Topics in Database Systems: Modern DBMS
CS848 Spring 2021

David Toman

BASIC DESIGNS
(and an overview of standard techniques)
Big Picture

Definability and Rewriting

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

Logical Schema and User Queries
Big Picture

Definability and Rewriting

Queries: range-restricted FOL over $S_L$ definable w.r.t. $\Sigma$ and $S_A$

Schema: range-restricted FOL $\Sigma := \Sigma^L \cup \Sigma^{LP} \cup \Sigma^P$

Data: CWA (complete information for $S_A$ symbols)

### 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>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 ($S_L$) --> Compiler --> Executable --> Answers
<table>
<thead>
<tr>
<th>$\psi$</th>
<th>$\psi$ (Relational Algebra over $S_A$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Sigma$</td>
<td>$\Sigma$ (instance of) $S_A$</td>
</tr>
</tbody>
</table>

Schema ($S_L \cup S_P$)  
Data ($S_A \subseteq S_P$)
```
Big Picture

Definability and Rewriting

<table>
<thead>
<tr>
<th>Queries</th>
<th>range-restricted FOL over $S_L$ definable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schema</td>
<td>range-restricted FOL $\Sigma := \Sigma^L \cup \Sigma^{LP} \cup \Sigma^{LP2}$</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

\[
\begin{align*}
\text{Query} (S_L) & \xrightarrow{\psi} \text{Compiler} \quad \Sigma \quad \text{(Relational Algebra over } S_A) \quad \text{Executable} \\
\text{Schema} (S_L \cup S_P) & \quad \Sigma \\
\text{Data} (S_A \subseteq S_P) & \quad \text{(instance of) } S_A \quad \text{Answers}
\end{align*}
\]
QUERY COMPILATION

PART I: PLANS AS FORMULAE AND STANDARD DESIGN
Queries over a Physical Design

Issues to resolve (today)

- What “formulas” do qualify as plans?
  - 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:

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

Notes:

- access path: \text{basetable}/4/0;
- additional \textit{r} attribute: address in physical storage

Query Plan:

\[
\text{q}(v0,v1,v2) \leftrightarrow \text{ex}(v3, \text{basetable}(v3,v0,v1,v2))
\]
Creating Table(s) and Base File(s)

Specification:

\[
\begin{align*}
\textbf{constraints} & : \\
\text{table}(x,y,z) & \iff \text{ex}(r,\text{basetable}(r,x,y,z)), \\
\textbf{query} & : \\
\text{q}(x,y,z) & \iff \text{table}(x,y,z)
\end{align*}
\]

Notes:

- **access path**: basetable/4/0;
- **additional** \( r \) **attribute**: address in physical storage

Query Plan:

\[
\text{q}(v0,v1,v2) \iff \text{ex}(v3,\text{basetable}(v3,v0,v1,v2))
\]
Access Path Code Templates

Array of records (C-structs)

Pseudo-code templates realizing a \texttt{first/next} protocol:

\begin{Verbatim}
function basetable-first()
  \begin{align*}
    & i := 0 \\
    & \text{return basetable-next }
  \end{align*}
\end{Verbatim}

\begin{Verbatim}
function basetable-next()
  \begin{align*}
    & \text{if } (i \geq N) \text{ return false} \\
    & x := \text{btarray}[i].xname \\
    & y := \text{btarray}[i].yname \\
    & z := \text{btarray}[i].zname \\
    & r := i++; \\
    & \text{return true}
  \end{align*}
\end{Verbatim}

⇒ \textbf{assuming} struct \{ \text{int } xname, yname, zname \} \text{ btarray}[N]

⇒ \textbf{variable } i \text{ renamed for each occurrence of } \texttt{basetable} \text{ in a plan.}

