Optimal Join Algorithms meet Top-$k$

Nikolaos Tziavelis, Wolfgang Gatterbauer, Mirek Riedewald
Northeastern University, Boston

Part 1: Top-$k$

Slides: https://northeastern-datalab.github.io/topk-join-tutorial/
DOI: https://doi.org/10.1145/3318464.3383132
Data Lab: https://db.khoury.northeastern.edu

This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 4.0 International License. See https://creativecommons.org/licenses/by-nc-sa/4.0/ for details.
Why "Optimal Join Algorithms meet Top-$k$"?

**Optimal Join algorithms**

Return all results over joins

⇒ How to avoid large intermediate results?

**Top-$k$**

Given $k$, return $k$ “best” results

⇒ How to avoid working on any lower ranked results?

**Ranked Enumeration (Any-$k$)**

Incrementally return the $k$ “best” results over joins (for any $k = 1, 2, ...$)

⇒ How to most effectively push sorting through joins?
Top-$k$

- middleware cost model (# accesses)
- small result size; wish: $O(k)$
- return only $k$-best results

Any-$k$

- ranking function
- most important results first
- return only $k$-best results
- return all results; wish: $O(r), r > n$
- incremental computation

Optimal Join Algorithms

- RAM cost model
- conjunctive queries
- query decompositions
- minimize intermediate results
- all results are equally important

Conjunctive queries wish to minimize intermediate results.
Outline tutorial

• Part 1: Top-$k$ (Wolfgang): ~20min
  – Top-$k$ selection problem
  – Threshold algorithm [Fagin+ '03]
  – Top-$k$ join problem
  – $J^*$ algorithm [Natsev+ '01]
  – Discussion on cost models

• Part 2: Optimal Join Algorithms (Mirek): ~30min

• Part 3: Ranked enumeration over joins (Nikolaos): ~40min
Top-$k$ Selection Query: overall setup

- $n$ objects $X_1, X_2, ..., X_n$ with $\ell$ numeric weight attributes $w_1, w_2, ..., w_\ell$
- weight of object = aggregate function over its weights $\rho(w_1, w_2, ..., w_\ell) = \rho(X)$
- Goal: Find top-$k$ objects according to some order (e.g. min)

<table>
<thead>
<tr>
<th>id</th>
<th>$w_1$</th>
<th>$w_2$</th>
<th>$w_3$</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>$X_2$</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>$X_3$</td>
<td>6</td>
<td>8</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>$X_4$</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>18</td>
</tr>
<tr>
<td>$X_5$</td>
<td>8</td>
<td>7</td>
<td>5</td>
<td>20</td>
</tr>
</tbody>
</table>

Example aggregate function: $\rho = \text{sum}\{w_1, w_2, w_3\}$

$\rho(X_i)$ for every $X_i \in T$ and every $X_j \notin T$

In most original papers assumed to be max!
Top-$k$ Selection Query: information in different relations

- Weights are stored in $\ell$ distinct relations $R_i$

<table>
<thead>
<tr>
<th></th>
<th>$w_1$</th>
<th>$w_2$</th>
<th>$w_3$</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>$X_2$</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>$X_3$</td>
<td>6</td>
<td>8</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>$X_4$</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>18</td>
</tr>
<tr>
<td>$X_5$</td>
<td>8</td>
<td>7</td>
<td>5</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>$w_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td>3</td>
</tr>
<tr>
<td>$X_2$</td>
<td>4</td>
</tr>
<tr>
<td>$X_3$</td>
<td>6</td>
</tr>
<tr>
<td>$X_4$</td>
<td>7</td>
</tr>
<tr>
<td>$X_5$</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>$w_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td>4</td>
</tr>
<tr>
<td>$X_2$</td>
<td>2</td>
</tr>
<tr>
<td>$X_3$</td>
<td>8</td>
</tr>
<tr>
<td>$X_4$</td>
<td>6</td>
</tr>
<tr>
<td>$X_5$</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>$w_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td>3</td>
</tr>
<tr>
<td>$X_2$</td>
<td>4</td>
</tr>
<tr>
<td>$X_3$</td>
<td>1</td>
</tr>
<tr>
<td>$X_4$</td>
<td>6</td>
</tr>
<tr>
<td>$X_5$</td>
<td>5</td>
</tr>
</tbody>
</table>
Top-$k$ Selection Query: sorted access

- Weights are stored in $\ell$ distinct relations $R_i$
  - each $R_i$ is sorted by attribute $w_i$

<table>
<thead>
<tr>
<th></th>
<th>$w_1$</th>
<th>$w_2$</th>
<th>$w_3$</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>$X_2$</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>$X_3$</td>
<td>6</td>
<td>8</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>$X_4$</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>18</td>
</tr>
<tr>
<td>$X_5$</td>
<td>8</td>
<td>7</td>
<td>5</td>
<td>20</td>
</tr>
</tbody>
</table>

