Topics in Database Systems: Modern DBMS

CS848 Spring 2022

David Toman

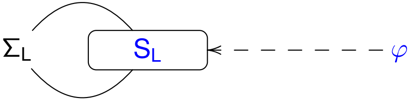
Basic Designs

(and an Overview of Standard Techniques)

Big Picture

Definability and Rewriting

Queries	range-restricted FOL (a.k.a. SQL)
Schema	range-restricted FOL $\Sigma := \Sigma^L \cup \Sigma^{LP} \cup \Sigma^P$
Data	CWA (complete information)

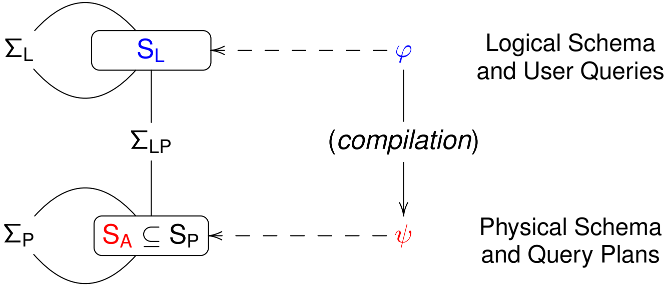


Logical Schema
and User Queries

Big Picture

Definability and Rewriting

Queries	range-restricted FOL over S_L *definable* w.r.t. Σ and S_A
Schema	range-restricted FOL $\Sigma := \Sigma^L \cup \Sigma^{LP} \cup \Sigma^P$
Data	CWA (complete information for S_A symbols)



[Borgida, de Bruijn, Franconi, Seylan, Straccia, Toman, Weddell: On Finding Query Rewritings under Expressive Constraints. SEBD 2010: 426-437]

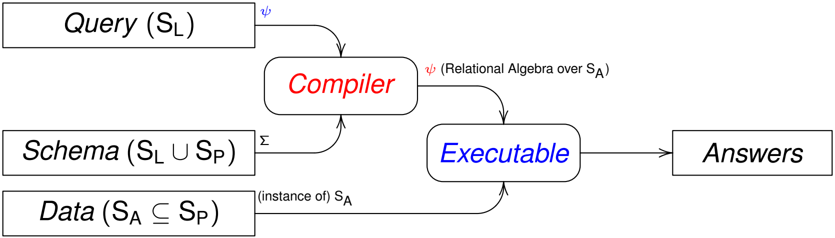
Big Picture

Definability and Rewriting

Queries	range-restricted FOL	over S_L *definable* w.r.t. Σ and S_A
Schema	range-restricted FOL	$\Sigma := \Sigma^L \cup \Sigma^{LP} \cup \Sigma^P$
Data	CWA (complete information for S_A symbols)	

- to users it looks like a *single model* (of the logical schema)
- implementation can pick from many models

  but *definable* queries answer the same in each of them



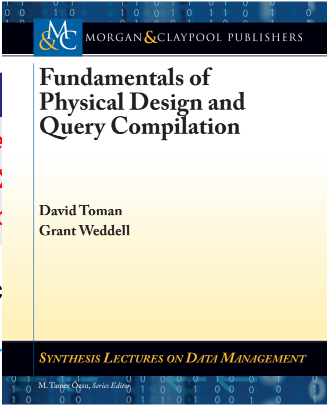
Big Picture

Definability and Rewriting

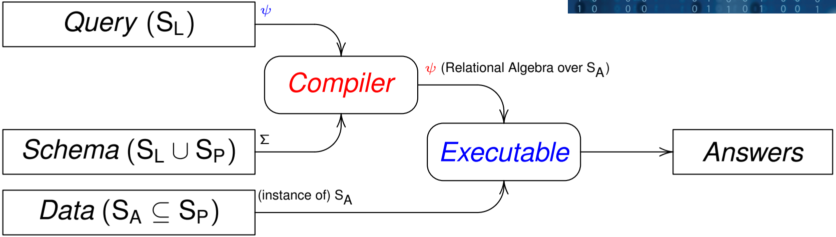
Queries	range-restricted FOL over S_L *definable*
Schema	range-restricted FOL $\Sigma := \Sigma^L \cup \Sigma^{LP} \cup$
Data	CWA (complete information for S_A symb

- to users it looks like a *single model* (of the logic
- implementation can pick from many models

but *definable* queries answer



©2011



QUERY COMPILATION

PART I: PLANS AS FORMULAE AND STANDARD DESIGN

Queries over a Physical Design

Issues to resolve (today)

- What "formulas" do qualify as *plans*?

  $\Rightarrow$ how do we interpret *logical connectives* as programs?
- Why do the *plans* implement the *user queries*?
- Are all (desired) *plans* captured by appropriate formulas?

