Fundamentals of Physical Design
Query Processing: Conjunctive Queries

David Toman

D. R. Cheriton School of Computer Science
University of Waterloo

Story so far . . .

Two approaches to *physical design*:

1. Current practice:

   Changes to logical schema + index selection

   . . . destroys physical data independence

2. Desired solution:

   Integrity constraints + index selection

   . . . preserves physical data independence

. . . but how do we now execute queries?

Story so far . . .

Two approaches to *physical design*:

1. Current practice:

   Changes to logical schema + index selection

   . . . destroys physical data independence

2. Desired solution:

   Integrity constraints + index selection

   . . . preserves physical data independence

. . . but how do we now execute queries?

Story so far . . .

Two approaches to *physical design*:

1. Current practice:

   Changes to logical schema + index selection

   . . . destroys physical data independence

2. Desired solution:

   Integrity constraints + index selection

   . . . preserves physical data independence

. . . but how do we now execute queries?

Query Language: Conjunctive Queries

Syntax:

`Q ::= A v`	class access
`  | v.Pf1 = u.Pf2`	equation
`  | true`	singleton
`  | from Q1,Q2`	natural join
`  | elim v1,...,vk Q`	selection (select distinct)

… usual "normal form" a.k.a. SELECT block

Definition (Meaning)

Let D be a database instance and φ_{Q} a formula corresponding to Q.

$$\mathrm{Q}(D) = \{\{v_1 = o_1, \ldots, v_k = o_k\} \mid D, \{v_1 = o_1, \ldots, v_k = o_k\} \models \varphi_{\mathrm{Q}}\}$$

… alternatively, an equivalent an algebraic definition

Query Language: Conjunctive Queries

Syntax:

`Q ::= A v`	class access
`  | v.Pf1 = u.Pf2`	equation
`  | true`	singleton
`  | from Q1,Q2`	natural join
`  | elim v1,...,vk Q`	selection (select distinct)

... usual "normal form" a.k.a. SELECT block

Definition (Meaning)

Let D be a database instance and $\varphi_{\mathtt{Q}}$ a formula corresponding to $\mathtt{Q}$.

$$\mathtt{Q}(D) = \{\{v_1 = o_1, \ldots, v_k = o_k\} \mid D, \{v_1 = o_1, \ldots, v_k = o_k\} \models \varphi_{\mathtt{Q}}\}$$

... alternatively, an equivalent an algebraic definition

Example

Query: given an employee id (:p), list name of the employee and addresses of their department:

```
elim ename, dcity, :p
from EMPLOYEE e, DEPARTMENT d,
     e.eid=:p, e.Dept=d,
     ename=e.Name, dcity=d.City
```

Graphical Representation:

Example

Query: given an employee id (`:p`), list name of the employee and addresses of their department:

```
elim ename, dcity, :p
from EMPLOYEE e, DEPARTMENT d,
     e.eid=:p, e.Dept=d,
     ename=e.Name, dcity=d.City
```

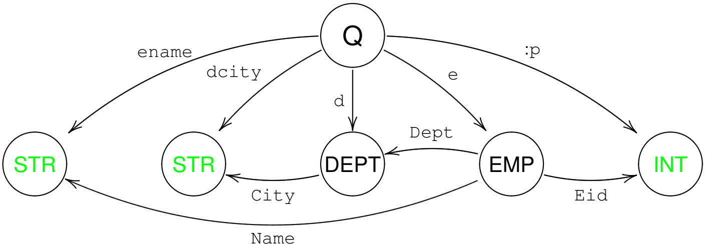
Graphical Representation:

Example

Query: given an employee id (`:p`), list name of the employee and addresses of their department:

```
elim ename, dcity, :p
from EMPLOYEE e, DEPARTMENT d,
     e.eid=:p, e.Dept=d,
     ename=e.Name, dcity=d.City
```

Graphical Representation:



Query Plans ∼ Patterns in QL

IDEA: Extend *binding patterns* to queries

BP(Q) is a pair (I, O) of path sets where
I are the expected input parameters and O the outputs.

