An Effective Framework for Processing Object-Oriented Database Languages

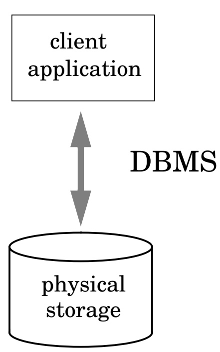
Leonidas Fegaras

Oregon Graduate Institute

(Ph.D. University of Massachusetts)

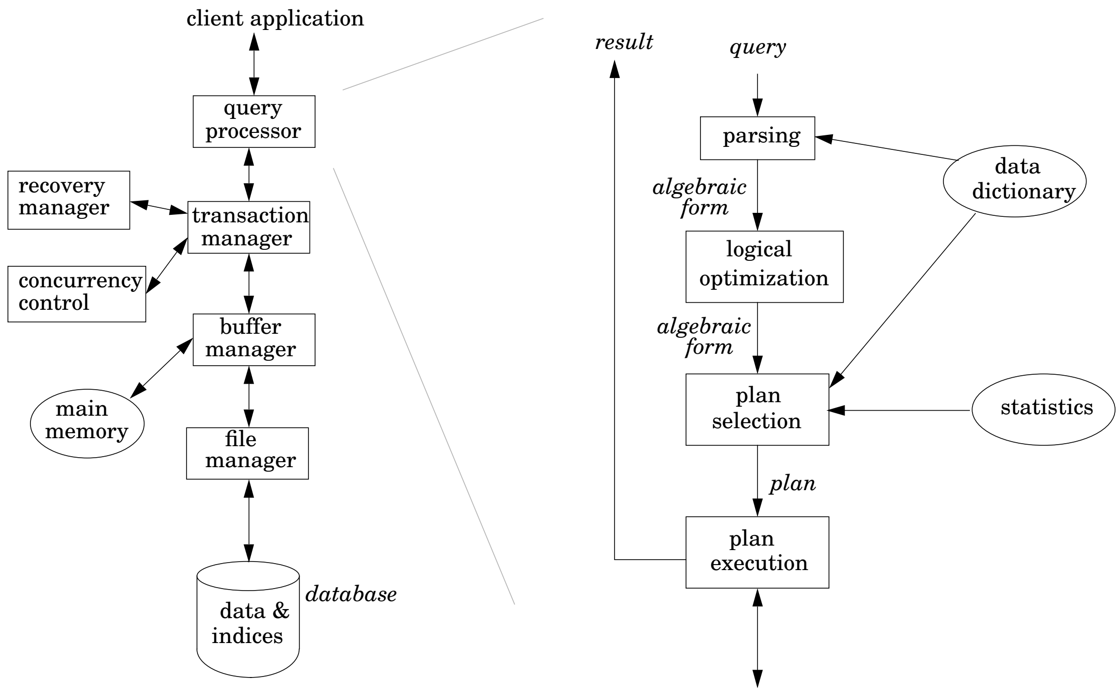
Maintaining Persistent Data

Data Base Management System:



Key issue: *data independence*.

Typical Database Architecture



The Gap Between Theory & Practice

Most commercial relational query languages are based on

- tuple calculus (SQL, Quel)
- domain calculus (QBE)

However, in some respects they go beyond the formal model:

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

Insufficient Modeling Power

Relational databases cannot effectively model many new applications:

- multimedia & World-Wide-Web,
- scientific databases,
- CAD,
- CASE,
- GIS,
- office automation.

New Requirements

New database languages must be able to handle:

- type extensibility;
- multiple collection types (e. g., sets, lists, trees, arrays);
- arbitrary nesting of type constructors;
- large objects (e.g., text, sound, image);
- temporal & spatial data;
- unstructured data;
- active rules;
- methods.

New Proposals for Database Languages

Relational extensions:

- UniSQL,
- POSTGRES/Illustra,
- SQL3.

Object-oriented databases:

- Gemstone,
- O2,
- OQL of ODMG-93.

Why Do We Need 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 algebra/calculus
object-oriented databases	$\longleftrightarrow$	?

What is an Effective Calculus?

Several aspects:

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

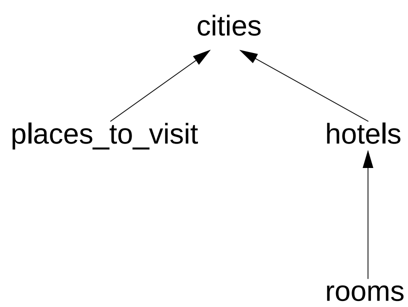
Rest of the Talk

- Monoids;
- *Algebra:* monoid homomorphisms;
- *Calculus:* monoid comprehensions;
- unnesting comprehensions;
- overview of the query translation framework;
- physical design specification;
- one example of query translation.