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

- \(x, y,\) and \(z\) to communicate the contents of \texttt{btarray}.
- \(i\) (and \(N\)) record scanning status (and size) of \texttt{btarray}.

\textbf{Note:} AP code (templates) for access paths must be provided.
Access Path Code Templates

Array of records (C-structs)

Pseudo-code templates realizing a first/next protocol:

\[
\text{function basetable-first()}
\]
\[
\quad i := 0
\]
\[
\quad \text{return basetable-next}
\]

\[
\text{function basetable-next()}
\]
\[
\quad \text{if } (i \geq N) \text{ return false}
\]
\[
\quad x := \text{btarray}[i].xname
\]
\[
\quad y := \text{btarray}[i].yname
\]
\[
\quad z := \text{btarray}[i].zname
\]
\[
\quad r := i++;
\]
\[
\quad \text{return true}
\]

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

⇒ variable \( i \) renamed for each occurrence of \text{basetable} in a plan.

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

1 \quad x, y and z to communicate the contents of \text{btarray}.

2 \quad i (and \( N \)) record scanning status (and size) of \text{btarray}.

Note: AP code (templates) for access paths must be provided.
Pseudo-code templates realizing a first/next protocol:

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

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

⇒ assuming struct { int xname, yname, zname } btarray[N]
⇒ variable i renamed for each occurrence of basetable in a plan.

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

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

**Note:** AP code (templates) for access paths must be provided.
1. Built-in “operations”:
   - arithmetic (\(\text{plus} / 3 / 2\), \(\text{times} / 3 / 2\), etc.)
   - string manipulation (\(\text{concat} / 3 / 2\), \(\text{substr} / 4 / 3\), etc.)
   - ...

2. data type tests (\(\text{is-integer} / 1 / 1\))

3. pointer dereference and field extraction from records

4. (page) reads from external storage

5. ...

(More Esoteric) Access Paths
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
  if empty \(S\) return false
  add \((x_1, \ldots, x_n)\) to \(S\)
return true
return false

function \(\{Q_1\}\)-next
while \(Q_1\)-next do
  if not \((x_1, \ldots, x_n) \in S\)
    add \((x_1, \ldots, x_n)\) to \(S\)
return true
return false
Conjunctive Query Plans: Semantics

function \((Q_1 \land Q_2)\)-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
**General Query Plans: Syntax**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (Q_1 \lor Q_2) )-first</td>
<td>( (Q_1 \lor Q_2) )-flag := true if ( Q_1 )-first return true (( Q_1 \lor Q_2 ))-flag := false return ( Q_2 )-first</td>
</tr>
<tr>
<td>( (Q_1 \lor Q_2) )-next</td>
<td>if ( (Q_1 \lor Q_2) )-flag if ( Q_1 )-next return true ( (Q_1 \lor Q_2) )-flag := false return ( Q_2 )-next</td>
</tr>
<tr>
<td>( \neg Q_1 )-first</td>
<td>if ( Q_1 )-first return false return true</td>
</tr>
<tr>
<td>( \neg Q_1 )-next</td>
<td>return false</td>
</tr>
</tbody>
</table>
What’s Missing?

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

...we touch on many of these in subsequent lectures.
What’s Missing?

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

...we touch on many of these in subsequent lectures
Adding an (search) index

