Topics in Database Systems: Modern DBMS

CS848 Spring 2022

David Toman

(POSTPROCESSING AND EFFICIENCY)

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++ !

Two-level Storage

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

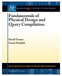
`emppages` returns (sequentially) disk pages containing `emp` records, and

`emprecords` given a disc page, returns `emp` records in that page.

⑤ List all employees with the same name
$(\exists z, u, v, w, t.\texttt{employee}(x_1, z, u, v) \land \texttt{employee}(x_2, z, w, t))$:

$$\begin{aligned}&\exists y, z, w, v, p, q.\texttt{emppages}(p) \land \texttt{emppages}(q)\\ &\quad \land \texttt{emprecords}(p, y) \land \texttt{emp-num}(y, x_1) \land \texttt{emp-name}(y, w)\\ &\quad \land \texttt{emprecords}(q, z) \land \texttt{emp-num}(z, x_2) \land \texttt{emp-name}(z, v)\\ &\qquad\qquad\qquad\qquad\qquad\qquad \land \texttt{compare}(w, v).\end{aligned}$$

⇒ this plan implements the *block nested loops join* algorithm.

. . . more examples in .

Fundamentals of Physical Design and Query Compilation

David Toman

Grant Weddell

DUPLICATES AND POST-PROCESSING

Query Context

Assume Q is a query plan that contains a subplan Q_1.

Write $Q[Q_1]$ to denote this and call $Q[\,]$, in which Q_1 has been replaced by a placeholder "[ ]", a *context*.

Given a context Q^c, a *user query* $\mathrm{Uq}_p(Q^c)$ abstracting *properties* of variables *within the context* is defined as follows.

$$\mathrm{Uq}_p(Q^c) \equiv \begin{cases} \top & Q^c = \text{“}[\,]\text{”} \\ \mathrm{Uq}(Q_2) \wedge \mathrm{Uq}_p(Q_1^c) & Q^c = \text{“}Q_1^c[Q_2 \wedge [\,]]\text{” or “}Q_1^c[[\,] \wedge Q_2]\text{”} \\ \exists x.\, \mathrm{Uq}_p(Q_1^c) & Q^c = \text{“}Q_1^c[\exists x.[\,]]\text{”} \\ \mathrm{Uq}_p(Q_1^c) & Q^c = \text{“}Q_1^c[\{[\,]\}]\text{”, “}Q_1^c[\neg[\,]]\text{”, “}Q_1^c[Q_2 \vee [\,]]\text{”} \\ & \qquad \text{or “}Q_1^c[[\,] \vee Q_2]\text{”} \end{cases}$$

$\Rightarrow$ extract as much information from Q and Q_1 as possible for $\mathcal{C}_1$ and $\mathcal{C}_2$.

Query Context

Assume Q is a query plan that contains a subplan Q_1.

Write $Q[Q_1]$ to denote this and call $Q[\,]$, in which Q_1 has been replaced by a placeholder "[ ]", a *context*.

Given a context Q^c, a *user query* $\mathrm{Uq}_p(Q^c)$ abstracting *properties* of variables *within the context* is defined as follows.

$$\mathrm{Uq}_p(Q^c) \equiv \begin{cases} \top & Q^c = \text{“}[\,]\text{”} \\ \mathrm{Uq}(Q_2) \wedge \mathrm{Uq}_p(Q_1^c) & Q^c = \text{“}Q_1^c[Q_2 \wedge [\,]]\text{” or “}Q_1^c[[\,] \wedge Q_2]\text{”} \\ \exists x.\, \mathrm{Uq}_p(Q_1^c) & Q^c = \text{“}Q_1^c[\exists x.[\,]]\text{”} \\ \mathrm{Uq}_p(Q_1^c) & Q^c = \text{“}Q_1^c[\{[\,]\}]\text{”, “}Q_1^c[\neg[\,]]\text{”, “}Q_1^c[Q_2 \vee [\,]]\text{”} \\ & \qquad\text{or “}Q_1^c[[\,] \vee Q_2]\text{”} \end{cases}$$

$\Rightarrow$ extract as much information from Q and Q_1 as possible for $\mathcal{C}_1$ and $\mathcal{C}_2$.

