Top-k queries

Tecnologie e Sistemi per la Gestione di Basi di Dati e Big Data M
Top-k queries: what and why

Top-k queries aim to retrieve, from a potentially (very) large result set, only the k (k \geq 1) best answers

- Best = most important/interesting/relevant/...

- The need for such kind of queries arises in a variety of modern scenarios, such as e-commerce, scientific DB’s, Web search, multimedia systems, etc.

- The definition of top-k queries requires a system able to “rank” objects (the 1st best result, the 2nd one, ...)

Ranking = ordering the DB objects based on their “relevance” to the query
We're sorry. There are no Flight + Hotel Combo deals available for your search criteria.

What you can do:
- Please change your search criteria by adding more connections and removing airline or time preferences. For international travel, please consider searching for different dates.
- If you would like help purchasing Flight + Hotel Combo for your search criteria, please call 888-TRAVELOCITY (1-888-872-8356) to speak to a Travelocity Agent.

1. Where are you going?
   - From: JFK
   - To: CUN

2. When are you going?
   - Depart: Dec 20
   - Return: Dec 28

3. What is your preferred number of stops?
   - Maximum Stops: 3

4. How many travelers are there?
   - Adults (13-64): 2
   - Seniors (65+): 0
   - Children (2-12): 0

No result!
**Lowest Price Trip - 2 Adults**

<table>
<thead>
<tr>
<th>Date</th>
<th>Departure City</th>
<th>Departure City Code</th>
<th>Departure Time</th>
<th>Arrival City</th>
<th>Arrival City Code</th>
<th>Arrival Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sun, Dec 21</td>
<td>New York-Kennedy, NY (JFK)</td>
<td>JFK</td>
<td>4:25pm to 10:59pm</td>
<td>Cancun, Mexico (CUN)</td>
<td>CUN</td>
<td></td>
</tr>
<tr>
<td>Sun, Dec 28</td>
<td>Cancun, Mexico (CUN)</td>
<td>CUN</td>
<td>7:10am to 3:12pm</td>
<td>New York-Kennedy, NY (JFK)</td>
<td>JFK</td>
<td></td>
</tr>
</tbody>
</table>

**Current Sort:**

- **Travelocity Picks**

<table>
<thead>
<tr>
<th>Hotel Name</th>
<th>Rating</th>
<th>Price per person</th>
<th>Total Trip</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sol Cabanas Del Caribe</td>
<td>⭐⭐⭐⭐</td>
<td>$1035</td>
<td>$2070</td>
</tr>
<tr>
<td>Camino Real Cancun</td>
<td>⭐⭐⭐⭐⭐</td>
<td>$1731</td>
<td>$3462</td>
</tr>
</tbody>
</table>

**Price Includes:**
- 7-night stay in 1 Deluxe Room (other room types), 2 round-trip tickets, and all taxes.
Even here no result, yet…

...the system is anyway able to return “good” answers!
Top-k queries: the naïve approach (1)

- There is a straightforward way to compute the result of any top-k query.
- Assume that, given a query q, there is a scoring function $S$ that assigns to each tuple t a numerical score, according to which tuples are ranked.
  - E.g. $S(t) = t.\text{Points} + t.\text{Rebounds}$

<table>
<thead>
<tr>
<th>Name</th>
<th>Points</th>
<th>Rebounds</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shaquille O'Neal</td>
<td>1669</td>
<td>760</td>
<td>...</td>
</tr>
<tr>
<td>Tracy McGrady</td>
<td>2003</td>
<td>484</td>
<td>...</td>
</tr>
<tr>
<td>Kobe Bryant</td>
<td>1819</td>
<td>392</td>
<td>...</td>
</tr>
<tr>
<td>Yao Ming</td>
<td>1465</td>
<td>669</td>
<td>...</td>
</tr>
<tr>
<td>Dwyane Wade</td>
<td>1854</td>
<td>397</td>
<td>...</td>
</tr>
<tr>
<td>Steve Nash</td>
<td>1165</td>
<td>249</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

ALGORITHM Top-k-naïve

Input: a query q, a dataset R

Output: the k highest-scored tuples with respect to S

1. for all tuples t in R: compute $S(t)$;                      // $S(t)$ is the “score” of t
2. sort tuples based on their scores;
3. return the first k highest-scored tuples;
4. end.
Top-k queries: the naïve approach (2)

- Processing top-k queries using the naïve algorithm is very expensive for large databases, as it requires sorting a large amount of data.

