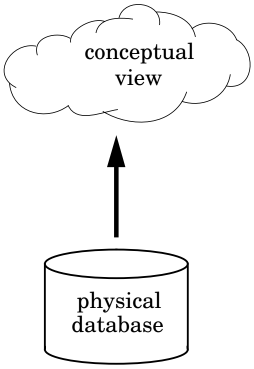
An Algebraic Framework for Physical OODB Design

Leonidas Fegaras David Maier

Oregon Graduate Institute

Data idependence for OODBs



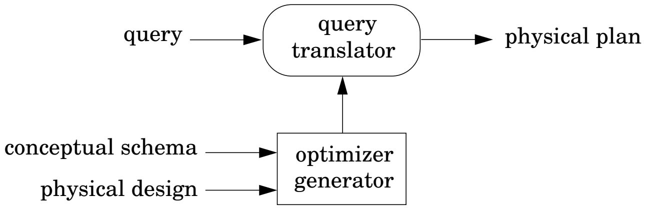
conceptual view

physical database

OODB query optimization is hard:

- richer type systems;
- more expressive query languages;
- more implementation choices:
  - clustering vs. normalization;
  - inverse links;
  - view materialization;
  - object partition;
  - join indices;
  - denormalization;
  - secondary indices.

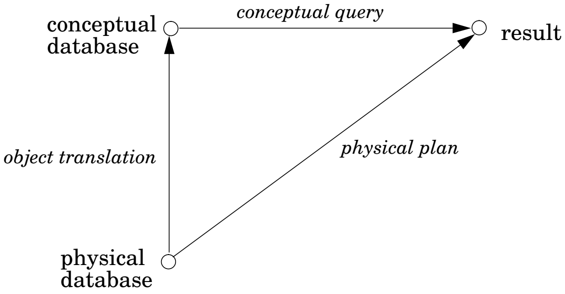
Query Translation in our Framework



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.

Framework Requirements



Both conceptual queries and physical plans must be expressed in the same language.

Need to avoid:

- the object translation overhead;
- generating the conceptual database.

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 }

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

Some Monoids

Collection Monoids

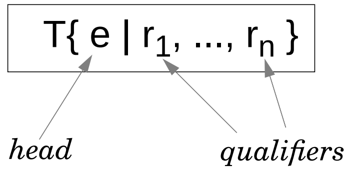
monoid	type	zero	unit(a)	merge
list	list(α)	[ ]	[a]	append
set	set(α)	{ }	{a}	∪
bag	bag(α)	{{ }}	{{a}}	⊎
***sorted*[f]**	list(α)	[ ]	[a]	list_merge[f]

Primitive Monoids

monoid	type	zero	unit(a)	merge
sum	integer	0	a	+
some	boolean	false	a	∨
all	boolean	true	a	∧

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.

---

set{ (a,b) | a ← x, b ← y } =

```
res = { };
for each a in x do
    for each b in y do
        res = res ∪ { (a,b) };
return res;
```

set{ (a,b) | a ← [1,2,3], b ← {{4,5}} } = { (1,4), (1,5), (2,4), (2,5), (3,4), (3,5) }

sum{ a | a ← [1,2,3], a≥2 } = 2+3 = 5

Formal Definition of a Monoid Comprehension

$\mathcal{M}\{ e \mid \}$	=	$\mathbf{unit}^{\mathcal{M}}(e)$
$\mathcal{M}\{ e \mid v \leftarrow \mathbf{zero}^{\mathcal{N}}, r_1, ..., r_n \}$	=	$\mathbf{zero}^{\mathcal{M}}$
$\mathcal{M}\{ e \mid v \leftarrow \mathbf{unit}^{\mathcal{N}}(u), r_1, ..., r_n \}$	=	$\mathbf{let}\ v=u\ \mathbf{in}\ \mathcal{M}\{ e \mid r_1, ..., r_n \}$
$\mathcal{M}\{ e \mid v \leftarrow \mathbf{merge}^{\mathcal{N}}(e_1,e_2), r_1, ..., r_n \}$	=	$\mathbf{merge}^{\mathcal{M}}(\mathcal{M}\{ e \mid v \leftarrow e_1, r_1, ..., r_n \},$ $\mathcal{M}\{ e \mid v \leftarrow e_2, r_1, ..., r_n \})$
$\mathcal{M}\{ e \mid pred, r_1, ..., r_n \}$	=	$\mathbf{if}\ pred\ \mathbf{then}\ \mathcal{M}\{ e \mid r_1, ..., r_n \}$ $\mathbf{else\ zero}^{\mathcal{M}}$

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*$\{ x \mid x \leftarrow e_1, x \in e_2 \}$

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 }

Example from 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:

1) T{ e | $\boxed{❶}$, x ← S{ u | $\boxed{❷}$ }, $\boxed{❸}$ }
→ T{ e | $\boxed{❶}$, $\boxed{❷}$, x ≡ u, $\boxed{❸}$ }

2) T{ e | $\boxed{❶}$, some{ pred | $\boxed{❷}$ }, $\boxed{❸}$ }
→ T{ e | $\boxed{❶}$, $\boxed{❷}$, pred, $\boxed{❸}$ }

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 ← hl,
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) }

