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:

\[ Q ::= A \lor v \quad \text{class access} \\
| \quad v.Pf1 = u.Pf2 \quad \text{equation} \\
| \quad \text{true} \quad \text{singleton} \\
| \quad \text{from } Q1,Q2 \quad \text{natural join} \\
| \quad \text{elim } v1,\ldots,vk \quad Q \quad \text{selection (select distinct)} \]

... usual “normal form” a.k.a. SELECT block

Definition (Meaning)

Let \( D \) be a database instance and \( \varphi_Q \) a formula corresponding to \( Q \).

\[ Q(D) = \{ \{ v_1 = o_1, \ldots, v_k = o_k \} \mid D, \{ v_1 = o_1, \ldots, v_k = o_k \} \models \varphi_Q \} \]

... alternatively, an equivalent an algebraic definition
Query Language: Conjunctive Queries

Syntax:

\[ Q ::= A \lor \text{class access} \]
\[ \mid v.Pf1 = u.Pf2 \quad \text{equation} \]
\[ \mid \text{true} \quad \text{singleton} \]
\[ \mid \text{from } Q1,Q2 \quad \text{natural join} \]
\[ \mid \text{elim } v1,...,vk \quad Q \quad \text{selection (select distinct)} \]

... usual “normal form” a.k.a. SELECT block

Definition (Meaning)

Let \( D \) be a database instance and \( \varphi_Q \) a formula corresponding to \( Q \).

\[ Q(D) = \{ \{ v_1 = o_1, \ldots, v_k = o_k \} \mid D, \{ v_1 = o_1, \ldots, v_k = o_k \} \models \varphi_Q \} \]

