Fundamentals of Physical Design: Constraints and Indices

David Toman

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

Recap of “State of Art”

- Current practice:

  Close coupling between Logical and Physical Schemata:

  $\Rightarrow$ physical design = logical schema revision + indices

  $\Rightarrow$ makes query optimization “easy”

  Logical schema revision $\Rightarrow$ changes in application DML (BAD)

- Alternative:

  Lose coupling supported by complex query optimization

  $\Rightarrow$ must support a wide variety of physical designs

Uniform Approach to Conceptual and Physical Design

DESIDERATA

Design a small number of primitives that support

- Conceptual/Logical schema development (including ICs)
- Physical schema development
- Linkage between the above two schemata

1. uniform DDL for both conceptual/logical and physical objects
2. capabilities (index) declarations for physical objects
3. integrity constraints to establish links between objects
4. no built-in assumptions (e.g., 2-level store)

Uniform Approach to Conceptual and Physical Design

DESIDERATA

Design a small number of primitives that support

- Conceptual/Logical schema development (including ICs)
- Physical schema development
- Linkage between the above two schemata

1. uniform DDL for both conceptual/logical and physical objects
2. capabilities (index) declarations for physical objects
3. integrity constraints to establish links between objects
4. no built-in assumptions (e.g., 2-level store)

Uniform Approach (cont.)

PLAN

The complete design is defined in terms of:

Integrity Constraints:

- ⇒ attach *attributes* to *classes/tables*
- ⇒ define *keys* and *foreign keys*
- ⇒ define *class hierarchies* (and *coverage*)
- ⇒ links conceptual and physical classes/tables

Index Declarations:

- ⇒ declare tables that can be scanned (binding patterns)
- ⇒ attaches *costs* to scanning these

. . . from now: a simple OO-style class/attribute based data model.

Uniform Approach (cont.)

PLAN

The complete design is defined in terms of:

Integrity Constraints:

- ⇒ attach *attributes* to *classes/tables*
- ⇒ define *keys* and *foreign keys*
- ⇒ define *class hierarchies* (and *coverage*)
- ⇒ links conceptual and physical classes/tables

Index Declarations:

- ⇒ declare tables that can be scanned (binding patterns)
- ⇒ attaches *costs* to scanning these

... from now: a simple OO-style class/attribute based data model.

Integrity Constraints in Description Logic(s)

Description Logic Syntax

Attributes and Path Functions: (denote total functions)

`Pf ::= Id`	identity	$\lambda x.x$
`  | f.Pf`	composition	$Pf \circ f$

Concept Descriptions: (denote sets of objects)

`C ::= A`	primitive	$(A \subseteq \Delta)$
`  | C1 and C2`	intersection	$C_1 \cap C_2$
`  | not C`	complement	$\Delta - C$
`  | all Pf C`	path type	$\{x \mid Pf(x) \in C\}$
`  | Pf1 = Pf2`	equation	$\{x \mid Pf_1(x) = Pf_2(x)\}$
`  | C: Pf1,...,Pfk -> Pf`	path FD	

$$\{x \mid \forall y \in C. \bigwedge_{i=1}^{k}(Pf_i(x) = Pf_i(y)) \to (Pf(x) = Pf(y))\}$$

Constraints: `C1 < C2` (denotes *subset relation*; schema = set of these)

⇒ "`C1<C2`" a *first order sentence*: satisfiability, logical implication, . . .

Integrity Constraints in Description Logic(s)

Description Logic Syntax

Attributes and Path Functions: (denote total functions)

`Pf::= Id`	identity	$\lambda x.x$
`  | f.Pf`	composition	$Pf \circ f$

Concept Descriptions: (denote sets of objects)

`C::= A`	primitive	$(A \subseteq \Delta)$
`  | C1 and C2`	intersection	$C_1 \cap C_2$
`  | not C`	complement	$\Delta - C$
`  | all Pf C`	path type	$\{x \mid Pf(x) \in C\}$
`  | Pf1 = Pf2`	equation	$\{x \mid Pf_1(x) = Pf_2(x)\}$
`  | C: Pf1,...,Pfk -> Pf`	path FD	

$$\{x \mid \forall y \in C. \bigwedge_{i=1}^{k}(Pf_i(x) = Pf_i(y)) \rightarrow (Pf(x) = Pf(y))\}$$

Constraints: `C1 < C2` (denotes *subset relation*; schema = set of these)

⇒ "`C1<C2`" a *first order sentence*: satisfiability, logical implication, ...

