Top-k join queries

- In a top-k join query we have \( n > 1 \) input relations and a scoring function \( S \) defined on the result of the join, i.e.:

  ```
  SELECT <some attributes>
  FROM R1, R2, ..., Rn
  WHERE <join and local conditions>
  ORDER BY S(p1, p2, ..., pm) [DESC]
  STOP AFTER k
  ```

  where \( p1, p2, ..., pm \) are scoring criteria (the “preferences”)

- Now we consider the general case of many-to-many (M-N) joins, e.g.:

  ```
  SELECT *
  FROM RESTAURANTS R, HOTELS H
  WHERE R.City = H.City
  AND R.Nation = 'Italy' AND H.Nation = 'Italy'
  ORDER BY R.Price + H.Price
  STOP AFTER 2
  ```

- Notice that the join is not anymore on the relations’ PK’s
### Top-k M-N join queries: example

#### Restaurants

<table>
<thead>
<tr>
<th>RName</th>
<th>City</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Al vecchio mulino</td>
<td>Bologna</td>
<td>25</td>
</tr>
<tr>
<td>La tavernetta</td>
<td>Roma</td>
<td>30</td>
</tr>
<tr>
<td>Tutti a tavola!</td>
<td>Bologna</td>
<td>40</td>
</tr>
<tr>
<td>Le delizie del palato</td>
<td>Milano</td>
<td>50</td>
</tr>
<tr>
<td>Acqua in bocca</td>
<td>Roma</td>
<td>70</td>
</tr>
</tbody>
</table>

#### Hotels

<table>
<thead>
<tr>
<th>HName</th>
<th>City</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>La pensioncina</td>
<td>Milano</td>
<td>40</td>
</tr>
<tr>
<td>Dormi Bene!</td>
<td>Milano</td>
<td>50</td>
</tr>
<tr>
<td>RonfRonf</td>
<td>Roma</td>
<td>60</td>
</tr>
<tr>
<td>La Cascina</td>
<td>Bologna</td>
<td>80</td>
</tr>
<tr>
<td>La Quiete</td>
<td>Bologna</td>
<td>85</td>
</tr>
<tr>
<td>Il Riposino</td>
<td>Roma</td>
<td>90</td>
</tr>
<tr>
<td>CheapSleep</td>
<td>Bologna</td>
<td>100</td>
</tr>
</tbody>
</table>

#### The “easy” case

- In the most favorable case we can access both (all) inputs using indexes on the join attributes (e.g., city).
- In this case the resulting algorithm is quite similar to TA:

1. Perform a round of s.a.'s
2. For each retrieved tuple:
   1. using the index on the join attributes, do r.a.'s to retrieve all the matches on other inputs
   2. keep only the best k so-resulting join combinations
   3. If one of such new join combinations is among the top-k combinations seen so far, keep it, otherwise discard it

   until the threshold condition is satisfied (i.e., when no unseen join combination can be better than any of the current top-k results)
The easy case: example

<table>
<thead>
<tr>
<th>RName</th>
<th>City</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Al vecchio mulino</td>
<td>Bologna</td>
<td>25</td>
</tr>
<tr>
<td>La tavernetta</td>
<td>Roma</td>
<td>30</td>
</tr>
<tr>
<td>Tutti a tavola!</td>
<td>Bologna</td>
<td>40</td>
</tr>
<tr>
<td>Le delizie del palato</td>
<td>Milano</td>
<td>50</td>
</tr>
<tr>
<td>Acqua in bocca</td>
<td>Roma</td>
<td>70</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HName</th>
<th>City</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>La pensioncina</td>
<td>Milano</td>
<td>40</td>
</tr>
<tr>
<td>Dormi Bene!</td>
<td>Milano</td>
<td>50</td>
</tr>
<tr>
<td>RonfRonf</td>
<td>Roma</td>
<td>60</td>
</tr>
<tr>
<td>La Cascina</td>
<td>Bologna</td>
<td>80</td>
</tr>
<tr>
<td>La Quiete</td>
<td>Bologna</td>
<td>85</td>
</tr>
<tr>
<td>Il Riposino</td>
<td>Roma</td>
<td>90</td>
</tr>
<tr>
<td>CheapSleep</td>
<td>Bologna</td>
<td>100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RName</th>
<th>Hname</th>
<th>City</th>
<th>TotPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Al vecchio mulino</td>
<td>La Cascina</td>
<td>Bologna</td>
<td>105</td>
</tr>
<tr>
<td>Al vecchio mulino</td>
<td>La Quiete</td>
<td>Bologna</td>
<td>110</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RName</th>
<th>Hname</th>
<th>City</th>
<th>TotPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Le delizie del palato</td>
<td>La pensioncina</td>
<td>Milano</td>
<td>90</td>
</tr>
<tr>
<td>Al vecchio mulino</td>
<td>La Cascina</td>
<td>Bologna</td>
<td>105</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RName</th>
<th>Hname</th>
<th>City</th>
<th>TotPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Le delizie del palato</td>
<td>La pensioncina</td>
<td>Milano</td>
<td>90</td>
</tr>
<tr>
<td>La tavernetta</td>
<td>RonfRonf</td>
<td>Roma</td>
<td>90</td>
</tr>
</tbody>
</table>

