OQL → Monoid Comprehensions

CS395T

Fall 2003

Main Points

- **Motivation:** Similarities between Object-oriented and Relational Database Models are *bulk types*.
- **Observation:** Use *monoids* to treat various bulk types uniformly.
- **Solution:** Monoid comprehension for queries + efficient execution plans (bypass plans)

What is a Monoid?

- From Group Theory, a Monoid is a Semigroup G with an Identity Element
- A Semigroup = (S, $F :: S \times S \rightarrow S$)
  S is a set
  F is an associative function:
  $(a \bullet b) \bullet c = a \bullet (b \bullet c)$
- *monoid = id + F*

What is a monoid? (cont')

- *monoid* M = (*zero*[M], *merge*[M])
- Constructor unit[M] :: $\alpha \rightarrow \alpha$ M
  E.g., unit[list]2 = [2]
- Set = ({ }, {a}, $\cup$), List = ([ ], [a], ++)

What is a monoid? (cont')

- *merge[list](unit[list]1, merge[list](unit[list]2, unit[list]3))*
  = [1] ++ ([2] ++ [3])
  = ([1] ++ [2]) ++ [3]
  = [1,2,3]

Monoid Homomorphisms

- $\text{hom}^{M \rightarrow N}(f)$ *zero*[M] => *zero*[N]
- $\text{hom}^{M \rightarrow N}(f)$ *unit*[M](x) => $f(x)$
- $\text{hom}^{M \rightarrow N}(f)$ *merge*[M] => *merge*[N]
- Compare with fold :: $(a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a$
  $\text{hom}^{M->N}$: (t -> N[t]) -> (M[t] -> N[t])

Monoid Comprehension

- Three Rules:
- N{e| } = unit[N](e)
- N{e | $x \leftarrow E$, l} = $\text{hom}^{M \rightarrow N}(\lambda x.$ N{e | l }) E
- N{e | p, l } = if p then N{e | l } else *zero*[N]

Stuck in nested loops?

- M{e | $x_1 \leftarrow E_1$, …, $x_n \leftarrow E_n$, P } computed:

  foreach x_1 in E_1
  …
  foreach x_n in E_n
  if P then *merge*

Not stuck in nested loops (cont')

- Flattening:
  M{*e* | *x*←*E*, *y*←N{*e*` | *x*`←*E*`,*h*`},*h*}
  = M{*e* | *x*←*E*, *x*`←*E*`,*y*≡*e*`,*h*`,*h*}

- For example,
  M{(*x*,*y*) | *x*←[1,2], *y*←N{*x*`*\*x* | *x*`←[0,1,2]}}
  = M{(*x*,*y*) | *x*←[1,2], *x*`←[0,1,2], *y*≡*x*`**x*}

Bypass-Comprehensions

- Three Rules:

$$\overline{M}\{e|\ \} = \langle unit[M](e), zero[M]\rangle$$

$$\overline{M}\{e|x \leftarrow E, l\} = \hom^{N\ \mathbf{a}\ pair[M]}\left(l\,x.\overline{M}\{e|l\}\right)E$$

$$\overline{M}\{e|p, l\} = \text{if } p \text{ then } \overline{M}\{e|l\} \text{else} \langle zero[M], unit[M](e)\rangle$$

- The third rule makes the second field complement of the first.

Bypass-Comprehension (cont')

$$\overline{bag}\{e|e \leftarrow [1,2,3,1], odd(e)\} = \langle\{\{1,3,1\}\}, \{\{2\}\}\rangle$$

f is a computationally expensive function :

$$\overline{bag}\{e|e \leftarrow [1,2,3,1], odd(e) \vee f(e)\}$$

Only f(2) needs to be worked out.