Outline

1. Iterator Protocols to communicate Sets
2. Atomic Plan Operations: Access Paths
3. Logical Connectives/Quantifiers as Plan Operators
4. Beyond Logical Operators: Dealing with Duplicates (not today)

Creating Table(s) and Base File(s)

Specification:

```
[
% constraints
table(x,y,z) <-> ex(r,basetable(r,x,y,z)),

% query
q(x,y,z) <-> table(x,y,z)
].
```

Notes:

- access path: `basetable/4/0`;
- additional `r` attribute: address in physical storage

Query Plan:

```
q(v0,v1,v2) <-> ex(v3,basetable(v3,v0,v1,v2))
```

Creating Table(s) and Base File(s)

Specification:

```
[
% constraints
table(x,y,z) <-> ex(r,basetable(r,x,y,z)),

% query
q(x,y,z) <-> table(x,y,z)
].
```

Notes:

- access path: `basetable/4/0`;
- additional `r` attribute: address in physical storage

Query Plan:

```
q(v0,v1,v2) <-> ex(v3,basetable(v3,v0,v1,v2))
```

Access Path Code Templates

Array of records (C-structs)

Pseudo-code templates realizing a `first/next` protocol:

```
function basetable-first()
  i := 0
  return basetable-next

function basetable-next()
  if (i ≥ N) return false
  x := btarray[i].xname
  y := btarray[i].yname
  z := btarray[i].zname
  r := i++;
  return true
```

$\Rightarrow$ assuming `struct { int xname, yname, zname } btarray[N]`

$\Rightarrow$ variable *i* renamed for each occurrence of `basetable` in a plan.

Global state records bindings of (possible copies of) variables:

1. *x*, *y* and *z* to communicate the contents of `btarray`;
2. *i* (and *N*) record scanning status (and size) of `btarray`

Note: AP code (templates) for access paths must be provided.

Access Path Code Templates

Array of records (C-structs)

Pseudo-code templates realizing a `first/next` protocol:

```
function basetable-first()
  i := 0
  return basetable-next

function basetable-next()
  if (i ≥ N) return false
  x := btarray[i].xname
  y := btarray[i].yname
  z := btarray[i].zname
  r := i++;
  return true
```

$\Rightarrow$ assuming `struct { int xname, yname, zname } btarray[N]`

$\Rightarrow$ variable i renamed for each occurrence of `basetable` in a plan.

Global state records bindings of (possible copies of) variables.

1. x, y and z to communicate the contents of `btarray`.
2. i (and N) record scanning status (and size) of `btarray`.

Note: AP code (templates) for access paths must be provided

Access Path Code Templates

Array of records (C-structs)

Pseudo-code templates realizing a `first`/`next` protocol:

```
function basetable-first()
  i := 0
  return basetable-next
```

```
function basetable-next()
  if (i ≥ N) return false
  x := btarray[i].xname
  y := btarray[i].yname
  z := btarray[i].zname
  r := i++;
  return true
```

⇒ assuming `struct { int xname, yname, zname } btarray[N]`

⇒ variable i renamed for each occurrence of `basetable` in a plan.

Global state records bindings of (possible copies of) variables.

1. x, y and z to communicate the contents of `btarray`.
2. i (and N) record scanning status (and size) of `btarray`.

Note: AP code (templates) for access paths must be provided.

(More Esoteric) Access Paths

1. Built-in "operations":
   - arithmetic (`plus`/3/2, `times`/3/2, etc.)
   - string manipulation (`concat`/3/2, `substr`/4/3, etc.)
   - ...
2. data type tests (`is-integer`/1/1)
3. pointer dereference and field extraction from records
4. (page) reads from external storage
5. ...

Conjunctive Query Plans: Semantics

```
function ($Q_1 \wedge Q_2$)-first
   if not $Q_1$-first return false
   while not $Q_2$-first do
      if not $Q_1$-next return false
   return true
```

```
function ($Q_1 \wedge Q_2$)-next
   if $Q_2$-next return true
   while $Q_1$-next do
      if $Q_2$-first return true
   return false
```

```
function ($\exists x.Q_1$)-first
  return $Q_1$-first
```

```
function ($\exists x.Q_1$)-next
  return $Q_1$-next
```

```
function {$Q_1$}-first
   if not exists store $S$
      create $S$
   if $Q_1$-first
      empty $S$
      add $(x_1, \ldots, x_n)$ to $S$
      return true
   return false
```

```
function {$Q_1$}-next
   while $Q_1$-next do
      if not $(x_1, \ldots, x_n) \in S$
         add $(x_1, \ldots, x_n)$ to $S$
         return true
   return false
```

