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 2.0 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:

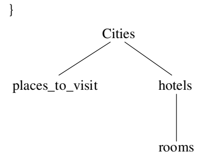
- coverage,
- ease of manipulation,
- ease of evaluation,
- uniformity.

Rest of the Talk

- Monoids,
- monoid comprehensions,
- unnesting comprehensions,
- monoid algebra,
- unnesting nested queries,
- λ-DB
- current research work,
- future research plans.

Case Study: ODMG 2.0 OQL

```
class City { extent Cities } {
  attribute string name;
  attribute list < struct( name: string, address: string ) > places_to_visit;
  relationship bag <Hotel> hotels inverse Hotel::location;
}
class Hotel { extent Hotels } {
  attribute string name;
  attribute set < struct( bed_num: int, price: int ) > rooms;
  relationship City location inverse City::hotels;
}
```




```
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 $\leftarrow$ Cities, h $\leftarrow$ c.hotels, p $\leftarrow$ 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 monoid is a *primitive monoid.*

Some Monoids

Collection monoids:

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

Primitive monoids:

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

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

accumulator head qualifiers

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

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

Example

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

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

Based on Abstract Algebra

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

$$H[\otimes,\oplus]\,(f)\,(Z_\otimes) = Z_\oplus$$
$$H[\otimes,\oplus]\,(f)\,(U_\otimes(a)) = f(a)$$
$$H[\otimes,\oplus]\,(f)\,(x \otimes y) = H[\otimes,\oplus]\,(f)\,(x) \oplus H[\otimes,\oplus]\,(f)\,(y)$$

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

h ({ })	= 0
h ({ a })	= f (a)
h (x $\cup$ y)	= h (x) + h (y)

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

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

Formal Semantics

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

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

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

Examples

$R \bowtie_{pred} S$	$= \cup\{\ (r,s)\ \mid\ r \leftarrow R,\ s \leftarrow S,\ pred\ \}$
flatten(R)	$= \cup\{\ s\ \mid\ r \leftarrow R,\ s \leftarrow r\ \}$
$R \cap S$	$= \cup\{\ r\ \mid\ r \leftarrow R,\ r \in S\ \}$
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;
```

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

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:

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

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

Example

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

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

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

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

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

- eliminates intermediate data structures;
- improves performance in many cases;
- allows operator mix-up between inner and outer queries;
- allows free movement of predicates 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;

In Comprehension form:

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

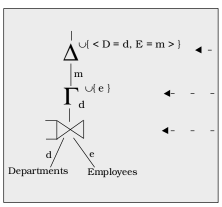
Lessons from Relational Databases

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

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



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:

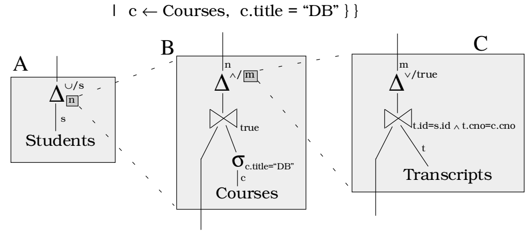
$R =\!\bowtie_p S$ left-outerjoin

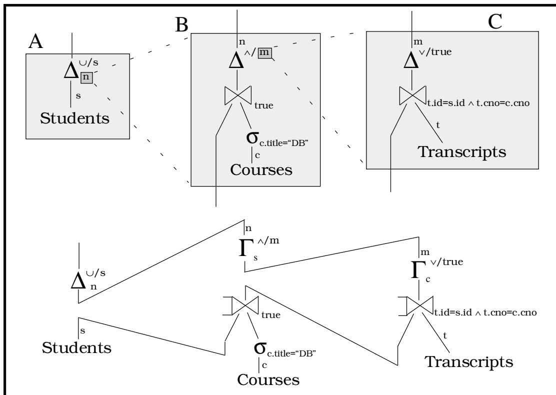
$=\!\mu_p^{path}(R)$ outer-unnest

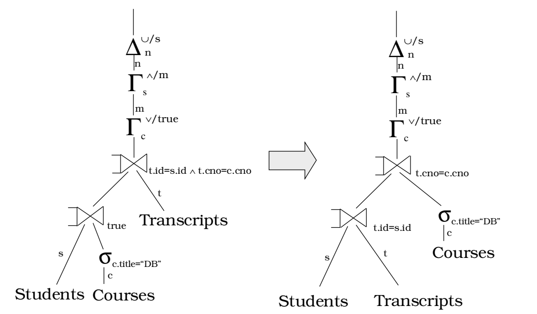
Example of Query Unnesting

Find all students who have taken all DB courses:

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







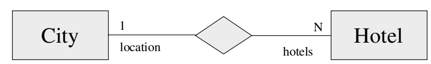
Translating Calculus to Algebra

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.

Using Relationships in Query Optimization



select h.name
from c **in** Cities,
h **in** c.hotels
where c.name = "Arlington"

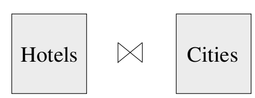
select h.name
from h **in** Hotels
where h.location.name = "Arlington"

Materialization of Path Expressions

select h.name
from h **in** Hotels
where h.location.name = "Arlington"

select h.name
from h **in** Hotels,
c **in** Cities
where h.location = OID(c)
and c.name = "Arlington"

Pointer Joins Between Class Extents



One-object-at-a-time traversals vs. pointer joins:

A path expression $x.A_1.A_2 \ldots A_n$ is translated into a sequence of pointer joins $C_1 \bowtie C_2 \bowtie \ldots \bowtie C_n$.