Top-k: advanced (2)

The easy case: join graphs (1)

- With n > 2 inputs, an important role is played by the “join graph”, i.e., the graph whose vertices are the n relations, and where there is an edge connecting Ri and Rj if the query includes a join predicate on such relations

```
```

A “clique” (fully connected) join graph

```
```

Top-k: advanced (2)
The easy case: join graphs (2)

- Unless we have a clique join graph, some r.a.'s might not be directly performed.
- Rather, one has to sequentially follow the edges of the join graph to obtain valid join combinations.

```
SELECT *
FROM R1, R2, R3
ORDER BY R1.X + R2.Y + R3.Z DESC
STOP AFTER 2
```

Example: after a s.a. on R1, r.a.'s can be performed only on R2; then, for each retrieved tuple of R2, corresponding r.a.'s are executed on R3.

The easy case: join graphs (3)

- When the join graph is cyclic, it is likely that many r.a.'s will lead to invalid join combinations, thus wasting a lot of work!

```
SELECT *
FROM R1, R2, R3
AND R3.C = R1.C
ORDER BY R1.X + R2.Y + R3.Z DESC
STOP AFTER 2
```

Example: after a s.a. on R1, r.a.'s can be performed only on R2; then, for each retrieved tuple of R2, corresponding r.a.'s are executed on R3.
The “difficult” case: no random accesses

- If join indexes are not available, the only alternative is to compute the result by using only sorted accesses, along the lines of NRA*
- The basic algorithm for this scenario is called Rank-Join [IAE03], and its description requires the following additional notation:
  - For each ranked list $L_j$, let $p_j^\text{max}$ denote the first (highest) score seen on $L_j$
  - Let $T$ be the maximum among the following $m$ values:
    \[ S(p_1^\text{max}, p_2^\text{max}, \ldots, p_m^\text{max}), S(p_1^\text{max}, p_2^\text{max}, \ldots, p_m^\text{max}), \ldots, S(p_1^\text{max}, p_2^\text{max}, \ldots, p_m^\text{max}) \]
  - This is also called the “corner bound”

- Let $j$ denote a generic join combination, i.e., $j = (t_1, t_2, \ldots, t_m)$
- The (at this point, obvious) observation is that one can halt when there are $k$ join combinations $j$ such that $S(j) \geq T$
- S.a.’s can be executed using a round-robin strategy, by accessing the list for which $p_j$ is maximum, etc.

Top-k: advanced (2)  Sistemi Informativi M  9

Visualizing the corner bound

- For simplicity, assume $p_1^\text{max} = p_2^\text{max} = 1$, and $S = \sum$
- In the figure it is: $p_1 = 0.4$ and $p_2 = 0.6$
- Thus, $T = \max\{(0.4+1),(1+0.6)\} = 1.6$

RankJoin stops when the current top-k results lie in this region

No unseen join combination can have a score higher than $T$, since in the best case an unseen tuple from $L_2$ will match a tuple from $L_1$ with score = 1
Is the corner bound the best possible one?