$R_1$

<table>
<thead>
<tr>
<th></th>
<th>$w_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td>3</td>
</tr>
<tr>
<td>$X_2$</td>
<td>4</td>
</tr>
<tr>
<td>$X_3$</td>
<td>6</td>
</tr>
<tr>
<td>$X_4$</td>
<td>7</td>
</tr>
<tr>
<td>$X_5$</td>
<td>8</td>
</tr>
</tbody>
</table>

$R_2$

<table>
<thead>
<tr>
<th></th>
<th>$w_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td>4</td>
</tr>
<tr>
<td>$X_2$</td>
<td>2</td>
</tr>
<tr>
<td>$X_3$</td>
<td>8</td>
</tr>
<tr>
<td>$X_4$</td>
<td>6</td>
</tr>
<tr>
<td>$X_5$</td>
<td>7</td>
</tr>
</tbody>
</table>

$R_3$

<table>
<thead>
<tr>
<th></th>
<th>$w_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td>3</td>
</tr>
<tr>
<td>$X_2$</td>
<td>4</td>
</tr>
<tr>
<td>$X_3$</td>
<td>1</td>
</tr>
<tr>
<td>$X_4$</td>
<td>6</td>
</tr>
<tr>
<td>$X_5$</td>
<td>5</td>
</tr>
</tbody>
</table>
Top-$k$ Selection Query: sorted access

- Weights are stored in $\ell$ distinct relations $R_i$
  - each $R_i$ is sorted by attribute $w_i$

Notice we sort in increasing order.
Top-$k$ Selection Query: "middleware" assumption

- **Weights** are stored in $\ell$ distinct relations $R_i$
  - each $R_i$ is sorted by attribute $w_i$
- **Goal**: Find top-$k$ with minimal access cost
  - get next object in $R_i$ sequentially: "sorted" sequential access cost $c_{seq}$
  - obtain the weight for a specific object in $R_i$: random access (index lookup) cost $c_{rand}$

**Assumption 1**: Middleware cost model:
- we aggregate rankings of other services.
  - we only pay for accesses to attribute lists
  - 2 types of access: sequential / random

<table>
<thead>
<tr>
<th>id</th>
<th>$w_1$</th>
<th>$w_2$</th>
<th>$w_3$</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>$X_2$</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>$X_3$</td>
<td>6</td>
<td>8</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>$X_4$</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>18</td>
</tr>
<tr>
<td>$X_5$</td>
<td>8</td>
<td>7</td>
<td>5</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>$w_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td>3</td>
</tr>
<tr>
<td>$X_2$</td>
<td>4</td>
</tr>
<tr>
<td>$X_3$</td>
<td>6</td>
</tr>
<tr>
<td>$X_4$</td>
<td>7</td>
</tr>
<tr>
<td>$X_5$</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>$w_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td>2</td>
</tr>
<tr>
<td>$X_2$</td>
<td>4</td>
</tr>
<tr>
<td>$X_4$</td>
<td>6</td>
</tr>
<tr>
<td>$X_5$</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>$w_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_3$</td>
<td>1</td>
</tr>
<tr>
<td>$X_1$</td>
<td>3</td>
</tr>
<tr>
<td>$X_2$</td>
<td>4</td>
</tr>
<tr>
<td>$X_5$</td>
<td>5</td>
</tr>
</tbody>
</table>

Notice we sort in increasing order.
Top-\(k\) Selection Query as a Join Problem

- Weights are stored in \(\ell\) distinct relations \(R_i\)
  - each \(R_i\) is sorted by attribute \(w_i\)
- Goal: Find top-\(k\) with minimal access cost
  - get next object in \(R_i\) sequentially: "sorted" sequential access cost \(c_{seq}\)
  - obtain the weight for a specific object in \(R_i\): random access (index lookup) cost \(c_{rand}\)

Assumption 1: Middleware cost model:
we aggregate rankings of other services.
- we only pay for accesses to attribute lists
- 2 types of access: sequential / random

\[
\begin{array}{c|c|c|c|c|c}
\text{id} & w_1 & w_2 & w_3 & \text{sum} \\
\hline
X_1 & 3 & 4 & 3 & 10 \\
X_2 & 4 & 2 & 4 & 10 \\
X_3 & 6 & 8 & 1 & 15 \\
X_4 & 7 & 6 & 6 & 18 \\
X_5 & 8 & 7 & 5 & 20 \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{id} & w_1 \\
\hline
X_1 & 3 \\
X_2 & 4 \\
X_3 & 6 \\
X_4 & 7 \\
X_5 & 8 \\
\end{array}
\quad
\begin{array}{c|c}
\text{id} & w_2 \\
\hline
X_1 & 4 \\
X_2 & 6 \\
X_4 & 7 \\
X_5 & 8 \\
\end{array}
\quad
\begin{array}{c|c}
\text{id} & w_3 \\
\hline
X_3 & 1 \\
X_1 & 3 \\
X_2 & 4 \\
X_3 & 5 \\
X_4 & 6 \\
\end{array}
\]

~ Joins on unique object id: 1-1 relationships

!!!select R_1.id, sum(w_1, w_2, w_3) as weight from R_1, R_2, R_3 where R_1.id=R_2.id and R_2.id=R_3.id order by weight limit 2!!!
Naive algorithm: retrieve all items