- BP(`A v`) is $(v.I, v.O)$ if an "`index A` I O" declaration exists;
- BP(`v.Pf1 = u.Pf2`)
  is $(\{$`v.Pf1`$\}, \{$`u.Pf2`$\})$ or $(\{$`u.Pf2`$\}, \{$`v.Pf1`$\})$;
- BP(`true`) is $(\{\}, \{\})$;
- BP(`from Q1,Q2`)
  is $(I_1 \cup (I_2 - O_1), O_1 \cup O_2)$ for BP(`Qi`) $= (I_i, O_i)$; and
- BP(`elim V Q`)
  is $(I, O \cap V)$ for BP(`Q`) $= (I, O)$ and $I \subseteq V$.

A query Q is a *plan* if BP(Q) $= (P, FV(Q))$ where P are parameters.

Query Compilation $\sim$ Equivalence under Constraints

Chase Step

Replace "`D x`" with "`from D x, E x`" if $\mathcal{T} \cup \mathcal{Q} \models$ `D<E`,
where $\mathcal{T}$ is the schema and $\mathcal{Q}$ are constraints induced by `Q`.

... easy to see that this preserves equivalence.

How can we use this??

1. (repeatedly) apply chase to `Q`;
2. extract plan by traversing result using `index` declarations;
3. (repeatedly) apply chase on the plan;
4. if results of (1) and (3) are the same:
   signal "success" otherwise signal "no plan"

Query Compilation $\sim$ Equivalence under Constraints

Chase Step

Replace "`D x`" with "`from D x, E x`" if $\mathcal{T} \cup \mathcal{Q} \models$ `D<E`,
where $\mathcal{T}$ is the schema and $\mathcal{Q}$ are constraints induced by `Q`.

. . . easy to see that this preserves equivalence.

How can we use this??

1. (repeatedly) apply chase to `Q`;
2. extract plan by traversing result using `index` declarations;
3. (repeatedly) apply chase on the plan;
4. if results of (1) and (3) are the same:
   signal "success" otherwise signal "no plan"

In practice (1-3) have to be interleaved as chase may not terminate

Query Compilation ∼ Equivalence under Constraints

Chase Step

Replace "`D x`" with "`from D x, E x`" if $\mathcal{T} \cup \mathcal{Q} \models$ `D<E`,
where $\mathcal{T}$ is the schema and $\mathcal{Q}$ are constraints induced by `Q`.

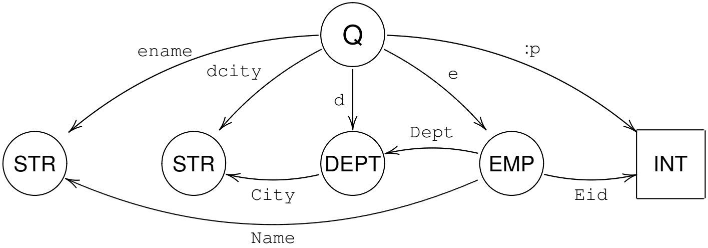
... easy to see that this preserves equivalence.

How can we use this??

1. (repeatedly) apply chase to `Q`;
2. extract plan by traversing result using `index` declarations;
3. (repeatedly) apply chase on the plan;
4. if results of (1) and (3) are the same:
signal "success" otherwise signal "no plan"

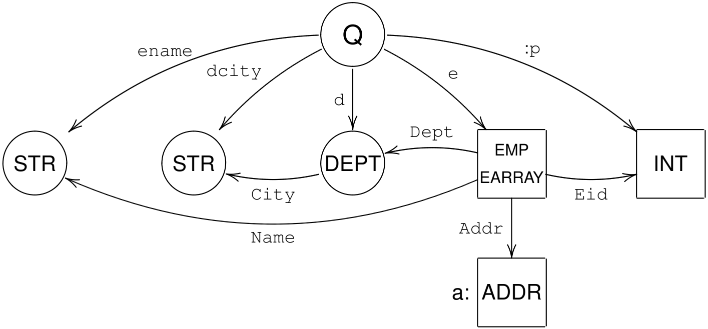
In practice (1-3) have to be interleaved as chase may not terminate

