Outline of Query Processing Techniques to Cover

For each we cover: a) Foundations; b) System implementations; and c) Open challenges.

1) Predefined Joins

2) Worst-case Optimal Joins (WCOJs)
   Handling Intermediate Size Growth for Cyclic Joins
   2.1. Foundations
   2.2 System Integration Approaches
       i) Index-based WCOJs (Graphflow & EmptyHeaded)
       ii) Hash-based WCOJs (Umbra)

3) Factorized Query Processing
Outline of Query Processing Techniques to Cover

For each we cover: a) Foundations; b) System implementations; and c) Open challenges.

1) Predefined Joins

2) Worst-case Optimal Joins (WCOJs)
   Handling Intermediate Size Growth for Cyclic Joins
   2.1. Foundations
   2.2 System Integration Approaches
      i) Index-based WCOJs (Graphflow & EmptyHeaded)
      ii) Hash-based WCOJs (Umbra)
         ● Optimization approaches (cost-based DP, GHD, rule-based)

3) Factorized Query Processing
Index-based Worst-case Optimal Joins
Index-based Worst-case Optimal Joins

Requires pre-sorted indexes which in graph terms map to adjacency list indexes.

Relational Systems such as **EMPTYHEADED**, **LogicBlox**, and **relationalAI** use sorted trie indexes.
Query Vertex Orderings (QVOs)
Query Vertex Orderings (QVOs)
Query Vertex Orderings (QVOs)

Plan$_1[a,b,c,d]$
Query Vertex Orderings (QVOs)

$\text{Plan}_1[a,b,c,d]$
Query Vertex Orderings (QVOs)

Plan$_1$[a,b,c,d]

Scan a → b
N-Way-Join C
Join d
Sink (accumulate)
Execution Simulation

Plan_{1}[a,b,c,d]

intermediate

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

output relation

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
</tbody>
</table>

Sink (accumulate)

Join d

N-Way-Join C

Scan a → b
Execution Simulation

Plan \(_1\)[a,b,c,d]

**intermediate**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**output relation**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Sink (accumulate)**

**Join d**

**N-Way-Join C**

**Scan a → b**

**Edges Relation**

<table>
<thead>
<tr>
<th>FROM</th>
<th>TO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1002</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>n</td>
<td>5</td>
</tr>
</tbody>
</table>

FROM → TO
Execution Simulation

Plan\textsubscript{1}[a,b,c,d]

<table>
<thead>
<tr>
<th>intermediate</th>
<th>output relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b c d</td>
<td>a b c d</td>
</tr>
<tr>
<td>Sink (accumulate)</td>
<td></td>
</tr>
<tr>
<td>Join d</td>
<td></td>
</tr>
<tr>
<td>N-Way-Join C</td>
<td></td>
</tr>
<tr>
<td>Scan a → b</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a b c d</th>
<th>2 7 11 14 17 33 52 61</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2 7 11 14 17 33 52 61</td>
</tr>
<tr>
<td>1</td>
<td>4 7 14 27</td>
</tr>
<tr>
<td>2</td>
<td>0 1 11 17 25 28</td>
</tr>
<tr>
<td>3</td>
<td>1002 1058</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>5 8 54</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>5 16 18</td>
</tr>
</tbody>
</table>
Execution Simulation

Plan $P_1[a,b,c,d]$

- **Scan** $a \rightarrow b$
- **N-Way-Join** $C$
- **Join** $d$
- **Sink** (accumulate)

Intermediate relation:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Output relation:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7</td>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>14</td>
<td>27</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>17</td>
</tr>
<tr>
<td>1002</td>
<td>1058</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>54</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>18</td>
<td></td>
</tr>
</tbody>
</table>

$Fw(0) = \{2, 7, 11, \ldots\}$
Execution Simulation

Plan_1[a,b,c,d]

Scan a → b
N-Way-Join C
Join d
Sink (accumulate)

Fw(0) = \{2, 7, 11, ...\}

intermediate
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

output relation
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7</td>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>14</td>
<td>27</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>17</td>
</tr>
<tr>
<td>1002</td>
<td>1058</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>54</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>18</td>
<td></td>
</tr>
</tbody>
</table>
Execution Simulation

Plan_1[a,b,c,d]

 intermediate output relation

\[
\begin{array}{cccc}
 a & b & c & d \\
 0 & 2 & - & - \\
\end{array}
\]

Fw(0) \cap Fw(2) = \{11, 17\}

Fw(0) = \{2, 7, 11, \ldots\}

Scan a \rightarrow b

Sink (accumulate)

Join d

N-Way-Join C

...
Execution Simulation

Plan$_1$[a,b,c,d]

\[
\begin{array}{cccc}
\text{intermediate} & a & b & c & d \\
0 & 2 & 11 & - \\
\end{array}
\]

Sink (accumulate)

\[
\begin{array}{cccc}
\text{output relation} & a & b & c & d \\
0 & 2 & 7 & 11 & 14 & 17 & 33 & 52 & 61 \\
1 & 4 & 7 & 14 & 27 \\
2 & 0 & 1 & 11 & 17 & 25 & 28 \\
3 & 1002 & 1058 \\
\ldots \\
11 & 5 & 8 & 54 \\
\ldots \\
n & 5 & 16 & 18 \\
\end{array}
\]

\[\text{Fw}(0) \cap \text{Fw}(2) = \{11, 17\}\]

\[\text{Fw}(0) = \{2, 7, 11, \ldots\}\]
Execution Simulation

Plan$_1$[a,b,c,d]

```
intermediate

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>-</td>
</tr>
</tbody>
</table>

Sink (accumulate)

Join D

N-Way-Join C

Scan a → b

output relation

```

Fw(0)=\{2,7,11,\ldots\}

Fw(11)=\{5,8,54\}

Fw(0) \cap Fw(2)=\{11,17\}

Fw(0)=\{2,7,11,\ldots\}

0 → 2 7 11 14 17 33 52 61

1 → 4 7 14 27

2 → 0 1 11 17 25 28

3 → 1002 1058

\ldots

11 → 5 8 54

\ldots

n → 5 16 18
Execution Simulation

Plan: \[a, b, c, d\]

<table>
<thead>
<tr>
<th>intermediate</th>
<th>output relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b c d</td>
<td>a b c d</td>
</tr>
<tr>
<td>0 2 11 5</td>
<td></td>
</tr>
</tbody>
</table>

Sink (accumulate)

Scan a \(\rightarrow\) b

N-Way-Join C

Join d

F_{w}(0) = \{2, 7, 11, \ldots\}

F_{w}(11) = \{5, 8, 54\}

F_{w}(0) \cap F_{w}(2) = \{11, 17\}

F_{w}(0) = \{2, 7, 11, \ldots\}

0 \rightarrow 2 7 11 14 17 33 52 61

1 \rightarrow 4 7 14 27

2 \rightarrow 0 1 11 17 25 28

3 \rightarrow 1002 1058

\ldots

11 \rightarrow 5 8 54

\ldots

n \rightarrow 5 16 18
Execution Simulation

Plan_1[a, b, c, d]

intermediate

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>5</td>
</tr>
</tbody>
</table>

output relation

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sink (accumulate)

Join d

N-Way-Join C

Scan a → b

Fw(11) = {5, 8, 54}

Fw(0) ∩ Fw(2) = {11, 17}

Fw(0) = {2, 7, 11, ...}

output relation

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>1002</td>
<td>1058</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>8</td>
<td>54</td>
</tr>
<tr>
<td>n</td>
<td>5</td>
<td>16</td>
<td>18</td>
</tr>
</tbody>
</table>
Execution Simulation

Plan_{1}[a,b,c,d]

Scan a → b

N-Way-Join C

Join d

Sink (accumulate)

intermediate

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>5</td>
</tr>
</tbody>
</table>

output relation

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>5</td>
</tr>
</tbody>
</table>

Fw(0) = {2, 7, 11, ...}
Fw(11) = {5, 8, 54}
Fw(0) ∩ Fw(2) = {11, 17}
Fw(0) = {2, 7, 11, ...}

$F_{w}(0) \cap F_{w}(2) = \{11, 17\}$

output relation

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>7</th>
<th>11</th>
<th>14</th>
<th>17</th>
<th>33</th>
<th>52</th>
<th>61</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>7</td>
<td>14</td>
<td>27</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>11</td>
<td>17</td>
<td>25</td>
<td>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1002</td>
<td>1058</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>8</td>
<td>54</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>5</td>
<td>16</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Plan\(_1[a,b,c,d]\)

Intermediate

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>5</td>
</tr>
</tbody>
</table>

Sink (accumulate)

N-Way-Join C

Join D

Scan a → b

Output relation

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>5</td>
</tr>
</tbody>
</table>

Fw(0) = \{2, 7, 11, \ldots\}

Fw(1) = \{4, 7, 14, 27\}

Fw(2) = \{0, 1, 11, 17\}

Fw(11) = \{5, 8, 54\}

Fw(0) \cap Fw(2) = \{11, 17\}

Fw(0) \cap Fw(11) = \{5, 8, 54\}

Output relation

<table>
<thead>
<tr>
<th>0</th>
<th>2</th>
<th>7</th>
<th>11</th>
<th>14</th>
<th>17</th>
<th>33</th>
<th>52</th>
<th>61</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>7</td>
<td>14</td>
<td>27</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>11</td>
<td>17</td>
<td>25</td>
<td>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1002</td>
<td>1058</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>8</td>
<td>54</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>5</td>
<td>16</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Execution Simulation

Plan\textsubscript{1}\\[a,b,c,d]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>5</td>
</tr>
</tbody>
</table>

Sink (accumulate)

N-Way-Join C

Fw(11) = \{5, 8, 54\}

Fw(0) \cap Fw(2) = \{11, 17\}

Fw(0) = \{2, 7, 11, \ldots\}

Scan a \rightarrow b

Fw(0)

\cap Fw(2) = \{11, 17\}

output relation

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>5</td>
</tr>
</tbody>
</table>

0 → 2 → 7 → 11 → 14 → 17 → 33 → 52 → 61

1 → 4 → 7 → 14 → 27

2 → 0 → 1 → 11 → 17 → 25 → 28

3 \rightarrow 1002 → 1058

\ldots

11 \rightarrow 5 → 8 → 54

\ldots

n \rightarrow 5 → 16 → 18
**Execution Simulation**

Plan$_1$[a,b,c,d]

---

**Intermediate**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>8</td>
</tr>
</tbody>
</table>

**Sink (accumulate)**

**N-Way-Join C**

**Join D**

**Scan a → b**

**Output relation**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>5</td>
</tr>
</tbody>
</table>

---

Fw(11) = {5, 8, 54}

Fw(0) ∩ Fw(2) = {11, 17}

Fw(0) = {2, 7, 11, ...}

---

0 → 2 7 11 14 17 33 52 61

1 → 4 7 14 27

2 → 0 1 11 17 25 28

3 → 1002 1058

... 11 → 5 8 54

... n → 5 16 18
Plan$_1$[a,b,c,d]

**Execution Simulation**

Intermediate

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>8</td>
</tr>
</tbody>
</table>

Output relation

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>5</td>
</tr>
</tbody>
</table>

Fw(11) = {5, 8, 54}

Fw(0) ∩ Fw(2) = {11, 17}

Fw(0) = {2, 7, 11, ...}

Fw(0) = {5, 8, 54}

Fw(0) = {5, 16, 18}
Execution Simulation

Plan \_1[a,b,c,d]

intermediate

\[
\begin{array}{cccc}
0 & 2 & 11 & 8 \\
0 & 2 & 11 & 5 \\
0 & 2 & 11 & 8 \\
\end{array}
\]

output relation

\[
\begin{array}{cccc}
2 & 7 & 11 & 14 & 17 & 33 & 52 & 61 \\
4 & 7 & 14 & 27 \\
0 & 1 & 11 & 17 & 25 & 28 \\
1002 & 1058 \\
5 & 8 & 54 \\
5 & 16 & 18 \\
\end{array}
\]

Sink (accumulate)

\[\text{Fw}(0) = \{2, 7, 11, \ldots\}\]

\[\text{Fw}(11) = \{5, 8, 54\}\]

\[\text{Fw}(0) \cap \text{Fw}(2) = \{11, 17\}\]

\[\text{Fw}(0) \cap \{5, 8, 54\}\]

Fw(0)
Execution Simulation

Plan$_1$[a,b,c,d]

intermediate

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>8</td>
</tr>
</tbody>
</table>

Sink (accumulate)

