Wei Wang

weiw at cse.unsw.edu.au
http://www.cse.unsw.edu.au/~weiw

School of Computer Science & Engineering
University of New South Wales

16 July, 2007
My research revolves around *query processing & optimization* issues.

Research interests:

- DB + IR
- Data Warehousing, Data Integration, and Data Mining
- XML Query Processing & Optimization
- Spatial Databases

This talk is based on our recent SIGMOD07 paper.
1 Introduction

2 Ranking Search Results

3 Efficient Query Processing

4 Experiments

5 Related Work

6 Conclusions
Example:

Fail to locate “World Wide Web” in the following paragraph.

... paradigm and have made the vast amount of information on the World Wide Web available to even casual users.

Should search within the scope of a “logical unit”.

Non-trivial to support such search

- Use regular expressions? ⇐ only works for linear layout of texts
- Find individual matches and grow into “clusters”
- Scale to disk-resident data?

Other issues that matter

- Ranking
- Efficiency
Keyword Search in Relational Databases

- What is the “logical unit” in databases?

Query: netvista maxtor
Keyword Search in Relational Databases

What is the “logical unit” in databases?

**Query:** netvista maxtor

<table>
<thead>
<tr>
<th>prodID</th>
<th>custID</th>
<th>comment</th>
<th>solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>P121</td>
<td>C111</td>
<td>“disk crashed after ... on an IBM Netvista X41”</td>
</tr>
<tr>
<td>c2</td>
<td>P131</td>
<td>C222</td>
<td>“lower-end IBM Netvista caught fire, ...”</td>
</tr>
<tr>
<td>c3</td>
<td>P131</td>
<td>C333</td>
<td>“IBM Netvista unstable with Maxtor HD”</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>prodID</th>
<th>model</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>P121</td>
<td>“D540X” “1.7G CPU ... , Maxtor 80G HD”</td>
</tr>
<tr>
<td>p2</td>
<td>P131</td>
<td>“D710P” “1.5G CPU ... , Maxtor 200G HD”</td>
</tr>
</tbody>
</table>

**Top-5 results:** c3, c1, c3 → p2, c1 → p1, c2 → p2

**Results in the form of JTT (Joined Tuple Tree)**
Applications

Keyword search in DB is good for average users:

- “What’s SQL?”
- “I don’t know what’s the cryptic attribute names in your database, but can’t you just show me all you know about John?”
- “I am looking for a movie, but I cannot remember its name, etc. All I know is that one actor is named Jim and the film was probably out in 1995”

Other applications:

- Enterprise search
- CMS
- CRM
- Explorative / interactive data analysis
IMDb Search

A search for "2001 hanks" found the following results:

Titles (Approx Matches) (Displaying 10 Results)
   a.k.a. "Tony Hawk's Gigantic Skate Park Tour Summer 2002" - USA (complete title)
3. TV Hanks and Babies (2006) (TV)
7. Hard und die 200.000 Küken (1952)
   a.k.a. "Nils Hounsfield show 2001" - Denmark (promotional title)
   a.k.a. "Jonathan 2001" - Germany (working title)
   a.k.a. "Hans Wams: My 20th Century" - (English title)
   a.k.a. "Jan's Return from the Paracamb Exile to Prague in the Summer of 2003" - (English)
   a.k.a. "Capcom vs SNK 2: Mark of the Millennium 2001" - USA

Companies (Approx Matches) (Displaying 2 Results)
2. Ralston-Purina Co. Inc. (Production)
## IMDb Search

A search for "2001 hanks" found the following results:

### Titles (Approx Matches) (Displaying 18 Results)

<table>
<thead>
<tr>
<th>Rank</th>
<th>Title</th>
<th>Year</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>2001: HAL's Legacy</td>
<td>2001</td>
<td>(TV)</td>
</tr>
<tr>
<td>2.</td>
<td>Gigantic Skate Park Tour: Summer 2002</td>
<td>2002</td>
<td>(TV)</td>
</tr>
<tr>
<td></td>
<td>aka &quot;Tony Hawk's Gigantic Skate Park Tour: Summer 2002&quot; - USA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>TV Hunks and Babes 2006</td>
<td>2006</td>
<td>(TV)</td>
</tr>
<tr>
<td>4.</td>
<td>Danish Music Awards 2001</td>
<td>2001</td>
<td>(TV)</td>
</tr>
<tr>
<td>5.</td>
<td>Norah Jones &amp; the Handsome Band: Live in 2004</td>
<td>2004</td>
<td>(V)</td>
</tr>
<tr>
<td>6.</td>
<td>Danish music awards 2001 - Forspil med Casper og Lasse</td>
<td>2001</td>
<td>(TV)</td>
</tr>
<tr>
<td>7.</td>
<td>Hans und die 200.000 Küken</td>
<td>1952</td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td>Niels Hausgaard live i Cirkusbygningen</td>
<td>2001</td>
<td>(TV)</td>
</tr>
<tr>
<td></td>
<td>aka &quot;Niels Hausgaard show 2001&quot; - Denmark</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>aka &quot;Jonathan 2001&quot; - Germany</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10.</td>
<td>Brasilleirhias</td>
<td>2001</td>
<td></td>
</tr>
<tr>
<td>11.</td>
<td>Eurovision 2001: More Than Just a Song Contest</td>
<td>2001</td>
<td>(TV)</td>
</tr>
<tr>
<td>12.</td>
<td>Spellemanværtskabsprisen</td>
<td>2001</td>
<td>(TV)</td>
</tr>
<tr>
<td>13.</td>
<td>2001 Maniacs</td>
<td>2005</td>
<td></td>
</tr>
<tr>
<td></td>
<td>aka &quot;Hans Warna: My 20th Century&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>aka &quot;Jan's Return from the Parisian Exile to Prague in the Summer of 2003&quot; - (English title)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17.</td>
<td>Stars 2001 - Die Aids-Gala</td>
<td>2001</td>
<td>(TV)</td>
</tr>
<tr>
<td>18.</td>
<td>Capcom tai SNK 2: Mireonea faiingu 2001-nen</td>
<td>2001</td>
<td>(VG)</td>
</tr>
<tr>
<td></td>
<td>aka &quot;Capcom vs SNK 2: Mark of the Millennium 2001&quot; - USA</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Companies (Approx Matches) (Displaying 2 Results)

<table>
<thead>
<tr>
<th>Rank</th>
<th>Company</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Hans Christian Andersen 2005 Fonden</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td>2.</td>
<td>Rhapsody 2001 Inc</td>
<td>Production</td>
</tr>
</tbody>
</table>
Query Results - Mozilla Firefox

SPARK
searching, probing & ranking

IMDb
Results 1 - 10 in "40.54 for "2001 Hanks"

movies: NAME: "Primetime Glick" (2001) {Tom Hanks/Ben Stiller (#2.1)} 31.54

actors: NAME: Hanks, Tom
actorplay: CHARACTER: Himself

movies: NAME: "Primetime Glick" (2001) {Tom Hanks/Ben Stiller (#2.1)}

actors: NAME: Hanks, Colin
actorplay: CHARACTER: Alex Whitman (1999-2001)

actorplay: CHARACTER: John Hanks

actorplay: CHARACTER: Hanks (Lt./Cmdr./Capt. Hanks)

Done
Top-k Keyword Search

Input:
- Relational schema: \( R = \{ R_1, R_2, \ldots, R_{|R|} \} \)
- Query keywords: \( Q = \{ q_1, q_2, \ldots, q_{|Q|} \} \)
- A constant: \( k \)
- A scoring function: \( sc(T, Q) \)

Output:
- \( k \) JTTs, \( T_1, T_2, \ldots, T_k \), with the highest scores.

Challenges:

Running Example

Input:
- \( R = \{ \text{COMPLAINT, PRODUCT} \} \)
- \( Q = \{ \text{netvista, maxtor} \} \)
- \( k = 3 \)

Output:
- \( c_3(4.3), c_1(4.0), c_3 \rightarrow p_2(2.9) \)
Top-k Keyword Search

- **Input:**
  - Relational schema:
    \[ \mathcal{R} = \{ R_1, R_2, \ldots, R_{|\mathcal{R}|} \} \]
  - Query keywords:
    \[ Q = \{ q_1, q_2, \ldots, q_{|Q|} \} \]
  - a constant: \( k \)
  - a scoring function: \( sc(T, Q) \)

- **Output:**
  - \( k \) JTTs, \( T_1, T_2, \ldots, T_k \), with the highest scores.

**Running Example**

- **Input:**
  - \( \mathcal{R} = \{ \text{Complaint}, \text{Product} \} \)
  - \( Q = \{ \text{netvista, maxtor} \} \)
  - \( k = 3 \)

- **Output:**
  - \( c_3(4.3), c_1(4.0), c_3 \rightarrow p_2(2.9) \)

**Challenges:**

- How to ensure the ranked output agrees with human judgement?
- How to retrieve such results efficiently?
2 Ranking Search Results
Our Proposed Ranking Function

\[ sc(T, Q) = sc_a(T, Q) \times sc_b(T, Q) \times sc_c(T, Q) \]

// IR score
// Completeness
// Size normalization
Example: Which one is better, $c_3 \rightarrow p_2$ or $c_2 \rightarrow p_2$?

