Logic in Database Systems Implementation

David Toman

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

Joint work with Alexander Hudek and Grant Weddell
## Data and Constraints

### The Textbook View of Relational Databases

<table>
<thead>
<tr>
<th>Data</th>
<th>represented as an instance of a <em>relational structure</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>Queries</td>
<td>access to data via <em>open formulae</em> (in an appropriate logic)</td>
</tr>
<tr>
<td>Constraints</td>
<td>data integrity enforced by <em>sentences</em> (in the same logic)</td>
</tr>
</tbody>
</table>

⇒ the instance is a *model* of the constraints

### What about CREATE VIEW Statements?

View declaration ~ a sentence ∀x. V(x) → ϕ

where V is a (new) relational symbol and ϕ is a *query* (in our logic).
Data and Constraints

The Textbook View of Relational Databases

Data represented as an instance of a relational structure
Queries access to data via open formulæ (in an appropriate logic)
Constraints data integrity enforced by sentences (in the same logic)

⇒ the instance is a model of the constraints

What about CREATE VIEW Statements?

View declaration ~ a sentence \( \forall x. V(x) \leftrightarrow \varphi \)
where \( V \) is a (new) relational symbol and \( \varphi \) is a query (in our logic).

⇒ now the instance is no longer a model (unless \( V \) is a materialized view)

What about CREATE INDEX Statements?

Index declaration (on \( R \) for a key \( x \)) ~ a sentence \( \forall r, x. I(r, x) \leftrightarrow \exists y. R(r, x, y) \)
Data and Constraints

The Textbook View of Relational Databases

Data represented as an instance of a relational structure
Queries access to data via open formulæ (in an appropriate logic)
Constraints data integrity enforced by sentences (in the same logic)

⇒ the instance is a model of the constraints

What about CREATE VIEW Statements?

View declaration ~ a sentence \( \forall x. V(x) \leftrightarrow \varphi \)
where \( V \) is a (new) relational symbol and \( \varphi \) is a query (in our logic).

⇒ now the instance is no longer a model (unless \( V \) is a materialized view)

What about CREATE INDEX Statements?

Index declaration (on \( R \) for a key \( x \)) ~ a sentence \( \forall r, x. I(r, x) \leftrightarrow \exists y. R(r, x, y) \)
Data and Constraints

The Textbook View of Relational Databases

Data represented as an instance of a *relational structure*

Queries access to data via *open formulæ* (in an appropriate logic)

Constraints data integrity enforced by *sentences* (in the same logic)

⇒ the instance is a *model* of the constraints

What about **CREATE VIEW** Statements?

View declaration ∼ a sentence $\forall x. V(x) \leftrightarrow \varphi$

where $V$ is a (new) relational symbol and $\varphi$ is a *query* (in our logic).

⇒ *now* the instance is *no longer a model* (unless $V$ is a *materialized view*)

What about **CREATE INDEX** Statements?

Index declaration (on $R$ for a key $x$) ∼ a sentence $\forall r, x. l(r, x) \leftrightarrow \exists y. R(r, x, y)$

what is then the difference between the primary index on $R$ and $R$ itself?
Data and Constraints

The Textbook View of Relational Databases

Data represented as an instance of a relational structure
Queries access to data via open formulæ (in an appropriate logic)
Constraints data integrity enforced by sentences (in the same logic)

⇒ the instance is a model of the constraints

What about CREATE VIEW Statements?

View declaration \( \sim \) a sentence \( \forall x. V(x) \leftrightarrow \varphi \)
where \( V \) is a (new) relational symbol and \( \varphi \) is a query (in our logic).

⇒ now the instance is no longer a model (unless \( V \) is a materialized view)

What about CREATE INDEX Statements?

Index declaration (on \( R \) for a key \( x \)) \( \sim \) a sentence \( \forall r, x. I(r, x) \leftrightarrow \exists \bar{y}. R(r, x, y) \)

... what is then the difference between the primary index on \( R \) and \( R \) itself?
Physical Data Independence

**IDEA:**
Separate the users’ view(s) of the data from the way it is physically represented.

- independent customized user views,
- changes to conceptual structure without affecting users,
- physical storage details hidden from users,
- changes to physical storage without affecting logical view,

Originally just two levels: physical and conceptual/logical [Codd1970].