DDL in DL Examples

Example (Department and Employee Tables)

```
  EMPLOYEE < (all Eid INT) and
             (all Name STRING) and
             (all Dept DEPARTMENT)
DEPARTMENT < (all City STRING) and
             (all Boss EMPLOYEE)
```

Example (Department and Employee Keys)

```
  EMPLOYEE < (EMPLOYEE: Eid -> Id)
DEPARTMENT < (DEPARTMENT: Boss.Eid -> Id)
  EMPLOYEE < (not (Dept.Boss = Id))
```

DDL in DL Examples

Example (Department and Employee Tables)

```
  EMPLOYEE < (all Eid INT) and
             (all Name STRING) and
             (all Dept DEPARTMENT)
DEPARTMENT < (all City STRING) and
             (all Boss EMPLOYEE)
```

Example (Department and Employee Keys)

```
  EMPLOYEE < (EMPLOYEE: Eid -> Id)
DEPARTMENT < (DEPARTMENT: Boss.Eid -> Id)
  EMPLOYEE < (not (Dept.Boss = Id))
```

Views via Integrity Constraints

Example (Employees Views)

```
  WATEMP < EMPLOYEE and
           (Dept.City = 'Waterloo')

           EMPLOYEE and
           (Dept.City = 'Waterloo') < WATEMP

TOKYOEMP < EMPLOYEE and
           (Dept.City = 'Tokyo')    < TOKYOEMP
```

Example (Coverage and Disjointness Constraints)

```
           EMPLOYEE < WATEMP or TOKYOEMP
WATEMP and TOKYOEMP < BOTTOM
```

Views via Integrity Constraints

Example (Employees Views)

```
   WATEMP < EMPLOYEE and
            (Dept.City = 'Waterloo')

            EMPLOYEE and
            (Dept.City = 'Waterloo') < WATEMP

 TOKYOEMP < EMPLOYEE and
            (Dept.City = 'Tokyo')    < TOKYOEMP
```

Example (Coverage and Disjointness Constraints)

```
            EMPLOYEE < WATEMP or TOKYOEMP
 WATEMP and TOKYOEMP < BOTTOM
```

Index Declarations

IDEA: use generalized binding patterns

A *extra-logical* declaration of the form

```
index A (Pf1, ..., Pfm) (Pf1', ..., PFn')
```

where

- `A` is the (primitive) class whose objects are indexed,
- `(Pf1, ..., Pfm)` are the *input parameters*, and
- `(Pf1', ..., Pfn')` are the *outputs*

in addition costs of getting the *first* and the *next* object
(details skipped in this presentation).

$\Rightarrow$ each index declaration has an associated “iterator”.

Example: Addresses and Field Extraction

Example (Employee Table...)

```
EMPLOYEE < (all Eid INT) and
           (all Name STRING) and
           (all Dept DEPARTMENT)
```

Example (...as an array of pointers to structs)

```
EMPLOYEE < EARRAY < EMPLOYEE and (all Addr ADDR)
index EARRAY (Eid) (Addr)

EMPLOYEE < ENAME < EMPLOYEE
index ENAME (Addr) (Name)

EMPLOYEE < EDEPT < EMPLOYEE
index EDEPT (Addr) (Dept.Boss.Eid)
```

Example: 2-level Storage

Example (Department Table...)

```
DEPARTMENT < (all City STRING) and
             (all Boss EMPLOYEE)
```

Example (...as a file of pages)

```
(all PgRef DPAGES) <
DEPARTMENT < DRECS < DEPARTNENT and
             (all PgRef DPAGES) and (all Addr ADDR)
DPAGES < (all Addr ADDR)

index DPAGES () (Addr)                  ; expensive
index DRECS  (PgRef.Addr) (Addr)        ; cheap
```

⇒ now we can distinguish cost of "page access" v.s. "record access"

Example: Clustered/Unclustered Index Access

Example (Clustered index on Employee(Dept))

```
EMPLOYEE or (all PgRef CLUST) < EPAGE < EMPLOYEE
      and (all PgRef CLUST) and (Dept = PgRef.Dept)
index EPAGE (PgRef.Addr) (Addr)
index CLUST (Dept) (Addr)
```

Example (Un-Clustered index on Employee(Name))

```
EMPLOYEE or (all PgRef EPAGES) < EPAGE < EMPLOYEE
 and (all PgRef EPAGES) and (all CRef UNCLUST)
 and (PgRef.PgId = Cref.PgId) and (Name = CRef.Name)
UNCLUST < (all PgId EPAGES)

index EPAGE (PgRef.Addr) (Addr)
index EPAGES (PgId) (Addr)
index UNCLUST (Name) (PgId)
```

Example: Denormalization

Example (EMPDEPT denormalization)

```
 EMPDEPT < (all Eid INT) and (all Name STRING) and
           (all City STRING) (all Boss EMPDEPT)

EMPLOYEE < (all De EMPDEPT) and (EMPLOYEE: De->Id)
       and (Eid = De.Eid) and (Name = De.Name)
       and (Dept.City = De.City)
       and (Dept.Boss = De.Boss)

index EMPDEPT () (Eid,Name,City,Boss)
```

What happens to DEPT?

