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.

<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,
- nested relational algebra,
- unnesting nested queries,
- implementation,
- current research work,
- future research plans.

Case Study: ODMG OQL

```plaintext
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
```

```plaintext
Cities
  places_to_visit
  hotels

  rooms
```

OQL:

```
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
```

Monoid comprehension:

```
∪( 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:

```
( ⊕, Z⊕ )
```

The *merge* function ⊕ is associative with *zero* Z⊕:

```
x ⊕ Z⊕ = Z⊕ ⊕ x = x
```

A parametric type (e.g. set(a)) is associated with a free monoid that has a *unit* U⊕:

```
( ⊕, Z⊕ , U⊕ )
```

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

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

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:

\[ \oplus \{ e \mid r_1, \ldots, r_n \} \]

where \( \oplus \) is a monoid and each \textit{qualifier} \( r_i \) is either:

- a \textit{generator} \( v \leftarrow u \), or
- a \textit{filter} \( \text{pred} \).

Examples

\[ \bigcup \{ (a,b) \mid a \leftarrow x, b \leftarrow y \} \]

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

\[ +\{ a \mid a \leftarrow [1,2,3], a \geq 2 \} = 2+3 = 5 \]
Based on Abstract Algebra

H[⊗, ⊕]( f ) is a homomorphism from a collection monoid ⊗

to any monoid ⊕.

\[
\begin{align*}
H[\otimes, \oplus] (f)(Z_{\otimes}) &= Z_{\oplus} \\
H[\otimes, \oplus] (f)(U_{\otimes}(a)) &= f(a) \\
H[\otimes, \oplus] (f)(x \otimes y) &= \left( H[\otimes, \oplus](f)(x) \oplus H[\otimes, \oplus](f)(y) \right)
\end{align*}
\]

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[⊗, ⊕]( f ) is the homomorphic extension of \( f \),
H[⊗, ⊕]( f ) ° U ⊗ = f is an adjunction.

Formal Semantics

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

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

\[ \text{flatten}(R) = \bigcup \{ s \mid r \leftarrow R, s \leftarrow r \} \]
\[ R \cap S = \bigcup \{ r \mid r \leftarrow R, r \in S \} \]
\[ \text{size}(R) = +\{ 1 \mid r \leftarrow R \} \]
\[ e \in R = \forall \{ r = e \mid r \leftarrow R \} \]
\[ R \subseteq S = \forall \{ r = s \mid s \leftarrow S \} \mid r \leftarrow R \]
Normalization

Canonical form:

\[ \bigoplus \{ e \mid x_1 \leftarrow \text{path}_1, \ldots, x_n \leftarrow \text{path}_n, \text{pred} \} \]

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

Two important normalization rules:

\[ \bigoplus \{ e \mid x \leftarrow \bigotimes \{ u \mid \text{condition} \}, \text{condition} \} \rightarrow \bigoplus \{ e \mid x \equiv u \}, \text{condition} \}

\[ \bigoplus \{ e \mid \text{condition}, \bigvee \{ \text{pred} \mid \text{condition} \} \}, \text{condition} \} \rightarrow \bigoplus \{ e \mid \text{condition}, \text{pred}, \text{condition} \} \]

Example

\[ \bigcup \{ \text{hotel.price} \mid \text{hotel} \leftarrow \bigcup \{ h \mid c \leftarrow \text{Cities, h \leftarrow c.hotels, c.name = “Arlington” }, \bigvee \{ r \cdot \text{bed_num} = 3 \mid r \leftarrow \text{hotel.rooms} \} \} \}

= \bigcup \{ \text{hotel.price} \mid c \leftarrow \text{Cities, h \leftarrow c.hotels, c.name = “Arlington”, hotel \equiv h, } \bigvee \{ r \cdot \text{bed_num} = 3 \mid r \leftarrow \text{hotel.rooms} \} \}

= \bigcup \{ h\cdot\text{price} \mid c \leftarrow \text{Cities, h \leftarrow c.hotels, c.name = “Arlington”, } \bigvee \{ r \cdot \text{bed_num} = 3 \mid r \leftarrow h\cdot\text{rooms} \} \}

= \bigcup \{ h\cdot\text{price} \mid c \leftarrow \text{Cities, h \leftarrow c.hotels, r \leftarrow h\cdot\text{rooms, c.name = “Arlington”, r\cdot\text{bed_num} = 3} \} \]
Unnesting OQL Queries

```sql
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;
```

```sql
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

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

```sql
∪( < D = d, E = ∪( e | e ← Employees, e.dno = d.dno ) >
  | d ← Departments )
```

Lessons from Relational Databases

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

```sql
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**

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

--- Reduce by \( \cup \): 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

\[\sigma_p(R) = \bigcup \{ r \mid r \leftarrow R, \ p(r) \}\]

\[R \bowtie_p S = \bigcup \{ (r,s) \mid r \leftarrow R, \ s \leftarrow S, \ p(r,s) \}\]

\[\Delta_p(\Theta_c/R) = \Theta \{ e(r) \mid r \leftarrow R, \ p(r) \}\]

\[\mu_p^\text{path}(R) = \bigcup \{ (r,s) \mid r \leftarrow R, \ s \leftarrow \text{path}(r), \ p(r,s) \}\]

\[\Gamma_p(\oplus/e/f)(R) = \bigcup \{ (f(r), \bigoplus \{ e(s) \mid s \leftarrow R, \ f(r)=f(s), \ p(s) \}\) \mid r \leftarrow R \}\]

Other operators:

\[R \Rightarrow_s p S\] left-outerjoin

\[=\mu_p^\text{path}(R)\] outer-unnest

Example of Query Unnesting

Find all students that have taken all DB courses:

\[\bigcup \{ s \mid s \leftarrow \text{Students,} \]
\[\land \{ \lor \{ \land \{ t.\text{cno} = c.\text{cno} \mid t \leftarrow \text{Transcript,} \ t.\text{id} = s.\text{id} \} \mid c \leftarrow \text{Courses,} \ c.\text{title} = "DB" \} \} \]
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.

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.

![Diagram of the Operator Ordering Algorithm](image_url)

### Handling Object Identity

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

```plaintext
++{ !x | x ← [ 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: \( \text{ext}(f) = \text{H}[\oplus, \oplus](f) \)
- boom hierarchy of types [Bird, Meertens, Backhouse];
- monad comprehensions [Wadler, Trinder, Buneman];
- normalization of monad comprehensions [Wong].