<table>
<thead>
<tr>
<th>prodID</th>
<th>custID</th>
<th>comment</th>
<th>solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_2</td>
<td>P131</td>
<td>C222</td>
<td>“lower-end IBM Netvista caught fire, . . .”</td>
</tr>
<tr>
<td>c_3</td>
<td>P131</td>
<td>C333</td>
<td>“IBM Netvista unstable with Maxtor HD”</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>prodID</th>
<th>model</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_2</td>
<td>“D710P”</td>
<td>“1.5G CPU . . . , Maxtor 200G HD”</td>
</tr>
</tbody>
</table>
**Score Function Part A — IR-style Ranking**

**Example:** Which one is better, \( c_3 \rightarrow p_2 \) or \( c_2 \rightarrow p_2 \)?

<table>
<thead>
<tr>
<th>Complaint</th>
<th>prodID</th>
<th>custID</th>
<th>comment</th>
<th>solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_2</td>
<td>P131</td>
<td>C222</td>
<td>“lower-end IBM Netvista caught fire, . . .”</td>
<td>“dial 911”</td>
</tr>
<tr>
<td>c_3</td>
<td>P131</td>
<td>C333</td>
<td>“IBM Netvista unstable with Maxtor HD”</td>
<td>“Really?”</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product</th>
<th>prodID</th>
<th>model</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_2</td>
<td>P131</td>
<td>“D710P”</td>
<td>“1.5G CPU . . ., Maxtor 200G HD”</td>
</tr>
</tbody>
</table>

1. Model JTT as a *virtual document*
2. Apply IR score functions

\[
s_{c_2}(T, Q) = \sum_{t \in Q \cap D} \left( \frac{1 + \ln(1 + \ln(tf))}{1 - s + s \cdot \frac{dl}{avdl}} \cdot \ln \frac{N + 1}{df} \right)
\]

<table>
<thead>
<tr>
<th></th>
<th>(tf_{\text{netvista}})</th>
<th>(tf_{\text{maxtor}})</th>
<th>(s_{c_2}(T, Q))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c_3 \rightarrow p_2)</td>
<td>1</td>
<td>2</td>
<td>1.71</td>
</tr>
<tr>
<td>(c_2 \rightarrow p_2)</td>
<td>1</td>
<td>1</td>
<td>1.26</td>
</tr>
</tbody>
</table>
Example: Which one is better, $c_2 \rightarrow p_2$ or $c'_3 \rightarrow p_2$?

<table>
<thead>
<tr>
<th>prodID</th>
<th>custID</th>
<th>comment</th>
<th>solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>P131</td>
<td>C222</td>
<td>“lower-end IBM Netvista caught fire, . . .”</td>
<td>“dial 911”</td>
</tr>
<tr>
<td>P131</td>
<td>C333</td>
<td>“IBM ThinkPad X41 unstable with Maxtor HD”</td>
<td>“Really?”</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>prodID</th>
<th>model</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P131</td>
<td>“D710P”</td>
<td>“1.5G CPU . . ., Maxtor 200G HD”</td>
</tr>
</tbody>
</table>
Example: Which one is better, \( c_2 \rightarrow p_2 \) or \( c'_3 \rightarrow p_2 \)?

<table>
<thead>
<tr>
<th>prodID</th>
<th>custID</th>
<th>comment</th>
<th>solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>P131</td>
<td>C222</td>
<td>“lower-end IBM Netvista caught fire, …”</td>
<td>“dial 911”</td>
</tr>
<tr>
<td>P131</td>
<td>C333</td>
<td>“IBM ThinkPad X41 unstable with Maxtor HD”</td>
<td>“Really?”</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>prodID</th>
<th>model</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P131</td>
<td>“D710P”</td>
<td>“1.5G CPU …, Maxtor 200G HD”</td>
</tr>
</tbody>
</table>

1. Intuition: for short queries, user prefer results matching *more* (distinct) keywords.
2. Derived from extended Boolean model:

\[
sc_b(T, Q) = 1 - \left( \frac{\sum_{1 \leq i \leq m} (1 - T.i)^p}{p} \right)
\]

<table>
<thead>
<tr>
<th></th>
<th>( tf_{\text{netvista}} )</th>
<th>( tf_{\text{maxtor}} )</th>
<th>( sc_b(T, Q) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c_2 \rightarrow p_2 )</td>
<td>1</td>
<td>1</td>
<td>0.65</td>
</tr>
<tr>
<td>( c'_3 \rightarrow p_2 )</td>
<td>0</td>
<td>2</td>
<td>0.29</td>
</tr>
</tbody>
</table>
**Example:** Which one is better, \( c_3 \) or \( c_2 \rightarrow p_2 \)?

<table>
<thead>
<tr>
<th>Complaint</th>
<th>prodID</th>
<th>custID</th>
<th>comment</th>
<th>solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c_2 )</td>
<td>P131</td>
<td>C222</td>
<td>“lower-end IBM Netvista caught fire, . . .”</td>
<td>“dial 911”</td>
</tr>
<tr>
<td>( c_3 )</td>
<td>P131</td>
<td>C333</td>
<td>“IBM Netvista unstable with Maxtor HD”</td>
<td>“Really?”</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product</th>
<th>prodID</th>
<th>model</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_2 )</td>
<td>P131</td>
<td>“D710P”</td>
<td>“1.5G CPU . . ., Maxtor 200G HD”</td>
</tr>
</tbody>
</table>
**Example:** Which one is better, $c_3$ or $c_2 \rightarrow p_2$?

<table>
<thead>
<tr>
<th>Complaint prodID custID comment</th>
<th>solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_2$ P131 C222 “lower-end IBM Netvista caught fire, . . .”</td>
<td>“dial 911”</td>
</tr>
<tr>
<td>$c_3$ P131 C333 “IBM Netvista unstable with Maxtor HD”</td>
<td>“Really?”</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product prodID model description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_2$ P131 “D710P 1.5G CPU . . . Maxtor 200G HD”</td>
</tr>
</tbody>
</table>

1. Size of a JTT matters!
2. \[ sc_c(T, Q) = (1 + s_1 - s_1 \cdot \text{size}(CN)) \cdot (1 + s_2 - s_2 \cdot \text{size}(CN^{nf})) \]

<table>
<thead>
<tr>
<th></th>
<th>$sc_a(T, Q)$</th>
<th>$sc_b(T, Q)$</th>
<th>$sc_c(T, Q)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_3$</td>
<td>1.26</td>
<td>0.75</td>
<td>1.00</td>
</tr>
<tr>
<td>$c_2 \rightarrow p_2$</td>
<td>1.26</td>
<td>0.75</td>
<td>0.56</td>
</tr>
</tbody>
</table>
Effectiveness

- Test the effectiveness against previous methods.

**Table:** Top-1 Result for Query “nikos clique” on DBLP

<table>
<thead>
<tr>
<th>Method</th>
<th>Top-1 Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Hristidis VLDB03]</td>
<td><strong>InProceeding:</strong> <em>Clique</em>-to-<em>Clique</em> Distance ...</td>
</tr>
<tr>
<td>[Liu SIGMOD06]</td>
<td><strong>Series Proceeding:</strong> Person: <strong>Nikos</strong> Karatzas</td>
</tr>
<tr>
<td></td>
<td><strong>Proceeding:</strong> Person: <strong>Nikos</strong> Mamoulis ← RPI → <strong>InProceeding:</strong> ...</td>
</tr>
<tr>
<td>Ours</td>
<td><strong>Person:</strong> <strong>Nikos</strong> Mamoulis ← RPI → <strong>InProceeding:</strong> ...</td>
</tr>
<tr>
<td></td>
<td><strong>Constraint-Based Algorithms for Computing Clique Intersection Joins</strong></td>
</tr>
</tbody>
</table>
Efficient Query Processing
Query Processing

Three steps:

1. Generate candidate tuples in every relation in the schema (using full-text indexes)

Running Example

\[ C^Q = [c_3, c_2] \]
\[ P^Q = [p_3] \]

<table>
<thead>
<tr>
<th>Complaint</th>
<th>prodID</th>
<th>custID</th>
<th>comment</th>
<th>solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_2</td>
<td>P131</td>
<td>C222</td>
<td>“lower-end IBM Netvista caught fire, . . .”</td>
<td>“dial 911”</td>
</tr>
<tr>
<td>c_3</td>
<td>P131</td>
<td>C333</td>
<td>“IBM Netvista unstable with Maxtor HD”</td>
<td>“Really?”</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product</th>
<th>prodID</th>
<th>model</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_2</td>
<td>P131</td>
<td>“D710P”</td>
<td>“1.5G CPU . . ., Maxtor 200G HD”</td>
</tr>
</tbody>
</table>
Query Processing . . .

Three steps:

2. Enumerate all possible Candidate Networks (CN) using a breath-first subgraph enumeration algorithm.

Running Example

“Valid” CN of size ≤ 3:

- $CQ$
- $PQ$
- $CQ \rightarrow PQ$
- $CQ \rightarrow P \leftarrow CQ$
- $CQ \rightarrow PQ \leftarrow CQ$

Note that $PQ \leftarrow C^* \rightarrow PQ$ is *not* a valid CN.
Query Processing . . .

Three steps:

3. Execute the CNs
   - Most algorithms differ here
   - Naive alg: execute every CN.
   - The key is how to optimize for the top-\( k \) retrieval.