Join d

N-Way-Join C

Scan a → b

output relation

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>8</td>
</tr>
</tbody>
</table>

\[ Fw(0) = \{2, 7, 11, \ldots\} \]

\[ Fw(1) = \{4, 7, 14, 27\} \]

\[ Fw(2) = \{0, 1, 11, 17\} \]

\[ Fw(3) = \{1002, 1058\} \]

\[ Fw(11) = \{5, 8, 54\} \]

\[ Fw(0) \cap Fw(2) = \{11, 17\} \]

\[ Fw(0) \cap Fw(1) = \{2, 7, 11, \ldots\} \]

output relation

<table>
<thead>
<tr>
<th>0</th>
<th>2</th>
<th>7</th>
<th>11</th>
<th>14</th>
<th>17</th>
<th>33</th>
<th>52</th>
<th>61</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>7</td>
<td>14</td>
<td>27</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>11</td>
<td>17</td>
<td>25</td>
<td>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1002</td>
<td>1058</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>8</td>
<td>54</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>5</td>
<td>16</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

...
Execution Simulation

Plan\(_{1}[a,b,c,d]\)

\[
\begin{array}{cccc}
 a & b & c & d \\
 0 & 2 & 11 & 8 \\
 0 & 2 & 11 & 5 \\
 0 & 2 & 11 & 8 \\
\end{array}
\]

Sink (accumulate)

Join \(d\)

N-Way-Join \(C\)

Scan \(a \rightarrow b\)

intermediate

output relation

\[Fw(0) \cap Fw(2) = \{11, 17\}\]

\[Fw(11) = \{5, 8, 54\}\]

\[Fw(0) = \{2, 7, 11, \ldots\}\]

\[Fw(0) = \{4, 7, 14, 27\}\]

\[Fw(0) = \{0, 1, 11, 17, 25, 28\}\]

\[Fw(0) = \{1002, 1058\}\]

\[Fw(0) = \{5, 8, 54\}\]

\[Fw(0) = \{5, 16, 18\}\]

output relation

\[
\begin{array}{cccc}
 a & b & c & d \\
 0 & 2 & 11 & 8 \\
 0 & 2 & 11 & 5 \\
 0 & 2 & 11 & 8 \\
\end{array}
\]
Execution Simulation

Plan_1[a,b,c,d]

**Intermediate**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>54</td>
</tr>
</tbody>
</table>

**Sink (accumulate)**

**N-Way-Join C**

**Join D**

Fw(11)={5,8,54}

Fw(0) ∩ Fw(2)={11,17}

Fw(0)={2,7,11,...}

**Output relation**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>8</td>
</tr>
</tbody>
</table>

Execution Simulation

Fw(0)={2,7,11,...}

output relation

\[
\begin{align*}
0 & \rightarrow 2 & 7 & 11 & 14 & 17 & 33 & 52 & 61 \\
1 & \rightarrow 4 & 7 & 14 & 27 \\
2 & \rightarrow 0 & 1 & 11 & 17 & 25 & 28 \\
3 & \rightarrow 1002 & 1058 \\
\cdots & & & & & & & & \\
n & \rightarrow 5 & 16 & 18
\end{align*}
\]
Execution Simulation

Plan$_1$[a,b,c,d]

intermediate

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>54</td>
</tr>
</tbody>
</table>

output relation

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>54</td>
</tr>
</tbody>
</table>

Sink (accumulate)

Fw(0) = \{2, 7, 11, 14, 17, 33, 52, 61\}

Fw(1) = \{4, 7, 14, 27\}

Fw(2) = \{0, 1, 11, 17, 25, 28\}

Fw(11) = \{5, 8, 54\}

Fw(0) \cap Fw(2) = \{11, 17\}

Fw(0) = \{2, 7, 11, ...\}

output relation

<table>
<thead>
<tr>
<th>0</th>
<th>2</th>
<th>7</th>
<th>11</th>
<th>14</th>
<th>17</th>
<th>33</th>
<th>52</th>
<th>61</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>7</td>
<td>14</td>
<td>27</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>17</td>
<td>25</td>
<td>28</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1002</td>
<td>1058</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>54</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Execution Simulation

Plan$_1$[a,b,c,d]

```plaintext
intermediate

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>54</td>
</tr>
</tbody>
</table>

Sink (accumulate)

Join D

N-Way-Join C

Scan a → b

output relation

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>54</td>
</tr>
</tbody>
</table>

Fw(0) = \{2,7,11,\ldots\}

Fw(0) ∩ Fw(2) = \{11,17\}

Fw(11) = \{5,8,54\}

```

0 → 2 7 11 14 17 33 52 61
1 → 4 7 14 27
2 → 0 1 11 17 25 28
3 → 1002 1058

... 

11 → 5 8 54

... 

n → 5 16 18

UNIVERSITY OF WATERLOO
Execution Simulation

Plan_{1}[a,b,c,d]

intermediate

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>54</td>
</tr>
</tbody>
</table>

Sink (accumulate)

Join d

N-Way-Join C

Scan a → b

output relation

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>54</td>
</tr>
</tbody>
</table>

Fw(0) \cap Fw(2) = \{11, 17\}

Fw(0) = \{2, 7, 11, ...\}

output relation

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>14</td>
<td>27</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>11</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>54</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

...
Execution Simulation

Plan$_1[a,b,c,d]$

intermediate

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>54</td>
</tr>
</tbody>
</table>

Sink (accumulate)

Join d

N-Way-Join C

Scan a → b

output relation

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>54</td>
</tr>
</tbody>
</table>

Fw(0) ∩ Fw(2) = {11, 17}

Fw(0) = {2, 7, 11, ...}

output relation

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>11</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7</td>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>14</td>
<td>27</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>17</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>17</td>
<td>25</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>17</td>
<td>28</td>
</tr>
<tr>
<td>1002</td>
<td>1058</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>54</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>18</td>
<td></td>
</tr>
</tbody>
</table>
Execution Simulation

Plan_{1}[a,b,c,d]

intermediate

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>17</td>
<td>54</td>
</tr>
</tbody>
</table>

Sink (accumulate)

Join d

N-Way-Join C

Scan a → b

output relation

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
<td>54</td>
</tr>
</tbody>
</table>

\[F_w(0) \cap F_w(2) = \{11, 17\}\]

\[F_w(0) = \{2, 7, 11, \ldots\}\]

output relation

- 0: 2 7 11 14 17 33 52 61
- 1: 4 7 14 27
- 2: 0 1 11 17 25 28
- 3: 1002 1058
- 11: 5 8 54
- n: 5 16 18
Multiple possible Query Vertex Orderings (QVOs)

Plan$_1$[a,b,c,d]

- Scan a → b
- N-Way-Join C
- Join d
- Sink (accumulate)

Plan$_2$[a,b,d,c]

- Scan a → b
- Join d
- N-Way-Join C
- Sink (accumulate)
Decomposition using Binary Joins

Plan 1: [a, b, c]

Plan 2: [a, b, d]
Decomposition using Binary Joins

Plan 1: [a, b, c]
- Scan a → b
- N-Way-Join C
- Build HT a
- Probe HT a
- Join d
- Scan a
- Pipeline 0
- Sink (accumulate)

Plan 2: [a, b, d]
- Pipeline 1
Decomposition using Binary Joins

Plan 1[a,b,c]

Plan 2[a,b,d]

Scan a → b

N-Way-Join C

pipeline 0

Build HT a

Probe HT a

Sink (accumulate)

Join d

Scan a

pipeline 1
Outline of Query Processing Techniques to Cover

For each we cover: a) Foundations; b) System implementations; and c) Open challenges.

1) Predefined Joins

2) Worst-case Optimal Joins (WCOJs)
   Handling Intermediate Size Growth for Cyclic Joins
   2.1. Foundations
   2.2 System Integration Approaches
      i) Index-based WCOJs (Graphflow & EmptyHeaded)
      ii) Hash-based WCOJs (Umbra)
         - Optimization approaches (cost-based DP, GHD rule-based)

3) Factorized Query Processing
How to Pick Good Query Vertex Ordering?

Query Vertex Orderings have 2 different effects on runtime.
(1) Number of intermediate results.
(2) Direction of adjacency lists intersected.
How to Pick Good Query Vertex Ordering?

Query Vertex Orderings have two different effects on runtime.

1. Number of intermediate results.
2. Direction of adjacency lists intersected.

Plan_1[a,b,c]
How to Pick Good Query Vertex Ordering?

Query Vertex Orderings have 2 different effects on runtime.

(1) Number of intermediate results.
(2) Direction of adjacency lists intersected.

Plan$_1$[a,b,c]  Plan$_2$[b,c,a]
How to Pick Good Query Vertex Ordering?

Query Vertex Orderings have 2 different effects on runtime.
(1) Number of intermediate results.
(2) Direction of adjacency lists intersected.

Plan$_1$[a,b,c]  Plan$_2$[b,c,a]  Plan$_3$[a,c,b]
How to Pick Good Query Vertex Ordering?

Query Vertex Orderings have 2 different effects on runtime.
(1) Number of intermediate results.
(2) Direction of adjacency lists intersected.

Plan 1: [a, b, c]
Plan 2: [b, c, a]
Plan 3: [a, c, b]

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Plan 1</th>
<th>Σ Adj List sizes</th>
<th>Runtime (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web-BerkStan</td>
<td>Plan 1</td>
<td>0.5B</td>
<td>2.6</td>
</tr>
<tr>
<td></td>
<td>Plan 2</td>
<td>55B (113.8x)</td>
<td>15.2 (5.8x)</td>
</tr>
<tr>
<td></td>
<td>Plan 3</td>
<td>55B (114.0x)</td>
<td>31.6 (12.2x)</td>
</tr>
</tbody>
</table>
Cost Metric: I-Cost

Cost of a worst-case optimal plan is total intersection-cost of all operators.

Plan [a,b,c]
Cost Metric: I-Cost

Cost of a worst-case optimal plan is total **intersection-cost** of all operators.

\[ I-cost: \text{ size of intersected adj lists throughout execution.} \]

\[
\sum_{Q_{k-1} \in Q_2 \cdots Q_{m-1}} \sum_{t \in Q_{k-1}} \sum_{(i, dir) \in A_{k-1}} |t[i].dir| \\
\text{s.t. (i, dir) is accessed}
\]
Cost Metric: I-Cost

Cost of a worst-case optimal plan is total **intersection-cost** of all operators.

**I-cost:** size of intersected adj lists throughout execution.

\[
\sum_{Q_{k-1} \in Q_2 \ldots Q_{m-1}} \sum_{t \in Q_{k-1}} \sum_{(i, dir) \in A_{k-1}} |t[i].dir| \\
\text{s.t.} (i, \text{dir}) \text{ is accessed}
\]

1) number of intermediate results.
Cost Metric: I-Cost

Cost of a worst-case optimal plan is total **intersection-cost** of all operators.

I-cost: size of intersected adj lists throughout execution.

\[
\sum_{Q_{k-1} \in Q_2 \ldots Q_{m-1}} \sum_{t \in Q_{k-1}} \sum_{(i, dir) \in A_{k-1}} |t[i].dir|
\]

1) number of intermediate results.

2) size of adj. lists
Cost Metric: I-Cost

Cost of a worst-case optimal plan is total intersection-cost of all operators.

\[ \text{I-cost: size of intersected adj lists throughout execution.} \]

\[ \sum_{Q_{k-1} \in Q_2 \ldots Q_{m-1}} \sum_{t \in Q_{k-1}} \sum_{(i, \text{dir}) \in A_{k-1}} |t[i].\text{dir}| \]

1) number of intermediate results.

2) size of adj. lists

I-Cost captures both effects & # of intermediate results and size of adj lists is estimated using a subgraph catalogue. Summary-based Cardinality estimation using sampling.
Graphflow Hybrid Plan

Dynamic Programming
Cost-Based Optimizer
Graphflow Hybrid Plan

Dynamic Programming
Cost-Based Optimizer

*Difference with classic optimizer:* At each level, considers multiway joins to a query vertices.
Example Graphflow Hybrid Plan

Dynamic Programming
Cost-Based Optimizer
Example Graphflow Hybrid Plan

Dynamic Programming
Cost-Based Optimizer
Example Graphflow Hybrid Plan

Dynamic Programming
Cost-Based Optimizer

Intersect

Binary Join

Intersect

Scan Edges
Example Graphflow Hybrid Plan

Dynamic Programming Cost-Based Optimizer

Scan $a \rightarrow b$

N-Way-Join $c$

Build HT on $c$

Probe HT on $c$

N-Way-Join $e$

Scan $c \rightarrow d$

pipeline 1

Sink (accumulate)

N-Way-Join $f$

Scan $a \rightarrow b$

pipeline 0

Intersect

Binary Join

Intersect

Scan Edges
Example Graphflow Hybrid Plan

Dynamic Programming
Cost-Based Optimizer

Scan $a \rightarrow b$

N-Way-Join $c$

Build HT on $c$

N-Way-Join $e$

Scan $c \rightarrow d$

pipeline 1

Sink (accumulate)

N-Way-Join $f$

Probe HT on $c$

Intersect

Binary Join

Intersect

Scan Edges
Example Graphflow Hybrid Plan

Dynamic Programming
Cost-Based Optimizer

Scan a → b
N-Way-Join c
Build HT on C
Intersect
N-Way-Join e
Scan c → d
Sink (accumulate)
N-Way-Join f
Probe HT on C
Scan a → b
pipeline 0
HT on C
blocks
header
RID / vID → slot
1 → (0, 24)
3 → (0, 92)
...
(a,b) matches
...