Conjunctive Query Plans: Semantics

```
function (Q1 ∧ Q2)-first
  if not Q1-first return false
  while not Q2-first do
    if not Q1-next return false
  return true
```

```
function (Q1 ∧ Q2)-next
  if Q2-next return true
  while Q1-next do
    if Q2-first return true
  return false
```

```
function (∃x.Q1)-first
  return Q1-first
```

```
function (∃x.Q1)-next
  return Q1-next
```

```
function {Q1}-first
  if not exists store S
    create S
  if Q1-first
    empty S
    add ⟨x1,...,xn⟩ to S
    return true
  return false
```

```
function {Q1}-next
  while Q1-next do
    if not ⟨x1,...,xn⟩ ∈ S
      add ⟨x1,...,xn⟩ to S
      return true
  return false
```

General Query Plans: Syntax

```
function (Q1 ∨ Q2)-first
  (Q1 ∨ Q2)-flag := true
  if Q1-first return true
  (Q1 ∨ Q2)-flag := false
  return Q2-first
```

```
function (Q1 ∨ Q2)-next
  if (Q1 ∨ Q2)-flag
    if Q1-next return true
  (Q1 ∨ Q2)-flag := false
  return Q2-next
```

```
function (¬Q1)-first
  if Q1-first return false
  return true
```

```
function (¬Q1)-next
  return false
```

What's Missing?

1. binding patterns (a.k.a. usage restrictions on access paths)
2. dealing with extra-logical phenomena: duplicates/ordering
3. cost model

. . . we touch on many of these in subsequent lectures

What's Missing?

1. binding patterns (a.k.a. usage restrictions on access paths)
2. dealing with extra-logical phenomena: duplicates/ordering
3. cost model

. . . we touch on many of these in subsequent lectures

Adding an (search) index

What about `create index indexx(indexyy)` on x(y) in `table`?

Specification:

```
[
% constraints
...
indexx(x,r) <-> ex([y,z],basetable(r,x,y,z)),
indexy(y,r) <-> ex([x,z],basetable(r,x,y,z)),
baselookup(r,x,y,z) <-> basetable(r,x,y,z),

% query
q(x,y) <-> ex([z,v,w],
                table(x,v,z) and table(z,w,y))
].
```

Notes:

- access paths: `baselookup/4/1`, `indexx`, `indexy/2/1`;

Adding an (search) index (cont)

```
q(x,y) <-> ex([z,v,w],
               table(x,v,z) and table(z,w,y))
```

Possible plans:

Table Scans:

```
q(v11,v12) <-> ex([v13,v14,v15,v16,v17],
               basetable(v13,v11,v14,v15) and
               basetable(v16,v15,v17,v12))
```

Index lookup:

```
q(v11,v12) <-> ex([v13,v14,v15,v16,v17],
               basetable(v13,v11,v14,v15) and
               indexx(v15,v16) and
               baselookup(v16,v17,v18,v12))
```

Adding an (search) index (cont)

```
q(x,y) <-> ex([z,v,w],
              table(x,v,z) and table(z,w,y))
```

Possible plans:

Table Scans:

```
q(vl1,vl2) <-> ex([vl3,vl4,vl5,vl6,vl7],
               basetable(vl3,vl1,vl4,vl5) and
               basetable(vl6,vl5,vl7,vl2))
```

Index lookup:

```
q(vl1,vl2) <-> ex([vl3,vl4,vl5,vl6,vl7],
               basetable(vl3,vl1,vl4,vl5) and
               indexx(vl5,vl6) and
               baselookup(vl6,vl7,vl8,vl2))
```

Adding an (search) index (cont)

```
q(x,y) <-> ex([z,v,w],
              table(x,v,z) and table(z,w,y))
```

Possible plans:

Table Scans:

```
q(vl1,vl2) <-> ex([vl3,vl4,vl5,vl6,vl7],
               basetable(vl3,vl1,vl4,vl5) and
               basetable(vl6,vl5,vl7,vl2))
```

Index lookup:

```
q(vl1,vl2) <-> ex([vl3,vl4,vl5,vl6,vl7],
               basetable(vl3,vl1,vl4,vl5) and
               indexx(vl5,vl6) and
               baselookup(vl6,vl7,vl8,vl2))
```

Adding an (search) index (cont)

```
q(x,y) <-> table(x,x,y)
```

(with a parameter x)

Possible plans:

Index lookup:

```
q(v11,v12) <-> ex([v13,v14],
                  indexx(v11,v13) and
                  baselookup(v13,v14,v14,v12))
```

Index intersection:

```
q(v11,v12) <-> ex([v13,v14,v15],
                  indexx(v11,v13) and
                  indexy(v11,v13) and
                  baselookup(v13,v14,v15,v12))
```

Adding an (search) index (cont)

```
q(x,y) <-> table(x,x,y)
```

(with a parameter x)

Possible plans:

Index lookup:

```
q(vl1,vl2) <-> ex([vl3,vl4],
                  indexx(vl1,vl3) and
                  baselookup(vl3,vl4,vl4,vl2))
```

Index intersection:

```
q(vl1,vl2) <-> ex([vl3,vl4,vl5],
                  indexx(vl1,vl3) and
                  indexy(vl1,vl3) and
                  baselookup(vl3,vl4,vl5,vl2))
```

