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

![Logical Schema Diagram]

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

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

<table>
<thead>
<tr>
<th>Record of <code>emp</code></th>
<th>Record of <code>dept</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>integer <code>num</code></td>
<td>integer <code>num</code></td>
</tr>
<tr>
<td>string <code>name</code></td>
<td>string <code>name</code></td>
</tr>
<tr>
<td>reference <code>dept</code></td>
<td>reference <code>manager</code></td>
</tr>
</tbody>
</table>

⇒ 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

<table>
<thead>
<tr>
<th>employee</th>
<th>works</th>
<th>department</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>emp</td>
<td>number</td>
</tr>
<tr>
<td>name</td>
<td>dept</td>
<td>name</td>
</tr>
</tbody>
</table>

⇒ 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
integer num
string name
reference dept

record dept of
integer num
string name
reference manager

⇒ 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

\[
\begin{align*}
\text{employee} & \quad \text{works} & \quad \text{department} \\
\text{number} & \quad \text{emp} & \quad \text{number} \\
\text{name} & \quad \text{dept} & \quad \text{name} \\
\end{align*}
\]

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

2. Physical Design: a \textit{linked list of emp records pointing to dept records}.

\[
\begin{align*}
\text{record emp of} & \quad \text{record dept of} \\
\text{integer num} & \quad \text{integer num} \\
\text{string name} & \quad \text{string name} \\
\text{reference dept} & \quad \text{reference manager} \\
\end{align*}
\]

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

Exercise:
Modify the rest of the development to account for the works table.
% 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(e) -> ex(y,ea_num(e,y)),  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 ($\exists z. \text{employee}(x, y, z)$):
   $$\exists a. \text{ea}(a) \land \text{ea-num}(a, x) \land \text{ea-name}(a, y)$$

2. List all department numbers with their manager names
   ($\exists z, u, v. \text{department}(x, z, u) \land \text{employee}(u, y, v)$):
   $$\exists e. \text{ea}(e) \land \text{ea-dept}(e, d) \land \text{da-num}(d, x) \land \text{da-mgr}(d, f) \land \text{ea-name}(f, y)$$

Needs “departments have at least one employee”. 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 ($\exists z.\text{employee}(x, y, z)$):
   \[
   \exists a.\text{ea}(a) \land \text{ea-num}(a, x) \land \text{ea-name}(a, y)
   \]
   or, in C-like syntax: for $a$ in ea do
   
   $x := a->\text{num}$;
   $y := a->\text{name}$;

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

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

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

2. List all department numbers with their manager names

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

$$\exists a.\text{ea}(a) \land \text{ea-dept}(a, d)$$
$$\land \text{da-num}(d, x) \land \text{da-mgr}(d, f) \land \text{ea-name}(f, y)$$
$$\Rightarrow \text{needs "departments have at least one employee".}$$
$$\Rightarrow \text{needs duplicate elimination during projection.}$$

$$\exists a.\text{ea}(a) \land \text{ea-dept}(a, d) \land \text{ea-name}(a, y)$$
$$\land \text{da-num}(d, x) \land \text{da-mgr}(d, f) \land \text{compare}(e, f)$$
$$\Rightarrow \text{needs "managers work in their own departments".}$$
$$\Rightarrow \text{NO duplicate elimination during projection.}$$
What can this do: navigating pointers

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

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

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

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

2. List all department numbers with their manager names

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

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

   $\Rightarrow$ needs “departments have at least one employee”.

   ...needs duplicate elimination during projection.

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

   $\Rightarrow$ 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 ($\exists z. \text{employee}(x, y, z)$):

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

2. List all department numbers with their manager names

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

   $$\exists e, d, f. \text{ea}(e) \land \text{ea-dept}(e, d)$$
   $$\land \text{da-num}(d, x) \land \text{da-mgr}(d, f) \land \text{ea-name}(f, y)$$
   $$\Rightarrow$$ needs “departments have at least one employee”.
   
   ... needs *duplicate elimination* during projection.

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

Hash Array | Separate Chaining Linked Lists | Dept Records

i: ...
i: ...
n: ...
j: ⊥
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:

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

$$\text{ex}(x_6.\text{Hash}(p, x_6) \land \text{ex}(x_5.\text{hasharraylookup}(x_6, x_5))$$

$$\land \text{ex}(x_4.\text{listscan}(x_5, x_4) \land \text{da-name}(x_4, p)$$

$$\land \text{da-num}(x_4, x_1) \land \text{ex}(x_3.(\text{da-mgr}(x_4, x_3)$$

$$\land \text{ea-name}(x_3, x_2))))))))}$$
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:

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

\[ \text{ex}(x_6.\text{Hash}(p, x_6) \land \text{ex}(x_5.(\text{hasharraylookup}(x_6, x_5) \land \text{listscan}(x_5, x_4) \land \text{da-name}(x_4, p) \land \text{da-num}(x_4, x_1) \land \text{ex}(x_3.(\text{da-mgr}(x_4, x_3) \land \text{ea-name}(x_3, x_2)))))) \]
What can this do: two-level store

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

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

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

... more examples in ...
D UP LICATES AND P OST-PROCESSING
Assume $\langle S_L \cup S_P, \Sigma \rangle$ is a physical design and $Q^c[Q]$ a query plan. Then the following rewrite rules hold.

$$Q^c[\{ R(x_1, \ldots, x_k) \}] \leftrightarrow Q^c[R(x_1, \ldots, x_k)]$$
$$Q^c[\{ Q_1 \land Q_2 \}] \leftrightarrow Q^c[\{ Q_1 \} \land \{ Q_2 \}]$$
$$Q^c[\{ \exists x. Q_1 \}] \overset{C_1}{\leftrightarrow} Q^c[\exists x. \{ Q_1 \}]$$
$$Q^c[\{ \neg Q_1 \}] \leftrightarrow Q^c[\neg Q_1]$$
$$Q^c[\neg \{ Q_1 \}] \leftrightarrow Q^c[\neg Q_1]$$
$$Q^c[\{ Q_1 \lor Q_2 \}] \overset{C_2}{\leftrightarrow} Q^c[\{ Q_1 \} \lor \{ Q_2 \}]$$

where $C_1 = \Sigma \cup \{ Q^c \land Q_1[y_1/x] \land Q_1[y_2/x] \} \models (y_1 \approx y_2)$
$$C_2 = \Sigma \cup \{ Q^c \} \models (Q_1 \land Q_2) \rightarrow \bot$$
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 $Uq_p(Q^c)$ abstracting properties of variables within the context is defined as follows.

$$Uq_p(Q^c) = \begin{cases} 
\top & Q^c = "[]" \\
Uq(Q_2) \land Uq_p(Q^c_1) & Q^c = "Q^c_1[Q_2 \land [\]"
\text{ or } "Q^c_1[[\] \land Q_2]"
\\
\exists x. Uq_p(Q^c_1) & Q^c = "Q^c_1[\exists x.[\]]"
\\
Uq_p(Q^c_1) & Q^c = "Q^c_1[{[\}]}, "Q^c_1[\lnot[\]"\text{ or } "Q^c_1[[\] \lor Q_2]"
\end{cases}$$

⇒ extract as much information from $Q$ and $Q_1$ as possible for $C_1$ and $C_2$. 
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 $Uq_p(Q^c)$ abstracting properties of variables within the context is defined as follows:

$$Uq_p(Q^c) = \begin{cases} 
\top & Q^c = "[\]" \\
Uq(Q_2) \land Uq_p(Q^c_1) & Q^c = "Q^c_1 [Q_2 \land [\]]" \text{ or } "Q^c_1 [\] \land Q_2" \\
\exists x. Uq_p(Q^c_1) & Q^c = "Q^c_1 [\exists x. [\]]" \\
Uq_p(Q^c_1) & Q^c = "Q^c_1 [\{} [\] \}, "Q^c_1 [\neg [\]]", "Q^c_1 [Q_2 \lor [\]]" \\
& \text{ or } "Q^c_1 [\] \lor Q_2" 
\end{cases}$$

⇒ extract as much information from $Q$ and $Q_1$ as possible for $C_1$ and $C_2$. 

David Toman (University of Waterloo)
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 $Uq_p(Q^c)$ abstracting properties of variables within the context is defined as follows.

$$
Uq_p(Q^c) \equiv \begin{cases} 
\top & Q^c = "[]"

Uq(Q_2) \wedge Uq_p(Q^c_1) & Q^c = "Q^c_1[Q_2 \wedge []]" \text{ or } "Q^c_1[[] \wedge Q_2]"

\exists x. Uq_p(Q^c_1) & Q^c = "Q^c_1[\exists x.[]]"

Uq_p(Q^c_1) & Q^c = "Q^c_1[[[]]], "Q^c_1[[\neg []]], "Q^c_1[Q_2 \lor []]"

\text{ or } "Q^c_1[[] \lor Q_2]"
\end{cases}
$$

⇒ extract as much information from $Q$ and $Q_1$ as possible for $C_1$ and $C_2$. 

David Toman (University of Waterloo)
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
  ⇒ creating hashtables = extra physical design/on the fly decision

• Merge(-Sort):
  ⇒ ????
  ⇒ 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
  - creating hashtables = extra physical design/on the fly decision

- Merge(-Sort):
  - sorting = extra physical design/on the fly decision
What about Merge-Joins et al??