Scan edges

Edges

Blocks

RID / vID → slot
1 → (0, 24)
3 → (0, 92)
...
(a,b) matches
...

header
blocks
Example EmptyHeaded Plan

Generalized Hypertree Decomposition
Cost-Based Optimizer
A GHD is similar to a syntax tree of a relational expression. Nodes represent a join and projection operation and edges indicate data dependencies.
A GHD is similar to a syntax tree of a relational expression. Nodes represent a join and projection operation and edges indicate data dependencies.
A GHD is similar to a syntax tree of a relational expression. Nodes represent a join and projection operation and edges indicate data dependencies.

EmptyHeaded picks GHD with minimum fractional hypertree width which is the maximum AGM bound of any of the leaves.
A GHD is similar to a syntax tree of a relational expression. Nodes represent a join and projection operation and edges indicate data dependencies.

EmptyHeaded picks GHD with minimum fractional hypertree width which is the maximum AGM bound of any of the leaves.
Example Graphflow Hybrid Plan

Dynamic Programming
Cost-Based Optimizer

Intersect
Binary Join
Intersect
Scan Edges

Generalized Hypertree Decomposition
Cost-Based Optimizer

Q
Q_i Q_j
Q_k Q_l
Q_k Q_l
Example Graphflow Hybrid Plan

Dynamic Programming
Cost-Based Optimizer

Generalized Hypertree Decomposition
Cost-Based Optimizer

Graphflow

EMPTHEAD
Example Graphflow Hybrid Plan

Dynamic Programming Cost-Based Optimizer

Generalized Hypertree Decomposition Cost-Based Optimizer

Binary Join

Intersect

Scan Edges

Q

Q_i

Q_j

Q_k

Q_l

WCO Subplan

WCO Subplan

WCO Subplan
Example Graphflow Hybrid Plan

Dynamic Programming Cost-Based Optimizer

Generates only WCO subplans followed by multiple binary joins.

Generalized Hypertree Decomposition Cost-Based Optimizer
Example Graphflow Hybrid Plan

Dynamic Programming
Cost-Based Optimizer

Generalized Hypertree Decomposition
Cost-Based Optimizer

Graphflow

Intersect

Binary Join

Intersect

Scan Edges
Example Graphflow Hybrid Plan

Dynamic Programming Cost-Based Optimizer

Generalized Hypertree Decomposition Cost-Based Optimizer

<table>
<thead>
<tr>
<th></th>
<th>Graphflow</th>
<th>EmptyHeaded (EH)</th>
<th>EH in Graphflow</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>24.7 secs</td>
<td>&gt; 30 mins</td>
<td>5.8 mins (14x)</td>
</tr>
</tbody>
</table>

Amazon (V=403K, E=3.4M)
Outline of Query Processing Techniques to Cover

For each we cover: a) Foundations; b) System implementations; and c) Open challenges.

1) Predefined Joins

2) Worst-case Optimal Joins (WCOJs)
   Handling Intermediate Size Growth for Cyclic Joins
   2.1. Foundations
   2.2 System Integration Approaches
      i) Index-based WCOJs (Graphflow & EmptyHeaded)
      ii) Hash-based WCOJs (Umbra)
         - Optimization approaches (cost-based DP, GHD, rule-based)

3) Factorized Query Processing
Index-based Worst-case Optimal Joins

Generic Join uses pre-sorted indexes which in graph terms map to adjacency list indexes.

Relational Systems such as EMPTYHEADED, LogicBlox, and relationalAI use sorted trie indexes.
Index-based Worst-case Optimal Joins

Generic Join uses pre-sorted indexes which in graph terms map to adjacency list indexes.

Relational Systems such as **EMPTYHEADED**, **LogicBlox**, and **relationalAI** use sorted trie indexes.

High-cost associated with keeping indexes up-to-date for a general database system!
- EmptyHeaded keeps all possible Tries with different sorts on the attributes → Not practical.
- The adjacency list indexes of Graphflow are specialized for subgraph query workloads.
Index-based Worst-case Optimal Joins

Generic Join uses pre-sorted indexes which in graph terms map to adjacency list indexes.

Relational Systems such as **EMPTYHEADED**, **LogicBlox**, and **relationalAI** use sorted trie indexes.

High-cost associated with keeping indexes up-to-date for a general database system!
- EmptyHeaded keeps all possible Tries with different sorts on the attributes → Not practical.
- The adjacency list indexes of Graphflow are specialized for subgraph query workloads.
  - What is needed?

  A generic WCOJ algorithm for a changing workload e.g., HTAP systems that does not rely on these pre-computed indexes that analytical systems can afford building and updating.
Hash-based Worst-case Optimal Joins
Hash-based Worst-case Optimal Joins

Building Hash Tries on-the-fly that can be used within the worst-case optimal join algorithm.
Hash-based Worst-case Optimal Joins

Building Hash Tries on-the-fly that can be used within the worst-case optimal join algorithm.
Hash-based Worst-case Optimal Joins

Building Hash Tries on-the-fly that can be used within the worst-case optimal join algorithm.

Attribute ordering for evaluation: [a, b, c]
Hash-based Worst-case Optimal Joins

Building Hash Tries on-the-fly that can be used within the worst-case optimal join algorithm.

Attribute ordering for evaluation: [a, b, c]
Hash-based Worst-case Optimal Joins

Building Hash Tries on-the-fly that can be used within the worst-case optimal join algorithm.

Attribute ordering for evaluation: [a, b, c]

Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels
Hash-based Worst-case Optimal Joins

Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a, b, c]
Hash-based Worst-case Optimal Joins

Attribute ordering: \([a, b, c]\)

Build Hash Tries for R1, R2, and R3 following attribute ordering \([a, b, c]\) in Trie-levels.
Build Hash Tries

Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a, b, c]

R1 (a, c)  R2 (b, c)  R3 (a, b, c)

R1

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>c</td>
</tr>
</tbody>
</table>

R2

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

R3

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Build Hash Tries

A multiway join of $K$ relations $\rightarrow$ Building $K$ Hash Tries.
Build Hash Tries

R1 \((a, c)\)

Build Hash Tries for R1, R2, and R3 following attribute ordering \([a, b, c]\) in Trie-levels.

Attribute ordering: \([a, b, c]\)
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a, b, c]
Build Hash Tries

R1 (a, c)

\[ h(a) \{ h(a_0), h(a_1), h(a_2), \ldots, h(a_n) \} \]

\[ \ldots \ldots \]

\[ \uparrow a_0 \leftarrow c_1 \downarrow a_2 \downarrow c_3 \uparrow a_6 \leftarrow c_7 \]

Attribute ordering: [a, b, c]
Build Hash Tries

Attribute ordering: [a, b, c]

R1 (a, c)

h(a) \{ h(a_0) h(a_1) h(a_2) ... h(a_n) \}

... ...

h(c) \{ h(c_4) h(c_1) = h(c_7) h(c_5) h(c_3) \}

... ...

\[ \begin{align*}
  a_2, & c_1, \quad \bullet \quad a_3, \quad c_4, \quad \bullet \quad a_0, \quad c_7 \\
  \end{align*} \]
Build Hash Tries

Attribute ordering: [a, b, c]

R1 (a, c)

h(a)

\[
\begin{align*}
\text{h}(a_0) & \quad \text{h}(a_1) & \quad \cdots & \quad \text{h}(a_7) \\
\end{align*}
\]

\[
\begin{align*}
\vdots \\
\end{align*}
\]

h(c)

\[
\begin{align*}
\text{h}(c_1) = \text{h}(c_7) & \quad \text{h}(c_4) & \quad \text{h}(c_3) \\
\end{align*}
\]

\[
\begin{align*}
\vdots \\
\end{align*}
\]

Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a, b, c]

R1 (a, c)

R2 (b, c)

R3 (a, b, c)
Build Hash Tries

R1 (a, c)

R2 (b, c)

Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a, b, c]
**Build Hash Tries**

**R1 (a, c)**

- \( h(a) \) (a₀, a₁, a₂, ..., aₙ)
- \( h(c) \) (c₁, c₂, c₃, ..., cₙ) with \( h(c₁) = h(c₇) \)

**R2 (b, c)**

- \( h(b) \) (b₀, b₁, b₂, ..., bₙ)
- \( h(c) \) as above

**Attribute ordering:** [a, b, c]

Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels.

University of Waterloo
Build Hash Tries

**R1 (a, c)**

\[ h(a_0), h(a_1), h(a_2), \ldots, h(a_n) \]

**R2 (b, c)**

\[ h(b_0), h(b_1), h(b_2), \ldots, h(b_n) \]

Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels.

Attribute ordering: [a, b, c]

- **R1**
  - (a, c)
  - a
  - c
- **R2**
  - (b, c)
  - b
  - c
- **R3**
  - (a, b, c)
  - a
  - b
  - c
Build Hash Tries

Attribute ordering: [a, b, c]

R1 (a, c)

\[
\begin{align*}
& h(a) \\
& \quad \downarrow h(a_0) | h(a_1) | h(a_2) \ldots | h(a_n) \\
& \quad \vdots \\
& h(c) \\
& \quad \downarrow h(c_4) | h(c_5) \ldots | h(c_{c_7}) \\
& \quad \vdots \\
& \quad \vdots \\
& \quad \vdots
\end{align*}
\]

\[
\begin{align*}
& a \quad \downarrow a_0 \quad | \quad a_1 \quad | \quad a_2 \quad | \quad a_3 \quad \ldots \quad | \quad a_n \quad | \quad c \quad | \quad c_1 \quad | \quad c_2 \quad | \quad c_3 \quad \ldots \quad | \quad c_{c_7}
\end{align*}
\]

R2 (b, c)

\[
\begin{align*}
& h(b) \\
& \quad \downarrow h(b_1) | h(b_2) \ldots | h(b_n) \\
& \quad \vdots \\
& h(c) \\
& \quad \downarrow h(c_4) | h(c_5) \ldots | h(c_{c_7}) \\
& \quad \vdots \\
& \quad \vdots \\
& \quad \vdots
\end{align*}
\]

\[
\begin{align*}
& b \quad \downarrow b_4 \quad \ldots \quad \ldots \quad \ldots \quad \downarrow b_n \quad \downarrow b_1 \quad \downarrow b_4 \quad | \quad b_2 \quad | \quad b_3 \quad \ldots \quad | \quad b_4 \quad | \quad c \quad | \quad c_1 \quad | \quad c_2 \quad | \quad c_3 \quad \ldots \quad | \quad c_{c_7}
\end{align*}
\]

Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

R1 (a, c) \quad R2 (b, c) \quad R3 (a, b, c)

Attribute ordering: [a, b, c]
Build Hash Tries

Attribute ordering: [a, b, c]

R1 (a, c)

R2 (b, c)

Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels
Build Hash Tries

Attribute ordering: [a, b, c]

R1 (a, c)

R2 (b, c)

R3 (a, b, c)

Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a, b, c]
Build Hash Tries

Attribute ordering: \([a, b, c]\)

R1 (a, c)

R2 (b, c)

R3 (a, b, c)

Build Hash Tries for R1, R2, and R3 following attribute ordering \([a, b, c]\) in Trie-levels
Build Hash Tries

Attribute ordering: [a, b, c]

R1 (a, c)

R2 (b, c)

R3 (a, b, c)

Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels
Build Hash Tries

R1 (a, c)

\[ h(a) \]

\[ h(a_1) h(a_2) h(a_3) \ldots h(a_n) \]

R2 (b, c)

\[ h(b) \]

\[ h(b_1) h(b_2) h(b_3) \ldots h(b_n) \]

R3 (a, b, c)

\[ h(a) \]

\[ h(a_1) h(a_2) \ldots h(a_n) \]

\[ h(b) \]

\[ h(b_1) h(b_2) \ldots h(b_n) \]

Attribute ordering: [a, b, c]
Build Hash Tries

Attribute ordering: \([a, b, c]\)

R1 (a, c)

R2 (b, c)

R3 (a, b, c)

Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a, b, c]
Build Hash Tries

R1 (a, c)

\[ h(a) \{ h(a_0), h(a_1), \ldots, h(a_n) \} \]

R2 (b, c)

\[ h(b) \{ h(b_0), h(b_1), \ldots, h(b_n) \} \]

R3 (a, b, c)

\[ h(c) \{ h(c_0), h(c_1), \ldots, h(c_n) \} \]