Running Example

\[
C \rightarrow P
\]

SELECT * FROM C, P
WHERE C.prodID = P.prodID
AND C.cid = Ci.cid
AND P.pid = Pj.pid

What if \( k \ll \) (size of join results)

Analogy: Mine Sweeper

CN \( C^Q \rightarrow P^Q \) (easily generalized to CNs with > 2 joins)

However, such probes are very expensive!
Query Processing . . .

Three steps:

1. Execute the CNs
   - Most algorithms differ here
   - Naive alg: execute every CN.
   - The key is how to optimize for the top-$k$ retrieval.

Running Example

<table>
<thead>
<tr>
<th>CN</th>
<th>CQ → PQ</th>
<th>CQ → P ← CQ</th>
<th>CQ → PQ ← CQ</th>
</tr>
</thead>
</table>

SELECT * FROM C, P
WHERE C.prodID = P.prodID
AND C.cid = ci.pid
AND P.pid = pj.pid

What if $k \ll \text{(size of join results)}$

Analogy: Mine Sweeper

CN $C^Q \rightarrow P^Q$ (easily generalized to CNs with > 2 joins)

However, such probes are very expensive!
Query Processing . . .

Three steps:

1. Execute the CNs
   - Most algorithms differ here
   - Naive alg: execute every CN.
   - The key is how to optimize for the top-k retrieval.

Running Example

- \( C^Q \)
- \( P^Q \)
- \( C^Q \rightarrow P^Q \)
- \( C^Q \rightarrow P \leftarrow C^Q \)
- \( C^Q \rightarrow P^Q \leftarrow C^Q \)

SELECT * FROM C, P
WHERE C.prodID = P.prodID

What if \( k \ll (\text{size of join results}) \)

Analogy: Mine Sweeper

CN \( C^Q \rightarrow P^Q \) (easily generalized to CNs with > 2 joins)

However, such probes are very expensive!
Query Processing . . .

Three steps:

3. Execute the CNs
   - Most algorithms differ here
   - Naive alg: execute every CN.
   - The key is how to optimize for the top-\(k\) retrieval.

Running Example

- \(C^Q\)
- \(P^Q\)
- \(C^Q \rightarrow P^Q\)
- \(C^Q \rightarrow P \leftarrow C^Q\)
- \(C^Q \rightarrow P^Q \leftarrow C^Q\)

SELECT * FROM C, P
WHERE C.prodID = P.prodID

What if \(k \ll (\text{size of join results})\)

SELECT * FROM C, P
WHERE C.prodID = P.prodID
AND C.cid = c_i.cid
AND P.pid = p_j.pid

Analogy: Mine Sweeper

CN \(C^Q \rightarrow P^Q\) (easily generalized to CNs with > 2 joins)

However, such probes are very expensive!
It is a Search Problem

An unknown space to search

Lessons learned:
- No optimization exists if we are dealing with a general search space.
- We should stop earlier with the help of a monotonic score upper bounding function.
- We should check candidates in an optimal order.
It is a Search Problem

An unknown space to search

Many cells has 0 score
An unknown space to search

Many cells has 0 score

In general, we cannot stop even if we found one result \((k = 1)\)
It is a Search Problem

An unknown space to search

Many cells has 0 score

In general, we cannot stop even if we found one result \( (k = 1) \)

But, if we know the scores* are monotonic wrt both axes

\* : non-zero scores

Lessons learned:

No optimization exists if we are dealing with a general search space.

We should stop earlier with the help of a monotonic score upper bounding function.

We should check candidates in an optimal order.
It is a Search Problem

An unknown space to search

Many cells has 0 score

In general, we cannot stop even if we found one result ($k = 1$)

But, if we know the scores* are monotonic wrt both axes

*: non-zero scores
It is a Search Problem

- An unknown space to search
- Many cells have 0 score
- In general, we cannot stop even if we found one result ($k = 1$)
- But, if we know the scores* are monotonic wrt both axes

* : non-zero scores

Lessons learned:
- No optimization exists if we are dealing with a general search space.
- We should stop earlier with the help of a monotonic score upper bounding function.
- We should check candidates in an optimal order.
It is a Search Problem

An unknown space to search

Many cells has 0 score

In general, we cannot stop even if we found one result ($k = 1$)

But, if we know the scores* are monotonic wrt both axes

*: non-zero scores

An easier search space to deal with
It is a Search Problem

An unknown space to search

Many cells has 0 score

In general, we cannot stop even if we found one result \((k = 1)\)

But, if we know the scores* are monotonic wrt both axes

\* : non-zero scores

An easier search space to deal with

What's the stopping criterion? (e.g., \(k = 1\))?
It is a Search Problem

An unknown space to search

Many cells has 0 score

In general, we cannot stop even if we found one result \((k = 1)\)

But, if we know the scores* are monotonic wrt both axes

*: non-zero scores

An easier search space to deal with

What's the stopping criterion? (e.g., \(k = 1\))?
It is a Search Problem

An unknown space to search

Many cells has 0 score

In general, we cannot stop even if we found one result ($k = 1$)

But, if we know the scores* are monotonic wrt both axes

*: non-zero scores

An easier search space to deal with

What’s the stopping criterion? (e.g., $k = 1$)?

Lessons learned:
- No optimization exists if we are dealing with a general search space.
- We should stop earlier with the help of a monotonic score upper bounding function.
- We should check candidates in an optimal order.
Finding a Monotonic Upper Bounding Function

- It is non-trivial to find a (good) monotonic upper bounding function for our scoring function.
  - Previous method (sorting on local scores) does not work
- Our scores are related to the *distribution* of tfs in a JTT
- Deriving $usc(T, Q)$:
  - $\text{sumidf} = \sum_w \text{idf}_w$
  - $\text{watf}(t) = \left( \frac{1}{\text{sumidf}} \right) \cdot \sum_w (\text{tf}_w(t) \cdot \text{idf}_w)$
  - $A = \text{sumidf} \cdot (1 + \ln(1 + \ln(\sum_t \text{watf}(t))))$
  - $B = \text{sumidf} \cdot \sum_t \text{watf}(t)$

Then,

\[
\begin{align*}
sc_a & \leq usc_a = \frac{1}{1-s} \cdot \min(A, B) \\
sc_b & = \text{const given the CN} \\
sc_c & = \text{const given the CN}
\end{align*}
\]

\[\Rightarrow \quad sc(T, Q) \leq usc(T, Q)\]

and $usc(T, Q)$ is monotonic wrt all $\text{watf}(t_i)$ ($\forall t_i \in T$)
With the discovery of $usc(T, Q)$, the Single Pipeline [Hristidis VLDB03] algorithm can be applied.

- Tuples (on each dimension) sorted in decreasing width order.
- Its search strategy is reminiscent of the Ripple Join.

\[
\begin{align*}
\text{p}_4 & \\
\text{p}_3 & \\
\text{p}_2 & \\
\text{p}_1 & \\
\text{c}_1 & \text{c}_2 & \text{c}_3 & \text{c}_4
\end{align*}
\]
With the discovery of $usc(T, Q)$, the *Single Pipeline* [Hristidis VLDB03] algorithm can be applied.

- Tuples (on each dimension) sorted in decreasing order.
- Its search strategy is reminiscent of the *Ripple Join*.

\[ c_1 \bowtie p_1 \]
\[ c_2 \bowtie \{c_1, \ldots, c_2\} \]
\[ c_3 \bowtie \{c_1, \ldots, c_2\} \]
\[ c_4 \bowtie \{c_1, \ldots, c_2\} \]
With the discovery of $usc(T, Q)$, the *Single Pipeline* [Hristidis VLDB03] algorithm can be applied.

- Tuples (on each dimension) sorted in decreasing $w_{\text{atf}}$ order.
- Its search strategy is reminiscent of the *Ripple Join*.

\[
\begin{align*}
p_4 & \quad c_1 \bowtie p_1 \\
p_3 & \\
p_2 & \\
p_1 &
\end{align*}
\]
The Single Pipeline Algorithm [Hristidis VLDB03]

- With the discovery of \(usc(T, Q)\), the *Single Pipeline* [Hristidis VLDB03] algorithm can be applied.
  - Tuples (on each dimension) sorted in decreasing \(wattf\) order.
  - Its search strategy is reminiscent of the *Ripple Join*.

\[c_1 \bowtie p_1\]
\[c_2 \bowtie [p_1, \ldots, p_1]\]
With the discovery of $usc(T, Q)$, the *Single Pipeline* [Hristidis VLDB03] algorithm can be applied.

- Tuples (on each dimension) sorted in decreasing weight order.
- Its search strategy is reminiscent of the *Ripple Join*.