[ANSI/X3/SPARC Standards Planning and Requirements Committee, Bachman, 1975]
Physical Data Independence

**IDEA:**
Separate the users’ view(s) of the data from the way it is physically represented.

- independent customized user views,
- changes to conceptual structure without affecting users,
- physical storage details hidden from users,
- changes to physical storage without affecting logical view,

Originally just two levels: physical and conceptual/logical [Codd1970].

[ANSI/X3/SPARC Standards Planning and Requirements Committee, Bachman, 1975]
Physical Data Independence

**IDEA:**
Separate the users’ view(s) of the data from the way it is physically represented.

- independent customized user views,
- changes to conceptual structure without affecting users,
- physical storage details hidden from users,
- changes to physical storage without affecting logical view,

Originally just two levels: **physical** and **conceptual/logical** [Codd1970].

[ANSI/X3/SPARC Standards Planning and Requirements Committee, Bachman, 1975]
## What to do? (Managing User Expectations)

### Definability and Rewriting

<table>
<thead>
<tr>
<th>Queries</th>
<th>range-restricted FOL (a.k.a. SQL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ontology/Schema</td>
<td>range-restricted FOL $\Sigma := \Sigma^L \cup \Sigma^{LP} \cup \Sigma^P$</td>
</tr>
<tr>
<td>Data</td>
<td>CWA (complete information)</td>
</tr>
</tbody>
</table>
What to do? (Managing User Expectations)

Definability and Rewriting

<table>
<thead>
<tr>
<th>Queries</th>
<th>range-restricted FOL over $S_L$ definable w.r.t. $\Sigma$ and $S_A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ontology/Schema</td>
<td>range-restricted FOL $\Sigma := \Sigma^L \cup \Sigma^{LP} \cup \Sigma^P$</td>
</tr>
<tr>
<td>Data</td>
<td>CWA (complete information for $S_A$ symbols)</td>
</tr>
</tbody>
</table>

What to do? (Managing User Expectations)

### Definability and Rewriting

<table>
<thead>
<tr>
<th>Queries</th>
<th>range-restricted FOL over $S_L$ definable w.r.t. $\Sigma$ and $S_A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ontology/Schema</td>
<td>range-restricted FOL $\Sigma := \Sigma^L \cup \Sigma^{LP} \cup \Sigma^P$</td>
</tr>
<tr>
<td>Data</td>
<td>CWA (complete information for $S_A$ symbols)</td>
</tr>
</tbody>
</table>

- 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

---

Diagram:

- **Query** ($S_L$)
  - $\psi$
- **Schema** ($S_L \cup S_P$)
  - $\Sigma$
- **Evaluator**
  - (instance of) $S_A$
- **Answers**
  - $\psi$ (Relational Algebra over $S_A$)
GRAND UNIFIED APPROACH TO QUERY COMPILATION

PART I: PLANS AS FORMULAE AND STANDARD DESIGN
Issues to resolve (today)

- What “formulas” do qualify as *plans*?
  - ⇒ 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
3. Logical Connectives/Quantifiers as Plan Operators
4. Beyond Logical Operators: Dealing with Duplicates (not today)
Creating Table(s) and Base File(s)

Specification:

\[
\begin{align*}
&\% \text{ constraints} \\
&\text{table}(x,y,z) \leftrightarrow \text{ex}(r,\text{p0basetable}(r,x,y,z)) \\
\&\% \text{ query} \\
&q(x,y,z) \leftrightarrow \text{table}(x,y,z) \\
\end{align*}
\]

Notes:

- additional \( r \) attribute in the physical storage
Access Path Code Templates

Pseudo-code templates realizing a \texttt{first/next} protocol might be given as follows (variables would be renamed for each occurrence of \texttt{p0basetable} in a query plan).

\begin{verbatim}
function p0basetable-first
  \texttt{i} := 0
  return p0basetable-next

function p0basetable-next
  \texttt{i} := \texttt{i} + 1
  if (\texttt{i} > \texttt{n}) return false
  \texttt{r} := \texttt{i}
  \texttt{x} := array[\texttt{i}].emp-num
  \texttt{y} := array[\texttt{i}].emp-name
  \texttt{z} := array[\texttt{i}].emp-salary
  return true
\end{verbatim}

Assumes a global state recording bindings of (possible copies of) variables.