Adding an (search) index (cont)

```
q(x,y) <-> table(x,x,y)
```

(with a parameter x)

Possible plans:

Index lookup:

```
q(vl1,vl2) <-> ex([vl3,vl4],
                  indexx(vl1,vl3) and
                  baselookup(vl3,vl4,vl4,vl2))
```

Index intersection:

```
q(vl1,vl2) <-> ex([vl3,vl4,vl5],
                  indexx(vl1,vl3) and
                  indexy(vl1,vl3) and
                  baselookup(vl3,vl4,vl5,vl2))
```

Index-only Plans

```
q(x) <-> ex(y,table(x,x,y))
```

(with a parameter x)

Possible plans:

Index lookup:

```
q(v11) <-> ex([v12,v13,v14],
               indexx(v11,v13) and
               baselookup(v13,v14,v14,v12))
```

Index intersection:

```
q(v11) <-> ex([v12,v13,v14,v15],
               indexx(v11,v13) and
               indexy(v11,v13))
```

Index-only Plans

```
q(x) <-> ex(y,table(x,x,y))
```

(with a parameter x)

Possible plans:

Index lookup:

```
q(vl1) <-> ex([vl2,vl3,vl4],
              indexx(vl1,vl3) and
              baselookup(vl3,vl4,vl4,vl2))
```

Index intersection:

```
q(vl1) <-> ex([vl2,vl3,vl4,vl5],
              indexx(vl1,vl3) and
              indexy(vl1,vl3))
```

Index-only Plans

```
q(x) <-> ex(y,table(x,x,y))
```

(with a parameter x)

Possible plans:

Index lookup:

```
q(vl1) <-> ex([vl2,vl3,vl4],
              indexx(vl1,vl3) and
              baselookup(vl3,vl4,vl4,vl2))
```

Index intersection:

```
q(vl1) <-> ex([vl2,vl3,vl4,vl5],
              indexx(vl1,vl3) and
              indexy(vl1,vl3))
```

Column Store

Specification:

```
[
table(x,y,z) <-> ex(r,basetable(r,x,y,z)),

% indices
columnx(r,x) <-> ex([y,z],basetable(r,x,y,z)),
columny(r,y) <-> ex([x,z],basetable(r,x,y,z)),
columnz(r,z) <-> ex([x,y],basetable(r,x,y,z)),
% keys
basetable(r,x1,y1,z1) and basetable(r,x2,y2,z2)
                  ->(x1=x2 and y1=y2 and z1=z2),
% query
q(x,y,z) <-> table(x,y,z)
].
```

Notes:

- APs: `columnx/2/0`, `columny/2/0`, and `columnz/2/0`;
- the key constraint is necessary (why?)

Horizontal Partition (sharding)

Specification:
```
[...
  % horizontal partitions
  hpp1(r,x,y,z) -> basetable(r,x,y,z),
  hpp2(r,x,y,z) -> basetable(r,x,y,z),
  hpp3(r,x,y,z) -> basetable(r,x,y,z),
  basetable(r,x,y,z) -> (hpp1(r,x,y,z) or
                hpp2(r,x,y,z) or hpp3(r,x,y,z)),

...].
```

Notes:

- APs: `hpp1/4/0`, `hpp2/4/0`, and `hpp3/4/0`;
- do we need “disjointness” of the partitions?

Subclass/Complement

Specification: [...

```
% superclass and coverage
basetable(r,x,y,z) -> super(r,x,y,z),
complement(r,x,y,z) -> super(r,x,y,z),
super(r,x,y,z) -> (complement(r,x,y,z)
                       or basetable(r,x,y,z)),

% disjointness
complement(r,x,y,z) and basetable(r,x,y,z)
                                        -> bot,

...].
```

Notes:

- do we need “disjointness”? “keys”?

QUERY COMPILATION

PART II: WHAT CAN IT DO?

What can this do?

GOAL

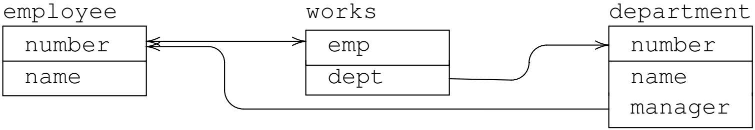
Generate query plans *that compete with hand-written programs in C*

1. linked data structures, pointers, . . .
2. access to search structures (index access and selection),
3. hash-based access to data (including hash-joins),
4. multi-level storage (aka disk/remote/distributed files), . . .
5. materialized views (FO-definable),
6. updates through logical schema (needs *id invention*!), . . .

. . . all without having to code (too much) in C/C++ !