Eliminating Duplicate Elimination (cont'd)

Assume $\langle \mathsf{S}_\mathsf{L} \cup \mathsf{S}_\mathsf{P}, \Sigma \rangle$ is a physical design and $Q^c[Q]$ a query plan. Then the following rewrite rules hold.

$$\begin{array}{rcl}
Q^c[\{R(x_1,\ldots,x_k)\}] & \leftrightarrow & Q^c[R(x_1,\ldots,x_k)] \\
Q^c[\{Q_1 \land Q_2\}] & \leftrightarrow & Q^c[\{Q_1\} \land \{Q_2\}] \\
Q^c[\{\exists x.Q_1\}] & \underset{\mathcal{C}_1}{\leftrightarrow} & Q^c[\exists x.\{Q_1\}] \\
Q^c[\{\neg Q_1\}] & \leftrightarrow & Q^c[\neg Q_1] \\
Q^c[\neg\{Q_1\}] & \leftrightarrow & Q^c[\neg Q_1] \\
Q^c[\{Q_1 \lor Q_2\}] & \underset{\mathcal{C}_2}{\leftrightarrow} & Q^c[\{Q_1\} \lor \{Q_2\}]
\end{array}$$

where $\mathcal{C}_1 = \Sigma \cup \{Q^c \land Q_1[y_1/x] \land Q_1[y_2/x]\} \models (y_1 \approx y_2)$

$\mathcal{C}_2 = \Sigma \cup \{Q^c\} \models (Q_1 \land Q_2) \to \bot$

Cut Operator

Query with *two* parameters checking for *correct* salaries:

$$\{\exists y, z.(\texttt{emp}(x, y, z) \wedge (\texttt{correct}(p_1, z) \vee \texttt{correct}(p_2, z)))\}.$$

we know that for every x there is just one answer:

$$\exists y, z.(\texttt{emp}(x, y, z) \wedge [(\texttt{correct}(p_1, z) \vee \texttt{correct}(p_2, z))]_1 \wedge !_1)).$$

```
function ([$Q_1$]$_i$)-first
    cut$i$ := false
    return $Q_1$-first

function ($!_i$)-first
    cut$i$ := true
    return true
```

```
function ([$Q_1$]$_i$)-next
    if cut$i$ return false
    return $Q_1$-next

function ($!_i$)-next
    return false
```

Cut Operator

Query with *two* parameters checking for *correct* salaries:

$$\{\exists y, z.(\texttt{emp}(x, y, z) \wedge (\texttt{correct}(p_1, z) \vee \texttt{correct}(p_2, z)))\}.$$

we know that for every x there is just one answer:

$$\exists y, z.(\texttt{emp}(x, y, z) \wedge [(\texttt{correct}(p_1, z) \vee \texttt{correct}(p_2, z))]_1 \wedge \,!_1)).$$

```
function ($[Q_1]_i$)-first
    cut$i$ := false
    return $Q_1$-first

function ($!_i$)-first
    cut$i$ := true
    return true
```

```
function ($[Q_1]_i$)-next
    if cut$i$ return false
    return $Q_1$-next

function ($!_i$)-next
    return false
```

Cut Operator

Query with *two* parameters checking for *correct* salaries:

$$\{\exists y, z.(\texttt{emp}(x, y, z) \wedge (\texttt{correct}(p_1, z) \vee \texttt{correct}(p_2, z)))\}.$$

we know that for every x there is just one answer:

$$\exists y, z.(\texttt{emp}(x, y, z) \wedge [(\texttt{correct}(p_1, z) \vee \texttt{correct}(p_2, z))]_1 \wedge\, !_1)).$$

```
function ($[Q_1]_i$)-first
   cut$i$ := false
   return $Q_1$-first

function ($!_i$)-first
   cut$i$ := true
   return true
```

```
function ($[Q_1]_i$)-next
   if cut$i$ return false
   return $Q_1$-next

function ($!_i$)-next
   return false
```