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

Specification:

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

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

Notes:

- **access paths**: `basetlookup/4/1, indexx(y)/2/1;`
Adding an (search) index (cont)

\[ q(x, y) \iff \exists [z, v, w], \]
\[ \text{table}(x, v, z) \text{ and } \text{table}(z, w, y) \]

Possible plans:

Table Scans:

\[ q(v_{11}, v_{12}) \iff \exists [v_{12}, v_{14}, v_{15}, v_{16}, v_{17}], \]
\[ \text{basetable}(v_{13}, v_{11}, v_{14}, v_{15}) \text{ and } \]
\[ \text{basetable}(v_{16}, v_{15}, v_{17}, v_{12}) \]

Index lookup:

\[ q(v_{11}, v_{12}) \iff \exists [v_{13}, v_{14}, v_{15}, v_{16}, v_{17}], \]
\[ \text{basetable}(v_{13}, v_{11}, v_{14}, v_{15}) \text{ and } \]
\[ \text{indexx}(v_{15}, v_{16}) \text{ and } \]
\[ \text{baselookup}(v_{16}, v_{17}, v_{18}, v_{12}) \]
Adding an (search) index (cont)

$q(x,y) \iff \text{ex}([z,v,w],
\text{table}(x,v,z) \text{ and table}(z,w,y))$

Possible plans:

Table Scans:

$q(vl1,vl2) \iff \text{ex}([vl2,vl4,vl5,vl6,vl7],
\text{basetable}(vl3,vl1,vl4,vl5) \text{ and }
\text{basetable}(vl6,vl5,vl7,vl2))$

Index lookup:

$q(vl1,vl2) \iff \text{ex}([vl3,vl4,vl5,vl6,vl7],
\text{basetable}(vl3,vl1,vl4,vl5) \text{ and }
\text{indexx}(vl5,vl6) \text{ and }
\text{baselookup}(vl6,vl7,vl8,vl2))$
Adding an (search) index (cont)

\[ q(x, y) \leftrightarrow \text{ex}([z, v, w], \text{table}(x, v, z) \text{ and } \text{table}(z, w, y)) \]

Possible plans:

Table Scans:

\[ q(vl1, vl2) \leftrightarrow \text{ex}([vl2, vl4, vl5, vl6, vl7], \text{basetable}(vl3, vl1, vl4, vl5) \text{ and } \text{basetable}(vl6, vl5, vl7, vl2)) \]

Index lookup:

\[ q(vl1, vl2) \leftrightarrow \text{ex}([vl3, vl4, vl5, vl6, vl7], \text{basetable}(vl3, vl1, vl4, vl5) \text{ and } \text{indexx}(vl5, vl6) \text{ and } \text{baselookup}(vl6, vl7, vl8, vl2)) \]
Adding an (search) index (cont)

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

(with a parameter \( x \))

Possible plans:

**Index lookup:**

\[ q(vl1, vl2) \leftrightarrow \text{ex}([vl3, vl4], \\text{indexx}(vl1, vl3) \text{ and } \text{baselookup}(vl3, vl4, vl4, vl2)) \]

**Index intersection:**

\[ q(vl1, vl2) \leftrightarrow \text{ex}([vl3, vl4, vl5], \\text{indexx}(vl1, vl3) \text{ and } \text{indexy}(vl1, vl3) \text{ and } \text{baselookup}(vl3, vl4, vl5, vl2)) \]
Adding an (search) index (cont)

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

(with a parameter x)

Possible plans:

Index lookup:

\[ q(vl1, vl2) \leftrightarrow \text{ex}([vl3, vl4], \]

\[ \text{indexx}(vl1, vl3) \text{ and } \]

\[ \text{baselookup}(vl3, vl4, vl4, vl2)) \]

Index intersection:

\[ q(vl1, vl2) \leftrightarrow \text{ex}([vl3, vl4, vl5], \]

\[ \text{indexx}(vl1, vl3) \text{ and } \]

\[ \text{indexy}(vl1, vl3) \text{ and } \]

\[ \text{baselookup}(vl3, vl4, vl5, vl2)) \]
Adding an (search) index (cont)

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

(with a parameter \( x \))

Possible plans:

**Index lookup:**

\[ q(vl1, vl2) \leftrightarrow \text{ex}([vl3, vl4], \]
\[ \text{indexx}(vl1, vl3) \text{ and} \]
\[ \text{baselookup}(vl3, vl4, vl4, vl2)) \]

**Index intersection:**

\[ q(vl1, vl2) \leftrightarrow \text{ex}([vl3, vl4, vl5], \]
\[ \text{indexx}(vl1, vl3) \text{ and} \]