Lists and Pointers (example)

1. Logical Schema



$\Rightarrow$ we merge `works` into `employee` as a `dept` attribute (to simplify)

2. Physical Design: a *linked list of* `emp` *records pointing to* `dept` *records.*

```
record emp of                     record dept of
       integer     num                   integer     num
       string      name                  string      name
       reference   dept                  reference   manager
```

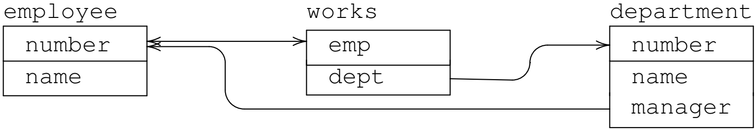
$\Rightarrow$ main difference: pointers rather than primary key-based foreign keys

Exercise

Modify the rest of the development to account for the `works` table.

Lists and Pointers (example)

1. Logical Schema



⇒ we merge `works` into `employee` as a `dept` attribute (to simplify)

2. Physical Design: a *linked list of `emp` records pointing to `dept` records*.

```
record emp of                      record dept of
       integer     num                    integer     num
       string      name                   string      name
       reference   dept                   reference   manager
```

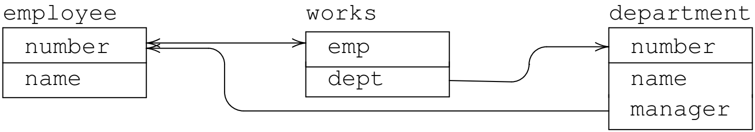
⇒ main difference: pointers rather than prinary key-based foreign keys

Exercise

Modify the rest of the development to account for the `works` table.

Lists and Pointers (example)

1. Logical Schema



⇒ we merge `works` into `employee` as a `dept` attribute (to simplify)

2. Physical Design: a *linked list of `emp` records pointing to `dept` records*.

```
record emp of                     record dept of
       integer    num                    integer    num
       string     name                   string     name
       reference  dept                   reference  manager
```

⇒ main difference: pointers rather than prinary key-based foreign keys

Exercise:

Modify the rest of the development to account for the `works` table.

Lists and Pointers (record declarations)

```
% record layout of emp and dept records and fields for:
%
%   struct emp  { int num, char[20] name, struct dept* dept };
%   struct dept { int num, char[20] name, struct mgr* emp };
%
%    ea/da addresses of emp/dept records
%    access paths: ea/1/0  (linked list of employee records),
%                  ea_num, ea_name, ea_dept, da_num, da_name,
%                  da_mgr/2/1 (field extractors "->" in C)
%    all attributes functional, "num" is a key;
%                   "dept" and "mgr" are pointers;
%
ea(e) -> ex(y,ea_num(e,y)),  ea_num(e,y) and ea_num(e,z)-> y=z,
                             ea_num(y,x) and ea_num(z,x)-> y=z,
ea(e) -> ex(y,ea_name(e,y)), ea_name(e,y) and ea_name(e,z)-> y=z,
ea(e) -> ex(y,ea_dept(e,y)), ea_dept(e,y) and ea_dept(e,z)-> y=z,
                             ea_dept(e,d) -> da(d),
```

. . . and the same for `da` et al.

Lists and Pointers (logical tables)

```
%
% user predicates over records
%
employee(x,y,z) <-> ex(e,baseemployee(e,x,y,z)), % record addr
%
ea(e)             <-> ex([x,y,z],baseemployee(e,x,y,z)),
ea_num(e,x)       <-> ex([y,z],baseemployee(e,x,y,z)),
ea_name(e,y)      <-> ex([x,z],baseemployee(e,x,y,z)),
ex(d,ea_dept(e,d) and da_num(d,z))
                  <-> ex([x,y],baseemployee(e,x,y,z)),
```

…and the same for `department` (we merged `works` into `employee`).

```
%
% business logic: managers work for their own departments
%
% employee(x,y,z) and department(u,v,x)-> z=u
da_mgr(x,e) and ea_dept(e,y) -> x=y    % pointer-based version
```

What can this do: navigating pointers

1. List all employee numbers and names ($\exists z.\texttt{employee}(x, y, z)$):

   $$\exists a.\texttt{ea}(a) \land \texttt{ea-num}(a, x) \land \texttt{ea-name}(a, y)$$

2. List all department numbers with their manager names

   ($\exists z, u, v.\texttt{department}(x, z, u) \land \texttt{employee}(u, y, v)$):

What can this do: navigating pointers

1. List all employee numbers and names ($\exists z.\texttt{employee}(x, y, z)$):

$$\exists a.\texttt{ea}(a) \wedge \texttt{ea-num}(a, x) \wedge \texttt{ea-name}(a, y)$$

or, in C-like syntax:

```
for a in ea do
    x := a->num;
    y := a->name;
```

2. List all department numbers with their manager names ($\exists z, u, v.\texttt{department}(x, z, u) \wedge \texttt{employee}(u, y, v)$):

What can this do: navigating pointers

1. List all employee numbers and names ($\exists z.\texttt{employee}(x, y, z)$):

$$\exists a.\texttt{ea}(a) \wedge \texttt{ea-num}(a, x) \wedge \texttt{ea-name}(a, y)$$

2. List all department numbers with their manager names ($\exists z, u, v.\texttt{department}(x, z, u) \wedge \texttt{employee}(u, y, v)$):

$$.\texttt{ea}(e) \wedge \texttt{ea-dept}(e, d) \wedge \texttt{da-num}(d, x) \wedge \texttt{da-mgr}(d, f) \wedge \texttt{ea-name}(f, y)$$

$\Rightarrow$ needs "departments have at least one employee".