- \texttt{x}, \texttt{y} and \texttt{z} to communicate the contents of \texttt{p0basetable}.
- \texttt{i} and \texttt{n} to record scanning status and size of \texttt{p0basetable}.

Note: Code templates for access paths must be provided by DBA.
Pseudo-code templates realizing a first/next protocol might be given as follows (variables would be renamed for each occurrence of `p0basetable` in a query plan).

```plaintext
function p0basetable-first
    i := 0
    return p0basetable-next

function p0basetable-next
    i := i + 1
    if (i > n) return false
    r := i
    x := array[i].emp-num
    y := array[i].emp-name
    z := array[i].emp-salary
    return true
```

Assumes a global state recording bindings of (possible copies of) variables.

1. `x`, `y` and `z` to communicate the contents of `p0basetable`.
2. `i` and `n` to record scanning status and size of `p0basetable`.

Note: Code templates for access paths must be provided by DBA.
Pseudo-code templates realizing a first/next protocol might be given as follows (variables would be renamed for each occurrence of p0basetable in a query plan).

```plaintext
function p0basetable-first
  i := 0
  return p0basetable-next

function p0basetable-next
  i := i + 1
  if (i > n) return false
  r := i
  x := array[i].emp-num
  y := array[i].emp-name
  z := array[i].emp-salary
  return true
```

Assumes a global state recording bindings of (possible copies of) variables.

1. x, y and z to communicate the contents of p0basetable.
2. i and n to record scanning status and size of p0basetable.

**Note:** Code templates for access paths must be provided by DBA.
Index-only Plans & Column Store

Specification:

\[ \text{table}(x,y,z) \leftrightarrow \text{ex}(r, \text{basetable}(r,x,y,z)), \]

% indices
\[ \text{p0indexx}(r,x) \leftrightarrow \text{ex}([y,z], \text{basetable}(r,x,y,z)), \]
\[ \text{p0indexy}(r,y) \leftrightarrow \text{ex}([x,z], \text{basetable}(r,x,y,z)), \]
\[ \text{p0indexz}(r,z) \leftrightarrow \text{ex}([x,y], \text{basetable}(r,x,y,z)), \]

% keys
\[ \text{basetable}(r,x1,y1,z1) \text{ and } \text{basetable}(r,x2,y2,z2) \rightarrow (x1=x2 \text{ and } y1=y2 \text{ and } z1=z2), \]

% query
\[ q(x,y,z) \leftrightarrow \text{table}(x,y,z) \]

Notes:

- key constraint necessary (why?)
Conjunctive Query Plans: Semantics

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

function \((Q_1 \land Q_2)\)-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 \(\langle x_1, \ldots, x_n \rangle\) to \(S\)
return true
return false

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

The *access paths* $S_A \subseteq S_P$ are predicate symbols *associated* with a *physical capability* realized by an iterator implementation.

⇒ some attributes can be designated as *parameters* (by convention the left-most ones)

Requirements (for access path $\mathcal{AP}/k + l/l$):

- given a fixed values for parameters there are only finitely many answers (bindings) to the remaining variables, i.e., the set

$$\{a_1, \ldots, a_k \mid I, \mathcal{V} = \exists x_1, \ldots, x_l. \mathcal{AP}(x_1, \ldots, x_l, y_1, \ldots, y_k) \land (\bigwedge x_i = p_i)\}$$

(where $\mathcal{V} = [y_1 = a_1, \ldots, y_k = a_k]$) is finite.

- the invocation of the iterator protocol *outputs* all and only the valuations $\mathcal{V}$ that satisfy the condition above.
Access Paths

The access paths $S_A \subseteq S_P$ are predicate symbols associated with a physical capability realized by an iterator implementation.

⇒ some attributes can be designated as parameters
(by convention the left-most ones)

Requirements (for access path $\text{AP}/k + l/l$):

- given a fixed values for parameters there are only finitely many answers (bindings) to the remaining variables, i.e., the set

$$\{a_1, \ldots, a_k \mid \mathcal{I}, \mathcal{V} \models \exists x_1, \ldots, x_l. \text{AP}(x_1, \ldots, x_l, y_1, \ldots, y_k) \land (\bigwedge x_i = p_i)\}$$

(where $\mathcal{V} = [y_1 = a_1, \ldots, y_k = a_k]$) is finite.