Example



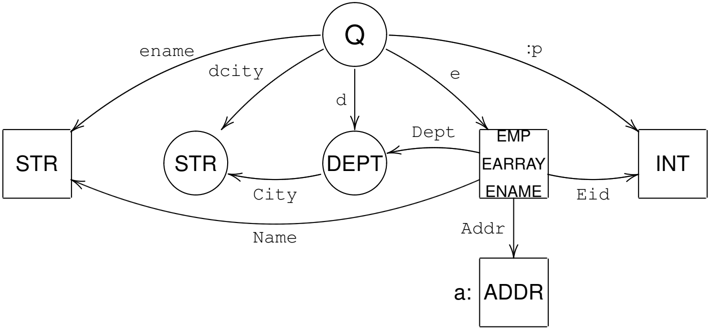
PLAN:
```
elim ename,dcity,:p
from true,
```

Example



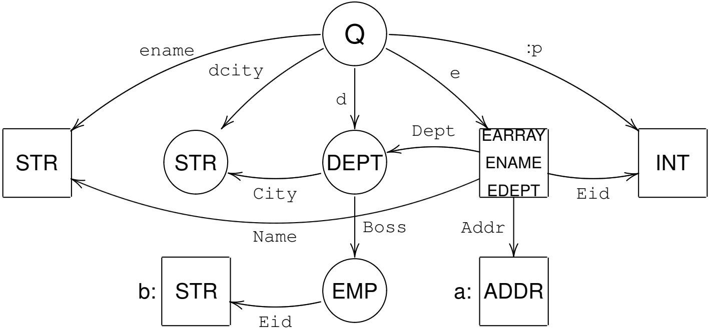
PLAN:
```
elim ename,dcity,:p
from true,
 ( e.Eid = :p, EARRAY e, a = e.Addr ),
```

Example



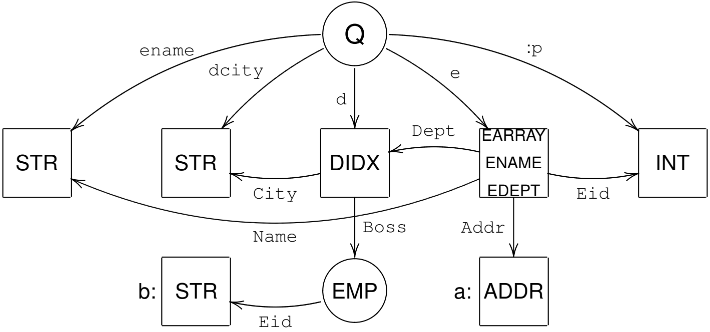
PLAN:
```
elim ename,dcity,:p
from true,
  ( e.Eid = :p, EARRAY e, a = e.Addr ),
  ( e.Addr = a, ENAME e, ename = e.Name ),
```

Example



PLAN:
```
elim ename,dcity,:p
from true,
 ( e.Eid = :p, EARRAY e, a = e.Addr ),
 ( e.Addr = a, ENAME e, ename = e.Name ),
 ( e.Addr = a, EDEPT e, b = e.Dept.Boss.Eid ),
```

Example



PLAN:
```
elim ename,dcity,:p
from true,
  ( e.Eid = :p, EARRAY e, a = e.Addr ),
  ( e.Addr = a, ENAME e, ename = e.Name ),
  ( e.Addr = a, EDEPT e, b = e.Dept.Boss.Eid ),
  ( d.Boss.Eid = b, DIDX d, dcity = d.City )
```

More about Plans

- Alternative plans (e.g., join-order selection?)