Attribute ordering: [a, b, c]
Probe - Enumeration Simulation

Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a, b, c]
Probe - Enumeration Simulation

Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a, b, c]

Trie Iterator Functions
up
down
next
lookup

hash
size
tuples
**Probe - Enumeration Simulation**

**I1**

- **R1 (a, c)**
  - $h(a_0), h(a_1), \ldots, h(a_7)$
  - $h(c_4), h(c_1) = h(c_7)$
  - $h(c_2), h(c_5)$

**I2**

- **R2 (b, c)**
  - $h(b_0), h(b_1), \ldots, h(b_7)$
  - $h(c_2), h(c_4)$
  - $h(c_1) = h(c_7)$

**I3**

- **R3 (a, b, c)**
  - $h(a_0), h(a_1), \ldots, h(a_n)$
  - $h(b_0), h(b_1), \ldots, h(b_n)$
  - $h(c_4), h(c_1) = h(c_7)$
  - $h(c_2), h(c_3)$

**Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels**

Attribute ordering: [a, b, c]
Probe - Enumeration Simulation

Attribute ordering: [a, b, c]

R1 (a, c)

R2 (b, c)

R3 (a, b, c)

Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a, b, c]
**Probe - Enumeration Simulation**

R1 (a, c)  

R2 (b, c)  

R3 (a, b, c)  

Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels.
Probe - Enumeration Simulation

R1 (a, c)

R2 (b, c)

R3 (a, b, c)

Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a, b, c]
Probe - Enumeration Simulation

Build Hash Tries for R1, R2, and R3 following attribute ordering \([a, b, c]\) in Trie-levels

Attribute ordering: \([a, b, c]\)

R1 \((a, c)\)

R2 \((b, c)\)

R3 \((a, b, c)\)
Probe - Enumeration Simulation

Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a, b, c]
Probe - Enumeration Simulation

Build Hash Tries for R1, R2, and R3 following attribute ordering \([a, b, c]\) in Trie-levels

Attribute ordering: \([a, b, c]\)
Probe - Enumeration Simulation

Attribute ordering: [a, b, c]

R1 (a, c)
R2 (b, c)
R3 (a, b, c)

Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a, b, c]
Probe - Enumeration Simulation

Attribute ordering: [a, b, c]

R1 (a, c)

R2 (b, c)

R3 (a, b, c)

Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

I1

I2

I3

smaller size
Probe - Enumeration Simulation

Attribute ordering: [a, b, c]

R1 (a, c)

R2 (b, c)

R3 (a, b, c)

Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a, b, c]
Probe - Enumeration Simulation

Build Hash Tries for R1, R2, and R3 following attribute ordering \([a, b, c]\) in Trie-levels

Attribute ordering: \([a, b, c]\)
Probe - Enumeration Simulation

Build Hash Tries for R1, R2, and R3 following attribute ordering \([a, b, c]\) in Trie-levels

Attribute ordering: \([a, b, c]\)

R1 (a, c)

R2 (b, c)

R3 (a, b, c)
Probe - Enumeration Simulation

smaller size

I1

R1 (a, c)

h(a)

h(a_0), h(a_1), h(a_2), ..., h(a_7)

... h(a)

h(c)

h(c_4), h(c_7) = h(c_7)

h(c_2), h(c_5)

... h(c)

a_0, c_1, a_2, c_4, a_3, c_7

R2 (b, c)

h(b)

h(b_1), h(b_2), h(b_3), ..., h(b_7)

... h(b)

h(c)

h(c_4), h(c_2), ..., h(c_1) = h(c_7)

h(c_5)

... h(c)

b_4, c_4, b_4, c_2, ..., b_4, c_1, b_4, c_7

R3 (a, b, c)

h(a)

h(a_0), h(a_1), ..., h(a_6)

... h(a)

h(b)

h(b_1), h(b_2), h(b_3), ..., h(b_7)

... h(b)

h(c)

h(c_4), h(c_5), ..., h(c_1) = h(c_7)

h(c_2)

... h(c)

b_4, c_4, b_4, c_2, ..., b_4, c_1, b_4, c_7

Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a, b, c]

R1

R2

R3

a

b

a

c

b


Probe - Enumeration Simulation

```
R1 (a, c)
R2 (b, c)
R3 (a, b, c)
```

Attribute ordering: [a, b, c]

Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

I1

I2

I3

smaller size
Probe - Enumeration Simulation

Build Hash Tries for R1, R2, and R3 following attribute ordering \([a, b, c]\) in Trie-levels.

Attribute ordering: \([a, b, c]\)

I1
- R1 \((a, c)\)
- I1
- h(a)
  - h(a_0)h(a_1)h(a_2)h(a_3)h(a_7)
- h(c)
  - h(c_1)h(c_2)=h(c_7)

I2
- R2 \((b, c)\)
- I2
- h(b)
  - h(b_0)h(b_1)h(b_2)h(b_3)h(b_7)
- h(c)
  - h(c_0)h(c_1)=h(c_2)

I3
- R3 \((a, b, c)\)
- I3
- h(a)
  - h(a_0)h(a_1)h(a_2)h(a_3)
- h(b)
  - h(b_0)h(b_1)h(b_2)h(b_3)h(b_7)
- h(c)
  - h(c_0)h(c_1)=h(c_7)

R1 \((a, c)\)
R2 \((b, c)\)
R3 \((a, b, c)\)

h(a)
- a_n b_4 c_4
- h(b)
  - a_n b_4 c_4
- h(c)
  - a_n b_4 c_5
Probe - Enumeration Simulation

Attribute ordering: [a, b, c]

Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

R1 (a, c)
R2 (b, c)
R3 (a, b, c)
Probe - Enumeration Simulation

Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels.

Attribute ordering: [a, b, c]

**R1 (a, c)**

- Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels.

**R2 (b, c)**

- Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels.

**R3 (a, b, c)**

- Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels.

Output relation:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_n</td>
<td>b_4</td>
<td>c_4</td>
<td></td>
</tr>
</tbody>
</table>
## Probe - Enumeration Simulation

### Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

<table>
<thead>
<tr>
<th>Attribute Ordering</th>
<th>Build Hash Tries</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1 (a, c)</td>
<td>R1</td>
</tr>
<tr>
<td>R2 (b, c)</td>
<td>R2</td>
</tr>
<tr>
<td>R3 (a, b, c)</td>
<td>R3</td>
</tr>
</tbody>
</table>

### Attribute Ordering: [a, b, c]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>b</td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### R1 (a, c)

#### I1

- h(a)
- h(a) \(\rightarrow\) a, c
- h(c) \(\Rightarrow\) h(c)
- h(c) = h(c)

### R2 (b, c)

#### I2

- h(b)
- h(b) \(\rightarrow\) b, c
- h(c) \(\Rightarrow\) h(c)
- h(c) = h(c)

### R3 (a, b, c)

#### I3

- h(a)
- h(a) \(\rightarrow\) a, b, c
- h(b) \(\Rightarrow\) h(b)
- h(c) \(\Rightarrow\) h(c)
- h(c) = h(c)

### Output Relation

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
</tbody>
</table>

![Umbra Logo]
Probe - Enumeration Simulation

Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a, b, c]

output relation

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_n</td>
<td>b_4</td>
<td>c_4</td>
</tr>
</tbody>
</table>

I1

I2

I3

smaller size
Probe - Enumeration Simulation

R1 (a, c)

R2 (b, c)

R3 (a, b, c)

Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a, b, c]

output relation

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_n</td>
<td>b_4</td>
<td>c_4</td>
</tr>
</tbody>
</table>

smaller size

I1

I2

I3
Probe - Enumeration Simulation

Build Hash Tries for R1, R2, and R3 following attribute ordering \([a, b, c]\) in Trie-levels

Attribute ordering: \([a, b, c]\)

I1

R1 \((a, c)\)

I2

R2 \((b, c)\)

I3

R3 \((a, b, c)\)

output relation

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a_n)</td>
<td>(b_4)</td>
<td>(c_4)</td>
</tr>
</tbody>
</table>
Probe - Enumeration Simulation

Build Hash Tries for R1, R2, and R3 following attribute ordering \([a, b, c]\) in Trie-levels

Attribute ordering: \([a, b, c]\)

Output relation:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>4</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>4</td>
<td>7</td>
</tr>
</tbody>
</table>

R1 \((a, c)\)

R2 \((b, c)\)

R3 \((a, b, c)\)
**Probe - Enumeration Simulation**

**R1** (a, c)  
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

**R2** (b, c)  
Attribute ordering: [a, b, c]

**R3** (a, b, c)  

**I1**  
smaller size

**I2**

**I3**

output relation

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_0</td>
<td>b_4</td>
<td>c_4</td>
</tr>
</tbody>
</table>
Probe - Enumeration Simulation

Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a, b, c]

I1

R1 (a, c)

I2

R2 (b, c)

I3

R3 (a, b, c)

output relation

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>h(a)</td>
<td>a_0</td>
<td>b_a</td>
<td>c_{a_0}</td>
</tr>
<tr>
<td>h(b)</td>
<td>b_{b_0}</td>
<td>h(b_1)</td>
<td>h(b_2)</td>
</tr>
<tr>
<td>h(c)</td>
<td>c_{c_0}</td>
<td>h(c_1) = h(c_7)</td>
<td>h(c_2)</td>
</tr>
</tbody>
</table>

smaller size

| h(a) | a_{a_0} | h(a) | a_{a_1} | h(a_3) | ... | h(a_7) |
| h(b) | h(b) | h(b) | h(b_2) | h(b_7) | ... |
| h(c) | h(c) | h(c_2) | ... | h(c_7) | ... |

R1 (a, c)

R2 (b, c)

R3 (a, b, c)

h(c)

I1

R1 (a, c)

I2

R2 (b, c)

I3

R3 (a, b, c)

R1 a b a
R2 c c b
R3 c
Probe - Enumeration Simulation

Build Hash Tries for R1, R2, and R3 following attribute ordering \([a, b, c]\) in Trie-levels

Attribute ordering: \([a, b, c]\)

Output relation:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>a_n</td>
<td>b_4</td>
<td>c_4</td>
</tr>
<tr>
<td>I2</td>
<td>a_n</td>
<td>b_4</td>
<td>c_7</td>
</tr>
<tr>
<td>I3</td>
<td>a_n</td>
<td>b_4</td>
<td>c_5</td>
</tr>
</tbody>
</table>
Probe - Enumeration Simulation

Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a, b, c]

I1

I2

I3

output relation

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_n</td>
<td>b_4</td>
<td>c_4</td>
</tr>
<tr>
<td>a_n</td>
<td>b_4</td>
<td>c_7</td>
</tr>
</tbody>
</table>
Probe - Enumeration Simulation

Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a, b, c]

R1 (a, c)
R2 (b, c)
R3 (a, b, c)

Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a, b, c]
Probe - Enumeration Simulation

Attribute ordering: \[a, b, c\]

R1 (a, c)

R2 (b, c)

R3 (a, b, c)

Build Hash Tries for R1, R2, and R3 following attribute ordering \([a, b, c]\) in Trie-levels

Attribute ordering: \([a, b, c]\)
Outline of Query Processing Techniques to Cover

For each we cover: a) Foundations; b) System implementations; and c) Open challenges.

1) Predefined Joins

2) Worst-case Optimal Joins (WCOJs)
   Handling Intermediate Size Growth for Cyclic Joins

   2.1. Foundations

   2.2 System Integration Approaches
      i) Index-based WCOJs (Graphflow & EmptyHeaded)
      ii) Hash-based WCOJs (Umbra)
         • Optimization approaches (cost-based DP, GHD, rule-based)

3) Factorized Query Processing
Query Optimization - Rule-based

Relation $R_1$

Relation $R_2$

Relation $R_3$
Query Optimization - Rule-based

Relation R₁ (a, c) ⨝ Relation R₂ (b, c) ⨝ Relation R₃ (a, b, c)
Query Optimization - Rule-based

Relation R₁

Relation R₂

Relation R₃

R₁ (a, c)

R₂ (b, c)

R₃ (a, b, c)
Query Optimization - Rule-based

Relation $R_1$ (a, c)
Relation $R_2$ (b, c)
Relation $R_3$ (a, b, c)

$R_1 \join R_2$ (5, 10, 20)

$R_3 (a, b, c)$

$R_1 (a, c)$
$R_2 (b, c)$
A binary join is replaced if it is classified as a *growing join*, i.e. its output cardinality is greater than the maximum of its input cardinalities.
Adopting Worst-Case Optimal Joins in Relational Database Systems

Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfonz Kemper, Thomas Neumann
Technische Universität München
{freitagm, bandle, tobiasschmidt, kemper, neumann}@in.tum.de

ABSTRACT
Worst-case optimal join algorithms are attractive from a theoretical point of view, as they offer asymptotically better runtime than binary joins on certain types of queries. In particular, they avoid enumerating large intermediate results by processing multiple input relations in a single multi-way join. However, existing implementations incur a sizable overhead in practice, primarily since they rely on suitable ordered index structures on their input. Systems that support worst-case optimal joins often focus on a specific problem domain, such as read-only graph analytic queries, where extensive precomputation allows them to mask these costs.