- the invocation of the iterator protocol outputs all and only the valuations $\mathcal{V}$ that satisfy the condition above.
(More Esoteric) Access Paths

1. Built-in “operations”:
   - arithmetic (\texttt{plus/3/2, times/3/2, etc.})
   - string manipulation (\texttt{concat/3/2, substr/4/3, etc.})
   - ...

2. Data type tests (\texttt{is-integer/1/1})

3. Pointer dereference and field extraction from records

4. (Page) reads from external storage

5. ...

David Toman (et al.)
Logic DBMS Implementation

Plans as Formulae 13 / 24
Horizontal Partition

Specification: [...

% horizontal partitions
p0hpp1(r,x,y,z) -> basetable(r,x,y,z),
p0hpp2(r,x,y,z) -> basetable(r,x,y,z),
p0hpp3(r,x,y,z) -> basetable(r,x,y,z),
basetable(r,x,y,z) -> (p0hpp1(r,x,y,z) or
p0hpp2(r,x,y,z) or p0hpp3(r,x,y,z)),
...

Notes:

■ do we need “disjointness” of the partitions?
Subclass/Complement

Specification: [ ... 

% superclass and coverage 
basetable(r,x,y,z) -> p0super(r,x,y,z), 
p0complement(r,x,y,z) -> p0super(r,x,y,z), 
p0super(r,x,y,z) -> (p0complement(r,x,y,z) 
or basetable(r,x,y,z)),

% disjointness 
p0complement(r,x,y,z) and basetable(r,x,y,z) 
-> bot,

...].

Notes: 
- do we need “disjointness”? “keys”?
General Query Plans: Syntax

The *query plans* induced by S add two final productions:

\[
Q ::= (Q_1 \lor Q_2) \quad \text{(concatenation)} \\
| \quad \neg Q \quad \text{(simple complement)}
\]
The query plans induced by S add two final productions:

\[ Q ::= (Q_1 \lor Q_2) \quad (\text{concatenation}) \]
\[ | \quad \neg Q \quad (\text{simple complement}) \]

---

function \((Q_1 \lor Q_2)\)-first
\[
(Q_1 \lor Q_2)\text{-flag} := \text{true} \\
\text{if } Q_1\text{-first return true} \\
(Q_1 \lor Q_2)\text{-flag} := \text{false} \\
\text{return } Q_2\text{-first}
\]

function \((Q_1 \lor Q_2)\)-next
\[
\text{if } (Q_1 \lor Q_2)\text{-flag} \\
\text{if } Q_1\text{-next return true} \\
(Q_1 \lor Q_2)\text{-flag} := \text{false} \\
\text{return } Q_2\text{-next}
\]

function \((\neg Q_1)\)-first
\[
\text{if } Q_1\text{-first return false} \\
\text{return true}
\]

function \((\neg Q_1)\)-next
\[
\text{return false}
\]
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
Grand Unified Approach to 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

1 Logical Schema

```
employee works department
number enumber
name dnumber
salary manager
```

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

- **record emp of**: integer num, string name, integer salary, reference dept
- **record dept of**: integer num, string name, reference manager

3 Access Paths: `empfile/1/0, emp-num/2/1, ...` (but no deptfile)

4 Integrity Constraints (many), e.g.,

\[
\forall x, y, z. \text{employee}(x, y, z) \rightarrow \exists w. \text{empfile}(w) \land \text{emp-num}(w, x),
\forall a, x. \text{empfile}(a) \land \text{emp-num}(a, x) \rightarrow \exists y, z. \text{employee}(x, y, z), ...
\]
What can this do: navigating pointers

1. List all employee numbers and names ($\exists z, w. \text{employee}(x, y, z, w)$):
   $$\exists a. \text{empfile}(a) \land \text{emp-num}(a, x) \land \text{emp-name}(a, y)$$

2. List all department numbers with their manager names
   ($\exists z, u, v. \text{department}(x, z, u) \land \text{employee}(u, y, v)$):
   $$\exists a. \text{empfile}(a) \land \text{emp-dept}(a, d) \land \text{dept-num}(d, x) \land \text{dept-mgr}(d, e) \land \text{emp-name}(e, y)$$
   $$\land \text{dept-num}(d, x) \land \text{dept-mgr}(d, b) \land \text{compare}(a, b)$$
What can this do: navigating pointers

1. List all employee numbers and names ($\exists z, w. \text{employee}(x, y, z, w)$):
   
   $$\exists a. \text{empfile}(a) \land \text{emp-num}(a, x) \land \text{emp-name}(a, y)$$

   or, in C-like syntax:
   
   ```
   for a in empfile do
     x := a->num;
     y := a->name;
   ```

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

   $$\forall a. \text{empfile}(a) \land \text{emp-name}(a, y) \land \text{emp-dept}(a, d) \land \text{dept-num}(d, x) \land \text{dept-mgr}(d, e) \land \text{compare}(a, e)$$

   needs "managers work in their own departments".