Incremental Query Context

Given a context Q^c, a *user query* $\text{Uq}_{ip}(Q^c)$ abstracting *incremental properties* of variables within the context is defined as follows.

$$\text{Uq}_{ip}(Q^c) \equiv \begin{cases} \top & Q^c = \text{“}[\,]\text{”} \\ \text{Uq}(Q_2) \wedge \text{Uq}_{ip}(Q_1^c) & Q^c = \text{“}Q_1^c[Q_2 \wedge [\,]]\text{”} \\ \exists x.\, \text{Uq}_{ip}(Q_1^c) & Q^c = \text{“}Q_1^c[\exists x.[\,]]\text{”} \\ \text{Uq}_{ip}(Q_1^c) & Q^c = \text{“}Q_1^c[\{[\,]\}]\text{”}, \text{“}Q_1^c[\neg[\,]]\text{”}, \text{“}Q_1^c[Q_2 \vee [\,]]\text{”}, \\ & \qquad \text{“}Q_1^c[[\,] \vee Q_2]\text{” or “}Q_1^c[[\,] \wedge Q_2]\text{”} \end{cases}$$

Cut Insertion

Observe that the rewrite rules for duplicate elimination are bidirectional, and can therefore determine situations in which such operators can be *added* to a query plan.

This is useful when formulating additional rewrite rules that determine when cut operators can be inserted in query plans without any impact on their ability to implement user queries.

Assume $\langle \mathsf{S_L} \cup \mathsf{S_P}, \Sigma \rangle$ is a physical design and $Q^c[\{Q_1\} \wedge Q_2]$ a query plan. Then the following rewrite rule holds.

$$Q^c[\{Q_1\} \wedge Q_2] \;\underset{\mathcal{C}}{\leftrightarrow}\; Q^c[[\{Q_1\}]_\ell \wedge (Q_2 \wedge\, !_\ell)]$$

$\mathcal{C}_1$ corresponds to the following condition, where $\mathrm{Out}(Q_1) = \{x_1, \ldots, x_k\}$ and where each y_i and z_j are fresh variable names not occurring in Q^c, Q_1 or Q_2.

$$\Sigma \cup \{\mathrm{Uq}_{ip}(Q^c) \wedge \mathrm{Uq}((Q_1 \wedge Q_2)[y_1/x_1, \ldots, y_k/x_k]) \wedge \mathrm{Uq}((Q_1 \wedge Q_2)[z_1/x_1, \ldots, z_k/x_k])\}$$
$$\models (y_1 \approx z_1) \wedge \cdots \wedge (y_k \approx z_k)$$

SORTED ACCESS

What about Merge-Joins et al??

Join Algorithms (in typical DBMS):

- Block Nested Loops:
  - ⇒ takes care of *block* access (done);
- Hash:
  - ⇒ free if appropriate *hashtable(s)* already exist
  - ⇒ creting hashtables = extra physical design/on the fly decision
- Merge(-Sort):
  - ⇒ ????   ⇐ NOW
  - ⇒ sorting = extra physical design/on the fly decision

What about Merge-Joins et al??

Join Algorithms (in typical DBMS):

- Block Nested Loops:

  ⇒ takes care of *block* access (done);

- Hash:

  ⇒ free if appropriate *hashtable(s)* already exist
  ⇒ creting hashtables = extra physical design/on the fly decision

- Merge(-Sort):

  ⇒ ???? ⇐ NOW
  ⇒ sorting = extra physical design/on the fly decision

What about Merge-Joins et al??

Join Algorithms (in typical DBMS):

- Block Nested Loops:
  ⇒ takes care of *block* access (done);
- Hash:
  ⇒ free if appropriate *hashtable(s)* already exist
  ⇒ creting hashtables = extra physical design/on the fly decision
- Merge(-Sort):
  ⇒ ???? ⇐ NOW
  ⇒ sorting = extra physical design/on the fly decision

What about Merge-Joins et al??