In this paper, we present a comprehensive implementation approach for worst-case optimal joins that is practicable within general-purpose relational database management systems supporting both hybrid transactional and analytical workloads. The key component of our approach is a novel hash-based worst-case optimal join algorithm that relies only on data structures that can be built efficiently during query execution. Furthermore, we implement a hybrid query optimizer that intelligently and transparently combines both binary and multi-way joins within the same query plan. We demonstrate that our approach far outperforms existing systems when worst-case optimal joins are beneficial while sacrificing no performance when they are not. Nevertheless, it is well-known that there are pathological cases in which any binary join plan exhibits suboptimal performance \cite{13}. The main shortcoming of binary joins is the generation of intermediate results that can become much larger than the actual query result \cite{20}.

Unfortunately, this situation is generally unavoidable in complex analytical settings where joins between non-key attributes are commonplace. For instance, a conceivable query on the TPC-H schema would be to look for parts within the same order that could have been delivered by the same supplier. Answering this query involves a self-join of lineitem and two non-key joins between lineitem and partsupp, all of which generate large intermediate results \cite{19}. Self joins that incur this issue are also prevalent in graph analytic queries such as searching for triangle patterns within a graph \cite{8}. On such queries, traditional RDBMS that employ binary join plans frequently exhibit disastrous performance or even fail to produce any result at all \cite{8,19,24}.

Consequently, there has been a long-standing interest in multi-way joins that avoid enumerating any potentially exploding intermediate results \cite{19,24}. Seminal theoretical advances recently enabled the development of worst-case optimal multi-way join algorithms which have runtime proportional to tight bounds on the worst-case size of the query result \cite{4,45}. As they can guarantee better asymptotic
Open Challenges
Open Challenges

1. Sorting-on-the-fly:
   - Direct comparison to hashing-on-the-fly
   - Should be slower to index but faster to join
   - Question: Which performs better and when?
Open Challenges

1. Sorting-on-the-fly:
   - Direct comparison to hashing-on-the-fly
   - Should be slower to index but faster to join
   - Question: Which performs better and when?

2. Databases Cracking (Idreos et al. CIDR 2017):
   - “... addressing index maintenance as part of query processing using continuous physical reorganization ...”
   - Don’t presort any lists
   - Sort the adjacency lists that are being integrated frequently during query processing
   - Seems easier to integrate into GDBMSs than RDBMSs.
3. Beyond WCOJ algorithms:

- Algorithms with instance optimality guarantees.
- Minesweeper [NGO et al., OIDS ‘14] Tetris [Khamis et al., TODS ‘16]
- **Fundamental Question:**
  - How do we know that a join algorithm’s output is correct?

- Answer: Any join algorithm implicitly provides a “proof” of its output through comparisons. Runtime ~ proof size.

- Next question: What is the minimum proof size?
Beyond WCOJ algorithms

There is an O(1) “proof” here with right indexing! Standard binary join or WCOJs would take at least linear time in the number of tuples.

Research Question: Can we make Tetris-like algorithms practical. (If you take a close look, algorithms have very high constants in their asymptotic runtimes)
Outline of Query Processing Techniques to Cover

For each we cover: a) Foundations; b) System implementations; and c) Open challenges.

1) Predefined Joins

2) Worst-case optimal joins

3) Factorized Query Processing
   Handling Intermediate Size Growth for Acyclic Joins
Outline of Query Processing Techniques to Cover

For each we cover: a) Foundations; b) System implementations; and c) Open challenges.

1) Predefined Joins

2) Worst-case optimal joins

3) Factorized Query Processing
   Handling Intermediate Size Growth for Acyclic Joins
   3.1. Foundations: Factorized Representations
   3.2. System Integration Approaches: FDB and Factorized Vector Execution in Graphflow
Outline of Query Processing Techniques to Cover

For each we cover: a) Foundations; b) System implementations; and c) Open challenges.

1) Predefined Joins

2) Worst-case optimal joins

3) Factorized Query Processing
   Handling Intermediate Size Growth for Acyclic Joins
   3.1. Foundations: Factorized Representations
   3.2. System Integration Approaches: FDB and Factorized Vector Execution in Graphflow
Flat Representation Example

FOLLOWS 1

a ----> b ----> c

FOLLOWS 2
Flat Representation Example

FOLLOW 1

FOLLOW 2

Relation \textit{FOLLOWS} (F1) \quad \text{FROM (a)} \quad \text{TO (b)}

Relation \textit{FOLLOWS} (F2) \quad \text{FROM (b)} \quad \text{TO (c)}
Flat Representation Example

Query Vertex Ordering: [b, a, c]
Flat Representation Example

Query Vertex Ordering: [b, a, c]

Output Relation

<table>
<thead>
<tr>
<th>b.ID</th>
<th>a.ID</th>
<th>c.ID</th>
</tr>
</thead>
</table>

Sink (accumulate)

Join C

Join a

Scan b

Scan b

n
n+1
n+2
n+3
2n
n-1

FOLLOWs 1

FOLLOWs 2

...
Flat Representation Example

Output Relation

<table>
<thead>
<tr>
<th>b.ID</th>
<th>a.ID</th>
<th>c.ID</th>
</tr>
</thead>
</table>

Query Vertex Ordering: [b, a, c]
Flat Representation Example

Query Vertex Ordering: [b, a, c]

Sink (accumulate)

n+1

n

Scan b

Join a

n+3

n+2

n+1

n-1

2n

...
Flat Representation Example

Query Vertex Ordering: \([b, a, c]\)

Output Relation

\[
\begin{array}{ccc}
\text{b.ID} & \text{a.ID} & \text{c.ID} \\
\end{array}
\]
Flat Representation Example

Output Relation

<table>
<thead>
<tr>
<th>b.ID</th>
<th>a.ID</th>
<th>c.ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>0</td>
<td>n+1</td>
</tr>
<tr>
<td>n</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>0</td>
<td>2n</td>
</tr>
</tbody>
</table>

Query Vertex Ordering: \([b, a, c]\)

```
Sink (accumulate)  
t(b:v_b,a:v_a,c:v_c)
```

```
Join C  
t(b:v_b,a:v_a)
```

```
Join a  
t(b:v_b)
```

```
Scan b
```
Flat Representation Example

Query Vertex Ordering: [b, a, c]

Output Relation

<table>
<thead>
<tr>
<th>b.ID</th>
<th>a.ID</th>
<th>c.ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>0</td>
<td>n+1</td>
</tr>
<tr>
<td>n</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>0</td>
<td>2n</td>
</tr>
</tbody>
</table>

n tuples

Sink (accumulate)

Join a

t (b:v_b, a:v_a)

Join C

t (b:v_b, a:v_a, c:v_c)

Scan b

t (b:v_b)
Flat Representation Example

Query Vertex Ordering: [b, a, c]

Output Relation

<table>
<thead>
<tr>
<th>b.ID</th>
<th>a.ID</th>
<th>c.ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>0</td>
<td>n+1</td>
</tr>
<tr>
<td>n</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>0</td>
<td>2n</td>
</tr>
</tbody>
</table>

n tuples
Flat Representation Example

Query Vertex Ordering: [b, a, c]

Output Relation

<table>
<thead>
<tr>
<th>b.ID</th>
<th>a.ID</th>
<th>c.ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>0</td>
<td>n+1</td>
</tr>
<tr>
<td>n</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>0</td>
<td>2n</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td>n+1</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td>2n</td>
</tr>
</tbody>
</table>

n tuples

Sink (accumulate)

Join C

Join A

Scan b
**Flat Representation Example**

Query Vertex Ordering: [b, a, c]

```
<table>
<thead>
<tr>
<th></th>
<th>b.ID</th>
<th>a.ID</th>
<th>c.ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>0</td>
<td>n+1</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>0</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>0</td>
<td>2n</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td>n+1</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td>2n</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>n-1</td>
<td>n+1</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>n-1</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>n-1</td>
<td>2n</td>
<td></td>
</tr>
</tbody>
</table>
```

Output Relation

**Flat Representation**

- **Sink (accumulate)**: 
  - \( t(b:v_b, a:v_a, c:v_c) \)
- **Join C**: 
  - \( t(b:v_b, a:v_a) \)
- **Join a**: 
  - \( t(b:v_b) \)
- **Scan b**: 
  - \( t(b:v_b) \)
Flat Representation Example

Query Vertex Ordering: [b, a, c]

Output Relation

<table>
<thead>
<tr>
<th>b.ID</th>
<th>a.ID</th>
<th>c.ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>0</td>
<td>n+1</td>
</tr>
<tr>
<td>n</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>0</td>
<td>2n</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td>n+1</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td>2n</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>n-1</td>
<td>n+1</td>
</tr>
<tr>
<td>n</td>
<td>n-1</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>n-1</td>
<td>2n</td>
</tr>
</tbody>
</table>
**Flat Representation Example**

Output Relation

<table>
<thead>
<tr>
<th>b.ID</th>
<th>a.ID</th>
<th>c.ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>0</td>
<td>n+1</td>
</tr>
<tr>
<td>n</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>0</td>
<td>2n</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td>n+1</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td>2n</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>n-1</td>
<td>n+1</td>
</tr>
<tr>
<td>n</td>
<td>n-1</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>n-1</td>
<td>2n</td>
</tr>
</tbody>
</table>

Query Vertex Ordering: [b, a, c]

- Sink (accumulate)
  - t (b:v_b, a:v_a, c:v_c)
- Join C
  - t (b:v_b, a:v_a)
- Join a
  - t (b:v_b)
- Scan b

Diagram showing relationships and tuples.
Flat Representation Example

FOLLOWS 1

FOLLOWS 2

Output Relation

<table>
<thead>
<tr>
<th>b.ID</th>
<th>a.ID</th>
<th>c.ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>0</td>
<td>n+1</td>
</tr>
<tr>
<td>n</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>0</td>
<td>2n</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td>n+1</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td>2n</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>n-1</td>
<td>n+1</td>
</tr>
<tr>
<td>n</td>
<td>n-1</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>n-1</td>
<td>2n</td>
</tr>
</tbody>
</table>

Query Vertex Ordering: \([b, a, c]\)
Flat Representation Example

Output Relation

<table>
<thead>
<tr>
<th>b.ID</th>
<th>a.ID</th>
<th>c.ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>0</td>
<td>n+1</td>
</tr>
<tr>
<td>n</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>0</td>
<td>2n</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td>n+1</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td>2n</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>n-1</td>
<td>n+1</td>
</tr>
<tr>
<td>n</td>
<td>n-1</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>n-1</td>
<td>2n</td>
</tr>
</tbody>
</table>

Query Vertex Ordering: [b, a, c]
Flat Representation Example

Output Relation

<table>
<thead>
<tr>
<th>b.ID</th>
<th>a.ID</th>
<th>c.ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>0</td>
<td>n+1</td>
</tr>
<tr>
<td>n</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>0</td>
<td>2n</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td>n+1</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td>2n</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>n-1</td>
<td>n+1</td>
</tr>
<tr>
<td>n</td>
<td>n-1</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>n-1</td>
<td>2n</td>
</tr>
</tbody>
</table>

Query Vertex Ordering: [b, a, c]
**Flat Representation Example**

- **a** and **c** are conditionally independent on **b**!

For a fixed **b** value, a change in **a** values does not change **c** values and vice-versa.

### Output Relation

<table>
<thead>
<tr>
<th>b.ID</th>
<th>a.ID</th>
<th>c.ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>0</td>
<td>n+1</td>
</tr>
<tr>
<td>n</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>0</td>
<td>2n</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td>n+1</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td>2n</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>n-1</td>
<td>n+1</td>
</tr>
<tr>
<td>n</td>
<td>n-1</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>n-1</td>
<td>2n</td>
</tr>
</tbody>
</table>

### Query Vertex Ordering:

$$[b, a, c]$$
Factorized Representation Example

\( a \) and \( c \) are conditionally independent on \( b \! \).

Query Vertex Ordering:
\([b, a, c]\)

Sink (accumulate)
\( t (b:v_b, a:v_a, c:v_c) \)

Join C
\( t (b:v_b, a:v_a) \)

Join A
\( t (b:v_b) \)

Scan b
Factorized Representation Example

a and c are conditionally independent on b!