$p_4$

$p_3$

$p_2$

$p_1$

$c_1 \Join p_1$

$c_2 \Join [p_1, \ldots, p_1]$
The Single Pipeline Algorithm [Hristidis VLDB03]

- With the discovery of $usc(T, Q)$, the *Single Pipeline* [Hristidis VLDB03] algorithm can be applied.
  - Tuples (on each dimension) sorted in decreasing $\omega_{atf}$ order.
  - Its search strategy is reminiscent of the *Ripple Join*.

\[
\begin{align*}
  & \quad p_4 \quad p_3 \quad p_2 \quad p_1 \\
  & c_1 \quad c_2 \quad c_3 \quad c_4 \\
\end{align*}
\]

- $c_1 \bowtie p_1$
- $c_2 \bowtie [p_1, \ldots, p_1]$
- $[c_1, \ldots, c_2] \bowtie p_2$
The Single Pipeline Algorithm [Hristidis VLDB03]

- With the discovery of $usc(T, Q)$, the *Single Pipeline* [Hristidis VLDB03] algorithm can be applied.
  - Tuples (on each dimension) sorted in decreasing $watf$ order.
  - Its search strategy is reminiscent of the *Ripple Join*.

\[
\begin{align*}
\text{c}_1 & \bowtie p_1 \\
\text{c}_2 & \bowtie [p_1, \ldots, p_1] \\
[c_1, \ldots, c_2] & \bowtie p_2
\end{align*}
\]

$\bowtie$ indicates a join operation in the context of database algorithms.
With the discovery of \(usc(T, Q)\), the Single Pipeline [Hristidis VLDB03] algorithm can be applied.

- Tuples (on each dimension) sorted in decreasing order.
- Its search strategy is reminiscent of the Ripple Join.
The Single Pipeline Algorithm [Hristidis VLDB03]

- With the discovery of $usc(T, Q)$, the Single Pipeline [Hristidis VLDB03] algorithm can be applied.
  - Tuples (on each dimension) sorted in decreasing $w^a_t$ order.
  - Its search strategy is reminiscent of the Ripple Join.

\[
\begin{align*}
  p_4 & \quad p_3 & \quad p_2 & \quad p_1 \\
  c_4 & \quad c_3 & \quad c_2 & \quad c_1 \\
\end{align*}
\]

- $c_1 \bowtie p_1$
- $c_2 \bowtie [p_1, \ldots, p_1]$
- $[c_1, \ldots, c_2] \bowtie p_2$
- $c_3 \bowtie [p_1, \ldots, p_2]$
With the discovery of $usc(T, Q)$, the \textit{Single Pipeline} [Hristidis VLDB03] algorithm can be applied.

- Tuples (on each dimension) sorted in decreasing width order.
- Its search strategy is reminiscent of the \textit{Ripple Join}.

\[ \begin{align*}
   & c_1 \bowtie p_1 \\
   & c_2 \bowtie [p_1, \ldots, p_1] \\
   & [c_1, \ldots, c_2] \bowtie p_2 \\
   & c_3 \bowtie [p_1, \ldots, p_2] \\
\end{align*} \]
With the discovery of $usc(T, Q)$, the *Single Pipeline* [Hristidis VLDB03] algorithm can be applied.

- Tuples (on each dimension) sorted in decreasing order.
- Its search strategy is reminiscent of the *Ripple Join*.

\[ c_1 \bowtie p_1 \]
\[ c_2 \bowtie [p_1, \ldots, p_1] \]
\[ [c_1, \ldots, c_2] \bowtie p_2 \]
\[ c_3 \bowtie [p_1, \ldots, p_2] \]
With the discovery of $usc(T, Q)$, the *Single Pipeline* [Hristidis VLDB03] algorithm can be applied.

- Tuples (on each dimension) sorted in decreasing $waf$ order.
- Its search strategy is reminiscent of the *Ripple Join*.

\[
\begin{align*}
\text{p}_4 & \leq 5.7 \leq 5.7 \leq 5.7 \leq 5.7 \\
\text{p}_3 & \leq 5.7 \leq 5.7 \leq 5.7 \leq 5.7 \\
\text{p}_2 & \leq \leq \leq 6.1 \leq \leq \leq \\
\text{p}_1 & \leq \leq \leq \leq \leq \\
\text{c}_1 & \leq \leq \leq \leq \\
\text{c}_2 & \leq \leq \leq \leq \\
\text{c}_3 & \leq \leq \leq \leq \\
\text{c}_4 & \leq \leq \leq \leq
\end{align*}
\]

- $c_1 \bowtie p_1$
- $c_2 \bowtie [p_1, \ldots, p_1]$
- $[c_1, \ldots, c_2] \bowtie p_2$
- $c_3 \bowtie [p_1, \ldots, p_2]$

The Single Pipeline Algorithm [Hristidis VLDB03]

- With the discovery of \(usc(T,Q)\), the *Single Pipeline* [Hristidis VLDB03] algorithm can be applied.
  - Tuples (on each dimension) sorted in decreasing \(\text{wattf}\) order.
  - Its search strategy is reminiscent of the *Ripple Join*.

\[
\begin{align*}
\text{p}_4 & \leq 5.7 & \leq 5.7 & \leq 5.7 & \leq 5.7 \\
\text{p}_3 & \leq 5.7 & \leq 5.7 & \leq 5.7 & \leq 5.7 \\
\text{p}_2 & \geq 6.1 & \geq 6.1 & \geq 6.1 & \geq 6.1 \\
\text{p}_1 & \leq 4.3 & \leq 4.3 & \leq 4.3 & \leq 4.3 \\
\end{align*}
\]

- \(c_1 \bowtie \text{p}_1\)
- \(c_2 \bowtie [p_1, \ldots, p_1]\)
- \([c_1, \ldots, c_2] \bowtie \text{p}_2\)
- \(c_3 \bowtie [p_1, \ldots, p_2]\)
The Single Pipeline Algorithm [Hristidis VLDB03]

- With the discovery of \(usc(T, Q)\), the *Single Pipeline* [Hristidis VLDB03] algorithm can be applied.
  - Tuples (on each dimension) sorted in decreasing \(watf\) order.
  - Its search strategy is reminiscent of the *Ripple Join*.

\[
\begin{align*}
&\text{p}_4 &\leq 5.7 &\leq 5.7 &\leq 5.7 &\leq 4.3 \\
&\text{p}_3 &\leq 5.7 &\leq 5.7 &\leq 5.7 &\leq 4.3 \\
&\text{p}_2 &\leq 6.1 &\leq 4.3 \\
&\text{p}_1 &\leq 4.3 \\
&c_1 &c_2 &c_3 &c_4
\end{align*}
\]

- \(c_1 \bowtie p_1\)
- \(c_2 \bowtie [p_1, \ldots, p_1]\)
- \([c_1, \ldots, c_2] \bowtie p_2\)
- \(c_3 \bowtie [p_1, \ldots, p_2]\)
With the discovery of $usc(T, Q)$, the *Single Pipeline* [Hristidis VLDB03] algorithm can be applied.

- Tuples (on each dimension) sorted in decreasing order.
- Its search strategy is reminiscent of the *Ripple Join*.

<table>
<thead>
<tr>
<th>$p_4$</th>
<th>$\leq 5.7$</th>
<th>$\leq 5.7$</th>
<th>$\leq 5.7$</th>
<th>$\leq 4.3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_3$</td>
<td>$\leq 5.7$</td>
<td>$\leq 5.7$</td>
<td>$\leq 5.7$</td>
<td>$\leq 4.3$</td>
</tr>
<tr>
<td>$p_2$</td>
<td></td>
<td>$6.1$</td>
<td></td>
<td>$\leq 4.3$</td>
</tr>
<tr>
<td>$p_1$</td>
<td></td>
<td></td>
<td></td>
<td>$\leq 4.3$</td>
</tr>
</tbody>
</table>

- $c_1 \bowtie p_1$
- $c_2 \bowtie [p_1, \ldots, p_1]$
- $[c_1, \ldots, c_2] \bowtie p_2$
- $c_3 \bowtie [p_1, \ldots, p_2]$
- $sc(c_3 \bowtie p_2) \geq MPFS \Rightarrow \text{STOP}$
Problem of the Single Pipeline Algorithm

- Single Pipeline algorithm is *not* optimal in the number of database probes.
  - Recall that you pay a price for each probe!

\[
\begin{align*}
\mathrm{p}_4 & \Rightarrow & c_1 & \bowtie & p_1 \\
\mathrm{p}_3 & \Rightarrow & c_2 & \bowtie & [p_1, \ldots, p_1] \\
\mathrm{p}_2 & \Rightarrow & 6.1 & \bowtie & [c_1, \ldots, c_2] \\
\mathrm{p}_1 & \Rightarrow & c_3 & \bowtie & [p_1, \ldots, p_2] \\
\end{align*}
\]
Single Pipeline algorithm is not optimal in the number of database probes.

Recall that you pay a price for each probe!

- $c_1 \bowtie p_1$
- $c_2 \bowtie [p_1, \ldots, p_1]$
- $[c_1, \ldots, c_2] \bowtie p_2$
- $c_3 \bowtie [p_1, \ldots, p_2]$

\[ \leq 4.3 \]
Problem of the Single Pipeline Algorithm

- Single Pipeline algorithm is \textit{not} optimal in the number of database probes.
  - Recall that you pay a price for each probe!

\begin{align*}
\text{\textcolor{blue}{c}_1} & \preceq 4.3 \\
\text{\textcolor{blue}{c}_2} & \preceq 4.3 \\
\text{\textcolor{red}{6.1}} & \preceq 4.3 \\
\text{\textcolor{blue}{c}_3} & \preceq 4.3 \\
\end{align*}

- \textcolor{blue}{c}_1 \preceq \textcolor{blue}{p}_1
- \textcolor{blue}{c}_2 \preceq [\textcolor{blue}{p}_1, \ldots, \textcolor{blue}{p}_1]
- \textcolor{blue}{c}_3 \preceq [\textcolor{blue}{p}_1, \ldots, \textcolor{blue}{p}_2]

\begin{align*}
\text{\textcolor{blue}{c}_1} & \preceq \textcolor{blue}{p}_1 \\
\text{\textcolor{blue}{c}_2} & \preceq [\textcolor{blue}{p}_1, \ldots, \textcolor{blue}{p}_1] \\
\text{\textcolor{blue}{c}_3} & \preceq [\textcolor{blue}{p}_1, \ldots, \textcolor{blue}{p}_2] \\
\end{align*}
Problem of the Single Pipeline Algorithm

- Single Pipeline algorithm is *not* optimal in the number of database probes.
  - Recall that you pay a price for each probe!