- Weights are stored in $\ell$ distinct relations $R_i$
  - each $R_i$ is sorted by attribute $w_i$
- Goal: Find top-$k$ with minimal access cost
  - get next object in $R_i$ sequentially: "sorted" sequential access cost $c_{seq}$
  - obtain the weight for a specific object in $R_i$: random access (index lookup) cost $c_{rand}$

Assumption 1: Middleware cost model:
we aggregate rankings of other services.
- we only pay for accesses to attribute lists
- 2 types of access: sequential / random

<table>
<thead>
<tr>
<th>$X_1$</th>
<th>$X_2$</th>
<th>$X_3$</th>
<th>$X_4$</th>
<th>$X_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>8</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Cost = $n \cdot \ell \cdot c_{sort}$
Assumption 2: The aggregate function $\rho$ is monotone: $\rho(w_1, w_2, ..., w_\ell) \leq \rho(w'_1, w'_2, ..., w'_\ell)$ if $w_i \leq w'_i$ for all $i$.

Part 3: tropical semiring (min, sum) is instance of "selective dioid" (i.e. $\min(a,b) = a$ or $b$).

$\rho$ is decomposable: $\rho(w_1, w_2, w_3) = \rho\{w_1, w_2, w_3\}$

Assumption 1: Middleware cost model:
we aggregate rankings of other services.
• we only pay for accesses to attribute lists
• 2 types of access: sequential / random

Select $R_1.id,$
\[\text{sum}(w_1, w_2, w_3) \text{ as weight}\]
From $R_1, R_2, R_3$
Where $R_1.id = R_2.id$
And $R_2.id = R_3.id$
Order by weight
Limit 2

- Weights are stored in $\ell$ distinct relations $R_i$
  - each $R_i$ is sorted by attribute $w_i$
- Goal: Find top-$k$ with minimal access cost
  - get next object in $R_i$ sequentially: "sorted" sequential access cost $c_{\text{seq}}$
  - obtain the weight for a specific object in $R_i$: random access (index lookup) cost $c_{\text{rand}}$
Important early work making these assumptions

- **Fagin’s algorithm:**
  - Fagin. Combining fuzzy information from multiple systems. PODS 1996. [https://doi.org/10.1145/237661.237715](https://doi.org/10.1145/237661.237715)
  - Fagin. Fuzzy queries in multimedia database systems. PODS 1998. [https://doi.org/10.1145/275487.275488](https://doi.org/10.1145/275487.275488)
  - Fagin. Combining fuzzy information from multiple systems. JCSS 1999. [https://doi.org/10.1006/jcss.1998.1600](https://doi.org/10.1006/jcss.1998.1600)

- **Threshold Algorithm (TA):**
  - Nepal, Ramakrishna. Query processing issues in image (multimedia) databases. ICDE 1999. [https://doi.org/10.1109/ICDE.1999.754894](https://doi.org/10.1109/ICDE.1999.754894)
  - Fagin, Lotem, Naor. Optimal aggregation algorithms for middleware. JCSS 2003. [https://doi.org/10.1016/S0022-0000(03)00026-6](https://doi.org/10.1016/S0022-0000(03)00026-6)

2014 Gödel Prize on "a framework to design and analyze algorithms where aggregation of information from multiple data sources is needed... introduced the notion of instance optimality"
Outline tutorial

• Part 1: Top-$k$ (Wolfgang): ~20min
  – Top-$k$ selection problem
  – Threshold algorithm [Fagin+ '03]
  – Top-$k$ join problem
  – J* algorithm [Natsev+ '01]
  – Discussion on cost models
• Part 2: Optimal Join Algorithms (Mirek): ~30min
• Part 3: Ranked enumeration over joins (Nikolaos): ~40min
Threshold algorithm [Fagin+ 03]

1. Access next objects in all $R_i$ sequentially

<table>
<thead>
<tr>
<th></th>
<th>$R_1$</th>
<th>$R_2$</th>
<th>$R_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>$w_1$</td>
<td>$w_2$</td>
<td>$w_3$</td>
</tr>
<tr>
<td>$X_1$</td>
<td>3</td>
<td>$X_2$</td>
<td>2</td>
</tr>
<tr>
<td>$X_2$</td>
<td>4</td>
<td>$X_1$</td>
<td>4</td>
</tr>
<tr>
<td>$X_3$</td>
<td>6</td>
<td>$X_4$</td>
<td>6</td>
</tr>
<tr>
<td>$X_4$</td>
<td>7</td>
<td>$X_5$</td>
<td>7</td>
</tr>
<tr>
<td>$X_5$</td>
<td>8</td>
<td>$X_3$</td>
<td>8</td>
</tr>
</tbody>
</table>

Threshold algorithm [Fagin+ 03]