Query Vertex Ordering: [b, a, c]

Sink (accumulate)
\[ t(b:v_b, a:v_a, c:v_c) \]

Join C
\[ t(b:v_b, a:v_a) \]

Join A
\[ t(b:v_b) \]

Scan b

Diagram:
- Node b is followed by nodes a and c.
- Vertices are ordered as [b, a, c].
- Independence relationships are shown with directed edges.
Factorized Representation Example

\[ \text{Query Vertex Ordering: } [b, a, c] \]

\( a \) and \( c \) are conditionally independent on \( b \)!
Factorized Representation Example

\( a \) and \( c \) are conditionally independent on \( b \! \)

\[
\begin{align*}
&\text{FOLLOWS 1} \\
&\text{FOLLOWS 2}
\end{align*}
\]
Factorized Representation Example

\(a\) and \(c\) are conditionally independent on \(b\)!

\[
\begin{array}{c}
\text{Sink (accumulate)} \\
\text{Join C} \\
\text{Join A} \\
\text{Scan b}
\end{array}
\]

\[
\begin{array}{cccccccc}
\emptyset & 1 & ... & n-1 & n+1 & n+2 & ... & 2n
\end{array}
\]

Query Vertex Ordering: [\(b, a, c\)]
Factorized Representation Example

a and c are conditionally independent on b!

Query Vertex Ordering: [b, a, c]

2n+1 fields
Factorized Representation Example

\(a\) and \(c\) are conditionally independent on \(b\)!

\[
\begin{align*}
\text{Query Vertex Ordering:} & \quad [b, a, c] \\
\text{Sink (accumulate)} & \quad t (b: v_b, a: v_a, c: v_c) \\
\text{Join C} & \quad t (b: v_b, a: v_a) \\
\text{Join A} & \quad t (b: v_b) \\
\text{Scan b} &
\end{align*}
\]

2n+1 fields vs. flat 3n^2 fields
Factorized Representation Example

\( a \) and \( c \) are conditionally independent on \( b \)!

Query Vertex Ordering: 
\([b, a, c]\)

- **f-tree**
  - \( f\)-tree
  - \( f\)-representation

- **Sink (accumulate)**
  - \( t(b:v_b, a:v_a, c:v_c) \)

- **Join C**
  - \( t(b:v_b, a:v_a) \)

- **Join A**
  - \( t(b:v_b) \)

- **Scan b**

2n+1 fields vs. flat 3n^2 fields
Factorized Representation Example

**a and c are conditionally independent on b!**

```
FOLLOWING 1  FOLLOWING 2
a ----> b ----> c
```

**F-rtree:**

```
\[ \begin{array}{c}
\text{Sink (accumulate)} \\
\text{Join C} \\
\text{Join A} \\
\text{Scan b}
\end{array} \]
```

**Query Vertex Ordering:**

\[ [b, a, c] \]

**Join a:**

\[ t(b:v_b,a:v_a) \]

**Sink (accumulate):**

\[ t(b:v_b,a:v_a,c:v_c) \]

**f-representation:**

```
\[ \begin{array}{c}
U_b \\
U_a \\
U_c \end{array} \]
```

**f-tree:**

```
\[ \begin{array}{c}
b \\
a \\
c \end{array} \]
```

**attribute-level description of the f-representation**

```
\[ \begin{array}{c}
U_b \\
U_a \\
U_c \end{array} \]
```

```
\[ \begin{array}{c}
\text{Join C} \\
\text{Join A} \\
\text{Scan b}
\end{array} \]
```

**2n+1 fields vs. flat 3n^2 fields**
Factorized Representation Example

\(a\) and \(c\) are conditionally independent on \(b\)!

Query Vertex Ordering: \([b, a, c]\)

Theory of factorization establishes:
\(\sigma(Q) \leq \text{AGM}(Q)\) where
\(\sigma(Q)\): worst-case size bound over f-representations.
In some cases, \(\sigma(Q) < \text{AGM}(Q)\)

2n+1 fields vs. flat 3n^2 fields
What Are The Possible F-Representations for a Query Q?
What Are The Possible F-Representations for a Query Q?

f-representation examples over f-trees
What Are The Possible F-Representations for a Query Q?

FOLLOWS 1  
FOLLOWS 2

a  b  c

f-representation examples over f-trees

F-tree validity:
Any two Q-dependent attributes lie along a root-to-leaf path in an f-tree.
What Are The Possible F-Representations for a Query Q?

F-tree validity:
Any two Q-dependent attributes lie along a root-to-leaf path in an f-tree:
- Any query vertex pair connected by a query edge are Q-dependent.
What Are The Possible F-Representations for a Query Q?

f-representation examples over f-trees

F-tree validity:
Any two Q-dependent attributes lie along a root-to-leaf path in an f-tree:
- Any query vertex pair connected by a query edge are Q-dependent.
- Any query vertex/edge pair are dependent based on theta joins.
What Are The Possible F-Representations for a Query Q?

F-tree validity:
- Any two Q-dependent attributes lie along a root-to-leaf path in an f-tree:
  - Any query vertex pair connected by a query edge are Q-dependent.
  - Any query vertex/edge pair are dependent based on theta joins.
What Are The Possible F-Representations for a Query Q?

F-representation examples over f-trees

WHERE \( a.p < c.p \)

a and b are Q-dependent!!
f-tree is not valid

F-tree validity:
Any two Q-dependent attributes lie along a root-to-leaf path in an f-tree:
- Any query vertex pair connected by a query edge are Q-dependent.
- Any query vertex/edge pair are dependent based on theta joins.
Outline of Query Processing Techniques to Cover

For each we cover: a) Foundations; b) System implementations; and c) Open challenges.

1) Predefined Joins

2) Worst-case optimal joins

3) Factorized Query Processing
   3.1. Foundations: Factorized Representations
   3.2. System Integration Approaches: FDB
FDB: A Query Engine for Factorised Relational Databases

Nurzhan Bakibayev, Dan Olteanu, and Jakub Závodný
Department of Computer Science, University of Oxford, OX1 3QD, UK
{nurzhan.bakibayev, dan.olteanu, jakub.zavodny}@cs.ox.ac.uk

ABSTRACT
Factorised databases are relational databases that use compact factorised representations at the physical layer to reduce data redundancy and boost query performance.

This paper introduces FDB, an in-memory query engine for select-project-join queries on factorised databases. Key components of FDB are novel algorithms for query optimisation and evaluation that exploit the succinctness brought by data factorisation. Experiments show that for data sets with many-to-many relationships FDB can outperform relational engines by orders of magnitude.

whereby each singleton relation \( v \) holds one value \( v \), each tuple is a product of singleton relations, and the relation is a union of products of singleton relations:

\[
\begin{align*}
\text{(Milk)} & \times \text{(Istanbul)} \times \text{(Adnan)} \cup \\
\text{(I1)} & \times \text{(Istanbul)} \times \text{(Yasemin)} \cup \\
\text{(I1)} & \times \text{(Imir)} \times \text{(Adnan)} \cup \\
\text{(I1)} & \times \text{(Istanbul)} \times \text{(Volkan)} \cup \ldots
\end{align*}
\]

A more compact equivalent representation can be obtained by algebraic factorisation using distributivity of product over union and commutativity of product and union:

\[
\begin{align*}
\text{(Milk)} & \times (\text{(I1)} \times (\text{(Istanbul)} \cup (\text{(Adnan)} \cup (\text{(Yasemin)}))))
\end{align*}
\]
FDB: Factorized Database Engine

```
0  n  n+1
/  \ /  \  /
1    n+2  
\  /  \  /
2   n+3  
\ //   /
...  
\ //   /
n-1  2n
```
FDB: Factorized Database Engine

FOLLOWS 1

a --- b --- c

0 --- n --- n+1
1 --- n+2
2 --- n+3
... --- ...
n-1 --- 2n

FOLLOWS 2

operator n

f-T_n

operator 2

f-T_2

operator 1

f-T_1

...
FDB: Factorized Database Engine

Query Vertex Ordering:
\[ b, a, c \]
FDB: Factorized Database Engine

Query Vertex Ordering: [b, a, c]
FDB: Factorized Database Engine

Query Vertex Ordering: [b, a, c]
FDB: Factorized Database Engine

Query Vertex Ordering: [b, a, c]
FDB: Factorized Database Engine

Query Vertex Ordering: [b, a, c]

Sink (accumulate)

Join C

Join a

Scan b

b \{ 0 \ 1 \ 2 \ ... \ n \ ... \ 2n-1 \ 2n \}

FOLLOWS 1

FOLLOWS 2

a
b
n
n+1
n+2
n+3
2n
...}

b
a
c

0
1
2
n-1
FDB: Factorized Database Engine

Query Vertex Ordering: \([b, a, c]\)

Sink (accumulate)

Join C

Join a

Scan b

\[
\begin{align*}
\text{FOLLOWS 1} & \quad \text{FOLLOWS 2} \\
a & b & c \\
0 & 1 & 2 & \ldots & n & \ldots & 2n-1 & 2n \\
\end{align*}
\]
FDB: Factorized Database Engine

Query Vertex Ordering:
[b, a, c]

Sink (accumulate)

Join C

Join a

Scan b

No users follow 0 in the graph
FDB: Factorized Database Engine

Query Vertex Ordering: [b, a, c]

Sink (accumulate)

Join c

No users follow 1 in the graph
FDB: Factorized Database Engine

Query Vertex Ordering: [b, a, c]

Sink (accumulate)

Join a

Scan b

No users follow 2 in the graph
FDB: Factorized Database Engine

Query Vertex Ordering: \([b, a, c]\)

Sink (accumulate)

Join C

Join a

Scan b

No users follow 2 in the graph
FDB: Factorized Database Engine

Query Vertex Ordering: [b, a, c]

Sink (accumulate)

Join C

Join a

Scan b

b \{ 0 \ 1 \ 2 \ \ldots \ n \ \ldots \ 2n-1 \ 2n \}
**FDB: Factorized Database Engine**

Query Vertex Ordering: 
[b, a, c]
FDB: Factorized Database Engine

Query Vertex Ordering: [b, a, c]
FDB: Factorized Database Engine

Query Vertex Ordering: [b, a, c]

Sink (accumulate)

Join C

Join a

Scan b

FOLLOWS 1
FOLLOWS 2

0
1
2
... 
n-1

n
n+1
n+2
n+3
...
2n

a
b
... 
c

0 1 2 ... n-1 2n-1 2n
FDB: Factorized Database Engine

Query Vertex Ordering: [b, a, c]
FDB: Factorized Database Engine

Query Vertex Ordering: [b, a, c]

Sink (accumulate)

Join C

Join a

Scan b

0 1 2 ... n-1 n 2n-1 2n

0 1 2 ... n-1

a

b

FOLLOWS 1

FOLLOWS 2

a b c

0 1 2 ... n-1 1 2 3 4 ... 2n 2n

Sink (accumulate)
FDB: Factorized Database Engine

Query Vertex Ordering: [b, a, c]
FDB: Factorized Database Engine

Query Vertex Ordering: [b, a, c]
FDB: Factorized Database Engine

Query Vertex Ordering: [b, a, c]
FDB: Factorized Database Engine

Query Vertex Ordering: [b, a, c]

Sink (accumulate)

Join C

Join a

Scan b

FOLLOWS 1

FOLLOWS 2

0

1

2

... 
n-1

n+1

n+2

n+3

... 
2n

0 1 2 ... n ... 2n-1 2n

0 1 2 ... n ... n n

n+1 n+2 n+3 ... 2n

b

a

c

b

a

b
FDB: Factorized Database Engine

Query Vertex Ordering: [b, a, c]

Sink (accumulate)

Join C

Join a

Scan b

n+1 ... n+2 do not follow any users in the graph
FDB: Factorized Database Engine

Query Vertex Ordering: [b, a, c]
FDB: Factorized Database Engine

- Implements core algorithms for SJP queries meeting all asymptotic bounds.
- Able to generate all factorized trees.

Query Vertex Ordering: [b, a, c]
FDB: Factorized Database Engine

Query Vertex Ordering: [b, a, c]

Sink (accumulate)

Join C

Join a

Scan b
FDB: Factorized Database Engine

Query Vertex Ordering: [b, a, c]

- Materializes more than necessary till the sink! Recursive deletes need to be handled.
FDB: Factorized Database Engine

- Materializes more than necessary till the sink! Recursive deletes need to be handled.
- Full materialization operators: loss of classic optimizations e.g., vectorized execution.

Query Vertex Ordering: [b, a, c]
Outline of Query Processing Techniques to Cover

For each we cover: a) Foundations; b) System implementations; and c) Open challenges.

1) Predefined Joins

2) Worst-case optimal joins

3) Factorized Query Processing
   Handling Intermediate Size Growth for Acyclic Joins
   3.1. Foundations: Factorized Representations
   3.2. System Integration Approaches: FDB and Factorized Vector Execution in Graphflow
Factorized Vector Execution
Factorized Vector Execution

Columnar Storage and List-based Processing for Graph Database Management Systems

Pranjal Gupta, Amine Mhedhbi, Semih Salihoglu
University of Waterloo
{pranjal.gupta, amine.mhedhbi, semih.salihoglu}@uwaterloo.ca

ABSTRACT
We revisit column-oriented storage and query processing techniques in the context of contemporary graph database management systems (GDBMSs). Similar to column-oriented RDBMSs, GDBMSs support read-heavy analytical workloads that however have fundamentally different data access patterns than traditional analytical workloads. We first derive a set of desiderata for optimizing storage and query processors of GDBMS based on their access patterns.