Case Study: ODMG-93 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#: *int*, price: *int* >) >;



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

OQL:

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

Monoid comprehension:

set { h.name | c $\leftarrow$ cities,
h $\leftarrow$ c.hotels,
p $\leftarrow$ c.places_to_visit,
c.name="Portland",
h.name=p.name }

Monoids

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

operator	*functionality*	e.g., sets
zero	the identity value	{ }
merge(x,y)	associative with identity **zero**	$x \cup y$
unit(a)	singleton construction	{ a }

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

Optional properties:
- *commutativity* : **merge**(x,y) = **merge**(y,x)
- *idempotence* : **merge**(x,x) = x

Some Monoids

Collection Monoids

monoid	type	zero	unit(a)	merge
list	list(α)	[ ]	[a]	append
set	set(α)	{ }	{a}	$\cup$
bag	bag(α)	{{ }}	{{a}}	$\uplus$
***sorted*[f]**	list(α)	[ ]	[a]	list_merge[f]

Primitive Monoids

monoid	type	zero	unit(a)	merge
sum	integer	0	a	+
prod	integer	1	a	*
some	boolean	false	a	$\vee$
all	boolean	true	a	$\wedge$

Monoid Homomorphisms

hom[T,S](f) is a *homomorphism* from a collection monoid T to any monoid S:

$$\alpha \xrightarrow{f} S \qquad T(\alpha) \xrightarrow{\mathbf{hom}[T,S](f)} S$$

hom[T,S](f)(**zero**[T])	=	**zero**[S]
hom[T,S](f)(**unit**[T](a))	=	f(a)
hom[T,S](f)(**merge**[T](x,y))	=	**merge**[S](**hom**[T,S](f)(x), **hom**[T,S](f)(y))

Example

Notation: $\lambda a\,.\,\{ a^2 \}$ is the function f such that: $f(a) = \{ a^2 \}$

hom[*bag,set*]($\lambda a\,.\,\{ a^2 \}$) $\{\!\{1, 2, 3\}\!\}$

= **hom**[*bag,set*](f) ($\{\!\{1\}\!\} \uplus \{\!\{2\}\!\} \uplus \{\!\{3\}\!\}$)

= $f(1) \cup f(2) \cup f(3)$

= $\{1\} \cup \{4\} \cup \{9\}$

= $\{1, 4, 9\}$

Other Examples

$5 \in x \quad = \quad$ **hom**[*set,some*](λa. (a=5))(x)

$\{8\} \cup \{5\} = \{8, 5\}$

8=5 5=5

false $\vee$ true = true

length(x) $\quad = \quad$ **hom**[*list,sum*](λa. 1)(x)

append([2], [3]) = [2, 3]

1 1

1 + 1 = 2

Monoid Comprehensions

A *monoid comprehension* takes the form:

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

head *qualifiers*

where T is a monoid and each qualifier r_i is either:

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

Examples

$$set\{ (a,b) \mid a \leftarrow x, b \leftarrow y \} = \left\{ \begin{array}{l} \text{res = \{ \};} \\ \textbf{for each}\ a\ \textbf{in}\ x\ \textbf{do} \\ \quad \textbf{for each}\ b\ \textbf{in}\ y\ \textbf{do} \\ \quad\quad \text{res} = \text{res} \cup \{ (a,b) \}; \\ \textbf{return}\ \text{res}; \end{array} \right.$$

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

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

Formal Definition of a Monoid Comprehension

$T\{ e \mid \}$	=	**unit**[T](e)
$T\{ e \mid v \leftarrow u, r_1, ..., r_n \}$	=	**hom**[S,T](λv. $T\{ e \mid r_1, ..., r_n \}$)(u) *where* S *is the type of* u
$T\{ e \mid pred, r_1, ..., r_n \}$	=	**if** pred **then** $T\{ e \mid r_1, ..., r_n \}$ **else zero**[T]

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

= **hom**[*list,set*](λa. **hom**[*bag,set*](λb. { (a,b) })
({{4,5}}))
([1,2,3])