1. Access next objects in all $R_i$ sequentially
   a. Set threshold $\tau$ to the aggregate of the weights last seen in sorted access

\begin{align*}
\tau &= \text{sum}(3, 2, 1) = 6
\end{align*}

<table>
<thead>
<tr>
<th></th>
<th>$R_1$</th>
<th></th>
<th>$R_2$</th>
<th></th>
<th>$R_3$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>$w_1$</td>
<td>id</td>
<td>$w_2$</td>
<td>id</td>
<td>$w_3$</td>
<td></td>
</tr>
<tr>
<td>$X_1$</td>
<td>3</td>
<td>$X_2$</td>
<td>2</td>
<td>$X_3$</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$X_2$</td>
<td>4</td>
<td>$X_1$</td>
<td>4</td>
<td>$X_1$</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>$X_3$</td>
<td>6</td>
<td>$X_4$</td>
<td>6</td>
<td>$X_2$</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>$X_4$</td>
<td>7</td>
<td>$X_5$</td>
<td>7</td>
<td>$X_5$</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>$X_5$</td>
<td>8</td>
<td>$X_3$</td>
<td>8</td>
<td>$X_4$</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Threshold algorithm [Fagin+ 03]

1. Access next objects in all $R_i$ sequentially
   a. Set threshold $\tau$ to the aggregate of the weights last seen in sorted access
   b. Use random accesses and compute the aggregate weights $\rho$ of all objects seen

<table>
<thead>
<tr>
<th>id</th>
<th>$w_1$</th>
<th>$w_2$</th>
<th>$w_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$X_2$</td>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>$X_3$</td>
<td>5</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>$X_4$</td>
<td>6</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>$X_5$</td>
<td>7</td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

$\tau = \text{sum}(3, 2, 1) = 6$
Threshold algorithm [Fagin+ 03]

1. Access next objects in all $R_i$ sequentially
   a. Set threshold $\tau$ to the aggregate of the weights last seen in sorted access
   b. Use random accesses and compute the aggregate weights $\rho$ of all objects seen

\[
\begin{array}{|c|c|c|c|}
\hline
\text{id} & w_1 & w_2 & w_3 \\
\hline
X_1 & 3 & 4 & 3 \\
X_2 & 2 &   &   \\
X_3 &   &   & 1 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{id} & w_1 \\
\hline
X_1 & 3 \\
X_2 & 4 \\
X_3 & 6 \\
X_4 & 7 \\
X_5 & 8 \\
\hline
\end{array}
\quad
\begin{array}{|c|c|}
\hline
\text{id} & w_2 \\
\hline
X_1 & 4 \\
X_4 & 6 \\
X_5 & 7 \\
\hline
\end{array}
\quad
\begin{array}{|c|c|}
\hline
\text{id} & w_3 \\
\hline
X_1 & 3 \\
X_2 & 4 \\
X_3 & 8 \\
X_4 & 6 \\
\hline
\end{array}
\]

$\tau = \text{sum}(3, 2, 1) = 6$

Threshold algorithm [Fagin+ 03]

1. Access next objects in all $R_i$ sequentially
   
a. Set threshold $\tau$ to the aggregate of the weights last seen in sorted access
   
b. Use random accesses and compute the aggregate weights $\rho$ of all objects seen

<table>
<thead>
<tr>
<th></th>
<th>$w_1$</th>
<th>$w_2$</th>
<th>$w_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>$X_2$</td>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>$X_3$</td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

$\tau = \text{sum}(3, 2, 1) = 6$
Threshold algorithm [Fagin+ 03]

1. Access next objects in all $R_i$ sequentially
   a. Set threshold $\tau$ to the aggregate of the weights last seen in sorted access
   b. Use random accesses and compute the aggregate weights $\rho$ of all objects seen

\[
\begin{array}{cccc}
\text{id} & w_1 & w_2 & w_3 \\
X_1 & 3 & 4 & 3 \\
X_2 & 4 & 2 & 4 \\
X_3 & 6 & 8 & 1 \\
\end{array}
\]

\[
\begin{array}{ccc}
R_1 & R_2 & R_3 \\
\begin{array}{cccc}
\text{id} & w_1 & & \\
X_1 & 3 & & \\
X_2 & 4 & & \\
X_3 & 6 & & \\
X_4 & 7 & & \\
X_5 & 8 & & \\
\end{array} & \\
\begin{array}{cccc}
\text{id} & w_2 & & \\
X_1 & 4 & & \\
X_4 & 6 & & \\
X_5 & 7 & & \\
X_3 & 8 & & \\
\end{array} & \\
\begin{array}{cccc}
\text{id} & w_3 & & \\
X_3 & 1 & & \\
X_2 & 4 & & \\
X_5 & 5 & & \\
X_4 & 6 & & \\
\end{array}
\end{array}
\]

$\tau = \text{sum}(3, 2, 1) = 6$

Threshold algorithm [Fagin+ 03]

1. Access next objects in all $R_i$ sequentially
   a. Set threshold $\tau$ to the aggregate of the weights last seen in sorted access
   b. Use random accesses and compute the aggregate weights $\rho$ of all objects seen

