotels: *bag* (hotel),
places_to_visit: *list* (< name: *string*, address: *string* >) >
extent Cities;

class hotel = < name: *string*,
rooms: *set* (< bed_num: *int*, price: *int* >) >



select distinct h.name
from c **in** Cities,
h **in** c.hotels,
p **in** c.places_to_visit
where c.name="Arlington"
and h.name=p.name

OQL:	Monoid comprehension:
select distinct h.name **from** c **in** Cities, h **in** c.hotels, p **in** c.places_to_visit **where** c.name="Arlington" **and** h.name=p.name	$\cup$\{ h.name | c ← Cities, h ← c.hotels, p ← c.places_to_visit, c.name="Arlington", h.name=p.name \}

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* $U_\oplus$:

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

A free monoid is a *collection monoid*;
any other 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:	$(*, 1)$
integer:	$(\max, 0)$
boolean:	$(\vee, \text{false})$
boolean:	$(\wedge, \text{true})$

Example

$$\{ 1, 2, 3 \} = \{ 1 \} \cup \{ 2 \} \cup \{ 3 \}$$

Additional Properties:

commutativity: $x \cup y = y \cup x$

idempotence: $x \cup x = x$

Monoid Comprehensions

A monoid comprehension takes the form:

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

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;
```

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

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

Based on Abstract Algebra

$H[\otimes, \oplus](f)$ is a *homomorphism* from a collection monoid $\otimes$ to any monoid $\oplus$.

$$
\begin{aligned}
H[\otimes, \oplus]\,(f)\,(Z_\otimes) &= Z_\oplus \\
H[\otimes, \oplus]\,(f)\,(U_\otimes(a)) &= f(a) \\
H[\otimes, \oplus]\,(f)\,(x \otimes y) &= H[\otimes, \oplus]\,(f)\,(x) \oplus H[\otimes, \oplus]\,(f)\,(y)
\end{aligned}
$$

For example, for $h = H[\cup,+]\,(f)$:

$$
\begin{aligned}
h(\{\}) &= 0 \\
h(\{a\}) &= f(a) \\
h(x \cup y) &= h(x) + h(y)
\end{aligned}
$$

$H[\otimes, \oplus]\,(f)$ is the homomorphic extension of f,

$H[\otimes, \oplus]\,(f) \circ U_\otimes = f$ is an adjunction.

Formal Semantics

$$
\begin{aligned}
\oplus\{\, e \mid \,\} &= U_\oplus(e) \\
\oplus\{\, e \mid v \leftarrow u, r_1, \ldots, r_n \,\} &= H[\otimes,\oplus]\,(\lambda v.\ \oplus\{\, e \mid r_1, \ldots, r_n \,\})\,(u) \\
\oplus\{\, e \mid pred, r_1, \ldots, r_n \,\} &= \textbf{if}\ pred\ \textbf{then}\ \oplus\{\, e \mid r_1, \ldots, r_n \,\} \\
&\qquad \textbf{else}\ Z_\oplus
\end{aligned}
$$

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

$$= H[{+}{+},\cup](\ \lambda a.\ H[\uplus,\cup](\ \lambda b.\ \{(a,b)\}\ )\ (\{\!\{4,5\}\!\})\ )\ ([1,2,3])$$

Examples

$R \bowtie_{pred} S$	$= \cup\{\ (r,s) \mid r \leftarrow R,\ s \leftarrow S,\ pred\ \}$
$\text{flatten}(R)$	$= \cup\{\ s \mid r \leftarrow R,\ s \leftarrow r\ \}$
$R \cap S$	$= \cup\{\ r \mid r \leftarrow R,\ r \in S\ \}$
$\text{size}(R)$	$= +\{\ 1 \mid r \leftarrow R\ \}$
$e \in R$	$= \vee\{\ r = e \mid r \leftarrow R\ \}$
$R \subseteq S$	$= \wedge\{\vee\{\ r = s \mid s \leftarrow S\ \} \mid r \leftarrow R\ \}$

Translating 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;
```

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

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:

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

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

Example

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

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

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


Why Bother with Query Unnesting?

Query unnesting

- has been shown to be effective in other domains (e.g. logic);
- eliminates intermediate data structures;
- improves performance in many cases;
- allows operator mix-up between inner and outer queries;
- simplifies physical algorithms (no need for complex predicates).

Reminiscent to loop fusion and deforestation in programming languages.