- When \( m=2 \) it can be proved that Rank-Join is instance optimal, i.e., the corner bound is tight (there could be a join combination \( j \) s.t. \( S(j) = T \))

- On the other hand, when \( m > 2 \) things are more complex to analyze, and the instance optimality is guaranteed only if the join conditions are considered as a “black box”
  - I.e., arguments to prove instance optimality do not consider the actual join conditions

- On the other hand, when the join predicates are taken into account, no algorithm based on the corner bound is instance optimal [SP08]
- The same negative result holds even for \( n = 2 \) inputs, but in which at least one of the inputs has 2 partial scores for each tuple (i.e., \( n < m \))

---

Rank-Join: example of non optimality

```
S = SUM

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>p1</th>
<th></th>
<th>A</th>
<th>p2</th>
<th></th>
<th>A</th>
<th>p3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1.0</td>
<td></td>
<td></td>
<td>y</td>
<td>1.0</td>
<td></td>
<td>z</td>
<td>1.0</td>
</tr>
<tr>
<td>x</td>
<td>0.95</td>
<td></td>
<td></td>
<td>a</td>
<td>0.7</td>
<td></td>
<td>x</td>
<td>0.8</td>
</tr>
<tr>
<td>w</td>
<td>0.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SELECT *
FROM R1, R2, R3
ORDER BY p1 + p2 + p3 DESC
STOP AFTER 1

<table>
<thead>
<tr>
<th>Res</th>
<th>j</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(a,a,a)</td>
<td>2.5</td>
</tr>
</tbody>
</table>
```

- After the first 3 s.a.’s rounds, the corner bound yields \( T = 0.9 + 1 + 1 = 2.9 \)
- However, no unseen tuple from L1 can lead to a join combination \( j \) with \( S(j) > 2.5 \), thus we could stop here, with just 9 s.a.’s!
- On the other hand, on this instance Rank-Join might incur an arbitrarily high cost, depending on how scores are distributed
  - Notice that the number of tuples in L1 between \((x,0.9)\) and \((w,0.5)\) is unbounded
A tight bounding scheme: results

- Besides showing the deficiencies of Rank-Join, [SP08] has also introduced a tight bounding scheme that guarantees instance optimality.
- The method, not described here, has the following major features:
  - It has **polynomial data complexity**, i.e., it runs in time polynomial in the number of tuples retrieved from the ranked lists.
  - It is **NP-hard under query complexity**, i.e., its running time grows exponentially with the number of inputs.
  - Interestingly, it relies on the concept of “tuple domination”, which is at the core of skyline queries.

Enlarging the perspective

- A challenge concerning top-k queries is how to incorporate ranking-based techniques in a relational DBMS.
- This is needed to improve performance for general top-k queries.
- The RankDB project (http://www.cs.uwaterloo.ca/~ilyas/RankDB/) has provided fundamental contributions towards the solution of this problem, leading to a prototype system, called RankSQL [LCI+05], in which ranking is treated as a “first-class” citizen.
- We sketch the basic concepts of RankSQL, in particular:
  - The “splitting and interleaving” requirements
  - The concept of “rank-relation” and the “ranking principle”
  - The “rank algebra” for rank-relations
Splitting and interleaving

Consider the following query SQL:

```
SELECT * 
FROM RESTAURANTS R, HOTELS H 
WHERE R.Area = H.Area 
    AND H.Stars = 3 
    AND R.Cuisine = 'chinese'
```

A possible access plan (P1) for this query is:

```
HOTELS                RESTAURANTS
\( \sigma_{\text{Stars} = 3} \) \( \sigma_{\text{Cuisine} = \text{chinese}} \) \( \pi \)
```

P1 is much better than the following “monolithic” plan (P2) in which ‘X’ denotes the Cartesian product:

```
\( \sigma_{(\text{Stars} = 3) \land (\text{Cuisine} = \text{chinese})} \) \( \pi \)
```

We can transform P2 into P1 by:

• SPLITTING the Boolean predicates into joins and selection, and
• INTERLEAVING them

The same should be done with the ranking function!

Rank-relations and the ranking principle

To make query optimizers “rank-aware” it is necessary to introduce the concept of rank-relation, i.e., a relation that consists of tuples with scores.

The definition is tightly related to the ranking principle, which formalizes the now-well-known fact that the “most promising” tuples should be processed first.