\[ \text{indexy}(vl1, vl3) \text{ and} \]
\[ \text{baselookup}(vl3, vl4, vl5, vl2)) \]
Index-only Plans

\[ q(x) \leftrightarrow \text{ex}(y, \text{table}(x, x, y)) \]

(with a parameter \( x \))

Possible plans:

Index lookup:

\[ q(v_1) \leftrightarrow \text{ex}([v_2, v_3, v_4], \text{index}(v_1, v_3) \text{ and } \text{baselookup}(v_3, v_4, v_4, v_2)) \]

Index intersection:

\[ q(v_1) \leftrightarrow \text{ex}([v_2, v_3, v_4, v_5], \text{index}(v_1, v_3) \text{ and } \text{index}(v_1, v_3) \text{ and } \text{index}(v_1, v_3)) \]
Index-only Plans

$q(x) \leftrightarrow \text{ex}(y, \text{table}(x, x, y))$

(with a parameter $x$)

Possible plans:

Index lookup:

$q(vl1) \leftrightarrow \text{ex}([vl2, vl3, vl4],$
\[\text{indexx}(vl1, vl3) \text{ and} \]
\[\text{baselookup}(vl3, vl4, vl4, vl2))$

Index intersection:

$q(vl1) \leftrightarrow \text{ex}([vl2, vl3, vl4, vl5],$
\[\text{indexx}(vl1, vl3) \text{ and} \]
\[\text{indexy}(vl1, vl3))$
Index-only Plans

\[ q(x) \leftrightarrow \exists y, \text{table}(x, x, y) \]  
(with a parameter \( x \))

Possible plans:

**Index lookup:**

\[ q(v_{l1}) \leftrightarrow \exists [v_{l2}, v_{l3}, v_{l4}], \]
\[ \\quad \text{indexx}(v_{l1}, v_{l3}) \text{ and } \]
\[ \quad \text{baselookup}(v_{l3}, v_{l4}, v_{l4}, v_{l2}) \]

**Index intersection:**

\[ q(v_{l1}) \leftrightarrow \exists [v_{l2}, v_{l3}, v_{l4}, v_{l5}], \]
\[ \quad \text{indexx}(v_{l1}, v_{l3}) \text{ and } \]
\[ \quad \text{indexy}(v_{l1}, v_{l3}) \]
Column Store

Specification:

\[
\begin{align*}
&\text{table}(x,y,z) \leftrightarrow \text{ex}(r, \text{basetable}(r,x,y,z)), \\
&\% \text{ indices} \\
&\text{columnx}(r,x) \leftrightarrow \text{ex}([y,z], \text{basetable}(r,x,y,z)), \\
&\text{columny}(r,y) \leftrightarrow \text{ex}([x,z], \text{basetable}(r,x,y,z)), \\
&\text{columnz}(r,z) \leftrightarrow \text{ex}([x,y], \text{basetable}(r,x,y,z)), \\
&\% \text{ keys} \\
&\text{basetable}(r,x_1,y_1,z_1) \text{ and } \text{basetable}(r,x_2,y_2,z_2) \\
&\quad \rightarrow (x_1=x_2 \text{ and } y_1=y_2 \text{ and } z_1=z_2), \\
&\% \text{ query} \\
&q(x,y,z) \leftrightarrow \text{table}(x,y,z)
\end{align*}
\]

Notes:

- APs: \texttt{columnx/2/0}, \texttt{columny/2/0}, and \texttt{columnz/2/0};
- the key constraint is necessary (why?)
Horizontal Partition (sharding)

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

Notes:

■ **APs:** hpp1/4/0, hpp2/4/0, and hpp3/4/0;
■ do we need “disjointness” of the partitions?
Subclass/Complement

Specification: [ ... 

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

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

...].

Notes:
- do we need “disjointness”? “keys”?
QUERY COMPILATION

PART II: WHAT CAN IT DO?
What can this do?

**GOAL**

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

1. linked data structures, pointers, . . .
2. access to search structures (index access and selection),
3. hash-based access to data (including hash-joins),
4. multi-level storage (aka disk/remot/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

employee

<table>
<thead>
<tr>
<th>number</th>
<th>name</th>
</tr>
</thead>
</table>

works

| emp | dept |

department

| number | name | manager |

⇒ 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