... 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
```
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.

\[
\begin{align*}
\text{BP}(A \ v) & \text{ is } (v.I, v.O) \text{ if an "index A I O" declaration exists}; \\
\text{BP}(v.Pf1 = u.Pf2) & \text{ is } (\{v.Pf1\}, \{u.Pf2\}) \text{ or } (\{u.Pf2\}, \{v.Pf1\}); \\
\text{BP}(\text{true}) & \text{ is } (\{} , \{}); \\
\text{BP}(\text{from Q1,Q2}) & \text{ is } (I_1 \cup (I_2 - O_1), O_1 \cup O_2) \text{ for } BP(Q_i) = (I_i, O_i); \text{ and } \\
\text{BP elim V Q} & \text{ is } (I, O \cap V) \text{ for } BP(Q) = (I, O) \text{ and } I \subseteq V.
\end{align*}
\]

A query \(Q\) is a plan if \(BP(Q) = (P, FV(Q))\) where \(P\) are parameters.
Query Compilation ~ Equivalence under Constraints

Chase Step

Replace “\(D \times\)” with “\(\text{from } D \times, E \times\)” if \(T \cup Q \models D<E\), where \(T\) is the schema and \(Q\) 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 \( \sim \) Equivalence under Constraints

Chase Step

Replace “\(D \ x\)” with “\(\text{from } D \ x, \ E \ x\)” if \(T \cup Q \models D<E\),
where \(T\) is the schema and \(Q\) are constraints induced by \(Q\).

\[\ldots\text{ easy to see that this preserves equivalence.}\]

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 \( \mathcal{T} \cup \mathcal{Q} \models \text{D}<\text{E} \), where \( \mathcal{T} \) is the schema and \( \mathcal{Q} \) are constraints induced by \( \mathcal{Q} \).

... easy to see that this preserves equivalence.

How can we use this??

1. (repeatedly) apply chase to \( \mathcal{Q} \);
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,

D. Toman (Waterloo)
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 ),
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?)
  ⇒ 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?)
  ⇒ 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:**

  $Q ::= A \nu$
  
  $| \nu.Pf_1 = u.Pf_2$  
  $| \text{true}$  
  $| \text{from } Q,Q$  
  $| \text{elim } v_1,\ldots,v_k Q$  
  $| \text{select } v_1,\ldots,v_k Q$

  $\Rightarrow$ **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:**

\[ Q ::= A \vee v \]

- \( v.Pf1 = u.Pf2 \) equation
- \( \text{true} \) singleton
- \( \text{from } Q,Q \) natural join
- \( \text{elim } v1,\ldots,vk \ Q \) selection (distinct)
- \( \text{select } v1,\ldots,vk \ Q \) selection (with duplicates)

⇒ 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:

\[
\text{select } V \text{ from } A_1 v_1, \ldots, A_m v_m, \\
(\text{elim } W \text{ from } B_1 w_1, \ldots, B_n w_n, R) \\
\text{where } R \text{ are all the equations.}
\]

Transformation Rule:

\[
\text{select } V \text{ from } A_1 v_1, \ldots, A_m v_m, \\
(\text{elim } W \text{ from } B_1 w_1, \ldots, B_n w_n, R) \\
is \text{semantically equivalent to} \\
\text{select } V \text{ from } A_1 v_1, \ldots, A_m v_m, B_1 w_1, \\
(\text{elim } W, w_1 \text{ from } B_2 w_2, \ldots, B_n w_n, R) \\
\text{if and only if } Q < Q: v_1, \ldots, v_n, W \rightarrow w_1.
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:

\[ Q ::= A \lor \quad \text{class access} \]
| \[ v.Pf_1 = u.Pf_2 \quad \text{equation} \]
| \[ \text{true} \quad \text{singleton} \]
| \[ \text{from } Q_1, Q_2 \quad \text{natural join} \]
| \[ \text{elim } v_1, \ldots, v_k \ Q \quad \text{selection (distinct)} \]
| \[ \text{select } v_1, \ldots, v_k \ Q \quad \text{selection (with duplicates)} \]
| \[ \text{empty } v_1, \ldots, v_k \quad \text{empty set} \]
| \[ Q_1 \cup \text{all } Q_2 \quad \text{concatenation (union-compatible)} \]

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

\[ \Rightarrow \text{union arises from the SCHEMA} \]
Handling OR in Schema

- additional expansion rule:
  
  **Chase Step**

  Replace “\((D \text{ or } E) \times\)” with
  
  “\(\text{elim } x (D \times \text{ union all } E \times)\)”

  
  and rules for handling duplicates:

  **Duplicates and Union Step**

  “\(\text{elim } V (Q1 \text{ union all } Q2)\)” rewrites to
  
  “\((\text{elim } V Q1) \text{ union all } (\text{elim } V Q2)\)”
  
  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 \textit{WATEMP} and \textit{TOKYOEMP}.

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

- Plan: \textit{“select eid (WATEMP e, eid=e.Eid) union all
      (TOKYOEMP e, eid=e.Eid)”}
Example: Horizontal Partition

Physical design: two disjoint indices for \texttt{WATEMP} and \texttt{TOKYOEMP}.

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

- Plan: \texttt{“select eid (WATEMP e, eid=e.Eid) union all}
  \texttt{(TOKYOEMP e, eid=e.Eid)”}
Example: Horizontal Partition

Physical design: two disjoint indices for \texttt{WATEMP} and \texttt{TOKYOEMP}.

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

- Plan: “select eid \texttt{WATEMP} e, eid=e.Eid) union all
     \texttt{TOKYOEMP} e, eid=e.Eid)”
Example: Horizontal Partition

Physical design: two disjoint indices for \texttt{WATEMP} and \texttt{TOKYOEMP}.

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

- Plan: \texttt{“select eid (WATEMP e, eid=e.Eid) union all (TOKYOEMP e, eid=e.Eid)”}
Example: Horizontal Partition

Physical design: two disjoint indices for \texttt{WATEMP} and \texttt{TOKYOEMP}.

- **Expansion of** “select eid from \texttt{EMPLOYEE} e, eid=e.Eid”
  1. “select eid from \texttt{EMPLOYEE} e,
     (WATEMP or TOKYOEMP) e, eid=e.Eid”
  2. “select eid from \texttt{EMPLOYEE} e,
     (elim e (WATEMP e union all TOKYOEMP e)),
     eid=e.Eid”
  3. “select eid from \texttt{EMPLOYEE} e,
     (elim e,eid (WATEMP e union all TOKYOEMP e),
     eid=e.Eid)”
  4. “select eid from \texttt{EMPLOYEE} e,
     (elim e,eid WATEMP e, eid=E.eid) union all
     (elim e,eid TOKYOEMP e, eid=e.Eid)”
  5. “select eid from \texttt{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 \textbf{WATEMP} and \textbf{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 \textit{BEST} approach known today that . . .

- handles duplicates, and
- accommodates binding patterns

. . . in practice commonly competitive with hand-written C code
. . . extensions to \textit{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