  ⇒ YES: non-determinism in extracting PLANs

- Does a PLAN always exist?

  ⇒ NO (i.e., the "current" design cannot support the query)

- If a PLAN exists, do we find it?

  ⇒ NO (in general—depends on integrity constraints)

  . . . e.g., we do not have an *empty query* construct.

More about Plans

- Alternative plans (e.g., join-order selection?)

  $\Rightarrow$ YES: non-determinism in extracting PLANs

- Does a PLAN always exist?

  $\Rightarrow$ NO (i.e., the "current" design cannot support the query)

- If a PLAN exists, do we find it?

  $\Rightarrow$ NO (in general—depends on integrity constraints)

  … e.g., we do not have an *empty query* construct.

More about Plans

- Alternative plans (e.g., join-order selection?)

  ⇒ YES: non-determinism in extracting PLANs

- Does a PLAN always exist?

  ⇒ NO (i.e., the "current" design cannot support the query)

- If a PLAN exists, do we find it?

  ⇒ NO (in general—depends on integrity constraints)
  . . . e.g., we do not have an *empty query* construct.

Database Trimmings: Duplicates et al.

- SQL (OQL) queries allow duplicate semantics:

Syntax:

`Q ::= A v`	class access
`  | v.Pf1 = u.Pf2`	equation
`  | true`	singleton
`  | from Q,Q`	natural join
`  | elim v1,...,vk Q`	selection (distinct)
`  | select v1,...,vk Q`	selection (with duplicates)

$\Rightarrow$ algebraic semantics

- IDEA: mark which variables do not need to be *deduplicated*

  + transformation that uses PFDs to manipulate the marking

Database Trimmings: Duplicates et al.

- SQL (OQL) queries allow duplicate semantics:

Syntax:

`Q ::= A v`	class access
`  | v.Pf1 = u.Pf2`	equation
`  | true`	singleton
`  | from Q,Q`	natural join
`  | elim v1,...,vk Q`	selection (distinct)
`  | select v1,...,vk Q`	selection (with duplicates)

$\Rightarrow$ algebraic semantics

- IDEA: mark which variables do not need to be *deduplicated*

  + transformation that uses PFDs to manipulate the marking

Duplicate Elimination Elimination

Normal Form for Queries w/Duplicates:

```
select V from A1 v1,..., Am vm,
      ( elim W from B1 w1,..., Bn wn, R)
```

where R are all the equations.

Transformation Rule:

```
select V from A1 v1,..., Am vm,
      ( elim W from B1 w1,..., Bn wn, R)
```

is semantically equivalent to

```
select V from A1 v1,..., Am vm, B1 w1,
      ( elim W, w1 from B2 w2,..., Bn wn, R)
```

if and only if `Q < Q: v1,...,vn, W -> w1`.

Duplicate Elimination Elimination

Normal Form for Queries w/Duplicates:

```
select V from A1 v1,..., Am vm,
      ( elim W from B1 w1,..., Bn wn, R)
```

where `R` are all the equations.

Transformation Rule:

```
select V from A1 v1,..., Am vm,
      ( elim W from B1 w1,..., Bn wn, R)
```

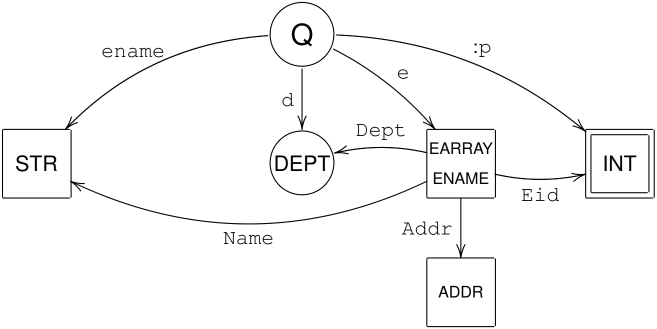
is semantically equivalent to

```
select V from A1 v1,..., Am vm, B1 w1,
      ( elim W, w1 from B2 w2,..., Bn wn, R)
```

if and only if `Q < Q: v1,...,vn, W -> w1`.