…needs *duplicate elimination* during projection.

$$.\texttt{ea}(e) \wedge \texttt{ea-dept}(e, d) \wedge \texttt{ea-name}(e, y) \wedge \texttt{da-num}(d, x) \wedge \texttt{da-mgr}(d, f) \wedge \texttt{compare}(e, f)$$

$\Rightarrow$ needs "managers work in their own departments".

…NO *duplicate elimination* during projection.

What can this do: navigating pointers

1. List all employee numbers and names ($\exists z.\texttt{employee}(x, y, z)$):

$$\exists a.\texttt{ea}(a) \wedge \texttt{ea-num}(a, x) \wedge \texttt{ea-name}(a, y)$$

2. List all department numbers with their manager names ($\exists z, u, v.\texttt{department}(x, z, u) \wedge \texttt{employee}(u, y, v)$):

$$\exists e, d, f.\texttt{ea}(e) \wedge \texttt{ea-dept}(e, d) \wedge \texttt{da-num}(d, x) \wedge \texttt{da-mgr}(d, f) \wedge \texttt{ea-name}(f, y)$$

⇒ needs "departments have at least one employee".

...needs *duplicate elimination* during projection.

$$.\texttt{ea}(e) \wedge \texttt{ea-dept}(e, d) \wedge \texttt{ea-name}(e, y) \wedge \texttt{da-num}(d, x) \wedge \texttt{da-mgr}(d, f) \wedge \texttt{compare}(e, f)$$

⇒ needs "managers work in their own departments".

...NO *duplicate elimination* during projection.

What can this do: navigating pointers

1. List all employee numbers and names ($\exists z.\texttt{employee}(x, y, z)$):

$$\exists a.\texttt{ea}(a) \wedge \texttt{ea-num}(a, x) \wedge \texttt{ea-name}(a, y)$$

2. List all department numbers with their manager names ($\exists z, u, v.\texttt{department}(x, z, u) \wedge \texttt{employee}(u, y, v)$):

$$\exists e, d, f.\texttt{ea}(e) \wedge \texttt{ea-dept}(e, d) \wedge \texttt{da-num}(d, x) \wedge \texttt{da-mgr}(d, f) \wedge \texttt{ea-name}(f, y)$$

$\Rightarrow$ needs "departments have at least one employee".

... needs *duplicate elimination* during projection.

$$\exists e, d, f.\texttt{ea}(e) \wedge \texttt{ea-dept}(e, d) \wedge \texttt{ea-name}(e, y) \wedge \texttt{da-num}(d, x) \wedge \texttt{da-mgr}(d, f) \wedge \texttt{compare}(e, f)$$

$\Rightarrow$ needs "managers work in their own departments".

... NO duplicate elimination during projection.

What can this do: navigating pointers

1. List all employee numbers and names ($\exists z.\texttt{employee}(x, y, z)$):

$$\exists a.\texttt{ea}(a) \wedge \texttt{ea-num}(a, x) \wedge \texttt{ea-name}(a, y)$$

2. List all department numbers with their manager names ($\exists z, u, v.\texttt{department}(x, z, u) \wedge \texttt{employee}(u, y, v)$):

$$\exists e, d, f.\texttt{ea}(e) \wedge \texttt{ea-dept}(e, d) \wedge \texttt{da-num}(d, x) \wedge \texttt{da-mgr}(d, f) \wedge \texttt{ea-name}(f, y)$$

$\Rightarrow$ needs "departments have at least one employee".

... needs *duplicate elimination* during projection.