This calls for redesigning columnar techniques in the context of GDBMSs. The contributions of this paper are as follows.

Guidelines and Desiderata: We begin in Section 3 by analyzing the properties of data access patterns in GDBMSs. For example, we observe that different components of data stored in GDBMSs can have some structure and the order in which operators access vertex and edge properties often follow the order of edges in adjacency lists. This analysis instructs a set of guidelines and desiderata for
Factorized Vector Execution

Query Vertex Ordering: [b, a, c, d]

FOLLOWS & LIVES_IN Edges
Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]

FOLLOWS & LIVES_IN Edges
Factorized Vector Execution

Query Vertex Ordering: [b, a, c, d]

FOLLOWS & LIVES_IN Edges
Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]

Sink (accumulate)
\[ t(b:v_b, a:v_a, c:v_c, d:v_d) \]

Join d
\[ t(b:v_b, a:v_a, c:v_c) \]

Join c
\[ t(b:v_b, a:v_a) \]

Join a
\[ t(b:v_b) \]

Scan b

FOLLOWS & LIVES_IN Edges
Factorized Vector Execution

Query Vertex Ordering: [b, a, c, d]

Sink (accumulate)

Join d

Join C

Join a

Scan b

b.id [0 1 ... n ... 1023]
pos → -1 | size → 1024

FOLLOWS & LIVES_IN Edges
Factorized Vector Execution

Query Vertex Ordering: [b, a, c, d]

Sink (accumulate)
- t(b:v₆ₐ:va₆:va₇:vc₇:vd₇)

Join d
- t(b:v₆ₐ:va₆:va₇:vc₇)

Join C
- t(b:v₆ₐ:va₆:va₇)

Join a
- t(b:v₆ₐ)

Scan b

FOLLOWS & LIVES_IN Edges

b.id | 0 | 1 | ... | n | ... | 1023
pos → -1 | size → 1024
Factorized Vector Execution

Query Vertex Ordering: [b, a, c, d]

Sink (accumulate)

\[ t(b:v_b, a:v_a, c:v_c, d:v_d) \]

Join d

\[ t(b:v_b, a:v_a, c:v_c) \]

Join C

\[ t(b:v_b, a:v_a) \]

Join a

\[ \text{Scan } b \]

FOLLOWS & LIVES_IN Edges

Sink (accumulate)

\[ \text{Scan } b \]
Factorized Vector Execution

Query Vertex Ordering: [b, a, c, d]

Sink (accumulate)
- \( t(b:v_b, a:v_a, c:v_c, d:v_d) \)

Join d
- \( t(b:v_b, a:v_a, c:v_c) \)

Join c
- \( t(b:v_b, a:v_a) \)

No users follow 0 in the graph

Scan b
- \( t(b:v_b) \)

FOLLOWS & LIVES_IN Edges

<table>
<thead>
<tr>
<th>pos</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1024</td>
</tr>
<tr>
<td>n</td>
<td>1023</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>n+1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>n+2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>n+3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>n+4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>n+5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>n+6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>n+7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>n+8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>n+9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>n+10</td>
<td></td>
</tr>
</tbody>
</table>

...
Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]

FOLLOWS & LIVES_IN Edges

No users follow 1 in the graph
Factorized Vector Execution

Query Vertex Ordering: [b, a, c, d]

Sink (accumulate)
\( t(b:v_b, a:v_a, c:v_c, d:v_d) \)

Join d
\( t(b:v_b, a:v_a, c:v_c) \)

Join c
\( t(b:v_b, a:v_a) \)

Join a
\( t(b:v_b) \)

Scan b

FOLLOW & LIVES_IN Edges
Factorized Vector Execution

Query Vertex Ordering: [b, a, c, d]

Sink (accumulate)
- t (b:v_b, a:v_a, c:v_c, d:v_d)

Join d
- t (b:v_b, a:v_a, c:v_c)

Join C
- t (b:v_b, a:v_a)

Join a
- t (b:v_b)

Scan b

FOLLOWS & LIVES_IN Edges

b.id: 0, 1, ..., n, ..., 1023
pos: n, size: 1024

a.id*: 0, 1, 2, ..., n-1
pos: -1, size: n
Factorized Vector Execution

Query Vertex Ordering: [b, a, c, d]

Sink (accumulate)
- t (b:v_b,a:v_a,c:v_c,d:v_d)

Join d
- t (b:v_b,a:v_a,c:v_c)

Join C
- t (b:v_b,a:v_a)

Join a
- Scan b

b.id | 0 1 ... n ... 1023
pos → n | size → 1024

a.id* | 0 1 2 ... n-1
pos → -1 | size → n

FOLLOWS & LIVES_IN Edges
Factorized Vector Execution

Query Vertex Ordering: [b, a, c, d]

Sink (accumulate)

Join d
t(b:v_b,a:v_a,c:v_c,d:v_d)

Join C
t(b:v_b,a:v_a,c:v_c)

Join a
t(b:v_b,a:v_a)

Scan b
t(b:v_b)

FOLLOWS & LIVES_IN Edges

b.id 0 1 ... n ... 1023
pos n size 1024

a.id* 0 1 2 ... n-1
pos -1 size n

b.id

a.id
Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]

Sink (accumulate)

```plaintext
Join d
```

```plaintext
Join c
t (b:v_b,a:v_a,c:v_c,d:v_d)
```

```plaintext
Join a
t (b:v_b,a:v_a)
```

```plaintext
Scan b
```

FOLLOWS & LIVES_IN Edges

<table>
<thead>
<tr>
<th>b.id</th>
<th>0</th>
<th>1</th>
<th>...</th>
<th>n</th>
<th>...</th>
<th>1023</th>
</tr>
</thead>
<tbody>
<tr>
<td>pos</td>
<td>n</td>
<td>size</td>
<td>1024</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```

<table>
<thead>
<tr>
<th>a.id</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>n-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>pos</td>
<td>-1</td>
<td>size</td>
<td>n</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b.id</th>
<th>n, n, n, ... , n</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.id</td>
<td>0, 1, 2, ... , n-1</td>
</tr>
</tbody>
</table>
Factorized Vector Execution

FOLLOWS & LIVES_IN Edges

Query Vertex Ordering: [b, a, c, d]

Sink (accumulate)

Join d

Join c

Join a

Scan b

n tuples
Factorized Vector Execution

Query Vertex Ordering: [b, a, c, d]

Sink (accumulate)

Join d

Join C

Join a

Scan b

FOLLOWS & LIVES_IN Edges
Factorized Vector Execution

Query Vertex Ordering: [b, a, c, d]

Sink (accumulate)

Join d

Join C

Join a

Scan b
Factorized Vector Execution

Query Vertex Ordering: [b, a, c, d]

Sink (accumulate)

Join d

Join C

Join a

Scan b

FOLLOWS & LIVES_IN Edges
Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]

Sink (accumulate)

Join d

Join c

Join a

Scan b
Factorized Vector Execution

Query Vertex Ordering: [b, a, c, d]

Sink (accumulate)

Join d

Join c

Join a

Scan b

<table>
<thead>
<tr>
<th>b.id</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.id</td>
<td>0</td>
</tr>
<tr>
<td>c.id</td>
<td>4</td>
</tr>
</tbody>
</table>

FOLLOWS & LIVES_IN Edges

FOLLOWS 1

FOLLOWS 2

LIVES_IN
Query Vertex Ordering:
[b, a, c, d]
Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b.id</td>
<td>0</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>pos</td>
<td>n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>1023</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a.id</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>pos</td>
<td>-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>n</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>c.id</td>
<td>n+1</td>
<td>...</td>
<td>2n</td>
</tr>
<tr>
<td>pos</td>
<td>n+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>n</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sink (accumulate)

Join d
t(b:v_b,a:v_a,c:v_c,d:v_d)

Join C
t(b:v_b,a:v_a,c:v_c)

Join a
t(b:v_b,a:v_a)

Scan b
Factorized Vector Execution

Query Vertex Ordering: [b, a, c, d]

FOLLOWS & LIVES_IN Edges

b.id | 0 1 ... n ... 1023 |
pos → n | size → 1024

a.id | 0 1 2 ... n-1 |
pos → -1 | size → n

c.id | n+1 ... 2n |
pos → -1 | size → n

<table>
<thead>
<tr>
<th>b.id</th>
<th>n,..., n</th>
<th>n,..., n</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.id</td>
<td>0,..., 0</td>
<td>1,..., 1</td>
</tr>
<tr>
<td>c.id</td>
<td>n+1,...,2n</td>
<td>n+1,...,2n</td>
</tr>
</tbody>
</table>
Factorized Vector Execution

Query Vertex Ordering: [b, a, c, d]

### Follows & Lives IN Edges

<table>
<thead>
<tr>
<th>id</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>n-1</th>
<th>n</th>
<th>n+1</th>
<th>...</th>
<th>2n</th>
</tr>
</thead>
<tbody>
<tr>
<td>pos</td>
<td>n</td>
<td>n</td>
<td>-1</td>
<td>...</td>
<td>-1</td>
<td>n</td>
<td>n-1</td>
<td>...</td>
<td>n-1</td>
</tr>
<tr>
<td>size</td>
<td>1023</td>
<td>1024</td>
<td>n</td>
<td>...</td>
<td>n+1</td>
<td>2n</td>
<td>n+1</td>
<td>...</td>
<td>2n</td>
</tr>
</tbody>
</table>

### Table

<table>
<thead>
<tr>
<th>Field</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>b.id</td>
<td>n, ..., n, n, ..., n, n</td>
</tr>
<tr>
<td>a.id</td>
<td>0, ..., 0, 1, ..., 1, 2</td>
</tr>
<tr>
<td>c.id</td>
<td>n+1, ..., 2n, n+1, ..., 2n</td>
</tr>
</tbody>
</table>
Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]

Sink (accumulate)
\[ t(b:v_b,a:v_a,c:v_c,d:v_d) \]

Join d
\[ t(b:v_b,a:v_a,c:v_c) \]

Join C
\[ t(b:v_b,a:v_a) \]

Join a
\[ t(b:v_b) \]

Scan b

<table>
<thead>
<tr>
<th>b.id</th>
<th>n, ..., n</th>
<th>n, ..., n</th>
<th>n, ..., n</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.id</td>
<td>0, ..., 0</td>
<td>1, ..., 1</td>
<td>2, ..., 2</td>
</tr>
<tr>
<td>c.id</td>
<td>n+1, ..., 2n</td>
<td>n+1, ..., 2n</td>
<td>n+1, ..., 2n</td>
</tr>
</tbody>
</table>

FOLLOWS & LIVES_IN Edges
Factorized Vector Execution

Query Vertex Ordering: [b, a, c, d]
Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]

Sink (accumulate)
\[ t(b:v_b,a:v_a,c:v_c,d:v_d) \]

Join d
\[ t(b:v_b,a:v_a,c:v_c) \]

Join C
\[ t(b:v_b,a:v_a) \]

Join a
\[ t(b:v_b) \]

Scan b
\[ t(b:v_b) \]

b.id 0 1 ... n ... 1023
pos -> n | size -> 1024

a.id 0 1 2 ... n-1
pos -> -1 | size -> n

c.id n+1 ... 2n
pos -> -1 | size -> n

FOLLOWS & LIVES_IN Edges
after flattening: \( n^2 \) tuples

<table>
<thead>
<tr>
<th></th>
<th>b.id</th>
<th>a.id</th>
<th>c.id</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n,...,n</td>
<td>n,...,n</td>
<td>n,...,n</td>
</tr>
<tr>
<td>0</td>
<td>n,...,n</td>
<td>n,...,n</td>
<td>n,...,n</td>
</tr>
<tr>
<td>n</td>
<td>n,...,n</td>
<td>n,...,n</td>
<td>n,...,n</td>
</tr>
<tr>
<td>2n</td>
<td>n,...,n</td>
<td>n,...,n</td>
<td>n,...,n</td>
</tr>
</tbody>
</table>

Follows & Lives_In Edges

After flattening: \( n^2 \) tuples
Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]

FOLLOWS & LIVES_IN Edges

after flattening:
\( n^2 \) tuples
Factorized Vector Execution

Query Vertex Ordering: [b, a, c, d]

Sink (accumulate)

Join d

Join c

Join a

Scan b

FOLLOWS & LIVES_IN Edges
Factorized Vector Execution

Query Vertex Ordering: [b, a, c, d]

FOLLOWS & LIVES_IN Edges
Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]

