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/Iliustra,
• 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.

<table>
<thead>
<tr>
<th>functional languages</th>
<th>lambda calculus</th>
</tr>
</thead>
<tbody>
<tr>
<td>relational databases</td>
<td>relational calculus</td>
</tr>
<tr>
<td>object-oriented databases</td>
<td>?</td>
</tr>
</tbody>
</table>

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

```plaintext
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
```

Monoids

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

\[ (\oplus, Z_0) \]

The merge function \( \oplus \) is associative with zero \( Z_0 \):

\[ x \oplus Z_0 = Z_0 \oplus x = x \]

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

\[ (\ominus, Z_0, U_a) \]

A free monoid is a collection monoid;
any other monoid is a primitive monoid.
Some Monoids

Collection monoids:
- set(a): \(( \cup, \{ \}, \lambda x. \{ x \} )\)
- bag(a): \(( \cup, \{ \}, \{ \}, \lambda x. \{ \{ x \} \})\)
- list(a): \(( ++, [], \lambda x. [x])\)

Primitive monoids:
- integer: \(( +, 0 )\)
- integer: \(( *, 1 )\)
- integer: \(( \max, 0 )\)
- boolean: \(( \lor, \text{false} )\)
- boolean: \(( \land, \text{true} )\)

Additional Properties:
- commutativity: \( x \cup y = y \cup x \)
- idempotence: \( x \cup x = x \)

Monoid Comprehensions

A monoid comprehension takes the form:

\[ \oplus \{ e | r_1, r_2, \ldots, r_n \} \]

where \( \oplus \) is a monoid and each qualifier \( r_i \) is either:
- a generator \( v \leftarrow u \), or
- a filter \( \text{pred} \).

Based on Abstract Algebra

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

\[
\begin{align*}
H[\otimes, \oplus](f) \{ e \} &= \{ e \} \\
H[\otimes, \oplus](f) \{ u \} &= H[\otimes, \oplus](f) \{ v \} \\
H[\otimes, \oplus](f) \{ x \otimes y \} &= H[\otimes, \oplus](f) \{ x \} \oplus H[\otimes, \oplus](f) \{ y \}
\end{align*}
\]

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

\[
\begin{align*}
h(\{ \} ) &= 0 \\
h(\{ a \} ) &= f(a) \\
h( x \cup y ) &= h( x ) + h( y )
\end{align*}
\]

\( H[\otimes, \oplus](f) \) is the homomorphic extension of \( f \).
\( H[\otimes, \oplus](f) \ast U_\otimes = f \) is an adjunction.

Formal Semantics

\[
\begin{align*}
\{ e | \} &= U_\oplus(\{ e \} ) \\
\{ e | v \leftarrow u, \ldots, v_n \} &= H[\otimes, \oplus](\lambda v. \{ e | v \leftarrow u, \ldots, v_n \}) (u) \\
\{ e | \text{pred}, r_1, \ldots, r_n \} &= \text{if pred then } \{ e | r_1, \ldots, r_n \} \\
& \quad \text{else } U_\otimes
\end{align*}
\]

\[
\begin{align*}
\{ a | a \leftarrow [1,2,3], b \leftarrow [4,5] \} &= H[\cup, \cup](\lambda a. H[\cup, \cup](\lambda b. \{ a | b \} )) ([1,2,3], [4,5]) \\
& \quad ([1,2,3])
\end{align*}
\]
Examples

\[ R \cup S = \{(r,s) \mid r \in R, s \in S, \text{pred}\} \]

\[ \text{flatten}(R) = \{(r,s) \mid s \in \text{pred}(R)\} \]

\[ R \cap S = \{r \mid r \in R, r \in S\} \]

\[ \text{size}(R) = +\{1 \mid r \in R\} \]

\[ e \in R = \{r = e \mid r \in R\} \]

\[ R \subseteq S = \{r \mid r \in R, r \in S\} \]

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;
)

union

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;
```

Normalization

Canonical form:

\[ \Theta( e \mid s_1 \leftarrow \text{path}_1, \ldots, s_n \leftarrow \text{path}_n, \text{pred}) \]

(path is a cascade of projections:  \(X_1.A_1 \ldots X_m.A_m\))

Two important normalization rules:

1. \[ \Theta( e \mid s_1 \leftarrow \Theta( u \mid \ldots, \ldots ) \mid \ldots ) \]
2. \[ \Theta( e \mid \ldots, \ldots, \text{pred} ) \mid \ldots \]

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:

```
∪ { < D = d, E = ∪ { e | e ← Employees, e.dno = d.dno } > 
  | d ← 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

```
∪ { < D = d, E = ∪ { e | e ← Employees, e.dno = d.dno } > 
  | d ← 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

```
σ_p(R) = ∪ { r | r ← R, p(r) }
R⋈σ S = ∪ { (r,s) | r ← R, s ← S, p(r,s) }
Δ_p(R) = ∪ { e(r) | r ← R, p(r) }
μ_pπ(R) = ∪ { (r,s) | r ← R, s ← path(r), p(r,s) }
Γ_p(R) = ∪ { ( f(r), ∪ { e(s) | s ← R, f(r)=f(s), p(s) } ) 
  | r ← R }
```

Other operators:

R ↷⋈ S left-outerjoin
R δ⋈ S outer-unnest

Example of Query Unnesting

Find all students who have taken all DB courses:

```
∪ { s | s ← Students, 
  σ c.title=ªDBº t.id=s.id 
    ∪ { t.cno = c.cno | t ← Transcript, 
        c ← Courses, c.title = "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 h in Hotels
where h.location.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 \( x.A_1.A_2...A_n \) is translated into a sequence of
pointer joins \( C_1 \circ C_2 \circ ... \circ C_n \).
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 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/

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 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.

### ODL Schema

```plaintext
module School {  
class Person {  
  extend Person
  attr long ssn;
  attr string name;
  attr string address;
};

typedef set < string > Degrees;

class Instructor extends Person {  
  extend Instructor
  attr long salary;
  attr Degrees degrees;
  relation Department dept;  
  inverse dept.employees : instructors;
  relation Instructors teaches ;
  inverse course :: taught by;
  short courses { string dept_name } ;
};
```

### OQL Example

```plaintext
#include odmg.h

#include <string>

int main (int argc, char * argv[]) {
  %initiate environment;
  begin;
  for each in sel e : e . ssn = 12345 from instructors, in e . to each s
  do cout << e . name << "  " << s . name << endl;
  %commit;
  %clean up;
};
```
Algebraic Form

\[ \text{reduce} \text{ bag,} \]
\[ \text{join bag,} \]
\[ \text{get bag,} \text{ Instructor s,} \text{e,} \]
\[ \text{and (eq (project (e, ssn)), 12345))} \]
\[ \text{get bag,} \text{ Course c,} \text{a nd(} \]
\[ \text{and (eq (project (c, taught_by)), OID(c))} \]
\[ \text{none),} \]
\[ \text{str} \text{ bind} (x, \text{project}(e, \text{name})), \]
\[ \text{bind} (y, \text{project}(c, \text{name})), \]
\[ \text{and (}) \]

Physical Plan

\[ \text{reduce} \text{ bag,} \]
\[ \text{MERGE JOIN bag,} \]
\[ \text{SORT INDEX SCAN bag,} \text{ Instructor s,} \text{e,} \text{and()} \]
\[ \text{INDEX I N S T R U C T O R sssn, 12345, 12345 } \]
\[ \text{ORDER (OID(s))} \]
\[ \text{ORDER (OID(e))} \]
\[ \text{ORDER (OID(c))} \]
\[ \text{ORDER (OID(e))} \]
\[ \text{ORDER (OID(c))} \]
\[ \text{ORDER (OID(e))} \]
\[ \text{ORDER (OID(c))} \]
\[ \text{str} \text{ bind} (x, \text{project}(e, \text{name})), \]
\[ \text{bind} (y, \text{project}(c, \text{name})), \]
\[ \text{and (}) \]

Handling Object Identity

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

\[ \text{++{ }x | x \leftarrow [\text{new}(1), \text{new}(2)], x := x+1 } \]

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.

Conclusion

I have presented:

- a uniform calculus based on comprehensions that captures many advanced features found in modern OODB languages;
- a normalization algorithm that unnest many forms of nested comprehensions;
- a lower-level algebra that reflects many DBMS physical algorithms;
- a translation algorithm from calculus to algebra that unnest 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: \( \text{ext}(f) = H[@.\oplus|f] \)
- 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)