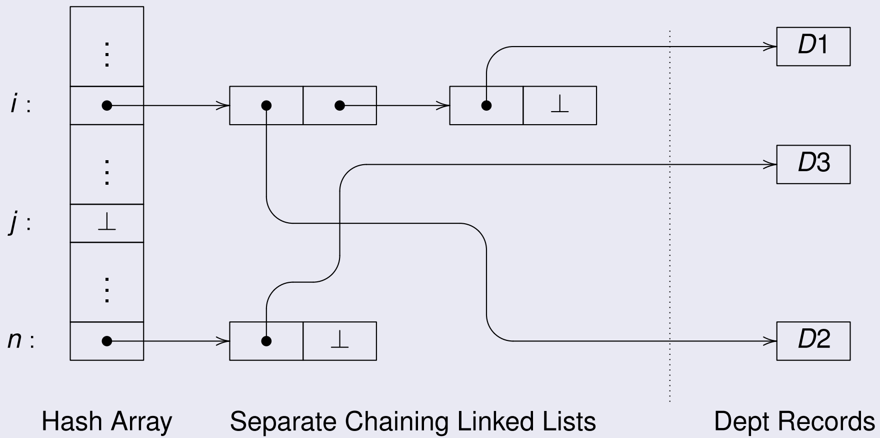
$$\exists e, d, f.\texttt{ea}(e) \wedge \texttt{ea-dept}(e, d) \wedge \texttt{ea-name}(e, y) \wedge \texttt{da-num}(d, x) \wedge \texttt{da-mgr}(d, f) \wedge \texttt{compare}(e, f)$$

$\Rightarrow$ needs "managers work in their own departments".

... NO *duplicate elimination* during projection.

What can it do: Hashing, Lists, et al.

Hash Index with (list-based) Separate Chaining



What can it do: Hashing, Lists, et al.

Hash Index on `department`'s `name`:

Access paths:

$S_A \supseteq \{\texttt{hash}/2/1, \texttt{hasharraylookup}/2/1, \texttt{listscan}/2/1\}.$

Physical Constraints:

$$\Sigma_{LP} \supseteq \{\forall x, y.((\texttt{deptfile}(x) \wedge \texttt{dept-name}(x, y)) \rightarrow \exists z, w.(\texttt{hash}(y, z) \wedge \texttt{hasharraylookup}(z, w) \wedge \texttt{listscan}(w, x))),$$
$$\forall x, y.(\texttt{hash}(x, y) \rightarrow \exists z.\texttt{hasharraylookup}(y, z)),$$
$$\forall x, y.(\texttt{listscan}(x, y) \rightarrow \texttt{deptfile}(y)) \}$$

Query:

$\exists y, z.(\texttt{department}(x_1, p, y) \wedge \texttt{employee}(y, x_2, z))\{p\}.$

```
ex(x6.Hash(p,x6) and ex(x5.(hasharraylookup(x6,x5)
      and ex(x4.listscan(x5,x4) and da-name(x4,p)
      and da-num(x4,x1) and ex(x3.(da-mgr(x4,x3)
                          and ea-name(x3,x2)))))))
```

What can it do: Hashing, Lists, et al.

Hash Index on `department`'s `name`:

Access paths:

$S_A \supseteq \{\texttt{hash}/2/1, \texttt{hasharraylookup}/2/1, \texttt{listscan}/2/1\}$.

Physical Constraints:

$$\Sigma_{LP} \supseteq \{\forall x, y.((\texttt{deptfile}(x) \wedge \texttt{dept-name}(x, y)) \rightarrow \exists z, w.(\texttt{hash}(y, z) \wedge \texttt{hasharraylookup}(z, w) \wedge \texttt{listscan}(w, x))),$$
$$\forall x, y.(\texttt{hash}(x, y) \rightarrow \exists z.\texttt{hasharraylookup}(y, z)),$$
$$\forall x, y.(\texttt{listscan}(x, y) \rightarrow \texttt{deptfile}(y)) \}$$

Query:

$\exists y, z.(\texttt{department}(x_1, p, y) \wedge \texttt{employee}(y, x_2, z))\{p\}$.

```
ex(x6.Hash(p,x6) and ex(x5.(hasharraylookup(x6,x5)
      and ex(x4.listscan(x5,x4) and da-name(x4,p)
      and da-num(x4,x1) and ex(x3.(da-mgr(x4,x3)
                            and ea-name(x3,x2)))))))
```

University of Waterloo

QUERY COMPILATION

PART III: CASE STUDY (TO THINK ABOUT . . .)

The LINUX-INFO System: A Case Study

GOAL:

to develop the LINUX-INFO system to monitor the operating systems deployed in their organization.

```
david@david-ryzen:/mnt/david/itb/itb2$ ps -efaux | head
USER   PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
root     2  0.0  0.0   0   0 ?   S    May07 0:00 [kthreadd]
root     3  0.0  0.0   0   0 ?   I<   May07 0:00  \_ [rcu_gp]
root     4  0.0  0.0   0   0 ?   I<   May07 0:00  \_ [rcu_par_gp]
root     6  0.0  0.0   0   0 ?   I<   May07 0:00  \_ [kworker/0:0H-
root     9  0.0  0.0   0   0 ?   I<   May07 0:00  \_ [mm_percpu_wq]
root    10  0.0  0.0   0   0 ?   S    May07 0:07  \_ [ksoftirqd/0]
root    11  0.0  0.0   0   0 ?   I    May07 5:31  \_ [rcu_sched]
root    12  0.0  0.0   0   0 ?   S    May07 0:01  \_ [migration/0]
...
```

LINUX-INFO System: Data and Metadata

Example of LINUX-INFO data important to APS.