Query Execution Plan Example

Film	
#	title
1	Foo
2	Goo
3	Hoo

Cast		
film	scene	actor
1	2	Duck
1	1	Goose
2	5	Oxen
3	1	Lion
3	1	Horse
3	4	Bear

Director	
film	director
1	Gozilla
1	Goose
2	Oxen
3	Buck
3	Horse

QEP Example - Rewrite

films = Film ►◄ Director ►◄ Cast

```
select distinct f.title
from films as f
where exists d in f.director : d in (select c.actor
                                     from f.cast as c
                                     where c.sense = 1)
```

⇒ **{f.title | f ¬ films,**
some{some{d=c.actor | c ¬ f.cast, c.scene=1} | d ¬ f.director}}

1. select e(x_1,…,x_n) from E_1 as x_1,…,E_n as x_n where p(x_1,…,x_n) = {{e(x_1,…,x_n) | x_1←E_1,…,x_n←E_n, p(x_1,…,x_n)}}
2. E_1 in E_2 = some{e= E_1 | e←E_2 }

QEP Example - Transform

```
{f.title | f ←films,
    some{some{d=c.actor | c ←f.cast, c.scene=1} | d ←f.director}}
=>
{f.title |
    f ←{<title = fp.title,
        directors = {{dp.director | dp←Director, dp.film = fp.#}},
        cast = [<actor = cp.actor, scene = cp.scene> | cp←Cast,
                                                       cp.film = fp.#]> | fp←Film},
    some{some{d=c.actor | c ←f.cast, c.scene=1} | d ←f.director}}
```

x ← {<e`> | x` ← E`}

I

QEP Example – Transform (cont')

```
{f.title |
    f ←{<title = fp.title,
        directors = {{dp.director | dp←Director, dp.film = fp.#}},
        cast = [<actor = cp.actor, scene = cp.scene> | cp←Cast,
                                                       cp.film = fp.#]> | fp←Film},
    some{some{d=c.actor | c ←f.cast, c.scene=1} | d ←f.director} }
=>
 {f.title |
    fp←Film,
    f ≡< title = fp.title,
        directors = {{dp.director | dp←Director, dp.film = fp.#}},
        cast = [<actor = cp.actor, scene = cp.scene> | cp←Cast, cp.film = fp.#]>,
    some{some{d=c.actor | c ←f.cast, c.scene=1} | d ←f.director} }
```

M{ e | y ← N{ e' | x' ← E' }, r } => M{ e | x' ← E', y ≡ e', r }
where y is bound to f

QEP Example – Transform (cont')

```
{f.title | fp←Film,
    f ≡< title = fp.title,
        directors = {{dp.director | dp←Director, dp.film = fp.#}},
        cast = [<actor = cp.actor, scene = cp.scene> | cp←Cast, cp.film = fp.#]>,
    some{some{d=c.actor
                | c ←f.cast,
                  c.scene=1}
            | d ←f.director} }
=>
{f.title | fp←Film,
    some{some{d=c.actor
                | c ←[<actor = cp.actor, scene = cp.scene> | cp←Cast, cp.film = fp.#],
                  c.scene=1}
            | d ←{{dp.director | dp←Director, dp.film = fp.#}} } }
```

Replace access to components of f with values from definition of f

QEP Example – Transform (cont')

{f.title | f_p←Film,
  some{some{d=c.actor
    | c ←[<actor = c_p.actor, scene = c_p.scene> | c_p←Cast, c_p.film = f_p.#],
      c.scene=1}
    | d ←{{d_p.director | d_p←Director, d_p.film = f_p.#}} } }
=>
{f.title | f_p←Film,
  some{some{d=c.actor
    | c_p←Cast, c_p.film = f_p.#,
      c ≡ <actor = c_p.actor, scene = c_p.scene>
      c.scene=1}
    | d ←{{d_p.director | d_p←Director, d_p.film = f_p.#}} } }

M{ e | a, y ← N{ e' | x' ← E' }, r } => M{ e | a, x' ← E', y ≡ e', r }
where y is bound to c

QEP Example – Transform (cont')