Join Algorithms (in typical DBMS):

- **Block Nested Loops:**
  - $\Rightarrow$ takes care of block access (done);

- **Hash:**
  - $\Rightarrow$ free if appropriate hashtable(s) already exist
  - $\Rightarrow$ creting hashtables = extra physical design/on the fly decision

- **Merge(-Sort):**
  - $\Rightarrow$ ????
  - $\Rightarrow$ 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
  - ⇒ creating 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)

<table>
<thead>
<tr>
<th>A:</th>
<th>1</th>
<th>3</th>
<th>6</th>
<th>8</th>
<th>11</th>
<th>17</th>
<th>...</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>B:</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>11</td>
<td>...</td>
<td>55</td>
<td></td>
</tr>
</tbody>
</table>
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)

\[
A: 1 \ 3 \ 6 \ 8 \ 11 \ 17 \ \ldots \ 50 \\
B: 3 \ 4 \ 5 \ 6 \ 11 \ \ldots \ 55
\]

1 \ < \ 3: \ next \ A
Merge-Joins Solution(s)

IDEA:

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

Example (Joining two sorted arrays with distinct values)

\[
A: \begin{array}{cccccccccc}
1 & 3 & 6 & 8 & 11 & 17 & \ldots & 50 \\
\end{array}
\hspace{1cm}
B: \begin{array}{cccccccccc}
3 & 4 & 5 & 6 & 11 & \ldots & 55 \\
\end{array}
\]
Merge-Joins Solution(s)

IDEA:

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

Example (Joining two sorted arrays with distinct values)

<table>
<thead>
<tr>
<th>A</th>
<th>1</th>
<th>3</th>
<th>6</th>
<th>8</th>
<th>11</th>
<th>17</th>
<th>...</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>11</td>
<td>...</td>
<td>55</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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)

\[
\begin{array}{c}
A: 1\ 3\ 6\ 8\ 11\ 17\ \ldots\ 50 \\
B: 3\ 4\ 5\ 6\ 11\ \ldots\ 55
\end{array}
\]

\[6 > 4: \text{next } B\]
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)

\[
A : 1 \ 3 \ 6 \ 8 \ 11 \ 17 \ \ldots \ 50 \\
\uparrow \\
B : 3 \ 4 \ 5 \ 6 \ 11 \ \ldots \ 55 \\
\uparrow \\
6 > 5: \text{next } B
\]
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)

\[ A : 1 \ 3 \ 6 \ 8 \ 11 \ 17 \ \ldots \ 50 \]
\[ B : 3 \ 4 \ 5 \ 6 \ 11 \ \ldots \ 55 \]

out 6: next B
Merge-Joins Solution(s)

IDEA:
• improve *ordered* access paths with *fingers*
  \( \Rightarrow \) modifies the behaviour of *get-first* depending on a parameter
• use standard Nested Loops Join

Example (Joining two sorted arrays with distinct values)

\[
A : \begin{array}{cccccc}
1 & 3 & 6 & 8 & 11 & 17 & \ldots & 50 \\
\end{array}
\]

\[
B : \begin{array}{cccccc}
3 & 4 & 5 & 6 & 11 & \ldots & 55 \\
\end{array}
\]

\( \uparrow \) \( \uparrow \)

6 < 11: next A
Merge-Joins Solution(s)

**IDEA:**

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

**Example (Joining two sorted arrays with distinct values)**

\[
\begin{array}{c}
A : 1 \ 3 \ 6 \ 8 \ 11 \ 17 \ \ldots \ 50 \\
B : 3 \ 4 \ 5 \ 6 \ 11 \ \ldots \ 55
\end{array}
\]

\[8 < 11: \text{next } A\]
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)

\[
A : \begin{array}{cccccccc}
1 & 3 & 6 & 8 & 11 & 17 & \ldots & 50 \\
\end{array}
\]

\[
B : \begin{array}{cccccccc}
3 & 4 & 5 & 6 & 11 & \ldots & 55 \\
\end{array}
\]

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.
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)
  - seamlessly 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)
• seamlessly 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]
...```
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.

1. There entities called process and file.
2. There are attributes called pno, pname, uname, and fname.
3. Each process entity has attributes pno, pname and uname.
4. Each file entity has attribute fname.
5. Processes are identified by their pno.
6. Files are identified by their fname.
7. There is a relationship uses between processes and files.
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.
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-structs 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, fss correspond appropriately to (open) file entities.

14 file-struct field of task-struct is an array of fds; 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.
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-structs, 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?
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-structs, 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?
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-structs, 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 \textit{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 approaches
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 approaches
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)