Join Algorithms (in typical DBMS):

- Block Nested Loops:

  ⇒ takes care of *block* access (done);

- Hash:

  ⇒ free if appropriate *hashtable(s)* already exist
  ⇒ creting hashtables = extra physical design/on the fly decision

- Merge(-Sort):

  ⇒ ???? ⇐ NOW
  ⇒ sorting = extra physical design/on the fly decision

Merge-Joins Solution(s)

IDEA:

- improve *ordered* access paths with *fingers*

  ⇒ modifies the behaviour of *get-first* depending on a parameter

- use standard Nested Loops Join

Example (Joining two sorted arrays with distinct values)

A :	1	3	6	8	11	17	...	50

B :	3	4	5	6	11	...	55

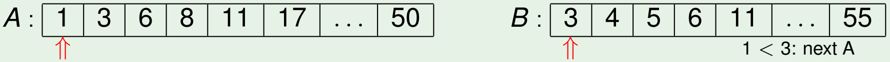
Merge-Joins Solution(s)

IDEA:

- improve *ordered* access paths with *fingers*

  ⇒ modifies the behaviour of *get-first* depending on a parameter
- use standard Nested Loops Join

Example (Joining two sorted arrays with distinct values)



A :

1	3	6	8	11	17	...	50
⇑							

B :

3	4	5	6	11	...	55
⇑						

$1 < 3$: next A

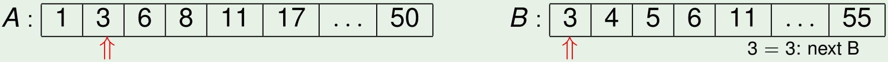
Merge-Joins Solution(s)

IDEA:

- improve *ordered* access paths with *fingers*

  ⇒ modifies the behaviour of *get-first* depending on a parameter
- use standard Nested Loops Join

Example (Joining two sorted arrays with distinct values)



A :

1	3	6	8	11	17	...	50
	⇑						

B :

3	4	5	6	11	...	55
⇑						

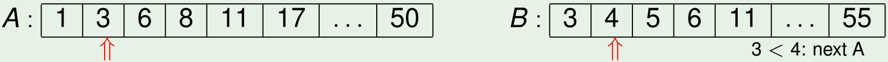
$3 = 3$: next B

Merge-Joins Solution(s)

IDEA:

- improve *ordered* access paths with *fingers*
  ⇒ modifies the behaviour of *get-first* depending on a parameter
- use standard Nested Loops Join

Example (Joining two sorted arrays with distinct values)



A :	1	3	6	8	11	17	...	50
		⇑						

B :	3	4	5	6	11	...	55
		⇑					

$3 < 4$: next A

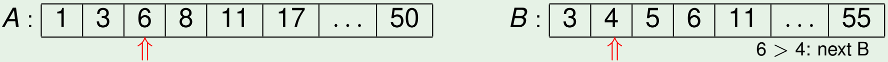
Merge-Joins Solution(s)

IDEA:

- improve *ordered* access paths with *fingers*

  ⇒ modifies the behaviour of *get-first* depending on a parameter
- use standard Nested Loops Join

Example (Joining two sorted arrays with distinct values)



A :

1	3	6	8	11	17	. . .	50

B :

3	4	5	6	11	. . .	55

6 $>$ 4: next B

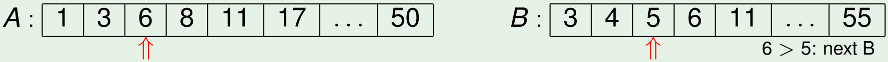
Merge-Joins Solution(s)

IDEA:

- improve *ordered* access paths with *fingers*

  ⇒ modifies the behaviour of *get-first* depending on a parameter
- use standard Nested Loops Join

Example (Joining two sorted arrays with distinct values)



A :

1	3	6	8	11	17	...	50

B :