1. process `gcc` is running
2. `gcc`'s process number is 1234.
3. the user running `gcc` is 145.
4. `gcc` uses file "foo.c"

Example of LINUX-INFO metadata specified by APS.

- There entities called `process` and `file`.
- There are attributes called `pno`, `pname`, `uname`, and `fname`.
- Each process entity has attributes `pno`, `pname` and `uname`.
- Each file entity has attribute `fname`.
- Processes are identified by their `pno`.
- Files are identified by their `fname`.
- There is a relationship `uses` between processes and files.

LINUX-INFO System: Data and Metadata

Example of LINUX-INFO data important to APS.

1. process `gcc` is running
2. `gcc`'s process number is 1234.
3. the user running `gcc` is 145.
4. `gcc` uses file "foo.c"

Example of LINUX-INFO metadata specified by APS.

4. There entities called `process` and `file`.
5. There are attributes called `pno`, `pname`, `uname`, and `fname`.
6. Each process entity has attributes `pno`, `pname` and `uname`.
7. Each file entity has attribute `fname`.
8. Processes are identified by their `pno`.
9. Files are identified by their `fname`.

There is a relationship `uses` between processes and files.

The LINUX System: Physical Design

A *physical design* for LINUX (selected by Linus Torvalds).

8. There are process records called `task-struct`.
9. Each `task-struct` record has record fields `pid`, `uid`, `comm`, and `file-struct`.
10. All `task-struct`s is organized as a tree data structure.
11. The `task-struct` records correspond one-to-one to `process` entities.
12. Record fields in `task-struct` encode the corresponding attribute values for `process` entities, for example, `pid` encodes an `pno`, etc.
13. Similarly, `fs`s correspond appropriately to (open) `file` entities.
14. `file-struct` field of `task-struct` is an array of `fd`s; an entry in this array indicates that the `process` corresponding to this `task-struct` is using the `file` represented by the `fd` record in the array.

LINUX-INFO System: Queries and Query Plans

A LINUX-INFO *user query* specified by APS.

14 Find the `files` used by `process` invoked by user 145.

A *query plan* selected by a *query compiler*.

14 Scan tree of `task-struct`s, for each check if its `uid` attribute is 145 and, if so scan the `file-struct array` in the `task-struct` and print out the names of files described by non-NULL file descriptors (`fd`).

Question:

Does the *physical design* allow APS to list all files known to the Linux system?

LINUX-INFO System: Queries and Query Plans

A LINUX-INFO *user query* specified by APS.

14 Find the `files` used by `process` invoked by user 145.

A *query plan* selected by a *query compiler*.

15 Scan tree of `task-struct`s, for each check if its `uid` attribute is 145 and, if so scan the `file-struct array` in the `task-struct` and print out the names of files described by non-NULL file descriptors (`fd`).

Question:

Does the *physical design* allow APS to list all files known to the Linux system?

LINUX-INFO System: Queries and Query Plans

A LINUX-INFO *user query* specified by APS.

14 Find the `files` used by `process` invoked by user 145.

A *query plan* selected by a *query compiler*.

15 Scan tree of `task-struct`s, for each check if its `uid` attribute is 145 and, if so scan the `file-struct array` in the `task-struct` and print out the names of files described by non-NULL file descriptors (`fd`).

Question:

Does the *physical design* allow APS to list all files known to the Linux system?

Take Home

Lots of open issues:

1. DB engine vs. Compilation aproaches
2. Main memory data organization

   ⇒ pointers and records accommodated *natively*
   ⇒ coded as combination of AP and physical tables
3. Data structures can be (commonly) decomposed to primitives (hash)
4. . . .

To try at Home

1. more query examples against employee-department schema
2. description of *LINUX-info* using constraints/APs

Project Idea(s)

- code generation from templates
  (e.g., `...as array` generates code similar to the code on s.7)

Take Home

Lots of open issues:

1. DB engine vs. Compilation aproaches
2. Main memory data organization

   $\Rightarrow$ pointers and records accommodated *natively*
   $\Rightarrow$ coded as combination of AP and physical tables
3. Data structures can be (commonly) decomposed to primitives (hash)
4. ...

To try at Home

1. more query examples against employee-department schema
2. description of *LINUX-info* using constraints/APs

Project Idea(s)

- code generation from templates
  (e.g., `...as array` generates code similar to the code on s.7)

Take Home

Lots of open issues:

1. DB engine vs. Compilation aproaches
2. Main memory data organization

   ⇒ pointers and records accommodated *natively*
   ⇒ coded as combination of AP and physical tables
3. Data structures can be (commonly) decomposed to primitives (hash)
4. …

To try at Home

1. more query examples against employee-department schema
2. description of *LINUX-info* using constraints/APs

Project Idea(s)

- code generation from templates

  (e.g., `...as array` generates code similar to the code on s.7)