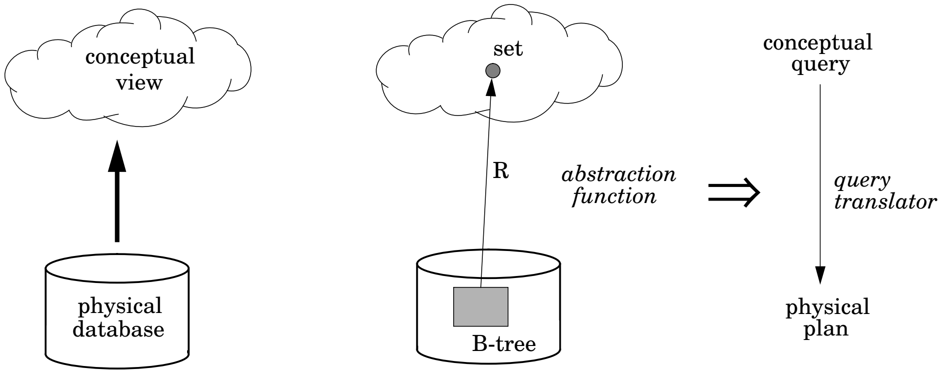
Unnesting Queries

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

Conceptual-to-Internal Mapping



conceptual view

physical database

set

R

B-tree

abstraction function

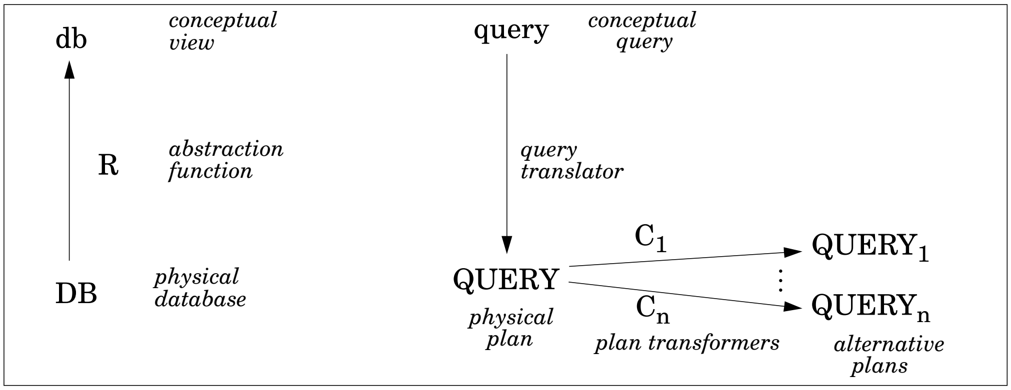
⇒

conceptual query

query translator

physical plan

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



db *conceptual view*

R *abstraction function*

DB *physical database*

query *conceptual query*

query translator

QUERY *physical plan*

C_1

C_n

plan transformers

QUERY_1

QUERY_n

alternative plans

$db = R(DB)$

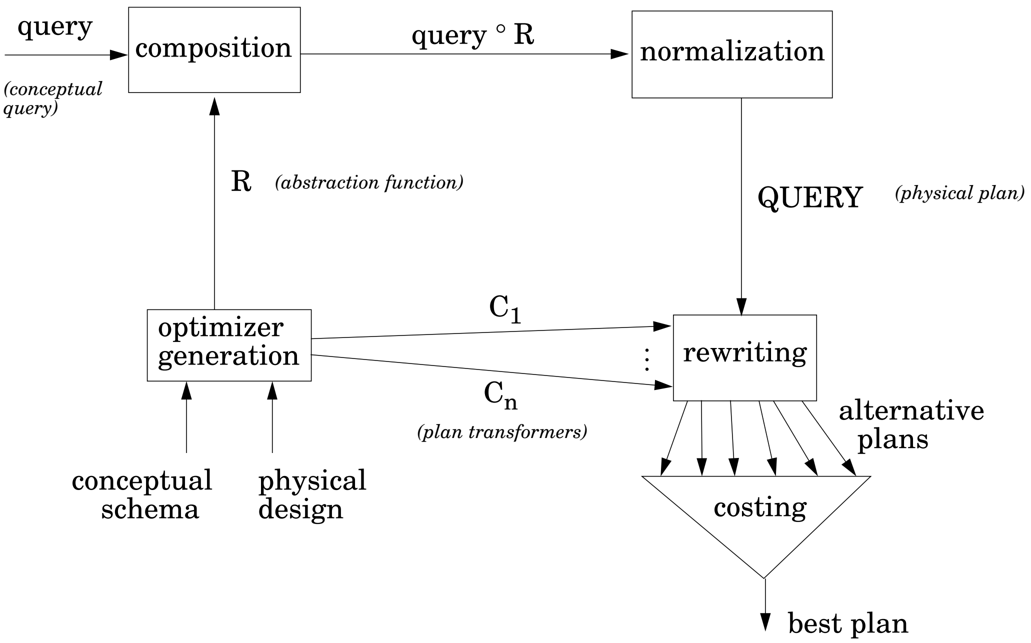
$QUERY(DB) = query(R(DB))$ physical plan

Plan transformers:

$DB = C_i(DB)$

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

Query Translator



query

(conceptual query)

composition

query ° R

normalization

R *(abstraction function)*

QUERY *(physical plan)*

optimizer generation

C_1

⋮

C_n

(plan transformers)

rewriting

alternative plans

conceptual schema

physical design

costing

best plan

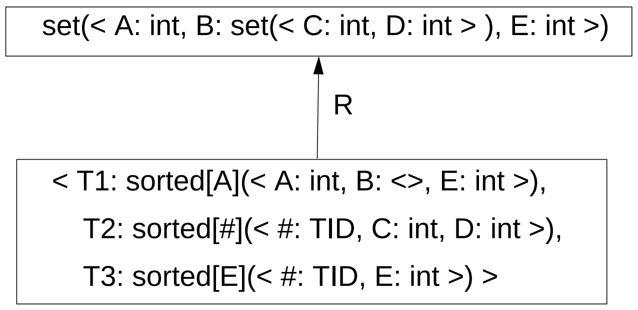
Example

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 key E.



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

< 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*)*

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.

Can be used for translating updates and for restructuring the database.

Conclusion

I have presented:

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