Example: Chase and Variable Tags

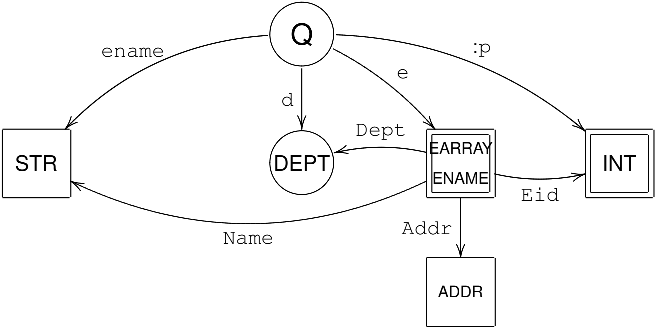
- **Query:** "`elim name from EMPLOYEE e, DEPARTMENT d,`
  `e.Eid = :p, e.Dept=d, name=e.Name`"



- Plan: "`select ename from`
  `(e.Eid=p, EARRAY e, a = e.Addr),`
  `(e.Addr = a, ENAME e, ename = e.Name)`"

Example: Chase and Variable Tags

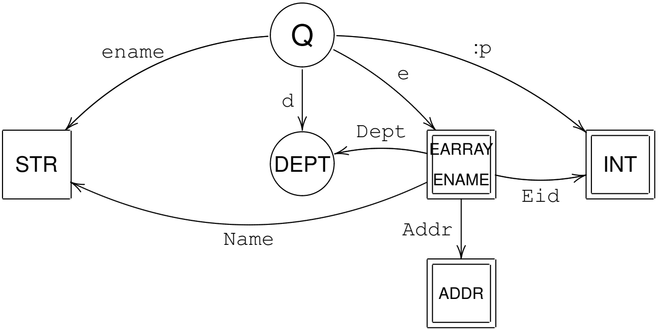
- **Query:** “`elim name from EMPLOYEE e, DEPARTMENT d, e.Eid = :p, e.Dept=d, name=e.Name`”



- Plan: “`select ename from (e.Eid=p, EARRAY e, a = e.Addr), (e.Addr = a, ENAME e, ename = e.Name)`”

Example: Chase and Variable Tags

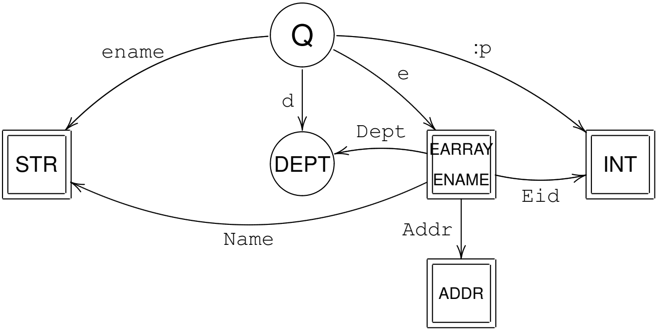
- **Query:** "`elim name from EMPLOYEE e, DEPARTMENT d, e.Eid = :p, e.Dept=d, name=e.Name`"



- Plan: "`select ename from (e.Eid=p, EARRAY e, a = e.Addr), (e.Addr = a, ENAME e, ename = e.Name)`"