<table>
<thead>
<tr>
<th></th>
<th>c_1</th>
<th>c_2</th>
<th>c_3</th>
<th>c_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_4</td>
<td></td>
<td></td>
<td></td>
<td>≤ 4.3</td>
</tr>
<tr>
<td>p_3</td>
<td>≤ 7.1</td>
<td></td>
<td></td>
<td>≤ 4.3</td>
</tr>
<tr>
<td>p_2</td>
<td></td>
<td>6.1</td>
<td></td>
<td>≤ 4.3</td>
</tr>
<tr>
<td>p_1</td>
<td></td>
<td></td>
<td></td>
<td>≤ 4.3</td>
</tr>
</tbody>
</table>

- c_1 \bowtie p_1
- c_2 \bowtie [p_1, \ldots, p_1]
- [c_1, \ldots, c_2] \bowtie p_2
- c_3 \bowtie [p_1, \ldots, p_2]
Problem of the Single Pipeline Algorithm

- Single Pipeline algorithm is *not* optimal in the number of database probes.
- Recall that you pay a price for each probe!

<table>
<thead>
<tr>
<th></th>
<th>c₁</th>
<th>c₂</th>
<th>c₃</th>
<th>c₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>p₁</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p₂</td>
<td></td>
<td></td>
<td>6.1</td>
<td></td>
</tr>
<tr>
<td>p₃</td>
<td>≤ 7.1</td>
<td>≤ 7.1</td>
<td>≤ 7.1</td>
<td>≤ 4.3</td>
</tr>
<tr>
<td>p₄</td>
<td></td>
<td></td>
<td></td>
<td>≤ 4.3</td>
</tr>
</tbody>
</table>

- c₁ ⊲ p₁
- c₂ ⊲ [p₁, . . . , p₁]
- [c₁, . . . , c₂] ⊲ p₂
- c₃ ⊲ [p₁, . . . , p₂]
Single Pipeline algorithm is not optimal in the number of database
probes.

Recall that you pay a price for each probe!

\begin{itemize}
  \item $p_4$ \begin{tabular}{cccc}
    $\leq 7.1$ & $\leq 7.1$ & $\leq 7.1$ & $\leq 4.3$
  
  \end{tabular}
  \item $c_1 \bowtie p_1$
  \item $c_2 \bowtie [p_1, \ldots, p_1]$
  \item $[c_1, \ldots, c_2] \bowtie p_2$
  \item $c_3 \bowtie [p_1, \ldots, p_2]$
  \item $[c_1, \ldots, c_3] \bowtie p_3$
\end{itemize}

\begin{itemize}
  \item $p_3$ \begin{tabular}{cccc}
    $\leq 7.1$ & $\leq 7.1$ & $\leq 7.1$ & $\leq 4.3$
  
  \end{tabular}
  \item $[c_1, \ldots, c_2] \bowtie p_2$
  \item $c_3 \bowtie [p_1, \ldots, p_2]$
  \item $[c_1, \ldots, c_3] \bowtie p_3$
\end{itemize}

\begin{itemize}
  \item $p_2$ \begin{tabular}{cccc}
    $\leq 7.1$ & 6.1 & $\leq 4.3$
  
  \end{tabular}
  \item $c_1 \bowtie p_1$
  \item $c_2 \bowtie [p_1, \ldots, p_1]$
  \item $[c_1, \ldots, c_2] \bowtie p_2$
  \item $c_3 \bowtie [p_1, \ldots, p_2]$
  \item $[c_1, \ldots, c_3] \bowtie p_3$
\end{itemize}

\begin{itemize}
  \item $p_1$ \begin{tabular}{cccc}
    $\leq 7.1$ & 6.1 & $\leq 4.3$
  
  \end{tabular}
  \item $c_1 \bowtie p_1$
  \item $c_2 \bowtie [p_1, \ldots, p_1]$
  \item $[c_1, \ldots, c_2] \bowtie p_2$
  \item $c_3 \bowtie [p_1, \ldots, p_2]$
  \item $[c_1, \ldots, c_3] \bowtie p_3$
\end{itemize}

\begin{itemize}
  \item $c_1$ $c_2$ $c_3$ $c_4$
\end{itemize}
Problem of the Single Pipeline Algorithm

- Single Pipeline algorithm is *not* optimal in the number of database probes.
- Recall that you pay a price for each probe!
Problem of the Single Pipeline Algorithm

- Single Pipeline algorithm is *not* optimal in the number of database probes.
  - Recall that you pay a price for each probe!

\[
\begin{align*}
\text{p}_4 & \quad \text{c}_1 & \quad \text{p}_1 \\
\text{p}_3 & \quad \leq 7.1 & \quad \leq 7.1 & \quad \leq 4.3 \\
\text{p}_2 & \quad \leq 6.1 & \quad \leq 4.3 \\
\text{p}_1 & \quad \leq 4.3 \\
\text{c}_1 & \quad \text{c}_2 & \quad \text{c}_3 & \quad \text{c}_4
\end{align*}
\]

- \( c_1 \bowtie p_1 \)
- \( c_2 \bowtie [p_1, \ldots, p_1] \)
- \([c_1, \ldots, c_2] \bowtie p_2\)
- \( c_3 \bowtie [p_1, \ldots, p_2] \)
- \([c_1, \ldots, c_3] \bowtie p_3 \)
Problem of the Single Pipeline Algorithm

- Single Pipeline algorithm is *not* optimal in the number of database probes.
  - Recall that you pay a price for each probe!

\[
\begin{align*}
\text{p}_4 & \quad \text{c}_1 \quad \text{c}_2 \quad \text{c}_3 \quad \text{c}_4 \\
\text{p}_3 & \quad 4.9 \quad \leq 7.1 \quad \leq 4.3 \\
\text{p}_2 & \quad 6.1 \quad \leq 4.3 \\
\text{p}_1 & \quad \leq 4.3 \\
\end{align*}
\]

- \( \text{c}_1 \bowtie \text{p}_1 \)
- \( \text{c}_2 \bowtie [\text{p}_1, \ldots, \text{p}_1] \)
- \( [\text{c}_1, \ldots, \text{c}_2] \bowtie \text{p}_2 \)
- \( \text{c}_3 \bowtie [\text{p}_1, \ldots, \text{p}_2] \)
- \( [\text{c}_1, \ldots, \text{c}_3] \bowtie \text{p}_3 \)
Problem of the Single Pipeline Algorithm

- Single Pipeline algorithm is *not* optimal in the number of database probes.
- Recall that you pay a price for each probe!

\[ \begin{align*}
  c_1 & \bowtie p_1 \\
  c_2 & \bowtie [p_1, \ldots, p_1] \\
  [c_1, \ldots, c_2] & \bowtie p_2 \\
  c_3 & \bowtie [p_1, \ldots, p_2] \\
  [c_1, \ldots, c_3] & \bowtie p_3
\end{align*} \]
Problem of the Single Pipeline Algorithm

- Single Pipeline algorithm is *not* optimal in the number of database probes.
- Recall that you pay a price for each probe!

\[
\begin{array}{c|c|c|c}
\text{p}_4 & \leq 4.5 & \leq 4.3 \\
\hline
\text{p}_3 & 4.9 & \leq 4.3 \\
\hline
\text{p}_2 & 6.1 & \leq 4.3 \\
\hline
\text{p}_1 & \leq 4.3 \\
\end{array}
\]

- \( c_1 \bowtie p_1 \)
- \( c_2 \bowtie [p_1, \ldots, p_1] \)
- \( [c_1, \ldots, c_2] \bowtie p_2 \)
- \( c_3 \bowtie [p_1, \ldots, p_2] \)
- \( [c_1, \ldots, c_3] \bowtie p_3 \)
Problem of the Single Pipeline Algorithm

- Single Pipeline algorithm is *not* optimal in the number of database probes.
  - Recall that you pay a price for each probe!

\[
\begin{align*}
\text{p}_4 & \leq 4.5 & \leq 4.5 & \leq 4.5 & \leq 4.3 \\
\text{p}_3 & & & 4.9 & \leq 4.3 \\
\text{p}_2 & & 6.1 & & \leq 4.3 \\
\text{p}_1 & & & & \leq 4.3 \\
\text{c}_1 & \text{c}_2 & \text{c}_3 & \text{c}_4
\end{align*}
\]

- \(c_1 \bowtie p_1\)
- \(c_2 \bowtie [p_1, \ldots, p_1]\)
- \([c_1, \ldots, c_2] \bowtie p_2\)
- \(c_3 \bowtie [p_1, \ldots, p_2]\)
- \([c_1, \ldots, c_3] \bowtie p_3\)

Wei Wang (UNSW)
SPARK
16 July, 2007 20 / 33
Problem of the Single Pipeline Algorithm

- Single Pipeline algorithm is *not* optimal in the number of database probes.
  - Recall that you pay a price for each probe!