<table>
<thead>
<tr>
<th>id</th>
<th>$w_1$</th>
<th>$w_2$</th>
<th>$w_3$</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>$X_2$</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>$X_3$</td>
<td>6</td>
<td>8</td>
<td>1</td>
<td>15</td>
</tr>
</tbody>
</table>

$k=2$

Focus only on top-$k$ (can purge rest)

$\tau = \text{sum}(3, 2, 1) = 6$

Threshold algorithm [Fagin+ 03]

1. Access next objects in all $R_i$ sequentially
   a. Set threshold $\tau$ to the aggregate of the weights last seen in sorted access
   b. Use random accesses and compute the aggregate weights $\rho$ of all objects seen
   c. Continue until the aggregate weights $\rho$ of the top-$k \leq \tau$

| $X_1$ | 3 | 4 | 3 | 10 |
| $X_2$ | 4 | 2 | 4 | 10 |
| $X_3$ | 6 | 8 | 1 | 15 |

Focus only on top-$k$ (can purge rest)

$\tau = \text{sum}(3, 2, 1) = 6$

10 \not\leq 6: continue: access next objects sequentially

Threshold algorithm [Fagin+ 03]

1. Access next objects in all $R_i$ sequentially
   a. Set threshold $\tau$ to the aggregate of the weights last seen in sorted access
   b. Use random accesses and compute the aggregate weights $\rho$ of all objects seen
   c. Continue until the aggregate weights $\rho$ of the top-$k \leq \tau$

   \[
   \begin{array}{c|cccc|c}
   \text{id} & w_1 & w_2 & w_3 & \text{sum} \\
   \hline
   X_1 & 3 & 4 & 3 & 10 \\
   X_2 & 4 & 2 & 4 & 10 \\
   X_3 & 6 & 8 & 1 & 15 \\
   \end{array}
   \]

   \[
   \begin{array}{c|c|c|c|c}
   R_1 & \text{id} & w_1 & \text{id} & w_2 & \text{id} & w_3 \\
   \hline
   X_1 & 3 & \text{--} & \text{--} & \text{--} & \text{--} \\
   X_2 & 4 & \text{--} & \text{--} & \text{--} & \text{--} \\
   X_3 & \text{--} & \text{--} & \text{--} & \text{--} \\
   X_4 & \text{--} & \text{--} & \text{--} & \text{--} \\
   X_5 & \text{--} & \text{--} & \text{--} & \text{--} \\
   \end{array}
   \]

   \[
   \begin{array}{c|c|c|c|c}
   R_2 & \text{id} & w_1 & \text{id} & w_2 & \text{id} & w_3 \\
   \hline
   X_1 & \text{--} & \text{--} & 4 & \text{--} & \text{--} \\
   X_2 & \text{--} & \text{--} & \text{--} & \text{--} & \text{--} \\
   X_4 & \text{--} & \text{--} & \text{--} & \text{--} & \text{--} \\
   X_5 & \text{--} & \text{--} & \text{--} & \text{--} & \text{--} \\
   \end{array}
   \]

   \[
   \begin{array}{c|c|c|c|c}
   R_3 & \text{id} & w_1 & \text{id} & w_2 & \text{id} & w_3 \\
   \hline
   X_3 & \text{--} & \text{--} & \text{--} & 1 & \text{--} \\
   X_2 & \text{--} & \text{--} & \text{--} & \text{--} & 4 \\
   X_5 & \text{--} & \text{--} & \text{--} & \text{--} & \text{--} \\
   \end{array}
   \]

   $\tau = \text{sum}(3,2,1) = 6$
   $\tau = \text{sum}(4,4,3) = 11$

   focus only on top-$k$
   (can purge rest)

   $10 \leq 11$: stop!

Threshold algorithm [Fagin+ 03]

- Why can we avoid looking at $X_4$?

From the monotonicity property: for any object not seen, the score of the object is bigger than the threshold $\tau$.

\[ \rho(X_4) \geq \tau \]

\[ \tau = \text{sum}(4, 4, 3) = 11 \]
Instance Optimality of Threshold Algorithm (TA)

• The TA algorithm is **instance cost-optimal**
  – within a constant factor of the best algorithm on any database*

• Let \( \text{cost}(A, D) = \text{access cost of algorithm } A \text{ on database } D \):
  – \( \text{cost}(\text{TA}, D) = O(\text{cost}(A, D)) \) for all \( A \) and \( D \)

* Excluding those that make “wild guesses” = random access to object without first seeing it with sorted access

Outline tutorial

• Part 1: Top-\(k\) (Wolfgang): \(\sim20\)min
  – Top-\(k\) selection problem
  – Threshold algorithm [Fagin+ '03]
  – Top-\(k\) join problem
  – J* algorithm [Natsev+ '01]
  – Discussion on cost models

• Part 2: Optimal Join Algorithms (Mirek): \(\sim30\)min

• Part 3: Ranked enumeration over joins (Nikolaos): \(\sim40\)min
Goal: Generalize TA setup to arbitrary join patterns

- Same cost model: measuring access cost
  - to simplify, we ignore random accesses