1. no additional info ⇒ we need a separate table (or NULLs)
2. every dept has an employee ⇒ additional constraints

Example: Denormalization

Example (EMPDEPT denormalization)

```
 EMPDEPT < (all Eid INT) and (all Name STRING) and
           (all City STRING) (all Boss EMPDEPT)

EMPLOYEE < (all De EMPDEPT) and (EMPLOYEE: De->Id)
       and (Eid = De.Eid) and (Name = De.Name)
       and (Dept.City = De.City)
       and (Dept.Boss = De.Boss)

index EMPDEPT () (Eid,Name,City,Boss)
```

What happens to `DEPT`?

1. no additional info ⇒ we need a separate table (or NULLs)
2. every dept has an employee ⇒ additional constraints

Other Idioms

- Horizontal Partitioning
  $\Rightarrow$ similar to `WATEMP`-`TOKYOEMP` example.
- Vertical Partitioning and (FK) Join Indices
  $\Rightarrow$ similar to *denormalization*
- Full Join Indices and Materialized Views
  $\Rightarrow$ depends on the expressive power of the constraints: ... needs full FOL
- ...

On the Power of Integrity Constraints

Highly Expressive Logics

⇒ First-order Logic (algebraic dependencies)
and extensions of FOL (fixpoints, ...)

⇒ Logical Implication undecidable

Decidable Logics

⇒ (certain) Description Logics

⇒ Logical Implication decidable

... to be combined with a decidable query language

⇒ Most features at modest cost

Weak Languages

⇒ status quo ($\sim$ projections of "base relations")

⇒ efficient but unable to cope with data independence

Summary

Take Home Message(s):

1. Integrity constraints *are key* to realizing the promise of *physical data independence*, and
2. Most of *physical design* issues (including appropriate costs) can be captured in such a framework.

To be solved ...

How to *optimize queries*?

$\Rightarrow$ Constraints $\sim$ (first-order) theories ($\mathcal{T}$)

$\Rightarrow$ Queries $\sim$ (first-order) formulae (Q)

$\Rightarrow$ Plans $\sim$ (first-order) formulae (of certain shape, P)

$$\mathcal{T} \models \forall \bar{x}.(Q \leftrightarrow P)$$

... additional issues: *database trimmings* (e.g., duplicates, etc.)

Summary

Take Home Message(s):

1. Integrity constraints *are key* to realizing the promise of *physical data independence*, and
2. Most of *physical design* issues (including appropriate costs) can be captured in such a framework.

To be solved ...

How to *optimize queries*?

⇒ Constraints ∼ (first-order) theories ($\mathcal{T}$)
⇒ Queries ∼ (first-order) formulae (Q)
⇒ Plans ∼ (first-order) formulae (of certain shape, P)

$\mathcal{T} \models \forall \bar{x}.(Q \leftrightarrow P)$ ... but how do we ... P???

... additional issues: *database trimmings* (e.g., duplicates, etc.)

Summary

Take Home Message(s):

1. Integrity constraints *are key* to realizing the promise of *physical data independence*, and
2. Most of *physical design* issues (including appropriate costs) can be captured in such a framework.

To be solved ...

How to *optimize queries*?

⇒ Constraints ∼ (first-order) theories ($\mathcal{T}$)
⇒ Queries ∼ (first-order) formulae (Q)
⇒ Plans ∼ (first-order) formulae (of certain shape, P)

$$\mathcal{T} \models \forall \bar{x}.(Q \leftrightarrow P)$$ ... but how do we *find* P???

... additional issues: *database trimmings* (e.g., duplicates, etc.)

Summary

Take Home Message(s):

1. Integrity constraints *are key* to realizing the promise of *physical data independence*, and
2. Most of *physical design* issues (including appropriate costs) can be captured in such a framework.

To be solved ...

How to *optimize queries*?

⇒ Constraints ∼ (first-order) theories ($\mathcal{T}$)
⇒ Queries ∼ (first-order) formulae (Q)
⇒ Plans ∼ (first-order) formulae (of certain shape, P)

$$\mathcal{T} \models \forall \bar{x}.(Q \leftrightarrow P)$$ ... but how do we find P???

... additional issues: *database trimmings* (e.g., duplicates, etc.)

Summary

Take Home Message(s):

1. Integrity constraints *are key* to realizing the promise of *physical data independence*, and
2. Most of *physical design* issues (including appropriate costs) can be captured in such a framework.

To be solved ...

How to *optimize queries*?

$\Rightarrow$ Constraints $\sim$ (first-order) theories ($\mathcal{T}$)
$\Rightarrow$ Queries $\sim$ (first-order) formulae (Q)
$\Rightarrow$ Plans $\sim$ (first-order) formulae (of certain shape, P)

$\mathcal{T} \models \forall \bar{x}.(Q \leftrightarrow P)$ ... but how do we find P???

... additional issues: *database trimmings* (e.g., duplicates, etc.)