{f.title | f_p←Film,
  some{some{d=c.actor
    | c_p←Cast, c_p.film = f_p.#,
      c ≡ <actor = c_p.actor, scene = c_p.scene>
      c.scene=1}
    | d ←{{d_p.director | d_p←Director, d_p.film = f_p.#}} } }
=>
{f.title | f_p←Film,
  some{some{d= c_p.actor
    | c_p←Cast, c_p.film = f_p.#, c_p.scene =1}
    | d ←{{d_p.director | d_p←Director, d_p.film = f_p.#}} } }

remove definition of "c"

QEP Example – Transform (cont')

{f.title | f_p←Film,
  some{some{d=c_p.actor
    | c_p←Cast, c_p.film = f_p.#, c_p.scene =1}
    | d ←{{d_p.director | d_p←Director, d_p.film = f_p.#}} } }
=>
{f.title | f_p←Film,
  some{some{d=c_p.actor
    | c_p←Cast, c_p.film = f_p.#, c_p.scene =1}
    | d_p←Director, d_p.film = f_p.#,
      d ≡ d_p.director } }

M{ e | y ← N{ e' | x' ← E' } } => M{ e | x' ← E', y ≡ e' }
where y is bound to d

QEP Example – Transform (cont')

{f.title | f_p←Film,
  some{some{d=c_p.actor | c_p←Cast, c_p.film = f_p.#, c_p.scene =1}
    | d_p←Director, d_p.film = f_p.#,
      d ≡ d_p.director } }
=>
{f.title | f_p←Film,
  some{some{d_p.director =c_p.actor | c_p←Cast, c_p.film = f_p.#, c_p.scene =1}
    | d_p←Director, d_p.film = f_p.# } }

Remove definition of "d"

QEP Example – Transform (cont')

{f.title | f_p←Film,
  some{some{d_p.director =c_p.actor | c_p←Cast, c_p.film = f_p.#, c_p.scene =1}
    | d_p←Director, d_p.film = f_p.# } }
=>
{f.title | f_p←Film,
  d_p←Director, d_p.film = f_p.#
  some{d_p.director =c_p.actor | c_p←Cast, c_p.film = f_p.#, c_p.scene =1}
  }

(variant of their rule)
M{ e | N{ e' | x' ← E' } } => M{ e | x' ← E', e' }

QEP Example – Transform (cont')

{f.title |
  f ←{<title = f_p.title,
    directors = {{d_p.director | d_p←Director, d_p.film = f_p.#}},
    cast = [<actor = c_p.actor, scene = c_p.scene> | c_p←Cast,
      c_p.film = f_p.#]> | f_p←Film},
  some{some{d=c.actor | c ←f.cast, c.scene=1} | d ←f.director}}
=>
{f_p.title | f_p← Film, d_p←Director, d_p.film = f_p.#,
  some{d_p.director=c_p.actor | c_p ←Cast, c_p.film=f_p.#, c_p.scene=1}}

summary of transformation

QEP Example – Transform (cont')

{f_p.title | f_p← Film, d_p←Director, d_p.film = f_p.#,
  some{d_p.director=c_p.actor | c_p ←Cast, c_p.film=f_p.#, c_p.scene=1}}

$Films_f$◀[f#=d.film] ($Directors_d$◀[c.actor=d.dir∧d.film=c.film] (S[c.scenes=1] $Cast_c$))

Translation to relational algebra with *semijoins* ◀:
A_x ◀[p] B_y = { x | x ← A, some{ p | y ← B }

How would this really be evaluated? Depends on estimations and indexes. Eg:

```
for c in index-scan(Cast.scenes, 1)
    for d in index-scan(Directors.film, c.film)
        if c.actor=d.dir then
            let f=index-find(Films.f#, d.film)
                yield(f.title)
```


Detecting Patterns

- Use Semi-join
- But how easy is it to detect pattern
- Results in a hybrid of monoid and calculus and relational algebra
- QEP is not executable yet. No backend support

Discussion

- Main point: monoid comprehensions can be efficiently evaluated using rewriting and bypass
- Bridge the gap between conceptual monoid queries and efficient implementations???
- Tighter and better integration between databases and functional programming?