- Many-to-many relationships
- No unique identifier per join result
- Arbitrary join conditions possible

natural join

```
select A_1, A_2, A_3, A_4,
    sum(w_1, w_2, w_3) as weight
from R_1, R_2, R_3
where R_1.A_2 = R_2.A_2
    and R_2.A_3 = R_3.A_3
order by weight
limit 1
```
Outline tutorial

• Part 1: Top-$k$ (Wolfgang): ~20min
  – Top-$k$ selection problem
  – Threshold algorithm [Fagin+ '03]
  – Top-$k$ join problem
  – J* algorithm [Natsev+ '01]
  – Discussion on cost models

• Part 2: Optimal Join Algorithms (Mirek): ~30min

• Part 3: Ranked enumeration over joins (Nikolaos): ~40min
J* Algorithm [Natsev+ 01]

- Idea: A* search on the Cartesian product to find top-$k$ join results
  - Keep Priority Queue (PQ) of partial results
  - Pop partial result with smallest lower bound (based on what has been seen) and access lists to extend it

---

Partial Solution | Next Tuple | Lower Bound
--- | --- | ---
() | $R_1: 1$ | $0+0+0=0$

---

![Diagram showing partial solutions and lower bounds](image_url)
J* Algorithm [Natsev+ 01]

- Idea: A* search on the Cartesian product to find top-$k$ join results
  - Keep Priority Queue (PQ) of partial results
  - Pop partial result with smallest lower bound (based on what has been seen) and access lists to extend it
  - If still incomplete, push back 2 new ones: one “longer”, one “deeper”

\[
\begin{array}{c|c|c|c}
\text{Partial Solution} & \text{Next Tuple} & \text{Lower bound} \\
\hline
(1) & R_2:1 & 1+0+0=1 \\
() & R_1:2 & 1+0+0=1 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c}
A_1 & A_2 & w_1 \\
\hline
\text{a} & \text{b} & 1 \\
\text{c} & \text{b} & 4 \\
\text{b} & \text{d} & 5 \\
\end{array}
\quad
\begin{array}{c|c|c|c}
A_2 & A_3 & w_2 \\
\hline
\text{d} & \text{c} & 1 \\
\text{b} & \text{c} & 2 \\
\text{b} & \text{a} & 3 \\
\end{array}
\quad
\begin{array}{c|c|c|c}
A_3 & A_4 & w_3 \\
\hline
\text{c} & \text{d} & 1 \\
\text{a} & \text{a} & 2 \\
\text{a} & \text{d} & 3 \\
\end{array}
\]

J* Algorithm [Natsev+ 01]

- Idea: A* search on the Cartesian product to find top-$k$ join results
  - Keep Priority Queue (PQ) of partial results
  - Pop partial result with smallest lower bound (based on what has been seen) and access lists to extend it
  - If still incomplete, push back 2 new ones: one “longer”, one “deeper”

---

Partial Solution

Next Tuple Lower bound

| (1)   | $R_2$: 1 | 1 + 0 + 0 = 1 |
| ()    | $R_1$: 2 | 1 + 0 + 0 = 1 |

J* Algorithm [Natsev+ 01]

- Idea: A* search on the Cartesian product to find top-k join results
  - Keep Priority Queue (PQ) of partial results
  - Pop partial result with smallest lower bound (based on what has been seen) and access lists to extend it
  - If still incomplete, push back 2 new ones: one “longer”, one “deeper”

<table>
<thead>
<tr>
<th>Partial Solution</th>
<th>Next Tuple</th>
<th>Lower Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>R₂:2</td>
<td>1+1+0=2</td>
</tr>
<tr>
<td>()</td>
<td>R₁:2</td>
<td>1+0+0=1</td>
</tr>
</tbody>
</table>

Partial Solution Next Tuple Lower Bound
(1) R₂:2 1+1+0=2
()
R₁:2 1+0+0=1

Invalid join condition, thus discard partial solution

J* Algorithm [Natsev+ 01]

- Idea: A* search on the Cartesian product to find top-k join results
  - Keep Priority Queue (PQ) of partial results
  - Pop partial result with smallest lower bound (based on what has been seen) and access lists to extend it
  - If still incomplete, push back 2 new ones: one "longer”, one “deeper”

Partial Solution       Next Tuple       Lower bound
(1)       R₂:2       1+1+0=2
()        R₁:2       1+0+0=1

**J* Algorithm [Natsev+ 01]**

- Idea: A* search on the Cartesian product to find top-$k$ join results
  - Keep Priority Queue (PQ) of partial results
  - Pop partial result with smallest lower bound (based on what has been seen) and access lists to extend it
  - If still incomplete, push back 2 new ones: one “longer”, one “deeper”

---

Partial Solution | Next Tuple | Lower bound
--- | --- | ---
(1) | R₂:2 | 1+1+0=2
(2) | R₂:1 | 4+0+0=4
(3) | R₁:3 | 4+0+0=4

---

<table>
<thead>
<tr>
<th>$R_1$</th>
<th>$A_1$</th>
<th>$A_2$</th>
<th>$w_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>d</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>b</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$R_2$</th>
<th>$A_2$</th>
<th>$A_3$</th>
<th>$w_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>c</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>c</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$R_3$</th>
<th>$A_3$</th>
<th>$A_4$</th>
<th>$w_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>d</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>d</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

