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 \( \text{emppages}/1/0 \) and \( \text{emprecords}/2/1 \):

- \( \text{emppages} \) returns (sequentially) disk pages containing \( \text{emp} \) records, and
- \( \text{emprecords} \) given a disc page, returns \( \text{emp} \) 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.

... more examples in ...
DUPLICATES AND POST-PROCESSING
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Write $Q[Q_1]$ to denote this and call $Q[]$, in which $Q_1$ has been replaced by a placeholder “[]”, a *context*.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.

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

⇒ extract as much information from $Q$ and $Q_1$ as possible for $C_1$ and $C_2$. 
Assume $Q$ is a query plan that contains a subplan $Q_1$.

Write $Q[Q_1]$ to denote this and call $Q[\cdot]$, in which $Q_1$ has been replaced by a placeholder “[ ]”, a context.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) \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 [\]]" \\
or "Q^c_1[[\]] \lor Q_2"
\end{cases}
$$

⇒ extract as much information from $Q$ and $Q_1$ as possible for $C_1$ and $C_2$. 
Eliminating Duplicate Elimination (cont’d)

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.

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

where $C_1 = \Sigma \cup \{Q^c \land Q_1[y_1/x] \land Q_1[y_2/x]\} \models (y_1 \approx y_2)$

$C_2 = \Sigma \cup \{Q^c\} \models (Q_1 \land Q_2) \rightarrow \bot$
Cut Operator

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

\[
\{ \exists y, z. (\text{emp}(x, y, z) \land (\text{correct}(p_1, z) \lor \text{correct}(p_2, z))) \}\.
\]

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

\[
\exists y, z. (\text{emp}(x, y, z) \land [(\text{correct}(p_1, z) \lor \text{correct}(p_2, z))]_1 \lor \neg_1)).
\]

function \([Q_1]\)\_first
cut/ := false
return \( Q_1 \)\_first

function \([Q_1]\)\_next
if cut/ return false
return \( Q_1 \)\_next

function \(!i\)\_first
cut/ := true
return true

function \(!i\)\_next
return false
Cut Operator

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

\[ \{\exists y, z. (\text{emp}(x, y, z) \land (\text{correct}(p_1, z) \lor \text{correct}(p_2, z)))\}\].

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

\[ \exists y, z. (\text{emp}(x, y, z) \land [(\text{correct}(p_1, z) \lor \text{correct}(p_2, z))]_1 \land \neg_1)\].

function \([Q_1]\)_\text{-first}

cut/ := false

return \(Q_1\)_\text{-first}

function \([Q_1]\)_\text{-next}

if cut/ return false

return \(Q_1\)_\text{-next}

function \(!i\)_\text{-first}

cut/ := true

return true

function \(!i\)_\text{-next}

return false
Cut Operator

Query with two parameters checking for correct salaries:

\[
\{ \exists y, z. (\text{emp}(x, y, z) \land (\text{correct}(p_1, z) \lor \text{correct}(p_2, z))) \}\.
\]

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

\[
\exists y, z. (\text{emp}(x, y, z) \land [(\text{correct}(p_1, z) \lor \text{correct}(p_2, z))]_1 \land !_1))
\]

function \([Q_1]_i\)-first

\[
cut_i := \text{false} \\
\text{return } Q_1\text{-first}
\]

function \(!i\)-first

\[
cut_i := \text{true} \\
\text{return } \text{true}
\]

function \([Q_1]_i\)-next

\[
\text{if } cut_i \text{ return } \text{false} \\
\text{return } Q_1\text{-next}
\]

function \(!i\)-next

\[
\text{return } \text{false}
\]
Incremental Query Context

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

$$Uq_{ip}(Q^c) \equiv \begin{cases} \top & Q^c = "[\]" \\ Uq(Q_2) \land Uq_{ip}(Q_1^c) & Q^c = "Q_1^c[Q_2 \land []]" \\ \exists x. Uq_{ip}(Q_1^c) & Q^c = "Q_1^c[\exists x. []]" \\ Uq_{ip}(Q_1^c) & Q^c = "Q_1^c[[[]]], "Q_1^c[\neg []], "Q_1^c[Q_2 \lor []], "Q_1^c[[] \lor Q_2]" or "Q_1^c[[] \land Q_2]" \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 $\langle S_L \cup S_P, \Sigma \rangle$ is a physical design and $Q^c[\{Q_1\} \land Q_2]$ a query plan. Then the following rewrite rule holds.