3	4	5	6	11	...	55

6 $>$ 5: next B

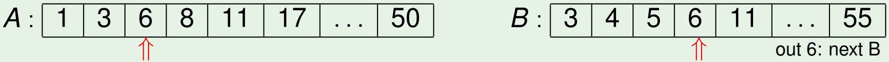
Merge-Joins Solution(s)

IDEA:

- improve *ordered* access paths with *fingers*

  ⇒ modifies the behaviour of *get-first* depending on a parameter
- use standard Nested Loops Join

Example (Joining two sorted arrays with distinct values)



A : 1 | 3 | 6 | 8 | 11 | 17 | . . . | 50

B : 3 | 4 | 5 | 6 | 11 | . . . | 55

out 6: next B

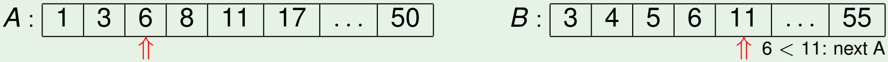
Merge-Joins Solution(s)

IDEA:

- improve *ordered* access paths with *fingers*

  ⇒ modifies the behaviour of *get-first* depending on a parameter
- use standard Nested Loops Join

Example (Joining two sorted arrays with distinct values)



A : 1 | 3 | 6 | 8 | 11 | 17 | . . . | 50

B : 3 | 4 | 5 | 6 | 11 | . . . | 55

6 $<$ 11: next A

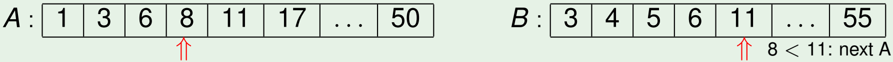
Merge-Joins Solution(s)

IDEA:

- improve *ordered* access paths with *fingers*

  ⇒ modifies the behaviour of *get-first* depending on a parameter
- use standard Nested Loops Join

Example (Joining two sorted arrays with distinct values)



A :

1	3	6	8	11	17	...	50

B :

3	4	5	6	11	...	55

$8 < 11$: next A

Merge-Joins Solution(s)

IDEA:

- improve *ordered* access paths with *fingers*

  ⇒ modifies the behaviour of *get-first* depending on a parameter
- use standard Nested Loops Join

Example (Joining two sorted arrays with distinct values)

A :	1	3	6	8	11	17	...	50

B :	3	4	5	6	11	...	55

out 11: next B

Merge-Joins Solution(s)

IDEA:

- improve *ordered* access paths with *fingers*

  ⇒ modifies the behaviour of *get-first* depending on a parameter
- use standard Nested Loops Join

Example (Joining two sorted arrays with distinct values)

A :

1	3	6	8	11	17	...	50

B :

3	4	5	6	11	...	55

etc.

Merge-Joins Solution(s)

IDEA:

- improve *ordered* access paths with *fingers*
  ⇒ modifies the behaviour of *get-first* depending on a parameter
- use standard Nested Loops Join

How Well are we doing?

- *simulates a merge join* provided the arrays are sorted
  ⇒ B **must be sorted and finger-modified** (i.e., has an parameter)
  ⇒ A no changes; what happens if A is **not sorted**?
- pay-as-you-go behaviour: ordered runs (in the A)
- seamlesly integrates with other operators
  ⇒ disjunction/concatenation, . . .
- can be extended to two-level access to data (how?)
- . . .

Merge-Joins Solution(s)

IDEA:

- improve *ordered* access paths with *fingers*

  ⇒ modifies the behaviour of *get-first* depending on a parameter
- use standard Nested Loops Join

How Well are we doing?

- *simulates a merge join* provided the arrays are sorted

  ⇒ B **must be sorted and finger-modified** (i.e., has an parameter)
  ⇒ A no changes; what happens if A is **not sorted**?
- pay-as-you-go behaviour: ordered runs (in the A)
- seamlesly integrates with other operators
  ⇒ disjunction/concatenation, . . .
- can be extended to two-level access to data (how?)
- . . .

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.

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`.
10. 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`.
10. 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*.

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?

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

   ⇒ 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

   ⇒ 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)