Other Examples

filter(pred) e	=	*set*{ x | x ← e, pred(x) }
flatten(e)	=	*set*{ x | s ← e, x ← s }
$e_1 \cap e_2$	=	*set*{ x | x ← e_1, x ∈ e_2 }
length(e)	=	*sum*{ 1 | x ← e }
∃a ∈ e: pred	=	*some*{ pred | a ← e }
∀a ∈ e: pred	=	*all*{ pred | a ← e }
nest(k) e	=	*set*{ < KEY= k(x), DATA= *set*{ y | y ← e, k(x)=k(y) } > | x ← e }
unnest(e)	=	*set*{ x | s ← e, x ← s.DATA }

Translating OQL

select distinct h.name
from hl **in** (**select** c.hotels
from c **in** cities
where c.name="Portland"),
h **in** hl
where exists r **in** h.rooms: (r.bed#=3)

set{ h.name | hl ← *bag*{ c.hotels | c ← cities, c.name="Portland" },
h ← hl,
some{ r.bed#=3 | r ← h.rooms } }

Program Normalization

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

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

Examples of normalization rules:

- T{ e | ❶ , x ← S{ u | ❷ }, ❸ }
  → T{ e | ❶ , ❷ , x ≡ u, ❸ }
- T{ e | ❶ , some{ pred | ❷ }, ❸ }
  → T{ e | ❶ , ❷ , pred, ❸ }

Example

set{ h.name | hl ← *bag*{ c.hotels | c ← cities, c.name=“Portland” },
h ← hl,
some{ r.bed#=3 | r ← h.rooms } }

= *set*{ h.name | c ← cities, c.name=“Portland”,
hl ≡ c.hotels,
h ← <u>hl</u>,
some{ r.bed#=3 | r ← h.rooms } }

Substitute c.hotels for hl

= *set*{ h.name | c ← cities, c.name=“Portland”,
h ← c.hotels,
some{ r.bed#=3 | r ← h.rooms } }

= *set*{ h.name | c ← cities,
h ← c.hotels,
r ← h.rooms,
(c.name=“Portland”) ∧ (r.bed#=3) }

Unnesting OQL Queries

```
select distinct h.name
 from hl in ( select c.hotels
                   from c in cities
                  where c.name="Portland" ),
      h in hl
 where exists r in h.rooms: ( r.bed#=3 )
```

normalization

```
select distinct h.name
 from c in cities,
      h in c.hotels,
      r in h.rooms
 where c.name="Portland" and r.bed#=3
```

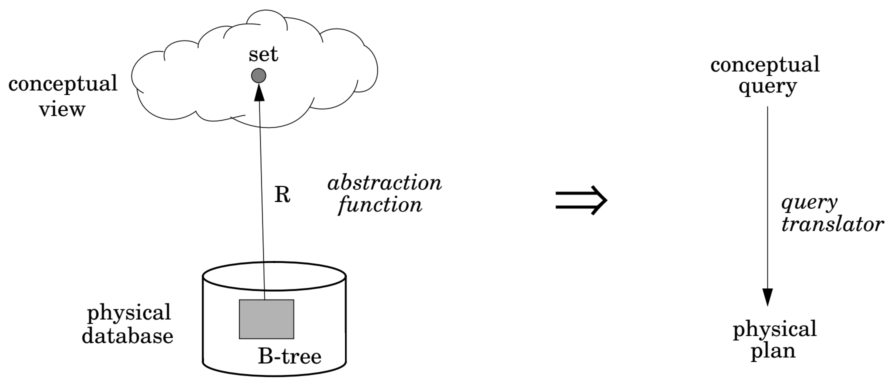

Query Translation

Need to support data independence.

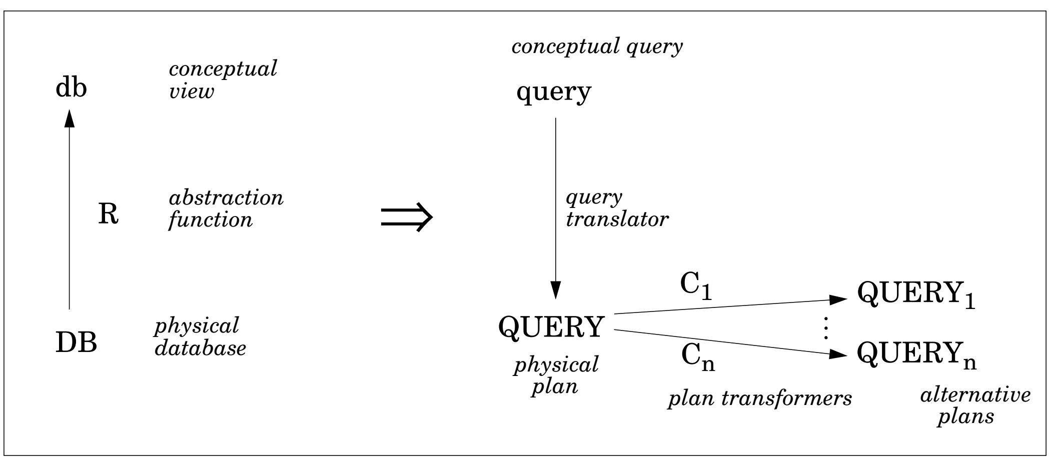
Tasks:

- the *database administrator* specifies the conceptual database schema;
- the *database implementor* specifies the *physical design*;
- an *application programmer* submits a query against the database;
- the *query translator* translates the query into a physical plan that reflects the physical design.

Conceptual-to-Internal Mapping



$$R(x) = \mathit{set}\{\, a \mid a \leftarrow x \,\}$$

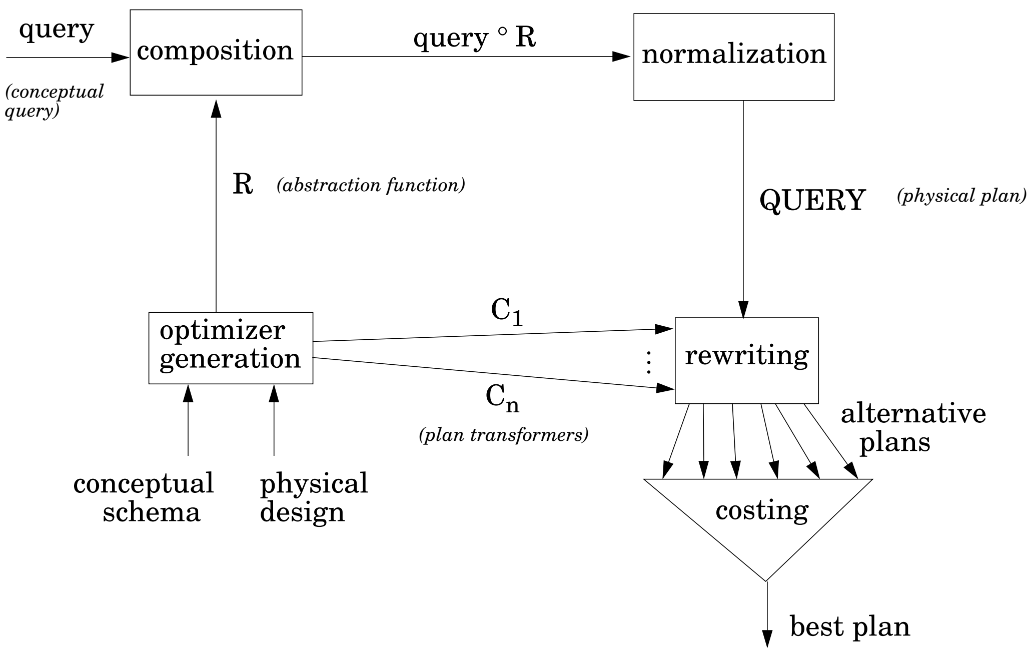


db	=	R(DB)	
QUERY(DB)	=	query(R(DB))	physical plan

Plan transformers:

DB	=	C_i(DB)	
$QUERY_i$(DB)	=	QUERY(C_i(DB))	alternative plans

Query Translator



Example

Conceptual Schema:

set(< A: int, B: set(< C: int, D: int >), E: int >)

Physical design:

- normalize the inner set;
- implement the outer set as a B-tree with key A;
- attach a secondary index to the outer set with a key E.

set(< A: int, B: set(< C: int, D: int >), E: int >) db *(conceptual database)*

↑ R

< T1: sorted[A](< A: int, B: <>, E: int >),
T2: sorted[#](< #: TID, C: int, D: int >),
T3: sorted[E](< #: TID, E: int >) > DB *(physical database)*

Abstraction function:

R(DB) = set{ < A = a.A,
B = set{ < C = b.C, D = b.D > | b ← DB.T2, b.#=@a },
E = a.E >
| a ← DB.T1 }

Plan Transformer: DB = C_1(DB) *(reconstructs* T1 *from* T3*)*

Conceptual query:

query(db) = sum{ y.C | x ← db, y ← x.B, x.A=10, y.D>5 }

Physical plan:

QUERY(DB) = query(R(DB))
= sum{ y.C | x ← R(DB), y ← x.B, x.A=10, y.D>5 }
= … *(after normalization)*
= sum{ b.C | a ← DB.T1, b ← DB.T2,
b.#=@a, a.A=10, b.D>5 }
(A sort-merge join!)