<table>
<thead>
<tr>
<th></th>
<th>$c_1$</th>
<th>$c_2$</th>
<th>$c_3$</th>
<th>$c_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_4$</td>
<td>$\leq 4.5$</td>
<td>$\leq 4.5$</td>
<td>$\leq 4.5$</td>
<td>$\leq 4.3$</td>
</tr>
<tr>
<td>$p_3$</td>
<td>$4.9$</td>
<td></td>
<td></td>
<td>$\leq 4.3$</td>
</tr>
<tr>
<td>$p_2$</td>
<td></td>
<td>$6.1$</td>
<td></td>
<td>$\leq 4.3$</td>
</tr>
<tr>
<td>$p_1$</td>
<td></td>
<td></td>
<td></td>
<td>$\leq 4.3$</td>
</tr>
</tbody>
</table>

- $c_1 \bowtie p_1$
- $c_2 \bowtie [p_1, \ldots, p_1]$
- $[c_1, \ldots, c_2] \bowtie p_2$
- $c_3 \bowtie [p_1, \ldots, p_2]$
- $[c_1, \ldots, c_3] \bowtie p_3$

$c_3 \bowtie p_3$ should **not** have been executed!
Our Skyline Sweeping Algorithm

- **Idea:** be lazy!
- Based on the skyline principle:
  - \( \text{usc}(c_i \bowtie p_j) \geq \text{usc}(c_{i+1} \bowtie p_j) \) and \( \text{usc}(c_i \bowtie p_j) \geq \text{usc}(c_i \bowtie p_{j+1}) \)
  
  \( \Rightarrow \) if the current “cell” fails, we should only examine its neighbors, as they are the best possible alternatives.
- Data structure: a priority queue of candidate cells
- Its search strategy is reminiscent of the J* algorithm.

```
<table>
<thead>
<tr>
<th>p1</th>
<th>p2</th>
<th>p3</th>
<th>p4</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>c2</td>
<td>c3</td>
<td>c4</td>
</tr>
</tbody>
</table>
```

Priority Queue

\[ \langle c_1, p_1 \rangle \]
Our Skyline Sweeping Algorithm

- **Idea:** be lazy!
- **Based on the skyline principle:**
  - \( \text{usc}(c_i \bowtie p_j) \geq \text{usc}(c_{i+1} \bowtie p_j) \) and \( \text{usc}(c_i \bowtie p_j) \geq \text{usc}(c_i \bowtie p_{j+1}) \)
  - \( \Rightarrow \) if the current “cell” fails, we should *only* examine its *neighbors*, as they are the best possible alternatives.
- **Data structure:** a priority queue of *candidate cells*
- **Its search strategy is reminiscent of the J* algorithm.**

\[
\begin{align*}
&\langle c_1, p_1 \rangle \langle c_2, p_2 \rangle \langle c_3, p_3 \rangle \langle c_4, p_4 \rangle \\
&\langle c_1, p_1 \rangle \langle c_2, p_1 \rangle \langle c_3, p_3 \rangle \langle c_4, p_2 \rangle \\
&\langle c_1, p_2 \rangle \langle c_2, p_1 \rangle \langle c_3, p_3 \rangle \langle c_4, p_4 \rangle \\
&\langle c_1, p_2 \rangle \langle c_2, p_2 \rangle \langle c_3, p_3 \rangle \langle c_4, p_4 \rangle \\
&\langle c_1, p_3 \rangle \langle c_2, p_1 \rangle \langle c_3, p_3 \rangle \langle c_4, p_4 \rangle \\
&\langle c_1, p_3 \rangle \langle c_2, p_2 \rangle \langle c_3, p_3 \rangle \langle c_4, p_2 \rangle \\
\end{align*}
\]

Priority Queue
Our Skyline Sweeping Algorithm

- **Idea:** be lazy!
- **Based on the skyline principle:**
  - \(usc(c_i \bowtie p_j) \geq usc(c_{i+1} \bowtie p_j)\) and \(usc(c_i \bowtie p_j) \geq usc(c_i \bowtie p_{j+1})\)
  - \(\Rightarrow\) if the current “cell” fails, we should *only* examine its *neighbors*, as they are the best possible alternatives.
- **Data structure:** a priority queue of *candidate cells*
- **Its search strategy is reminiscent of the J* algorithm.*

```
\begin{center}
\begin{tikzpicture}
  \draw[step=1cm, thin, lightgray] (0,0) grid (4,4);
  \node[fill=blue!30, draw=black] at (0,0) {c_1};
  \node[fill=blue!30, draw=black] at (1,0) {c_2};
  \node[fill=blue!30, draw=black] at (2,0) {c_3};
  \node[fill=blue!30, draw=black] at (3,0) {c_4};
  \node[fill=blue!30, draw=black] at (0,1) {p_1};
  \node[fill=blue!30, draw=black] at (1,1) {p_2};
  \node[fill=blue!30, draw=black] at (2,1) {p_3};
  \node[fill=blue!30, draw=black] at (3,1) {p_4};
  \node[fill=white, draw=black] at (0,3) {\text{Priority Queue}};
\end{tikzpicture}
\end{center}
```
Our Skyline Sweeping Algorithm

- Idea: be lazy!
- Based on the skyline principle:
  - $usc(c_i \bowtie p_j) \geq usc(c_{i+1} \bowtie p_j)$ and $usc(c_i \bowtie p_j) \geq usc(c_i \bowtie p_{j+1})$
  - $\Rightarrow$ if the current “cell” fails, we should only examine its neighbors, as they are the best possible alternatives.
- Data structure: a priority queue of candidate cells
- Its search strategy is reminiscent of the J* algorithm.

```
P4
  P3
  P2
  P1
  C1  C2  C3  C4
```
Our Skyline Sweeping Algorithm

- Idea: be lazy!
- Based on the skyline principle:
  - $usc(c_i \bowtie p_j) \geq usc(c_{i+1} \bowtie p_j)$ and $usc(c_i \bowtie p_j) \geq usc(c_i \bowtie p_{j+1})$
  - $\implies$ if the current “cell” fails, we should only examine its neighbors, as they are the best possible alternatives.
- Data structure: a priority queue of candidate cells
- Its search strategy is reminiscent of the J* algorithm.

Priority Queue

\begin{array}{cccc}
  p_1 & & & \\
  & p_2 & & \\
  & & p_3 & \\
  p_4 & & & \\
\end{array}

• $c_1 \bowtie p_1$

Priority Queue
Our Skyline Sweeping Algorithm

- Idea: be lazy!
- Based on the skyline principle:
  - \( \text{usc}(c_i \bowtie p_j) \geq \text{usc}(c_{i+1} \bowtie p_j) \) and \( \text{usc}(c_i \bowtie p_j) \geq \text{usc}(c_i \bowtie p_{j+1}) \)
  \[ \Rightarrow \] if the current “cell” fails, we should only examine its neighbors, as they are the best possible alternatives.
- Data structure: a priority queue of candidate cells
- Its search strategy is reminiscent of the J* algorithm.

\[
\begin{align*}
\text{Priority Queue} & \quad \langle c_1, p_2 \rangle \\
& \quad \langle c_2, p_1 \rangle \\
& \quad \text{c1} \bowtie p_1
\end{align*}
\]
Our Skyline Sweeping Algorithm

- **Idea:** be lazy!
- Based on the **skyline** principle:
  - \( \text{usc}(c_i \Join p_j) \geq \text{usc}(c_{i+1} \Join p_j) \) and \( \text{usc}(c_i \Join p_j) \geq \text{usc}(c_i \Join p_{j+1}) \)
  \[ \Rightarrow \] if the current “cell” fails, we should **only** examine its *neighbors*, as they are the best possible alternatives.
- **Data structure:** a priority queue of *candidate cells*
- Its search strategy is reminiscent of the \( J^* \) **algorithm**.

```
\begin{align*}
\text{\textbf{c}}_1 & \Join \text{p}_1 \\
\text{\textbf{c}}_2 & \Join \text{p}_1 \\
\text{\textbf{c}}_3 & \Join \text{p}_1 \\
\text{\textbf{c}}_4 & \Join \text{p}_1 \\
\end{align*}
```

Priority Queue
Our Skyline Sweeping Algorithm

- **Idea:** be lazy!
- **Based on the skyline principle:**
  - $\text{usc}(c_i \bowtie p_j) \geq \text{usc}(c_{i+1} \bowtie p_j)$ and $\text{usc}(c_i \bowtie p_j) \geq \text{usc}(c_i \bowtie p_{j+1})$
  - $\Rightarrow$ if the current “cell” fails, we should only examine its neighbors, as they are the best possible alternatives.
- **Data structure:** a priority queue of candidate cells
- **Its search strategy** is reminiscent of the J* algorithm.

$p_4$

$p_3$

$p_2$

$p_1$

$c_1$ $c_2$ $c_3$ $c_4$

$\langle c_2, p_1 \rangle$

Priority Queue
Our Skyline Sweeping Algorithm

- Idea: be lazy!
- Based on the skyline principle:
  - $usc(c_i \bowtie p_j) \geq usc(c_{i+1} \bowtie p_j)$ and $usc(c_i \bowtie p_j) \geq usc(c_i \bowtie p_{j+1})$
  - $\Rightarrow$ if the current “cell” fails, we should *only* examine its neighbors, as they are the best possible alternatives.
- Data structure: a priority queue of *candidate cells*
- Its search strategy is reminiscent of the J* algorithm.