**Rank-relation:**

Given a relation R and a monotone scoring function \( S(p_1, p_2, \ldots, p_m) \), the rank-relation \( R_P \) with \( P \subseteq \{p_1, p_2, \ldots, p_m\} \), is the relation R augmented with a ranking defined as follows:

• The score of a tuple \( t \) in \( R_P \) is the maximal possible score, \( S^P(t) \), with respect to \( S \), where \( S^P(t) \) is computed by substituting in \( S \) the values of \( p_j(t) \), if \( p_j \in P \), otherwise 1 (or the maximum possible value for \( p_j \)).
• Tuples in \( R_P \) are ranked by decreasing values of \( S^P(t) \) (ranking principle).

Notice that \( S^P(t) = S(t) \) when \( P = \{p_1, p_2, \ldots, p_m\} \), and that \( R_{\emptyset} \equiv R \)
The RankSQL algebra

- The RankSQL algebra extends the semantics of RA to rank-relations.
- It introduces a new rank operator, $\mu$, which applies to a rank-relation $R_p$ a not-evaluated-yet preference $p$ ($p \notin P$), yielding the new rank-relation $R_{p,\mu(p)}$.
- The $\mu$ operator is the basis to split the scoring function $S(p_1,p_2,\ldots,p_m)$, since:

$$R_{p_1,p_2,\ldots,p_m} = \mu_{p_1}(\mu_{p_2}(\ldots(\mu_{p_m}(R))))$$

i.e., the final ranking can be obtained by applying one-by-one the preferences.
- Any order of evaluation is admissible, since $\mu_{p_1}(\mu_{p_2}(R_p)) = \mu_{p_2}(\mu_{p_1}(R_p))$.
- Interleaving with selections and joins is now possible, e.g., for selection:

$$\mu_p(\sigma_c(R_{p,\mu})) = \sigma_c(\mu_p(R_{p,\mu}))$$

Because of the ranking principle, a $\mu_p$ operator can return a tuple iff it is guaranteed that there is no unseen tuple $t'$ such that $S_{p,\mu(p)}(t') > S_{p,\mu(p)}(t)$.
- This can be done as soon as $\mu_p$ fetches a tuple $t''$ such that $S_{p,\mu(p)}(t'') \geq S_{p,\mu(p)}(t)$.
- Index (and sequential) scans are also treated as operators, since they could be used in place of $\mu$ to rank tuples according to a preference $p$.

Top-k: advanced (2) Sistemi Informativi M 17

---

The rank operator in action

<table>
<thead>
<tr>
<th>Hotel</th>
<th>$p_1$</th>
<th>$p_2$</th>
<th>$p_3$</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>h1</td>
<td>0.7</td>
<td>0.8</td>
<td>0.9</td>
<td>2.4</td>
</tr>
<tr>
<td>h2</td>
<td>0.9</td>
<td>0.85</td>
<td>0.8</td>
<td>2.55</td>
</tr>
<tr>
<td>h3</td>
<td>0.5</td>
<td>0.45</td>
<td>0.75</td>
<td>1.7</td>
</tr>
<tr>
<td>h4</td>
<td>0.4</td>
<td>0.7</td>
<td>0.95</td>
<td>2.05</td>
</tr>
</tbody>
</table>

**SELECT** *
**FROM** HOTELS H
**ORDER BY** $p_1+p_2+p_3$ **DESC**
**STOP AFTER** 1

Top-k: advanced (2) Sistemi Informativi M 18
The rank operator in action

```
SELECT * FROM HOTELS H
ORDER BY p1+p2+p3 DESC
STOP AFTER 1

Hotel ... p1 p2 p3 score
h1    0.9 1.0 1.0 2.9
h2    0.9 1.0 1.0 2.9
h3    0.9 1.0 1.0 2.9
h4    0.9 1.0 1.0 2.9
...

SELECT * FROM HOTELS H
ORDER BY p1+p2+p3 DESC
STOP AFTER 1

Hotel ... p1 p2 p3 score
h1    0.9 1.0 1.0 2.9
h2    0.9 1.0 1.0 2.9
h3    0.9 1.0 1.0 2.9
h4    0.9 1.0 1.0 2.9
...
```
The rank operator in action

```
SELECT *
FROM HOTELS H
ORDER BY p1+p2+p3 DESC
STOP AFTER 1
```