Example: Chase and Variable Tags

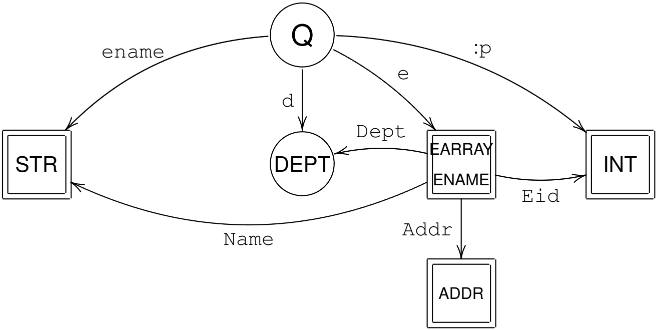
- **Query:** "`elim name from EMPLOYEE e, DEPARTMENT d, e.Eid = :p, e.Dept=d, name=e.Name`"



- Plan: "`select ename from (e.Eid=p, EARRAY e, a = e.Addr), (e.Addr = a, ENAME e, ename = e.Name)`"

Example: Chase and Variable Tags

- Query: "elim name from EMPLOYEE e, DEPARTMENT d,
  e.Eid = :p, e.Dept=d, name=e.Name"



- Plan: "select ename from
  (e.Eid=p, EARRAY e, a = e.Addr),
  (e.Addr = a, ENAME e, ename = e.Name)"

Bigger Languages: Positive Queries w/Duplicates

Syntax:

`Q ::= A v`	class access
`  | v.Pf1 = u.Pf2`	equation
`  | true`	singleton
`  | from Q1,Q2`	natural join
`  | elim v1,...,vk Q`	selection (distinct)
`  | select v1,...,vk Q`	selection (with duplicates)
`  | empty v1,...,vk`	empty set
`  | Q1 union all Q2`	concatenation (union-compatible)

…input query is still conjunctive (w/duplicate semantics)

⇒ `union` arises from the SCHEMA

Handling OR in Schema

- additional *expansion rule*:

Chase Step

Replace "`(D or E) x`" with
"`elim x ( D x union all E x )`"

- and rules for handling duplicates:

Duplicates and Union Step

"`elim V ( Q1 union all Q2 )`" rewrites to
"`(elim V Q1) union all (elim V Q2)`"
if (abstractions of) `Q1` and `Q2` are disjoint

Handling OR in Schema

- additional *expansion rule*:

Chase Step

Replace "`(D or E) x`" with
"`elim x ( D x union all E x )`"

- and rules for handling duplicates:

Duplicates and Union Step

"`elim V ( Q1 union all Q2 )`" rewrites to
"`(elim V Q1) union all (elim V Q2)`"
if (abstractions of) `Q1` and `Q2` are disjoint

Example: Horizontal Partition

Physical design: two disjoint indices for `WATEMP` and `TOKYOEMP`.

- Expansion of "`select eid from EMPLOYEE e, eid=e.Eid`"

1. "`select eid from EMPLOYEE e,`
   `(WATEMP or TOKYOEMP) e, eid=e.Eid`"
2. "`select eid from EMPLOYEE e,`
   `( elim e (WATEMP e union all TOKYOEMP e) ),`
   `eid=e.Eid`"
3. "`select eid from EMPLOYEE e,`
   `( elim e,eid (WATEMP e union all TOKYOEMP e),`
   `eid=e.Eid)`"
4. "`select eid from EMPLOYEE e,`
   `(elim e,eid WATEMP e, eid=E.eid) union all`
   `(elim e,eid TOKYOEMP e, eid=e.Eid)`"
5. "`select eid from EMPLOYEE e, ((WATEMP e,eid=e.Eid)`
   `union all (TOKYOEMP e,eid=e.Eid) )`"

- Plan: "`select eid (WATEMP e, eid=e.Eid) union all`
  `(TOKYOEMP e, eid=e.Eid)`"

Example: Horizontal Partition

Physical design: two disjoint indices for `WATEMP` and `TOKYOEMP`.