But Some Queries are Difficult to Unnest

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

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

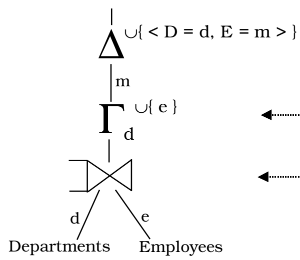
Lessons from Relational Databases

```
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

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



Reduce by ∪: form a set of tuples

Nest by d and form a set of e's

Left-outerjoin

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

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

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

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

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

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

Other operators:

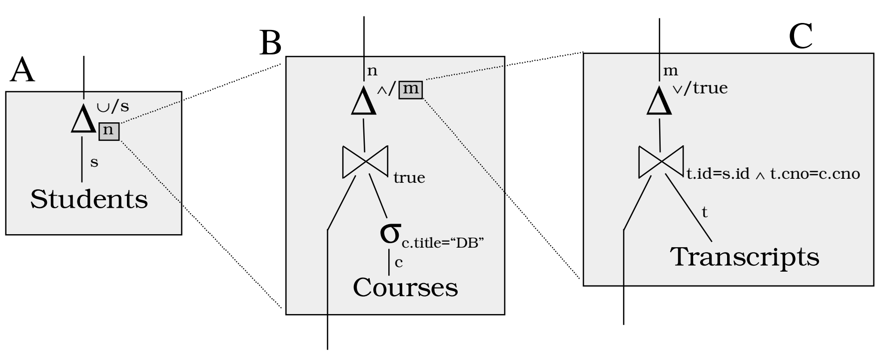
$R =\!\bowtie_p S$ left-outerjoin

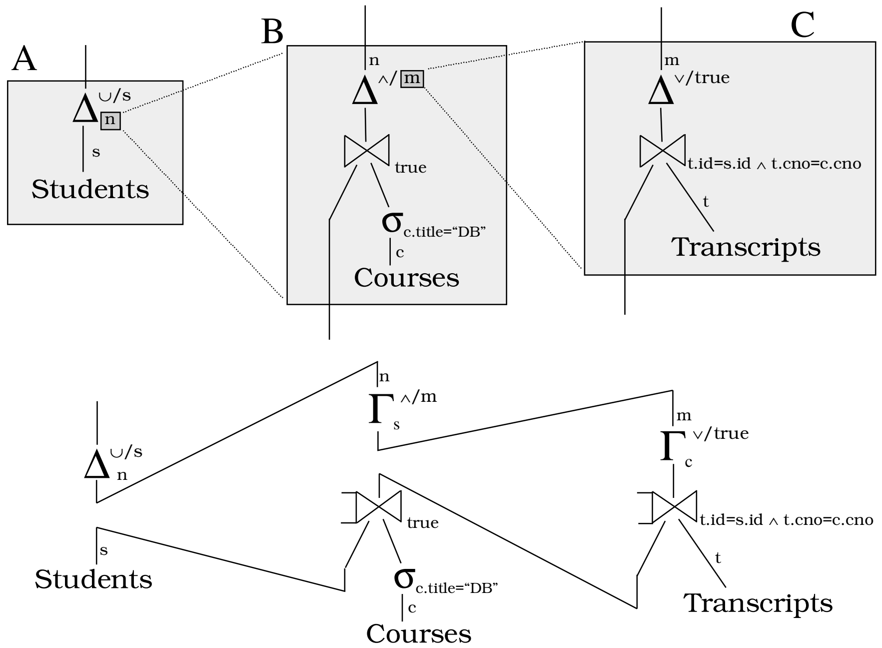
$=\!\mu_p^{path}(R)$ outer-unnest

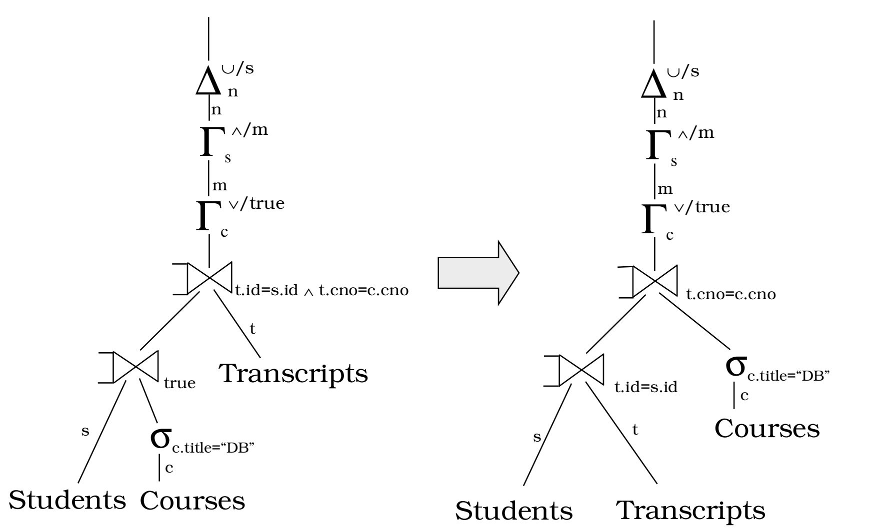
Example of Query Unnesting

Find all students that have taken all DB courses:

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




A
B
C
∪/s
n
s
Students
n
∧/
m
true
σc.title="DB"
c
Courses
m
∨/true
t.id=s.id ∧ t.cno=c.cno
t
Transcripts
Γs ∧/m
Γc ∨/true
Δn ∪/s


∪/s
n
n
∧/m
s
m
∨/true
c
t.id=s.id ∧ t.cno=c.cno
t
true
Transcripts
s
σc.title="DB"
c
Students
Courses
t.cno=c.cno
t.id=s.id
t
Students
Transcripts

Translating Calculus to Algebra

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

- is simple & compositional;
- requires 9 rules only;
- is linear to the query size;
- is sound and complete.

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

Implementation

A prototype OQL optimizer already in existence at UTA.
Components:

- normalization of comprehensions,
- normalization of predicates,
- query unnesting,
- materialization of path expressions,
- algebraic optimizations,
- translation into physical plans.

Expressed in an optimizer specification framework (OPTGEN) based on attribute grammars.
Currently evaluates plans in memory, but we are connecting it to SHORE.

Other Optimization Techniques

- Finding good algorithms to evaluate operators;
- handling encapsulation and methods;
- finding a good order to evaluate operators;
- handling object identity and object updates;