---

**J* Algorithm [Natsev+ 01]**

- **Idea:** A* search on the Cartesian product to find top-$k$ join results
  - Keep Priority Queue (PQ) of partial results
  - Pop partial result with smallest lower bound (based on what has been seen) and access lists to extend it
  - If still incomplete, push back 2 new ones: one “longer”, one “deeper”

---

**Table:**

<table>
<thead>
<tr>
<th>Partial Solution</th>
<th>Next Tuple</th>
<th>Lower bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>R₂:2</td>
<td>1+1+0=2</td>
</tr>
<tr>
<td>(2)</td>
<td>R₂:1</td>
<td>4+0+0=4</td>
</tr>
<tr>
<td>()</td>
<td>R₁:3</td>
<td>4+0+0=4</td>
</tr>
</tbody>
</table>

**Diagram:**

- **$R_1$**
  - $A_1$: a, b
  - $A_2$: b, c
  - $w_1$: 1, 4
- **$R_2$**
  - $A_2$: d, c
  - $A_3$: b, c
  - $w_2$: 1, 2
- **$R_3$**
  - $A_3$: c, d
  - $A_4$: a, a
  - $w_3$: 1, 2

---

J* Algorithm [Natsev+ 01]

- Idea: A* search on the Cartesian product to find top-\(k\) join results
  - Keep Priority Queue (PQ) of partial results
  - Pop partial result with smallest lower bound (based on what has been seen) and access lists to extend it
  - If still incomplete, push back 2 new ones: one “longer”, one “deeper”

---

<table>
<thead>
<tr>
<th>Partial Solution</th>
<th>Next Tuple</th>
<th>Lower bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,2)</td>
<td>(R_3:1)</td>
<td>1+2+0=3</td>
</tr>
<tr>
<td>(1)</td>
<td>(R_2:3)</td>
<td>1+2+0=3</td>
</tr>
<tr>
<td>(2)</td>
<td>(R_2:1)</td>
<td>4+0+0=4</td>
</tr>
<tr>
<td>()</td>
<td>(R_1:3)</td>
<td>4+0+0=4</td>
</tr>
</tbody>
</table>

---

![Diagram of J* Algorithm](https://doi.org/10.5555/645927.672365)
J* Algorithm [Natsev+ 01]

- Idea: A* search on the Cartesian product to find top-k join results
  - Keep Priority Queue (PQ) of partial results
  - Pop partial result with smallest lower bound (based on what has been seen) and access lists to extend it
  - If still incomplete, push back 2 new ones: one “longer”, one “deeper”

Partial Solution  | Next Tuple | Lower bound
--- | --- | ---
(1,2) | R₃:1 | 1+2+0=3
(1) | R₂:3 | 1+2+0=3
(2) | R₂:1 | 4+0+0=4
() | R₁:3 | 4+0+0=4

---

J* Algorithm [Natsev+ 01]

• Idea: A* search on the Cartesian product to find top-$k$ join results
  – Keep Priority Queue (PQ) of partial results
  – Pop partial result with smallest lower bound (based on what has been seen) and access lists to extend it
  – If still incomplete, push back 2 new ones: one “longer”, one “deeper”

---

<table>
<thead>
<tr>
<th>Partial Solution</th>
<th>Next Tuple</th>
<th>Lower bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,2,1)</td>
<td>R₂:2</td>
<td>1+2+1=4</td>
</tr>
<tr>
<td>(1,2)</td>
<td>R₂:3</td>
<td>1+2+0=3</td>
</tr>
<tr>
<td>(1)</td>
<td>R₂:3</td>
<td>1+2+0=3</td>
</tr>
<tr>
<td>(2)</td>
<td>R₂:1</td>
<td>4+0+0=4</td>
</tr>
<tr>
<td>()</td>
<td>R₁:3</td>
<td>4+0+0=4</td>
</tr>
</tbody>
</table>

---

J* Algorithm [Natsev+ 01]

- Idea: A* search on the Cartesian product to find top-$k$ join results
  - Keep Priority Queue (PQ) of partial results
  - Pop partial result with smallest lower bound (based on what has been seen) and access lists to extend it
  - If still incomplete, push back 2 new ones: one “longer”, one “deeper”

Partial Solution | Next Tuple | Lower bound
--- | --- | ---
(1,2,1) |  | 1+2+1=4
(1,2) | R₂:2 | 1+2+1=4
(1,3) | R₃:2 | 1+3+0=4
(2) | R₃:1 | 4+0+0=4
() | R₁:3 | 4+0+0=4

---

**J* Algorithm [Natsev+ 01]**

- Idea: A* search on the Cartesian product to find top-\(k\) join results
  - Keep Priority Queue (PQ) of partial results
  - Pop partial result with smallest lower bound (based on what has been seen) and access lists to extend it
  - If still incomplete, push back 2 new ones: one “longer”, one “deeper”