```
employee
  number
  name

works
  emp
  dept

department
  number
  name
  manager
```

⇒ 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

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

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

⇒ 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 (record declarations)

% record layout of emp and dept records and fields for:
% struct emp { int num, char[20] name, struct dept* dept };
% struct dept { int num, char[20] name, struct mgr* emp };

% ea/da addresses of emp/dept records
% access paths: ea/1/0 (linked list of employee records),
% ea_num, ea_name, ea_dept, da_num, da_name,
% da_mgr/2/1 (field extractors "->" in C)
% all attributes functional, "num" is a key;
% "dept" and "mgr" are pointers;

ea(e) -> ex(y,ea_num(e,y)), ea_num(e,y) and ea_num(e,z)-> y=z,
ea(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,

...and the same for da et al.
Lists and Pointers (logical tables)

% user predicates over records%

employee(x,y,z) \iff \text{ex}(e, \text{baseemployee}(e,x,y,z)), \% record addr%

ea(e) \iff \text{ex}([x,y,z], \text{baseemployee}(e,x,y,z)),
ea_num(e,x) \iff \text{ex}([y,z], \text{baseemployee}(e,x,y,z)),
ea_name(e,y) \iff \text{ex}([x,z], \text{baseemployee}(e,x,y,z)),
ex(d, ea\_dept(e,d) \text{ and } da\_num(d,z)) \iff \text{ex}([x,y], \text{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) \implies z=u
da_mgr(x,e) and ea\_dept(e,y) \implies 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
   ($\exists z, u, v.\text{department}(x, z, u) \land \text{employee}(u, y, v)$):
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: 

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

2. List all department numbers with their manager names

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

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

$\Rightarrow$ 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
   ($\exists z, u, v.\text{department}(x, z, u) \land \text{employee}(u, y, v)$):
   \[
   \exists e, d, f. \text{ea}(e) \land \text{ea-dept}(e, d) \\
   \land \text{da-num}(d, x) \land \text{da-mgr}(d, f) \land \text{ea-name}(f, y) \\
   \Rightarrow \text{needs “departments have at least one employee”}.
   \]
   \[
   \exists e, d. \text{ea}(e) \land \text{ea-dept}(e, d) \land \text{ea-name}(e, y) \\
   \land \text{da-num}(d, x) \land \text{da-mgr}(d, f) \land \text{compare}(e, f) \\
   \Rightarrow \text{needs “managers work in their own departments”}.
   \]
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

$i$: \ldots \rightarrow \bullet \rightarrow \bullet \rightarrow \bullet \rightarrow \bot \rightarrow D1

$j$: \ldots \rightarrow \bot \rightarrow D3

$n$: \ldots \rightarrow \bullet \rightarrow \bot \rightarrow D2
What can it do: Hashing, Lists, et al.

Hash Index on department's name:

Access paths:

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

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:

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

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:

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

\[ \text{ex}(x_6.\text{Hash}(p, x_6) \text{ and ex}(x_5.(\text{hasharraylookup}(x_6, x_5) \text{ and ex}(x_4.\text{listscan}(x_5, x_4) \text{ and da-name}(x_4, p) \text{ and da-num}(x_4, x_1) \text{ and ex}(x_3.(\text{da-mgr}(x_4, x_3) \text{ and ea-name}(x_3, x_2))))))) \]
QUERY COMPILATION

PART III: CASE STUDY (TO THINK ABOUT . . . )
The LINUX-INFO System: A Case Study

GOAL:

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

david@david-ryzen:/mnt/david/itb/itb2$ ps -efaux | head

USER   PID %CPU %MEM  VSZ   RSS TTY STAT START TIME COMMAND
root   2  0.0  0.0    0    0  ?  S   May07  0:00  [kthreadd]
root   3  0.0  0.0    0    0  ?  I<  May07  0:00  \_ [rcu_gp]
root   4  0.0  0.0    0    0  ?  I<  May07  0:00  \_ [rcu_par_gp]
root   6  0.0  0.0    0    0  ?  I<  May07  0:00  \_ [kworker/0:0H-]
root   9  0.0  0.0    0    0  ?  I<  May07  0:00  \_ [mm_percpu_wq]
root  10  0.0  0.0    0    0  ?  S   May07  0:07  \_ [ksoftirqd/0]
root  11  0.0  0.0    0    0  ?  I   May07  5:31  \_ [rcu_sched]
root  12  0.0  0.0    0    0  ?  S   May07  0:01  \_ [migration/0]

...
**LINUX-INFO System: Data and Metadata**

Example of LINUX-INFO data important to APS.

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

Example of LINUX-INFO metadata specified by APS.

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

Find the files used by process invoked by user 145.

A query plan selected by a query compiler.

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 **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. ...

Try at Home

- more query examples against employee-department schema
- 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)