H1: p1, p2, p3

```
<table>
<thead>
<tr>
<th>Hotel</th>
<th>p1</th>
<th>p2</th>
<th>p3</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>h1</td>
<td>0.9</td>
<td>1.0</td>
<td>1.0</td>
<td>2.9</td>
</tr>
<tr>
<td>h2</td>
<td>0.9</td>
<td>0.85</td>
<td>1.0</td>
<td>2.75</td>
</tr>
<tr>
<td>h3</td>
<td>0.9</td>
<td>1.0</td>
<td>1.0</td>
<td>2.9</td>
</tr>
<tr>
<td>h4</td>
<td>0.9</td>
<td>1.0</td>
<td>1.0</td>
<td>2.9</td>
</tr>
</tbody>
</table>
```

H2: p1, p2

```
<table>
<thead>
<tr>
<th>Hotel</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>h2</td>
<td>2.75</td>
</tr>
</tbody>
</table>
```

H3: p1

```
```

H4: p2, p3

```
```

The rank operator in action

```
SELECT *
FROM HOTELS H
ORDER BY p1+p2+p3 DESC
STOP AFTER 1
```

H1: p1, p2, p3

```
<table>
<thead>
<tr>
<th>Hotel</th>
<th>p1</th>
<th>p2</th>
<th>p3</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>h1</td>
<td>0.7</td>
<td>1.0</td>
<td>1.0</td>
<td>2.7</td>
</tr>
<tr>
<td>h2</td>
<td>0.9</td>
<td>0.85</td>
<td>1.0</td>
<td>2.75</td>
</tr>
<tr>
<td>h3</td>
<td>0.7</td>
<td>1.0</td>
<td>1.0</td>
<td>2.7</td>
</tr>
<tr>
<td>h4</td>
<td>0.7</td>
<td>1.0</td>
<td>1.0</td>
<td>2.7</td>
</tr>
</tbody>
</table>
```

H2: p1, p2

```
<table>
<thead>
<tr>
<th>Hotel</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>h2</td>
<td>2.75</td>
</tr>
</tbody>
</table>
```

H3: p1

```
```

H4: p2, p3

```
```

Top-k: advanced (2)
### The rank operator in action

#### Table 1: Hotel Ratings

<table>
<thead>
<tr>
<th>Hotel</th>
<th>p1</th>
<th>p2</th>
<th>p3</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>h1</td>
<td>0.7</td>
<td>0.8</td>
<td>1.0</td>
<td>2.5</td>
</tr>
<tr>
<td>h2</td>
<td>0.9</td>
<td>0.85</td>
<td>0.8</td>
<td>2.55</td>
</tr>
<tr>
<td>h3</td>
<td>0.7</td>
<td>1.0</td>
<td>1.0</td>
<td>2.7</td>
</tr>
<tr>
<td>h4</td>
<td>0.7</td>
<td>1.0</td>
<td>1.0</td>
<td>2.7</td>
</tr>
<tr>
<td>...</td>
<td>0.7</td>
<td>1.0</td>
<td>1.0</td>
<td>2.7</td>
</tr>
</tbody>
</table>

**SQL Query:**

```
SELECT *
FROM HOTELS H
ORDER BY p1 + p2 + p3 DESC
STOP AFTER 1
```

---

### The rank operator in action

#### Table 2: Hotel Ratings

<table>
<thead>
<tr>
<th>Hotel</th>
<th>p1</th>
<th>p2</th>
<th>p3</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1</td>
<td>0.7</td>
<td>0.8</td>
<td>1.0</td>
<td>2.5</td>
</tr>
<tr>
<td>H2</td>
<td>0.9</td>
<td>0.85</td>
<td>0.8</td>
<td>2.55</td>
</tr>
<tr>
<td>H3</td>
<td>0.7</td>
<td>1.0</td>
<td>1.0</td>
<td>2.7</td>
</tr>
<tr>
<td>H4</td>
<td>0.7</td>
<td>1.0</td>
<td>1.0</td>
<td>2.7</td>
</tr>
<tr>
<td>...</td>
<td>0.7</td>
<td>1.0</td>
<td>1.0</td>
<td>2.7</td>
</tr>
</tbody>
</table>