   NO duplicate elimination during projection.

David Toman (et al.)
Logic DBMS Implementation
What can it do? 21 / 24
What can this do: navigating pointers

1. List all employee numbers and names ($\exists z, w.\text{employee}(x, y, z, w)$):
   $$\exists a.\text{empfile}(a) \land \text{emp-num}(a, x) \land \text{emp-name}(a, y)$$

2. List all department numbers with their manager names
   ($\exists z, u, v.\text{department}(x, z, u) \land \text{employee}(u, y, v)$):
   $$\exists a.\text{empfile}(a) \land \text{emp-dept}(a, d)$$
   $$\land \text{dept-num}(d, x) \land \text{dept-mgr}(d, e) \land \text{emp-name}(e, y)$$
   $$\Rightarrow \text{needs "departments have at least one employee".}$$
   $$\text{... needs duplicate elimination during projection.}$$
   $$\exists a.\text{empfile}(a) \land \text{emp-name}(a, y) \land \text{emp-dept}(a, d)$$
   $$\land \text{dept-num}(d, x) \land \text{dept-mgr}(d, b) \land \text{compare}(a, b)$$
   $$\Rightarrow \text{needs "managers work in their own departments".}$$
   $$\text{NO duplicate elimination during projection.}$$
What can this do: navigating pointers

1. List all employee numbers and names \((\exists z, w.\text{employee}(x, y, z, w))\):
   \[
   \exists a.\text{empfile}(a) \land \text{emp-num}(a, x) \land \text{emp-name}(a, y)
   \]

2. List all department numbers with their manager names
   \((\exists z, u, v.\text{department}(x, z, u) \land \text{employee}(u, y, v))\):
   \[
   \exists a, d, e.\text{empfile}(a) \land \text{emp-dept}(a, d) \\
   \land \text{dept-num}(d, x) \land \text{dept-mgr}(d, e) \land \text{emp-name}(e, y)
   \Rightarrow \text{needs “departments have at least one employee”}
   \]
   \[
   \exists a, d, e.\text{empfile}(a) \land \text{emp-name}(a, y) \land \text{emp-dept}(a, d) \\
   \land \text{dept-num}(d, x) \land \text{dept-mgr}(d, b) \land \text{compare}(a, b)
   \Rightarrow \text{needs “managers work in their own departments”}
   \]
What can this do: navigating pointers

1. List all employee numbers and names ($\exists z, w. \text{employee}(x, y, z, w)$):
   $$\exists a. \text{empfile}(a) \land \text{emp-num}(a, x) \land \text{emp-name}(a, y)$$

2. List all department numbers with their manager names
   ($\exists z, u, v. \text{department}(x, z, u) \land \text{employee}(u, y, v)$):
   $$\exists a, d, e. \text{empfile}(a) \land \text{emp-dept}(a, d)$$
   $$\land \text{dept-num}(d, x) \land \text{dept-mgr}(d, e) \land \text{emp-name}(e, y)$$
   $$\Rightarrow \text{needs “departments have at least one employee”}.$$
What can this do: navigating pointers

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

   $$\exists a.\text{empfile}(a) \land \text{emp-num}(a, x) \land \text{emp-name}(a, y)$$

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

   $$\exists a, d, e.\text{empfile}(a) \land \text{emp-dept}(a, d)$$
   $$\land \text{dept-num}(d, x) \land \text{dept-mgr}(d, e) \land \text{emp-name}(e, y)$$

   $$\Rightarrow \text{needs “departments have at least one employee”}.$$  
   $$\ldots \text{needs duplicate elimination during projection.}$$

   $$\exists a, b, d.\text{empfile}(a) \land \text{emp-name}(a, y) \land \text{emp-dept}(a, d)$$
   $$\land \text{dept-num}(d, x) \land \text{dept-mgr}(d, b) \land \text{compare}(a, b)$$

   $$\Rightarrow \text{needs “managers work in their own departments”}.$$  
   $$\ldots \text{NO duplicate elimination during projection.}$$
What can it do: Hashing, Lists, et al.