- maintaining materialized views.

Evaluation Order of Operators

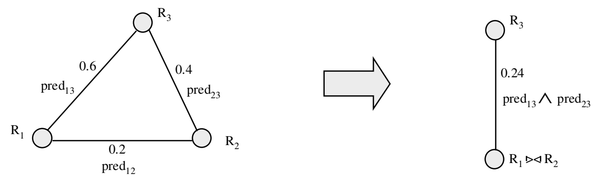
Problem: After query unnesting, all operators are promoted to the same level. Order of evaluation?

Complexity: $O(m! \times k^m)$
(for m operators, where each one can be evaluated in k ways)

Heuristics:
- dynamic programming of System R: $O(3^n)$
- iterative improvement;
- simulated annealing.

The Operator Ordering Algorithm

- Based on query graphs;
- similar to Kruskal's spanning tree algorithm, but with a twist:



- is polynomial: $O(n^3)$, can handle 100 joins in 32 msecs;
- handles nestings & unnestings as graph dependencies;
- can handle disjunctions and dependent predicates;
- beats other related proposals in performance.

Handling Object Identity

Object monoid calculus (= monoid calculus + SML-style objects):

$$++\{\ !x \mid x \leftarrow [\ new(1),\ new(2)\ ],\ \ x := !x+1\ \}$$

It returns:

$$[\ 2,\ 3\ ]$$

Characteristics of the optimization framework:

- it is based on denotational semantics (state transformers & nondeterminism);
- the state is always single-threaded;
- the resulting programs do destructive updates;
- normalization eliminates unnecessary state manipulation;
- it allows equational reasoning and optimization.

Conclusion

I have presented:

- a uniform calculus based on comprehensions that captures many advanced features found in modern OODB languages;
- a normalization algorithm that unnests nested comprehensions;
- a lower-level algebra that reflects many DBMS physical algorithms;
- a translation algorithm from calculus to algebra that unnests all forms of query nesting.

Future Work

Model extensions: vectors, encapsulation & methods, unstructured data, temporal data & version control.

Problems:

- realistic cost analysis,
- using & maintaining materialized views,
- supporting dynamic query plans.

Implementation:

- building a full-fledged OQL optimizer,
- applying the framework to SQL3,
- evaluating performance using an application domain.

Related Work

- Monoid homomorphisms [Tannen et al]
  - SRU
  - monads: $ext(f) = H[\oplus,\oplus](f)$
- boom hierarchy of types [Bird, Meertens, Backhouse];
- monad comprehensions [Wadler, Trinder, Buneman];
- normalization of monad comprehensions [Wong].