**SQL Query:**

```
SELECT *
FROM HOTELS H
ORDER BY p1 + p2 + p3 DESC
STOP AFTER 1
```

---

**Note:** Evaluation of p2 can be delayed.
The rank operator in action

<table>
<thead>
<tr>
<th>Hotel</th>
<th>p1</th>
<th>p2</th>
<th>p3</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>h1</td>
<td>0.7</td>
<td>0.8</td>
<td>1.0</td>
<td>2.5</td>
</tr>
<tr>
<td>h2</td>
<td>0.9</td>
<td>0.85</td>
<td>0.8</td>
<td>2.55</td>
</tr>
<tr>
<td>h3</td>
<td>0.5</td>
<td>1.0</td>
<td>1.0</td>
<td>2.5</td>
</tr>
<tr>
<td>h4</td>
<td>0.5</td>
<td>1.0</td>
<td>1.0</td>
<td>2.5</td>
</tr>
<tr>
<td>...</td>
<td>0.5</td>
<td>1.0</td>
<td>1.0</td>
<td>2.5</td>
</tr>
</tbody>
</table>

SELECT *
FROM HOTELS H
ORDER BY p1+p2+p3 DESC
STOP AFTER 1

Top-k: advanced (2)
The rank operator in action

<table>
<thead>
<tr>
<th>Hotel</th>
<th>p1</th>
<th>p2</th>
<th>p3</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>h1</td>
<td>0.7</td>
<td>0.8</td>
<td>0.9</td>
<td>2.4</td>
</tr>
<tr>
<td>h2</td>
<td>0.9</td>
<td>0.85</td>
<td>0.8</td>
<td>2.55</td>
</tr>
<tr>
<td>h3</td>
<td>0.5</td>
<td>0.45</td>
<td>1.0</td>
<td>1.95</td>
</tr>
<tr>
<td>h4</td>
<td>0.5</td>
<td>1.0</td>
<td>1.0</td>
<td>2.5</td>
</tr>
<tr>
<td>...</td>
<td>0.5</td>
<td>1.0</td>
<td>1.0</td>
<td>2.5</td>
</tr>
</tbody>
</table>

**H_{p1, p2, p3}**

<table>
<thead>
<tr>
<th>Hotel</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>h2</td>
<td>2.55</td>
</tr>
<tr>
<td>h1</td>
<td>2.4</td>
</tr>
</tbody>
</table>

**H_{p1, p2}**

<table>
<thead>
<tr>
<th>Hotel</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>h2</td>
<td>2.75</td>
</tr>
<tr>
<td>h1</td>
<td>2.5</td>
</tr>
<tr>
<td>h3</td>
<td>1.95</td>
</tr>
</tbody>
</table>

**H_{p1}**

<table>
<thead>
<tr>
<th>Hotel</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>h2</td>
<td>2.9</td>
</tr>
<tr>
<td>h1</td>
<td>2.7</td>
</tr>
<tr>
<td>h3</td>
<td>2.5</td>
</tr>
</tbody>
</table>

(*) Evaluation of p3 can be omitted

```
SELECT *
FROM HOTELS H
ORDER BY p1+p2+p3 DESC
STOP AFTER 1
```

The rank operator in action

<table>
<thead>
<tr>
<th>Hotel</th>
<th>p1</th>
<th>p2</th>
<th>p3</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>h1</td>
<td>0.7</td>
<td>0.8</td>
<td>0.9</td>
<td>2.4</td>
</tr>
<tr>
<td>h2</td>
<td>0.9</td>
<td>0.85</td>
<td>0.8</td>
<td>2.55</td>
</tr>
<tr>
<td>h3</td>
<td>0.5</td>
<td>0.45</td>
<td>1.0</td>
<td>1.95</td>
</tr>
<tr>
<td>h4</td>
<td>0.5</td>
<td>1.0</td>
<td>1.0</td>
<td>2.5</td>
</tr>
<tr>
<td>...</td>
<td>0.5</td>
<td>1.0</td>
<td>1.0</td>
<td>2.5</td>
</tr>
</tbody>
</table>

**H_{p1, p2, p3}**

<table>
<thead>
<tr>
<th>Hotel</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>h2</td>
<td>2.55</td>
</tr>
<tr>
<td>h1</td>
<td>2.4</td>
</tr>
</tbody>
</table>

**H_{p1, p2}**

<table>
<thead>
<tr>
<th>Hotel</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>h2</td>
<td>2.75</td>
</tr>
<tr>
<td>h1</td>
<td>2.5</td>
</tr>
<tr>
<td>h3</td>
<td>1.95</td>
</tr>
</tbody>
</table>