Relational database technology to the rescue:

- we know how to rearrange joins to gain better performance;
- we know what algorithms to use to evaluate joins;
- we know how to select the best access paths to data (using indexes).

λ-DB A High-Performance OODB System

λ-DB is an OODB system built on top of the SHORE object management system. The system

- can handle most ODMG ODL declarations;
- can process most ODMG OQL queries;
- supports embedded OQL in C++;
- supports transactions, updates, macros, and methods with OQL body.

Available at: `http://lambda.uta.edu/lambda-DB/manual/`

Query Optimization in λ-DB

The query optimizer:

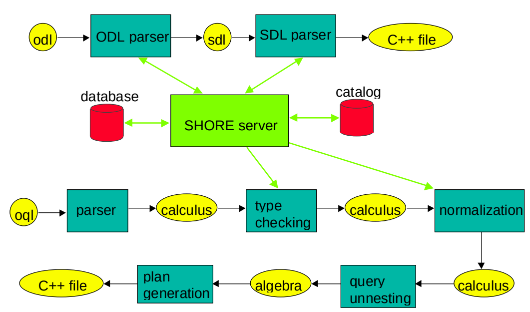
- unnests all nested queries;
- materializes path expressions into pointer joins;
- performs semantic optimizations (using ODL relationships);
- uses a cost-based polynomial-time heuristic for join ordering;
- uses a rule-based cost-driven optimizer to produce physical plans.

The Evaluation Engine

The query evaluator:

- translates evaluation plans into C++ code;
- supports pipelining (stream-based processing);
- supports many evaluation algorithms (indexed nested loop, pointer join, sort-merge join);
- supports the creation, maintenance, and use of indexes.

Architecture



ODL Schema

```
module School {
class Person (extent Persons key ssn)
{ attribute long ssn;
  attribute string name;
  attribute string address;
};
typedef set<string> Degrees;
class Instructor extends Person (extent Instructors)
{ attribute long salary;
  attribute Degrees degrees;
  relationship Department dept
        inverse Department::instructors;
  relationship set<Course> teaches
        inverse Course::taught_by;
   short courses(in string dept_name);
};
```


OQL Example

```
#include <odmg.h>
%module School;
int main (int argc, char* argv[])
{   %initialize(argc,argv);
    %begin;
    %foreach v in select x: e.name, y: c.name
                    from e in Instructors,
                         c in e.teaches
                   where e.ssn = 12345
                 do cout << v.x << " " << v.y << endl;
    %commit;
    %cleanup;
};
```

Algebraic Form

```
reduce(bag,
       join(bag,
            get(bag,Instructors,e,
                     eq(project(e,ssn),12345))),
            get(bag,Courses,c,and()),
               and(eq(project(c,taught_by),OID(c))),
               none),
        s,
        struct(bind(x,project(e,name)),
                 bind(y,project(c,name))),
        and())
```


Physical Plan

```
REDUCE(bag,
       MERGE_JOIN(bag,
                  SORT(INDEX_SCAN(bag,Instructors,e,and()
                           _index_Instructor_0,12345,123
                             order(OID(e))),
                  SORT(TABLE_SCAN(bag,Courses,c,and()),
                             order(project(c,taught_by)
                     and(eq(project(c,taught_by),OID(e))),
                     none,
                     true,
                     order(OID(e)),
                     order(project(c,taught_by))),
            s,
            stru(bind(x,project(e,name)),
                 bind(y,project(c,name))),
            and())
```


Handling Object Identity

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

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

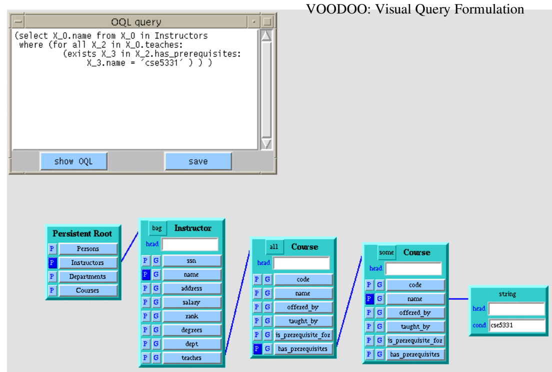
It returns:

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

Characteristics of the optimization framework:

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

VOODOO: Visual Query Formulation



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 many forms of 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 Research Plans

I am planning to extend my current work by

- developing more optimization techniques for OODBs;
- developing better cost estimation functions and using better cost-based optimization techniques;
- developing a framework for semantic query optimization;
- handling and optimizing active rules;
- developing a framework for maintaining materialized views;
- handling vectors and arrays and optimizing data cube queries (used in on-line analytical processing);
- handling unstructured and semistructured data;
- specifying and optimizing world-wide-web queries.

Related Work on Algebras

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

Related Work on Query Unnesting

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)