<table>
<thead>
<tr>
<th>Partial Solution</th>
<th>Next Tuple</th>
<th>Lower bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,2,1)</td>
<td>R₃:2</td>
<td>1+2+1=4</td>
</tr>
<tr>
<td>(1,2)</td>
<td>R₃:1</td>
<td>1+2+1=4</td>
</tr>
<tr>
<td>(1,3)</td>
<td>R₃:1</td>
<td>1+3+0=4</td>
</tr>
<tr>
<td>(2)</td>
<td>R₂:1</td>
<td>4+0+0=4</td>
</tr>
<tr>
<td>()</td>
<td>R₁:3</td>
<td>4+0+0=4</td>
</tr>
</tbody>
</table>

**Partial Solution**

- **Next Tuple**
- **Lower bound**

![Diagram](https://via.placeholder.com/150)

---

J* w/ iterative deepening [Natsev+ 01] & Rank Join [Ilyas+ 04]

- To guarantee instance optimality for J*, go deeper only after producing all results (iterative deepening) [Natsev+ 01]
- Rank-Join [Ilyas+ 04]: Instead of A* type of search use a threshold value similarly to TA. Also instance-optimal in terms of accesses
- Many variants and much follow-up work (different join strategies, heuristics to prioritize relations, etc.)

![Diagram](https://via.placeholder.com/150)

**References**


Figures from [Ilyas+ 04]


Similar access cost, but different times in practice. Is # of access cost thus a reasonable cost model?
Outline tutorial

• Part 1: Top-\(k\) (Wolfgang): \(~20\text{min}\)
  – Top-\(k\) selection problem
  – Threshold algorithm [Fagin+ '03]
  – Top-\(k\) join problem
  – J* algorithm [Natsev+ '01]
    – Discussion on cost models
• Part 2: Optimal Join Algorithms (Mirek): \(~30\text{min}\)
• Part 3: Ranked enumeration over joins (Nikolaos): \(~40\text{min}\)
Middleware cost model vs. in-database join computations

<table>
<thead>
<tr>
<th></th>
<th>( R_1 )</th>
<th>( R_2 )</th>
<th>( R_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A_1 )</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( A_2 )</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( w_1 )</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>( w_2 )</td>
<td>10</td>
<td>20</td>
<td>100</td>
</tr>
<tr>
<td>( w_3 )</td>
<td>90</td>
<td>900</td>
<td></td>
</tr>
</tbody>
</table>


Middleware cost model vs. in-database join computations

- J* and Rank-Join produce $n^2$ partial results to find top-1 result
  - Are number of accesses a realistic measure for in-database join computation?
    - E.g. if tables are available in a database, we don't have to fetch tuples over a network.

⇒ How to most effectively push sorting through joins?

<table>
<thead>
<tr>
<th>RAM cost model</th>
</tr>
</thead>
<tbody>
<tr>
<td>In-memory join comp.</td>
</tr>
<tr>
<td>quadratic cost</td>
</tr>
<tr>
<td>in-memory processing: join time matters</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Middleware cost model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimize access depth</td>
</tr>
<tr>
<td>linear cost</td>
</tr>
<tr>
<td>Information retrieval: latency/ access cost matters</td>
</tr>
</tbody>
</table>

Assuming sorted accesses only. If random accesses allowed, another slightly more complicated example shows the same issue.


A natural question:

What can one do under a RAM cost model for general conjunctive queries?

Assuming sorted accesses only. If random accesses allowed, another slightly more complicated example shows the same issue.


\( n^2 \) partial results to find top-1 result

- Are number of accesses a realistic measure for in-database join computation?

E.g. if tables are available in a database, we don't have to fetch tuples over a network.

\( \Rightarrow \) How to most effectively push sorting through joins?
An excerpt of rich literature, once access determines cost ...

- What if the ranking function is the distance from a desired (high-dimensional) point?
  - [Bruno+ TODS’02]: Rewrite as a range query and restart if #results < k
- What if we are allowed to pre-compute data structures and learn the ranking function at query time?
  - [Tsaparas+ ICDE’03]: Find linear ranking functions that act as “separators” (i.e., they change the top-k set)
  - [Chang+ SIGMOD’00]: Construct convex hulls for linear ranking functions
  - [Hristidis+ SIGMOD’01, Das+ VLDB’06]: Materialize ranked views for some selected ranking functions
- What if the ranking function is non-monotone?
  - [Zhang+ SIGMOD’06]: Use continuous function optimization methods
- What if the query model is different?
  - "SMART" [Wu+ VLDB’10]: Query contains disjunctions, partial results allowed to be returned
- ...

Please see dedicated tutorials and surveys on top-k

Outline tutorial

• Part 1: Top-$k$ (Wolfgang): ~20min
  – Top-$k$ selection problem
  – Threshold algorithm [Fagin+ '03]
  – Top-$k$ join problem
  – J* algorithm [Natsev+ '01]
  – Discussion on cost models

• Part 2: Optimal Join Algorithms (Mirek): ~30min

• Part 3: Ranked enumeration over joins (Nikolaos): ~40min