**H_{p1}**

<table>
<thead>
<tr>
<th>Hotel</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>h2</td>
<td>2.9</td>
</tr>
<tr>
<td>h1</td>
<td>2.7</td>
</tr>
<tr>
<td>h3</td>
<td>2.5</td>
</tr>
</tbody>
</table>

```
SELECT *
FROM HOTELS H
ORDER BY p1+p2+p3 DESC
STOP AFTER 1
```
RankSQL: an example (1)

```
SELECT * -- example adapted from [LCI+05]
FROM A,B,C
WHERE A.X = B.X AND B.Y = C.Y
AND A.W = 1 AND B.Z = 1
ORDER BY A.p1 + A.p2 + B.p3 + B.p4 + C.p5 DESC
STOP AFTER k
```

A traditional (not rank-aware) plan for processing the query could just exploit a Top-Sort operator to avoid sorting all the tuples produced by the second (top-most) join.

---

Plan 1 (traditional)

```
Top-Sort_{k,S}
```

```
Sort-Y
IdxScan_Y
```

```
Sort-Merge
```

```
A
```

```
B
```

---

Plan 2)

```
Top-Scan_k,S
```

```
Rank-Join
```

```
IdxScan_{p3}
```

```
H_{p4}
```

```
A
```

```
B
```

---

Plan 3)

```
Top-Scan_k,S
```

```
Rank-Join
```

```
IdxScan_{p5}
```

```
C
```

---

Plan 4)

```
Top-Scan_k,S
```

```
Rank-Join
```

```
IdxScan_{p4}
```

```
C
```

---

RankSQL: an example (2)

With rank-aware operators several alternative plans are possible.

---

Plan 4)

```
Top-Scan_k,S
```

```
Rank-Join
```

```
IdxScan_{p5}
```

```
C
```

---

Plan 2)

```
Top-Scan_k,S
```

```
Rank-Join
```

```
IdxScan_{p3}
```

```
B
```

---

Plan 3)

```
Top-Scan_k,S
```

```
Rank-Join
```

```
IdxScan_{p1}
```

```
SeqScan
```

```
A
```

```
B
```

---

Plan 1 (traditional)

```
Top-Sort_{k,S}
```

```
Sort-Merge
```

```
A
```

```
B
```
Experimential results (from [LCI+05+])

- None of the plans is always the best → need for optimization
- The (optimal) traditional plan is almost never the best one

On the placement of the Top operator

- As a final issue concerning top-k queries, let us consider the problem of where the Top operator can be placed in an access plan
- The default placement is at the top of the access plan
  - Notice that this was also the case for the rank-aware plans in the previous example
- However, in some cases it is possible to earlier discard tuples in excess, for instance:

  ```sql
  SELECT E.*, D.Dname
  FROM EMP E, DEPT D
  WHERE E.DNO = D.DNO
  ORDER BY E.Salary DESC
  STOP AFTER 100
  ```

- Here we can just join the top-100 employees
Safe Top placement rule

- If we anticipate the evaluation of the Top operator, we must be sure that none of its output tuples is subsequently discarded by other operators.
- This can be verified by looking at:
  - DB integrity constraints (FK, PK, NOT NULL, ...), and
  - The query predicates that remain to be evaluated after the Top operator is executed.

```
SELECT E.*, D.Dname
FROM EMP E, DEPT D
WHERE E.DNO = D.DNO
ORDER BY E.Salary DESC
STOP AFTER 100
```

- Here the Top can be pushed-down the join provided E.DNO is declared as a foreign key with non-null values.

Recap

- For general (M-N) top-k join queries, the simplest case to deal with is when random accesses are possible, in which case the principles of TA algorithm apply.
- The Rank-Join operator has been designed for scenarios in which r.a.’s are not possible.
- Rank-Join with the “corner bound” is instance optimal only when join conditions are not taken into account or when there are only 2 inputs, each with a single partial score.
- The RankSQL algebra represents a relevant contribution in making DBMS’s fully “rank-aware”.
- Its design principles derive from the requirements of “splitting and interleaving” the evaluation of the scoring function.
- RankSQL manages rank-relations, in which tuples are ranked according to the “ranking principle”.
- The novel rank operator evaluates a single preference.