- Expansion of “`select eid from EMPLOYEE e, eid=e.Eid`”
  1. “`select eid from EMPLOYEE e,`
     `(WATEMP or TOKYOEMP) e, eid=e.Eid`”
  2. “`select eid from EMPLOYEE e,`
     `( elim e (WATEMP e union all TOKYOEMP e) ),`
     `eid=e.Eid`”
  3. “`select eid from EMPLOYEE e,`
     `( elim e,eid (WATEMP e union all TOKYOEMP e),`
     `eid=e.Eid)`”
  4. “`select eid from EMPLOYEE e,`
     `(elim e,eid WATEMP e, eid=E.eid) union all`
     `(elim e,eid TOKYOEMP e, eid=e.Eid)`”
  5. “`select eid from EMPLOYEE e, ((WATEMP e,eid=e.Eid)`
     `union all (TOKYOEMP e,eid=e.Eid) )`”
- Plan: “`select eid (WATEMP e, eid=e.Eid) union all`
  `(TOKYOEMP e, eid=e.Eid)`”

Example: Horizontal Partition

Physical design: two disjoint indices for `WATEMP` and `TOKYOEMP`.

- Expansion of "`select eid from EMPLOYEE e, eid=e.Eid`"
  1. "`select eid from EMPLOYEE e,`
     `(WATEMP or TOKYOEMP) e, eid=e.Eid`"
  2. "`select eid from EMPLOYEE e,`
     `( elim e (WATEMP e union all TOKYOEMP e) ),`
     `eid=e.Eid`"
  3. "`select eid from EMPLOYEE e,`
     `( elim e,eid (WATEMP e union all TOKYOEMP e),`
     `eid=e.Eid)`"
  4. "`select eid from EMPLOYEE e,`
     `(elim e,eid WATEMP e, eid=E.eid) union all`
     `(elim e,eid TOKYOEMP e, eid=e.Eid)`"
  5. "`select eid from EMPLOYEE e, ((WATEMP e,eid=e.Eid)`
     `union all (TOKYOEMP e,eid=e.Eid) )`"
- Plan: "`select eid (WATEMP e, eid=e.Eid) union all`
  `(TOKYOEMP e, eid=e.Eid)`"

Example: Horizontal Partition

Physical design: two disjoint indices for `WATEMP` and `TOKYOEMP`.

- Expansion of "`select eid from EMPLOYEE e, eid=e.Eid`"

  1. "`select eid from EMPLOYEE e, (WATEMP or TOKYOEMP) e, eid=e.Eid`"
  2. "`select eid from EMPLOYEE e, ( elim e (WATEMP e union all TOKYOEMP e) ), eid=e.Eid`"
  3. "`select eid from EMPLOYEE e, ( elim e,eid (WATEMP e union all TOKYOEMP e), eid=e.Eid)`"
  4. "`select eid from EMPLOYEE e, (elim e,eid WATEMP e, eid=E.eid) union all (elim e,eid TOKYOEMP e, eid=e.Eid)`"
  5. "`select eid from EMPLOYEE e, ((WATEMP e,eid=e.Eid) union all (TOKYOEMP e,eid=e.Eid) )`"

- Plan: "`select eid (WATEMP e, eid=e.Eid) union all (TOKYOEMP e, eid=e.Eid)`"

Example: Horizontal Partition

Physical design: two disjoint indices for WATEMP and TOKYOEMP.

- Expansion of "select eid from EMPLOYEE e, eid=e.Eid"
  1. "select eid from EMPLOYEE e,
     (WATEMP or TOKYOEMP) e, eid=e.Eid"
  2. "select eid from EMPLOYEE e,
     (elim e (WATEMP e union all TOKYOEMP e)),
     eid=e.Eid"
  3. "select eid from EMPLOYEE e,
     (elim e,eid (WATEMP e union all TOKYOEMP e),
     eid=e.Eid)"
  4. "select eid from EMPLOYEE e,
     (elim e,eid WATEMP e, eid=E.eid) union all
     (elim e,eid TOKYOEMP e, eid=e.Eid)"
  5. "select eid from EMPLOYEE e, ((WATEMP e,eid=e.Eid)
     union all (TOKYOEMP e,eid=e.Eid))"
- Plan: "select eid (WATEMP e, eid=e.Eid) union all
  (TOKYOEMP e, eid=e.Eid)"

Example: Horizontal Partition

Physical design: two disjoint indices for `WATEMP` and `TOKYOEMP`.