Alternative plan:

QUERY_1(DB) = QUERY(C_1(DB))
= … *(a plan that uses the secondary index* T3*)*

Our Query Translation Framework is Purely Algebraic

It formalizes and generalizes many ad-hoc techniques used in relational query translation and optimization:

- the mapping from logical queries to physical plans is *meaning preserving*;
- the data translation overhead is completely eliminated by normalization;
- alternative physical plans are derived by a very simple cost-directed term-rewriting system that uses 3 types of rules:
  - associativity of monoids;
  - commutativity of some monoids;
  - plan transformers.

Physical Design Specification

A *physical design language* is provided that is

- declarative,
- extensible.

Captures many physical designs:

- object clustering;
- horizontal/vertical partitioning;
- schema normalization;
- join indices;
- multiple access paths via secondary indices.

Conclusion

I have presented:

- a uniform calculus that captures many new database language features:
  - supports expression nesting;
  - allows arbitrary nesting of type constructors;
  - supports multiple collection types;
  - handles aggregates & predicates directly.
- a normalization algorithm that unnests nested comprehensions;
- an effective algebraic model for query translation that facilitates data independence.

Current and Future Work

Model extensions:

- vectors and arrays;
- object identity & mutable objects;
- updates;
- methods.

Implementation:

- translators from OQL and a subset of SQL3 to the monoid calculus;
- a query optimizer.

Restriction

Partial order ≤
between monoids

$$\begin{tikzcd}
& \mathbf{CI}\ (\textit{set, some, all}) & \\
(\textit{bag, sum})\ \mathbf{C} \arrow[ur] & & \mathbf{I}\ (\textit{ordered-set}) \arrow[ul] \\
& \bullet\ (\textit{list}) \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

C: *Commutative*
I: *Idempotent*

hom[T,S] is *well-formed* if and only if $\boxed{T \leq S}$

bag-cardinality(x) = **hom**[*bag,sum*](λa.1)(x)
is well-formed, since *bag* ≤ *sum*

set-cardinality(x) = **hom**[*set,sum*](λa.1)(x)
is **not** well-formed, since *set* > *sum*

$$\begin{tikzcd}
\{1\} \arrow[d, "\text{set-card}"'] & \{1\} \arrow[d, "\text{set-card}"] \arrow[r, "\cup"] & \{1\} \arrow[d, "\text{set-card}"'] \\
1 & 1 \arrow[r, "+"] & 1
\end{tikzcd}$$

Related Work

- *monoid homomorphisms* [V. Tannen et al];
  - SRU
  - ext(f) A = **hom**[T,T](f) A
- *boom hierarchy of types* [R. Bird, L. Meertens, R. Backhouse];
- *monad comprehensions* [P. Wadler, P. Trinder, P. Buneman];
- *normalization* [L. Wong].

In Terms of Category Theory

The collection monoid $T(\alpha)$ is a **free monoid** generated by α:

Adjunction:

$$\begin{tikzcd}
\alpha \arrow[r, "\mathbf{unit}{[T]}"] \arrow[dr, "f"'] & T(\alpha) \arrow[d, dashed, "\mathbf{hom}{[T,S]}(f)"] \\
 & S
\end{tikzcd}$$

unit[T] is a natural transformation: $id \stackrel{\cdot}{\to} T$.

hom[T,S](f) is the **homomorphic extension** of f.

Monads vs. Monoids

$$\begin{tikzcd}
\alpha \arrow[r, "{[.]}"] \arrow[dr, "f"'] & T(\alpha) \arrow[d, dashed, "ext(f)"] \\
 & T(\alpha)
\end{tikzcd}
\qquad
\begin{tikzcd}
\alpha \arrow[r, "\mathbf{unit}{[T]}"] \arrow[dr, "f"'] & T(\alpha) \arrow[d, dashed, "\mathbf{hom}{[T,S]}(f)"] \\
 & S
\end{tikzcd}$$

The Kleisli triple (T, [.], ext(f)), which represents a monad, is a monoid:
the Kleisli composition $f \bullet g \;=\; ext(f) \circ g$ is associative with zero [.].