$$Q^c[\{Q_1\} \land Q_2] \iff_{C_1} Q^c[[\{Q_1\}]_\ell \land (Q_2 \land !\ell)]$$

$C_1$ corresponds to the following condition, where $\text{Out}(Q_1) = \{x_1, \ldots, x_k\}$ and where each $y_i$ and $z_j$ are fresh variable names not occurring in $Q^c$, $Q_1$ or $Q_2$.

$$\Sigma \cup \{U_{q_{ip}}(Q^c) \land U_q((Q_1 \land Q_2)[y_1/x_1, \ldots, y_k/x_k]) \land U_q((Q_1 \land Q_2)[z_1/x_1, \ldots, z_k/x_k])\}$$

$$\models (y_1 \approx z_1) \land \cdots \land (y_k \approx z_k)$$
SORTED ACCESS
Join Algorithms (in typical DBMS):

- Block Nested Loops:
  - takes care of \textit{block} access (done);

- Hash:
  - free if appropriate \textit{hashtable(s)} already exist
  - creating hashtables = extra physical design/on the fly decision

- Merge(-Sort):
  - \textit{?? ??} \iff NOW
  - sorting = extra physical design/on the fly decision
What about Merge-Joins et al??

Join Algorithms (in typical DBMS):

- **Block Nested Loops:**
  \[ \Rightarrow \text{takes care of block access (done)}; \]

- **Hash:**
  \[ \Rightarrow \text{free if appropriate hashtable(s) already exist} \]
  \[ \Rightarrow \text{creating hashtables = extra physical design/on the fly decision} \]

- **Merge(-Sort):**
  \[ \Rightarrow \text{????} \equiv \text{NOW} \]
  \[ \Rightarrow \text{sorting = extra physical design/on the fly decision} \]
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:**
  ⇒ 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)

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

\[
\begin{align*}
A : & \rightarrow 1 \ 3 \ 6 \ 8 \ 11 \ 17 \ \ldots \ 50 \\
B : & \rightarrow 3 \ 4 \ 5 \ 6 \ 11 \ \ldots \ 55 \\
\end{align*}
\]

1 < 3: 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)

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

\[3 = 3: \text{next } B\]
Merge-Joins Solution(s)

IDEA:

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

Example (Joining two sorted arrays with distinct values)

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

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)**

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

6 > 4: next B
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)

\[
A : \begin{array}{cccccccccc}
1 & 3 & 6 & 8 & 11 & 17 & \cdots & 50 \\
\end{array}
\quad B : \begin{array}{cccccccccc}
3 & 4 & 5 & 6 & 11 & \cdots & 55 \\
\end{array}
\]

6 > 5: next B
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)

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

out 6: next B
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)

\[
\begin{align*}
A : & \quad 1 \quad 3 \quad 6 \quad 8 \quad 11 \quad 17 \quad \ldots \quad 50 \\
B : & \quad 3 \quad 4 \quad 5 \quad 6 \quad 11 \quad \ldots \quad 55 \\
\end{align*}
\]

\[\uparrow\quad 6 < 11: \text{next A}\]
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
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)

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

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

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.
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

**How Well are we doing?**

- *simulates a merge join* provided the arrays are sorted
  \[ \Rightarrow \] B **must be sorted and finger-modified** (i.e., has an parameter)
  \[ \Rightarrow \] 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
  \[ \Rightarrow \] 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]
...

What can it do?
**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.

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.
**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.
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 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)
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)