Hash Index with (list-based) Separate Chaining

Hash Array | Separate Chaining Linked Lists | Dept Records

$i$:

$j$:

$n$:

$D1$

$D3$

$D2$
What can it do: Hashing, Lists, et al.

Hash Index on department’s name:

Access paths:

\[ S_A \supseteq \{\text{hash/2/1}, \text{hasharraylookup/2/1}, \text{listscan/2/1}\} \]

Physical Constraints:

\[ \Sigma_{LP} \supseteq \{ \forall x, y.((\text{deptfile}(x) \land \text{dept-name}(x, y)) \rightarrow \exists z, w. (\text{hash}(y, z) \land \text{hasharraylookup}(z, w) \land \text{listscan}(w, x))), \]

\[ \forall x, y.(\text{hash}(x, y) \rightarrow \exists z. \text{hasharraylookup}(y, z)), \]

\[ \forall x, y.(\text{listscan}(x, y) \rightarrow \text{deptfile}(y)) \}

Query:

\[ \exists y, z. (\text{department}(x_1, p, y) \land \text{employee}(y, x_2, z))\{p\}. \]

\[ \forall x_6. (\text{Hash}(p, ?x_6) \land ?x_5. (\text{Hasharraylookup}(?x_5, ?x_5) \land ?x_4. (\text{Listscan}(?x_5, ?x_4) \land ?x_0. (\text{Dept-name}(?x_4, ?x_0) \land \text{Cmp}(p, ?x_0)) \land \text{Dept-num}(?x_4, x_1) \land ?x_3. (\text{Dept-manager}(?x_4, ?x_3) \land \text{Emp-name}(?x_3, x_2))))) \]
What can it do: Hashing, Lists, et al.

Hash Index on department’s name:

Access paths:
\[ S_A \supseteq \{ \text{hash/2/1}, \text{hasharraylookup/2/1}, \text{listscan/2/1} \} \].

Physical Constraints:
\[ \Sigma_{LP} \supseteq \{ \forall x, y.((\text{deptfile}(x) \land \text{dept-name}(x, y)) \rightarrow \exists z, w.(\text{hash}(y, z) \land \text{hasharraylookup}(z, w) \land \text{listscan}(w, x))), \forall x, y.(\text{hash}(x, y) \rightarrow \exists z.\text{hasharraylookup}(y, z)), \forall x, y.(\text{listscan}(x, y) \rightarrow \text{deptfile}(y)) \} \]

Query:
\[ \exists y, z.(\text{department}(x_1, p, y) \land \text{employee}(y, x_2, z))\{p\}. \]

\[ \text{E?x6.}(\text{Hash}(p, ?x6)^\text{E?x5.}(\text{Hasharraylookup}(?x6, ?x5)^\text{E?x4.}(\text{Listscan}(?x5, ?x4)^\text{E?s0.}(\text{Dept-name}(?x4, ?s0)^\text{Cmp}(p, ?s0))\text{Dept-num(?x4,x1)})^\text{E?x3.}(\text{Dept-manager(?x4,?x3)^Emp-name(?x3,x2))})))) \]
What can this do: two-level store

The access path `empfile` is refined by `emppages/1/0` and `emprecords/2/1`:

- `emppages` returns (sequentially) disk pages containing `emp` records, and
- `emprecords` given a disc page, returns `emp` records in that page.

5 List all employees with the same name

\( (\exists z, u, v, w, t. \text{employee}(x_1, z, u, v) \land \text{employee}(x_2, z, w, t)) : \)

\( \exists y, z, w, v, p, q. \text{emppages}(p) \land \text{emppages}(q) \)
\( \land \text{emprecords}(p, y) \land \text{emp-num}(y, x_1) \land \text{emp-name}(y, w) \)
\( \land \text{emprecords}(q, z) \land \text{emp-num}(z, x_2) \land \text{emp-name}(z, v) \)
\( \land \text{compare}(w, v). \)

\( \Rightarrow \) this plan implements the *block nested loops join* algorithm.

\[ \text{...more examples in ...} \]