<table>
<thead>
<tr>
<th>$p_4$</th>
<th>$c_1$</th>
<th>$c_2$</th>
<th>$c_3$</th>
<th>$c_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$p_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$p_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$\langle c_2, p_1 \rangle$

Priority Queue
Our Skyline Sweeping Algorithm

- **Idea**: be lazy!
- **Based on the skyline principle**:
  - \( \text{usc}(c_i \bowtie p_j) \geq \text{usc}(c_{i+1} \bowtie p_j) \) and \( \text{usc}(c_i \bowtie p_j) \geq \text{usc}(c_i \bowtie p_{j+1}) \)
  \( \Rightarrow \) if the current “cell” fails, we should *only* examine its neighbors, as they are the best possible alternatives.
- **Data structure**: a priority queue of *candidate cells*
- **Its search strategy is reminiscent of the J* algorithm.**

\[
\begin{align*}
\langle c_1, p_1 \rangle & \quad \langle c_2, p_1 \rangle \\
\langle c_2, p_2 \rangle & \quad \langle c_3, p_3 \rangle \\
\langle c_4, p_4 \rangle & \quad \langle c_5, p_5 \rangle
\end{align*}
\]

Priority Queue
Our Skyline Sweeping Algorithm

- Idea: be lazy!
- Based on the skyline principle:
  - \( \text{usc}(c_i \bowtie p_j) \geq \text{usc}(c_{i+1} \bowtie p_j) \) and \( \text{usc}(c_i \bowtie p_j) \geq \text{usc}(c_i \bowtie p_{j+1}) \)
  - \( \Rightarrow \) if the current “cell” fails, we should only examine its neighbors, as they are the best possible alternatives.
- Data structure: a priority queue of candidate cells
- Its search strategy is reminiscent of the J* algorithm.

```
\begin{array}{cccc}
  \langle c_1, p_3 \rangle \\
  \langle c_2, p_1 \rangle \\
  \langle c_2, p_2 \rangle \\
  \langle c_3, p_3 \rangle \\
  \langle c_4, p_4 \rangle
\end{array}
```

Priority Queue
Our Skyline Sweeping Algorithm

- Idea: **be lazy!**
- Based on the **skyline** principle:
  - \( \text{usc}(c_i \bowtie p_j) \geq \text{usc}(c_{i+1} \bowtie p_j) \) and \( \text{usc}(c_i \bowtie p_j) \geq \text{usc}(c_i \bowtie p_{j+1}) \)
  - \( \Rightarrow \) if the current “cell” fails, we should *only* examine its *neighbors*, as they are the best possible alternatives.
- Data structure: a priority queue of **candidate cells**
- Its search strategy is reminiscent of the J\* algorithm.

### Data Structure Illustration

<table>
<thead>
<tr>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>p2</td>
<td>p3</td>
<td>p4</td>
</tr>
</tbody>
</table>

- \( \langle c_3, p_3 \rangle \)
- \( \langle c_1, p_4 \rangle \)
- \( \langle c_4, p_1 \rangle \)
- \( \langle c_2, p_4 \rangle \)
- \( \langle c_4, p_2 \rangle \)

Priority Queue

**Diagram:**

- **c_1 \bowtie p_1**
- **c_1 \bowtie p_2**
- **...**
Towards Optimal Probing

- We still pay the price of bounding a non-monotonic function with (a few) monotonic upper bounding functions
  - See an example below
  - Lots of candidates with high $usc(T, Q)$ return much lower (real) score
  - Unnecessary (expensive) checking
  - Cannot stop earlier

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>-1.5</td>
<td>-1</td>
<td>-0.5</td>
<td>0</td>
<td>0.5</td>
<td>1</td>
<td>1.5</td>
</tr>
<tr>
<td>10</td>
<td>15</td>
<td>20</td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>40</td>
<td></td>
</tr>
</tbody>
</table>
Towards Optimal Probing

- We still pay the price of bounding a non-monotonic function with (a few) monotonic upper bounding functions
  - See an example below
  - Lots of candidates with high usc(T, Q) return much lower (real) score
  - Unnecessary (expensive) checking
  - Cannot stop earlier

Mandelbrot function

![Mandelbrot function graph](image-url)
Block Pipeline Algorithm

- Idea: be even lazier!
  - Partition the space (into blocks) and derive tighter upper bounds for each partitions ⇐ “local knowledge”
  - “Unwilling” to check a candidate until we are quite sure about its “prospect” \( \text{bsc}(T, Q) \)

- Details:
  - Group tuples according to their tf signatures — \( \langle \text{tf}_{w_1}(t), \ldots, \text{tf}_{w_m}(t) \rangle \)
  - Form blocks — candidate JTTs that has the same tf signatures on all dimensions.
  - The signature of a block \( b \) is
    \[
    \text{sig}(b) = \langle \sum_{t \in T} \text{tf}_{w_1}(t), \ldots, \sum_{t \in T} \text{tf}_{w_m}(t) \rangle
    \]
  - A non-monotonic upper bounding function
    \[
    \text{bsc}(b, Q) = \sum_{w \in Q \cap b} \frac{1 + \ln(1 + \ln(\text{sig}_w(b)))}{1 - s} \cdot \ln(\text{idf}_w) \cdot \text{sc}_b(b, Q) \cdot \text{sc}_c(\text{CN}(T), Q)
    \]
    
    \[
    \text{sc}(t, Q) \leq \text{bsc}(b, Q) \leq \text{usc}(t, Q)
    \]
Example of the Block Pipeline Algorithm

\[ p_1 \rightarrow c_1 \rightarrow c_2 \rightarrow c_3 \rightarrow c_4 \rightarrow p_2 \rightarrow \ldots \]

\[ \text{Priority Queue} \]

\[ \text{usc} = 9.7 \]
\[ \text{usc} = 8.3 \]
\[ \text{usc} = 8.3 \]
\[ \text{usc} = 2.0 \]

\[ \text{bsc} = 3.5 \]
\[ \text{bsc} = 6.0 \]
\[ \text{bsc} = 7.1 \]
\[ \text{bsc} = 8.3 \]

\[ \text{sc} (c_3 \triangleleft p_2) \geq 6.0 \Rightarrow \]

Wei Wang (UNSW)
Example of the Block Pipeline Algorithm

\[(n:1, m:0) (n:0, m:1)\]

\[
\begin{align*}
\text{p}_1
\end{align*}
\]

\[
\begin{align*}
\text{p}_2
\end{align*}
\]

\[
\begin{align*}
\text{p}_3
\end{align*}
\]

\[
\begin{align*}
\text{p}_4
\end{align*}
\]

\[
\begin{align*}
\text{c}_1 & \quad \text{c}_2 & \quad \text{c}_3 & \quad \text{c}_4
\end{align*}
\]

\[
\begin{align*}
(n:1, m:0) & \quad (n:0, m:1)
\end{align*}
\]
Example of the Block Pipeline Algorithm

\[ (n:1, m:0) \ (n:0, m:1) \]

\[ \begin{align*}
\text{p}_4 & : \begin{array}{cccc}
\text{c}_1 & \text{c}_2 & \text{c}_3 & \text{c}_4 \\
\text{p}_3 & & & \\
\text{p}_2 & & & \\
\text{p}_1 & & & \\
\end{array} \\
\end{align*} \]

\( (n:1, m:0) \ (n:0, m:1) \)
Example of the Block Pipeline Algorithm

\[
\begin{array}{cccc}
\text{c}_1 & \text{c}_2 & \text{c}_3 & \text{c}_4 \\
\hline
\text{p}_1 & \text{usc} = 9.7 & \text{usc} = 8.3 & \text{usc} = 2.0 \\
\text{p}_2 & \text{usc} = 9.7 & \text{usc} = 8.3 & \text{usc} = 2.0 \\
\text{p}_3 & \text{usc} = 8.3 & \text{usc} = 8.3 & \text{usc} = 2.0 \\
\text{p}_4 & \text{usc} = 8.3 & \text{usc} = 2.0 & \text{usc} = 2.0 \\
\end{array}
\]
Example of the Block Pipeline Algorithm

Priority Queue

\[
\begin{array}{cccc}
\text{c}_1 & \text{c}_2 & \text{c}_3 & \text{c}_4 \\
\end{array}
\]

\[(n:1, m:0) \quad (n:0, m:1)\]
Example of the Block Pipeline Algorithm

Priority Queue

\begin{align*}
\text{usc} &= 9.7 \\
\text{usc} &= 8.3 \\
\text{usc} &= 8.3 \\
\text{usc} &= 2.0 \\
\text{bsc} &= 3.5 \\
\text{bsc} &= 6.0 \\
\text{bsc} &= 7.1 \\
\text{bsc} &= 3.5
\end{align*}

\( (n:1, m:0) (n:0, m:1) \)
Example of the Block Pipeline Algorithm

Priority Queue

| p4    | bsc = 6.0 | usc = 2.0 |
| p3    | bsc = 6.0 | usc = 2.0 |
| p2    | bsc = 3.5 | usc = 8.3 |
| p1    | bsc = 3.5 | usc = 8.3 |