- The problem is even worse if the input consists of more than one relation:

  \[ S(t) = t.\text{Points} + t.\text{Rebounds} \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Points</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shaquille O'Neal</td>
<td>1669</td>
<td>…</td>
</tr>
<tr>
<td>Tracy McGrady</td>
<td>2003</td>
<td>…</td>
</tr>
<tr>
<td>Kobe Bryant</td>
<td>1819</td>
<td>…</td>
</tr>
<tr>
<td>Yao Ming</td>
<td>1465</td>
<td>…</td>
</tr>
<tr>
<td>Dwyane Wade</td>
<td>1854</td>
<td>…</td>
</tr>
<tr>
<td>Steve Nash</td>
<td>1165</td>
<td>…</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Rebounds</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shaquille O'Neal</td>
<td>760</td>
<td>…</td>
</tr>
<tr>
<td>Tracy McGrady</td>
<td>484</td>
<td>…</td>
</tr>
<tr>
<td>Kobe Bryant</td>
<td>392</td>
<td>…</td>
</tr>
<tr>
<td>Yao Ming</td>
<td>669</td>
<td>…</td>
</tr>
<tr>
<td>Dwyane Wade</td>
<td>397</td>
<td>…</td>
</tr>
<tr>
<td>Steve Nash</td>
<td>249</td>
<td>…</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

- Now we have first to join all tuples, which is also a costly operation.

- Note that in the above example the join is 1-1, but in general it can be M-N (each tuple can join with an arbitrary number of tuples).
Expressing a top-k query in SQL requires the capability of:

1) **Ordering** the tuples according to their scores
2) **Limiting** the output cardinality to k tuples

We first consider the case in which the following template query written in *standard SQL* is used, in which only point 1) above is present:

```sql
SELECT <some attributes>
FROM R
WHERE <Boolean conditions>
ORDER BY S(...) [DESC]
```
Limits of the ORDER BY solution

Consider the following queries:

A) SELECT * 
   FROM UsedCarsTable 
   WHERE Vehicle = 'Audi/A4' AND Price <= 21000 
   ORDER BY 0.8*Price + 0.2*Mileage

B) SELECT * 
   FROM UsedCarsTable 
   WHERE Vehicle = 'Audi/A4' 
   ORDER BY 0.8*Price + 0.2*Mileage

- The values 0.8 and 0.2, also called “weights”, are a way to normalize our preferences on Price and Mileage

- Query A will likely lose some relevant answers! (near-miss)
  - e.g., a car with a price of $21,500 but very low mileage

- Query B will return as result all Audi/A4 in the DB! (information overload)
  - …and the situation is horrible if we don’t specify a vehicle type!!
Before considering other solutions, let’s take a closer look at how the DBMS
server sends the result of a query to the client application.

On the client side we work “1 tuple at a time” by using, e.g., `rs.next()`

However this does not mean that a result set is shipped (transmitted) 1 tuple at a time from the server to the client.

Most (all?) DBMSs implement a feature known as row blocking, aiming at reducing the transmission overhead.

**Row blocking:**

1. The DBMS allocates some buffers (a “block”) on the server side.
2. It fills the buffers with tuples of the query result.
3. It ships the whole block of tuples to the client.
4. The client consumes (reads) the tuples in the block.
5. Repeat from 2 until no more tuples (rows) are in the result set.