FOLLOWS & LIVES_IN Edges
Factorized Vector Execution

Query Vertex Ordering: [b, a, c, d]

FOLLOWS & LIVES_IN Edges
Factorized Vector Execution

Query Vertex Ordering: [b, a, c, d]

FOLLOWS & LIVES_IN Edges

Sink output buffer

- **Fixed-size**: b.id (4-bytes)
- **Variable-size**: (size, a.id), (size, c.id, d.id)

Sink (accumulate)

- Scan b
- Join a
- Join C
- Join d

Fixed-size: b.id (4-bytes)
Variable-size: (size, a.id), (size, c.id, d.id)
Factorized Vector Execution

Query Vertex Ordering: [b, a, c, d]

Sink output buffer

<table>
<thead>
<tr>
<th>fixed-size:</th>
<th>variable-size:</th>
</tr>
</thead>
<tbody>
<tr>
<td>b.id (4-bytes)</td>
<td>(size, a.id), (size, c.id, d.id)</td>
</tr>
</tbody>
</table>

Sink (accumulate)

Scan b

Join a

<table>
<thead>
<tr>
<th>Join C</th>
</tr>
</thead>
<tbody>
<tr>
<td>t(b:v_b,a:v_a,c:v_c)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Join d</th>
</tr>
</thead>
<tbody>
<tr>
<td>t(b:v_b,a:v_a,c:v_c,d:v_d)</td>
</tr>
</tbody>
</table>

Join vertex ordering: [b, a, c, d]
Factorized Vector Execution

Query Vertex Ordering: [b, a, c, d]

Sink output buffer

fixed-size: b.id (4-bytes)
variable-size: (size, a.id), (size, c.id, d.id)

Sink (accumulate)

0 1 ... n ... 1023
pos → n+1 | size → 1024

0 1 2 ... n-1
pos → n+1 | size → 1024

Sink output buffer

fixed-size: b.id (4-bytes)
variable-size: (size, a.id), (size, c.id, d.id)

Sink (accumulate)
	t (b:v_b, a:v_a, c:v_c, d:v_d)

Join d

t (b:v_b, a:v_a, c:v_c)

Join C

t (b:v_b, a:v_a)

Join a

t (b:v_b)

Scan b
Factorized Vector Execution

Query Vertex Ordering: [b, a, c, d]
Factorized Vector Execution

Query Vertex Ordering: [b, a, c, d]

Sink output buffer

FOLLOWS & LIVES_IN Edges
Factorized Vector Execution

Query Vertex Ordering: [b, a, c, d]

Sink output buffer

Sink (accumulate)
**Factorized Vector Execution**

Query Vertex Ordering: \([b, a, c, d]\)

Sink output buffer

- **fixed-size**: \(b\.id\) (4-bytes)
- **variable-size**: \((\text{size}, a\.id), (\text{size}, c\.id), (\text{size}, d\.id)\)

<table>
<thead>
<tr>
<th>n</th>
<th>n+1</th>
<th>...</th>
<th>n-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>...</td>
<td>2n</td>
</tr>
<tr>
<td>n+2</td>
<td>n+3</td>
<td>...</td>
<td>2n</td>
</tr>
</tbody>
</table>

Sink (accumulate)

- \(t(b: v_b, a: v_a, c: v_c, d: v_d)\)
- \(t(b: v_b, a: v_a)\)
- \(t(b: v_b)\)
- \(t(b: v_c)\)
Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]

Sink output buffer

Sink (accumulate)

Join d

Join C

Join a

Scan b
Factorized Vector Execution

Query Vertex Ordering: [b, a, c, d]

Sink output buffer

Sink (accumulate)
Factorized Vector Execution

Query Vertex Ordering: [b, a, c, d]

FOLLOWS & LIVES_IN Edges

Sink output buffer

Scan b

Join a

Join C

Join d

Sink (accumulate)
Factorized Vector Execution

Query Vertex Ordering: [b, a, c, d]

Sink output buffer

Sink (accumulate)

Join d

Join C

Join a

Scan b

FOLLOWS & LIVES_IN Edges
Query Vertex Ordering: [b, a, c, d]

Same for BHT (variable-sizes in overflow pages) → smaller HT

Sink (accumulate)
Build HT

Scan b
Join a
Join C
Join d

Sink output buffer

3n+1 fields vs. flat
4n^2 fields

FOLLOWS & LIVES_IN Edges

Factorized Vector Execution
Factorized Vector Benefits & Limits

- \( b.\text{id} \):
  - \( 0, 1, \ldots, n \) → 1023
  - \( \text{pos} \rightarrow n+1 \) | \( \text{size} \rightarrow 1024 \)

- \( a.\text{id}^{*} \):
  - \( 0, 1, 2, \ldots, n-1 \)
  - \( \text{pos} \rightarrow -1 \) | \( \text{size} \rightarrow n \)

- \( c.\text{id}^{*} \):
  - \( n+1, \ldots, 2n \)

- \( d.\text{id} \):
  - \( 7, \ldots, 8 \)
  - \( \text{pos} \rightarrow -1 \) | \( \text{size} \rightarrow n \)
Factorized Vector Benefits & Limits

Pipelined factorization with f-tree saving: reading column scans & Filtering

restricted f-trees (height = 1) to allow pipelining. Other redundancy is possible
Factorized Vector Benefits & Limits

Pipelined factorization with f-tree saving: reading column scans & Filtering

restricted f-trees (height = 1) to allow pipelining. Other redundancy is possible

Performance speedups up to 19x on LDBC 100
Other Work

Aggregation and Ordering in Factorised Databases

Nurzhan Bakibayev, Tomáš Kočiský, Dan Olteanu, and Jakub Závodný
Department of Computer Science, University of Oxford, OX1 3QD, UK
{nurzhan.bakibayev, tomas.kocisky, dan.olteanu, jakub.zavodny}@cs.ox.ac.uk

ABSTRACT
A common approach to data analysis involves understanding and manipulating succinct representations of data. In earlier work, we put forward a succinct representation system for relational data called factorised databases and reported on the main-memory query engine FDB for select-project-join queries on such databases.

In this paper, we extend FDB to support a larger class of practical queries with aggregates and ordering. This requires novel optimisation and evaluation techniques. We show how factorisation coupled with partial aggregation can effectively reduce the number of operations needed for query evaluation. We also show how factorisations of query results can support enumeration of tuples in desired orders as efficiently as listing them from the unfactorised, sorted results.

We experimentally observe that FDB can outperform off-the-shelf relational engines by orders of magnitude.
**Other Work**

**LMFAO: An Engine for Batches of Group-By Aggregates**

Layered Multiple Functional Aggregate Optimization

Maximilian Schleich  
University of Washington  
schleich@cs.washington.edu

Dan Olteanu  
University of Zurich  
olteanu@ifi.uzh.ch

**ABSTRACT**

LMFAO is an in-memory optimization and execution engine for large batches of group-by aggregates over joins. Such database workloads capture the data-intensive computation of a variety of data science applications.

We demonstrate LMFAO for three popular models: ridge linear regression with batch gradient descent, decision trees with CART, and clustering with k-means.

**PVLDB Reference Format:**
DOI: https://doi.org/10.14778/3415478.3415515
Redundancy in F-Representations Example
Redundancy in F-Representations Example

FOLLOWS 1

FOLLOWS 2

2n + n^2 fields
Redundancy in F-Representations Example

\[ \begin{array}{c}
a \\
\mid \\
b \\
\mid \\
c
\end{array} \]

\[ \begin{array}{c}
U_a \\
\mid \\
0 \ldots 2 \ldots \\
\mid \\
x \ldots x \\
\mid \\
U_b \mid U_b \\
\mid \\
1 \ldots 1 \\
\mid \\
x \ldots x \\
\mid \\
U_c \\
\mid \\
2 \ldots n
\end{array} \]

\[ \begin{array}{c}
U_c \\
\mid \\
2 \ldots n
\end{array} \]

\[ \begin{array}{c}
U_c \\
\mid \\
2 \ldots n
\end{array} \]

FOLLOWS 1

FOLLOWS 2

2n + n^2 fields
Redundancy in F-Representations Example

\[
\begin{align*}
\text{a} & \xrightarrow{\text{FOLLOWS 1}} \text{b} \xrightarrow{\text{FOLLOWS 2}} \text{c} \\
\text{d-tree} & \\
da & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \verticalalignment:
\text{Definition for values rooted in b per b assignment}
Theory of factorization establishes:
\[ \sigma^\uparrow(Q) \leq \sigma(Q) \leq \text{AGM}(Q), \]
where
\[ \sigma(Q) : \text{worst-case size bound over f-representations.} \]
\[ \sigma^\uparrow(Q) : \text{worst-case size bound over d-representations.} \]

In some cases, \[ \sigma^\uparrow(Q) < \sigma(Q) < \text{AGM}(Q) \]
Open Challenges

1) How to integrate the use of d-representations in query plans?

2) How to generate and cost such plans?

3) How to characterize the benefits of factorized representations in terms of query and dataset characteristics?
D-Representation for Expression Reuse Example
D-Representation for Expression Reuse Example

WHERE P(a, b, c, d, e)
RETURN b.p, c.p_{c1}, c.p_{c2}, d.p, e.p_{e1}
D-Representation for Expression Reuse Example

Query Vertex Ordering: [a, b, c, d, e]

WHERE P(a, b, c, d, e)
RETURN b.p, c.p, c.p, c.p, d.p, e.p

Scan necessary columns and apply filters
D-Representation for Expression Reuse Example

Query Vertex Ordering: [a, b, c, d, e]

WHERE P(a, b, c, d, e)
RETURN b.p_b, c.p_c1, c.p_c2, d.p_d, e.p_e1

Op_F c  Scan necessary columns and apply filters

same computation given same b assignment
D-Representation for Expression Reuse Example

Query Vertex Ordering: 
\[[a, b, c, d, e]\]

WHERE \( P(a, b, c, d, e) \)
RETURN \( b.p_b, c.p_{c1}, c.p_{c2}, d.p_d, e.p_{e1} \)

Scan necessary columns and apply filters

same computation given same \( b \) assignment
D-Representation for Expression Reuse Example

Query Vertex Ordering: [a, b, c, d, e]

Scan necessary columns and apply filters

same computation given same b assignment

same computation given same c assignment
D-Representation for Expression Reuse Example

Query Vertex Ordering: [a, b, c, d, e]

WHERE P(a, b, c, d, e)
RETURN b.p_b, c.p_c1, c.p_c2, d.p_d, e.p_e1

Scan necessary columns and apply filters

same computation given same b assignment
same computation given same c assignment
D-Representation for Expression Reuse Example

Query Vertex Ordering:

\[ [a, b, c, d, e] \]

WHERE \( P(a, b, c, d, e) \)
RETURN \( b.p, c.p_{c1}, c.p_{c2}, d.p, e.p_{e1} \)

Scan necessary columns and apply filters
D-Representation for Expression Reuse Example

Query Vertex Ordering:
\[ [a, b, c, d, e] \]

WHERE \( P(a, b, c, d, e) \)
RETURN \( b.p_b, c.p_{c1}, c.p_{c2}, d.p_d, e.p_{e1} \)

Scan necessary columns and apply filters

same computation given same \( b \) assignment

same computation given same \( c \) assignment

same computation given same \( d \) assignment
D-Representation for Expression Reuse Example

Query Vertex Ordering:
[a, b, c, d, e]

WHERE  P(a, b, c, d, e)
RETURN b.p_{b}, c.p_{c1}, c.p_{c2}, d.p_{d}, e.p_{e1}

Scan necessary columns and apply filters

Scan_{F} a  \quad \text{Join}_{F} b  \quad \text{Join}_{F} c  \quad \text{Join}_{F} d  \quad \text{Join}_{F} e  \quad \text{Sink} (\text{Accumulate})

same computation given same b assignment

\text{Join}_{F} d  \quad \text{Join}_{F} e  \quad \text{Sink} (\text{Accumulate})

same computation given same c assignment

\text{Join}_{F} e  \quad \text{Sink} (\text{Accumulate})

same computation given same d assignment
D-Representation for Expression Reuse Example

Query Vertex Ordering:
\[ [a, b, c, d, e] \]

WHERE  \( P(a, b, c, d, e) \)
RETURN  \( b.p_b, c.p_{c1}, c.p_{c2}, d.p_d, e.p_{e1} \)

Scan necessary columns and apply filters

same computation given same \( b \) assignment

same computation given same \( c \) assignment

same computation given same \( d \) assignment


Extending In-Memory Relational Database Engines with Native Graph Support. Hassan et al. In EDBT, 2017.


IBM DB2 Graph: Supporting Synergistic and Retrofittable Graph Queries Inside IBM Db2. Tian et al., In SIGMOD 2020.


Will be on the academic job market for 2023! (http://amine.io/)