- Expansion of "`select eid from EMPLOYEE e, eid=e.Eid`"

1. "
```
select eid from EMPLOYEE e,
                (WATEMP or TOKYOEMP) e, eid=e.Eid
```
"
2. "
```
select eid from EMPLOYEE e,
        ( elim e (WATEMP e union all TOKYOEMP e) ),
                                            eid=e.Eid
```
"
3. "
```
select eid from EMPLOYEE e,
      ( elim e,eid (WATEMP e union all TOKYOEMP e),
                                           eid=e.Eid)
```
"
4. "
```
select eid from EMPLOYEE e,
        (elim e,eid WATEMP e, eid=E.eid) union all
               (elim e,eid TOKYOEMP e, eid=e.Eid)
```
"
5. "
```
select eid from EMPLOYEE e, ((WATEMP e,eid=e.Eid)
               union all (TOKYOEMP e,eid=e.Eid) )
```
"

- Plan: "`select eid (WATEMP e, eid=e.Eid) union all (TOKYOEMP e, eid=e.Eid)`"

Example: Horizontal Partition

Physical design: two disjoint indices for `WATEMP` and `TOKYOEMP`.

- Expansion of “`select eid from EMPLOYEE e, eid=e.Eid`”
  1. “`select eid from EMPLOYEE e,`
     `(WATEMP or TOKYOEMP) e, eid=e.Eid`”
  2. “`select eid from EMPLOYEE e,`
     `( elim e (WATEMP e union all TOKYOEMP e) ),`
     `eid=e.Eid`”
  3. “`select eid from EMPLOYEE e,`
     `( elim e,eid (WATEMP e union all TOKYOEMP e),`
     `eid=e.Eid)`”
  4. “`select eid from EMPLOYEE e,`
     `(elim e,eid WATEMP e, eid=E.eid) union all`
     `(elim e,eid TOKYOEMP e, eid=e.Eid)`”
  5. “`select eid from EMPLOYEE e, ((WATEMP e,eid=e.Eid)`
     `union all (TOKYOEMP e,eid=e.Eid) )`”
- Plan: “`select eid (WATEMP e, eid=e.Eid) union all`
  `(TOKYOEMP e, eid=e.Eid)`”

Where does this Fail?

1. Input: general first-order queries:

   ⇒ best approaches so far ala QGM i.e., block-by block

2. Negations in schema: what to do with "`(not A) x`"?

   ⇒ restrictions on the schema language?

   ⇒ more general "rewriting rules"?

3. Completeness?

   ⇒ conjunctive query over conjunctive materialized views

   . . . needs *negation* in the plan!

Where does this Fail?

1. Input: general first-order queries:

   ⇒ best approaches so far ala QGM i.e., block-by block

2. Negations in schema: what to do with "`(not A) x`"?

   ⇒ restrictions on the schema language?

   ⇒ more general "rewriting rules"?

3. Completeness?

   ⇒ conjunctive query over conjunctive materialized views

   . . . needs *negation* in the plan!

Summary

This is the *BEST* approach known today that . . .

- handles duplicates, and
- accommodates binding patterns

. . . in practice commonly competitive with hand-written C code

. . . extensions to *ordering constraints* “in progress”

Next time:

How to deal with all first-order queries:

why is it worth reading older papers (on Logic).

. . . with speculation on dealing w/duplicates and binding patterns

Summary

This is the *BEST* approach known today that ...

- handles duplicates, and
- accommodates binding patterns

... in practice commonly competitive with hand-written C code

... extensions to *ordering constraints* "in progress"

Next time:

How to deal with all first-order queries:

why is it worth reading older papers (on Logic).

... with speculation on dealing w/duplicates and binding patterns