Priority Queue

| 8.3 / 6.0 |
| 8.3 / ??  |
| 9.7 / 3.5 |
| 2.0 / ??  |

 Wei Wang (UNSW)
 SPARK
 16 July, 2007 24 / 33
Example of the Block Pipeline Algorithm

Priority Queue

\[\begin{align*}
\text{p}_1 & \quad \text{bsc} = 3.5 \quad \text{usc} = 8.3 \\
\text{p}_2 & \quad \text{bsc} = 3.5 \quad \text{usc} = 8.3 \\
\text{p}_3 & \quad \text{bsc} = 6.0 \quad \text{usc} = 2.0 \\
\text{p}_4 & \\
\end{align*}\]

\(n:1, m:0\) \quad \text{(n:0, m:1)}

\(c_1\) \quad \text{c}_2 \quad \text{c}_3 \quad \text{c}_4

\begin{align*}
(\text{n:1, m:0}) & \quad (\text{n:0, m:1})
\end{align*}
Example of the Block Pipeline Algorithm

Priority Queue

\[(n:1, m:0) (n:0, m:1)\]

\begin{align*}
\text{p4} & : \text{bsc = 6.0} & \text{usc = 2.0} \\
\text{p3} & : \text{bsc = 3.5} & \text{usc = 7.1} \\
\text{p2} & : \text{bsc = 6.0} & \text{usc = 8.3} \\
\text{p1} & : \text{bsc = 6.0} & \text{usc = 9.7} \\
\text{c1} & & \text{c2} & \text{c3} & \text{c4} \\
\end{align*}

\begin{itemize}
  \item 8.3 / 7.1
  \item 8.3 / 6.0
  \item 9.7 / 3.5
  \item 2.0 / ??
\end{itemize}
Example of the Block Pipeline Algorithm

Priority Queue

\[
\begin{align*}
(p_1, \ n:1, \ m:0) \quad & (p_2, \ n:0, \ m:1) \\
(p_3, \ n:0, \ m:0) \\
(p_4, \ n:1, \ m:1)
\end{align*}
\]
Example of the Block Pipeline Algorithm

Priority Queue

\[
\begin{align*}
\text{c}_1 & \quad \text{c}_2 & \quad \text{c}_3 & \quad \text{c}_4 \\
\text{p}_4 & \quad \text{p}_3 & \quad \text{p}_2 & \quad \text{p}_1 \\
\end{align*}
\]

\[
\begin{align*}
\text{bsc} = & 6.0 & \text{usc} = & 2.0 \\
\text{bsc} = & 3.5 & \text{usc} = & 3.0 \\
\text{bsc} = & 3.5 & \text{usc} = & 2.0 \\
\end{align*}
\]
Example of the Block Pipeline Algorithm

Priority Queue

\[(n:1, m:0) \ (n:0, m:1)\]

\[
\begin{array}{cccc}
  p_4 & p_3 & p_2 & p_1 \\
  \text{bsc} = 6.0 & \text{usc} = 2.0 & \text{bsc} = 3.5 & \text{bsc} = 3.5 \\
  \text{usc} = 2.0 & \text{usc} = 8.3 & \text{usc} = 8.3 & \text{usc} = 8.3 \\
  \text{bsc} = 3.5 & \text{bsc} = 3.5 & \text{bsc} = 3.5 & \text{bsc} = 3.5 \\
\end{array}
\]

- \(8.3 / 6.0\)
- \(9.7 / 3.5\)
- \(2.0 / ??\)
Example of the Block Pipeline Algorithm

Priority Queue

\[(n:1, m:0) \quad (n:0, m:1)\]

\[
\begin{align*}
(p_4) & : \text{bsc} = 6.0, \text{usc} = 2.0 \\
(p_3) & : \text{bsc} = 6.0, \text{usc} = 2.0 \\
(p_2) & : \text{bsc} = 3.5, \text{usc} = 6.1 \\
(p_1) & : \text{bsc} = 3.5, \text{usc} = 2.0
\end{align*}
\]

\[
\begin{align*}
(p_4) & : 8.3 / 6.0 \\
(p_3) & : 9.7 / 3.5 \\
(p_1) & : 2.0 / ??
\end{align*}
\]
Example of the Block Pipeline Algorithm

<table>
<thead>
<tr>
<th></th>
<th>p4</th>
<th>p3</th>
<th>p2</th>
<th>p1</th>
</tr>
</thead>
<tbody>
<tr>
<td>bsc</td>
<td>6.0</td>
<td>3.5</td>
<td>3.5</td>
<td>3.5</td>
</tr>
<tr>
<td>usc</td>
<td>2.0</td>
<td>2.0</td>
<td>6.1</td>
<td>2.0</td>
</tr>
</tbody>
</table>

Priority Queue

8.3 / 6.0
9.7 / 3.5
2.0 / ??

Wei Wang (UNSW) 16 July, 2007 24 / 33
Example of the Block Pipeline Algorithm

Priority Queue

\[ sc(c_3 \otimes p_2) \geq 6.0 \Rightarrow \text{STOP} \]
Other Issues

- Generalization to Multiple CNs
  - Initially, push lower-left points of each CN into heap, and sort them all together

- Progressively output results

- Optimization: Range Parameterized Join
  - Why?
    - Low join selectivity (i.e., only few candidates really join)
    - High db connection overhead
  - How?
    - Evaluate a block of candidates in one SQL! (e.g.,)
    - Group tuples by: tf signature / local score (watf) / RID
Introduction

Ranking Search Results

Efficient Query Processing

Experiments

Related Work

Conclusions
Experiment Setup

- **Data**
  - DBLP: \(\approx 0.9M\) tuples, 6 tables, 18 queries
  - IMDB: \(\approx 10M\) tuples, 8 tables, 22 queries
  - Mondial: \(\approx 10K\) tuples, 28 tables, 35 queries

- **Settings**
  - **ORACLE** 10g Express / MySQL v5.0.18
  - JDK 1.5 + JDBC
  - 1.8GHz CPU / 512M memory / Debian GNU/Linux 3.1

- **Comparison**
  - Effectiveness: [VLDB 03], [SIGMOD 06], and ours
  - Efficiency: Sparse, GP from [VLDB 03], and our SS and BP

- **Metrics**
  - Effectiveness
    - \#-Rel: \(\text{COUNT(top-1 answer is relevant)}\)
    - R-Rank: \(1 / \text{(position of the first relevant result)}\)
  - Efficiency
    - Elapsed time
    - ...

## Effectiveness

<table>
<thead>
<tr>
<th>Method</th>
<th>Top-1 Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Hristidis VLDB03]</td>
<td>InProceeding: <strong>Clique</strong>-to-<strong>Clique</strong> Distance . . .</td>
</tr>
<tr>
<td>[Liu SIGMOD06]</td>
<td>Series → InProceeding: Maximum <strong>Clique</strong> Transversals</td>
</tr>
<tr>
<td>Ours</td>
<td>Person: <strong>Nikos</strong> Mamoulis ← RPI → InProceeding: Constraint-Based Algorithms for Computing <strong>Clique</strong> Intersection Joins</td>
</tr>
</tbody>
</table>

Table: Top-1 Result for Query “nikos clique” on DBLP
Table: Effectiveness on the DBLP Dataset Based on Top-20 Results

<table>
<thead>
<tr>
<th></th>
<th>[Hristidis VLDB03]</th>
<th>[Liu SIGMOD06]</th>
<th>$p = 1.0$</th>
<th>$p = 1.4$</th>
<th>$p = 2.0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Rel</td>
<td>2</td>
<td>2</td>
<td>16</td>
<td>16</td>
<td>18</td>
</tr>
<tr>
<td>R-Rank</td>
<td>$\leq 0.243$</td>
<td>$\leq 0.333$</td>
<td>0.926</td>
<td>0.935</td>
<td>1</td>
</tr>
</tbody>
</table>
Efficiency ...
1 Introduction

2 Ranking Search Results

3 Efficient Query Processing

4 Experiments

5 Related Work

6 Conclusions
Related Work

Based on the data model

1. **Graph-based:**
   - Proximity Search [VLDB 98]
   - BANKS I [ICDE 02], BANKS II [VLDB 05]
   - Recent work [PODS 06, ICDE 07, SIGMOD 07]

   **Strength** Fast query response
   **Weakness** Main-memory based; hard to integrate IR-style ranking functions

2. **Relation-based:**
   - DBXplorer [VLDB 03]
   - DISCOVER I [VLDB 02], DISCOVER II [VLDB 03]
   - Recent work [IDEAS 05, SIGMOD 06]

   **Strength** Ease of maintenance and deployment; allow sophisticated ranking
   **Weakness** Query response time
Say “no” to SQL, say “Yes” to keyword search.
Conclusions

- Say “no” to SQL, say “Yes” to keyword search.

OR

[Image of IMDB website search interface]
Conclusions

- Say “no” to SQL, say “Yes” to keyword search.
Conclusions

- Say “no” to SQL, say “Yes” to keyword search.
- It is effective: Meaningful query results with appropriate rankings
Conclusions

- Say “no” to SQL, say “Yes” to keyword search.
- It is effective: Meaningful query results with appropriate rankings
- It is efficient: Second-level response time for $\approx 10^7$ tuple DB on a commodity PC
Thank You!