<table>
<thead>
<tr>
<th>OBJ</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>t07</td>
<td>10</td>
</tr>
<tr>
<td>t24</td>
<td>20</td>
</tr>
<tr>
<td>t16</td>
<td>32</td>
</tr>
</tbody>
</table>
Why row blocking is not enough? I.e.: why do we need “k”?  
- Rationale: just fetch the tuples you need

E.g.: In DB2 the block size is established when the application connects to the DB (default size: 32 KB)

If the buffers can hold, say, 1000 tuples but the application just looks at the first, say, 10, we waste resources:
- We fetch from disk and process too many (1000) tuples
- We transmit too many data (1000 tuples) over the network

If we reduce the block size, then we might incur a large transmission overhead for queries with large result sets
- Bear in mind that we don’t have “just one query”: our application might consist of a mix of queries, each one with its own requirements

Also observe that the DBMS “knows nothing” about the client’s intention, i.e., it will optimize and evaluate the query so as to deliver the whole result set (more on this later)
Top-k queries in SQL (extended)

- The first step to support top-k queries is simple: *extend SQL with a new clause that explicitly limits the cardinality of the result:*

```sql
SELECT <some attributes>
FROM <some relation(s)>
WHERE <Boolean conditions>
[GROUP BY <some grouping attributes>]
ORDER BY S(...) [DESC]
STOP AFTER k
```

where k is a positive integer

- This is the syntax proposed in [CK97], most DBMSs have proprietary (equivalent) extensions, e.g.:
  - `FETCH FIRST k ROWS ONLY` o `LIMIT k` (DB2), `LIMIT TO k ROWS` (ORACLE),...
  - [CK97] also allows a numerical expression, uncorrelated with the rest of the query, in place of k
Semantics of top-k queries

- Consider a top-k query with the clause `STOP AFTER k`
- Conceptually, the rest of the query is evaluated as usual, leading to a table T
- Then, only the first k tuples of T become part of the result
- If T contains at most k tuples, `STOP AFTER k` has no effect
- If more than one set of tuples satisfies the ORDER BY directive, any of such sets is a valid answer (non-deterministic semantics)

```sql
SELECT * FROM R
ORDER BY Price
STOP AFTER 3
```

```
<table>
<thead>
<tr>
<th>OBJ</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>t15</td>
<td>50</td>
</tr>
<tr>
<td>t24</td>
<td>40</td>
</tr>
<tr>
<td>t26</td>
<td>30</td>
</tr>
<tr>
<td>t14</td>
<td>30</td>
</tr>
<tr>
<td>t21</td>
<td>40</td>
</tr>
</tbody>
</table>
```

Both are valid results

- If no ORDER BY clause is present, then any set of k tuples from T is a valid (correct) answer
Top-k queries: examples (1)

- The best NBA player (considering points and rebounds):
  
  ```sql
  SELECT *
  FROM NBA
  ORDER BY Points + Rebounds DESC
  STOP AFTER 1
  ```

- The 2 cheapest chinese restaurants
  
  ```sql
  SELECT *
  FROM RESTAURANTS
  WHERE Cuisine = 'chinese'
  ORDER BY Price
  STOP AFTER 2
  ```

- The top-5% highest paid employees
  
  ```sql
  SELECT E.*
  -- a top-k query with a numerical expression
  FROM EMP E
  ORDER BY E.Salary DESC
  STOP AFTER (SELECT COUNT(*)/20 FROM EMP)
  ```
Top-k queries: examples (2)

- The top-5 Audi/A4 (based on price and mileage)

  ```sql
  SELECT *  
  FROM USEDCARS  
  WHERE Vehicle = 'Audi/A4'  
  ORDER BY 0.8*Price + 0.2*Mileage 
  STOP AFTER 5 
  ```

- The 2 hotels closest to the Bologna airport

  ```sql
  SELECT H.*  
  -- a top-k distance join query  
  FROM HOTELS H, AIRPORTS A  
  WHERE A.Code = 'BLQ'  
  ORDER BY distance(H.Location,A.Location)  
  STOP AFTER 2 
  ```

Location is a “point” UDT (User-defined Data Type)
distance is a UDF (User-Defined Function)
Evaluation of top-k queries

Concerning evaluation, there are two basic aspects to consider:
- query type: 1 relation, many relations, aggregate results, ...
- access paths: no index, indexes on all/some ranking attributes

The simplest case to analyze is the top-k selection query, where only 1 relation is involved:

```
SELECT <some attributes>
FROM R
WHERE <Boolean conditions>
ORDER BY S(...) [DESC]
STOP AFTER k
```
Top-k queries: algebraic representation

- In order to concisely reason on alternative evaluation strategies, we have first to extend the relational algebra (RA)
- To this end, we introduce a logical Top operator, denoted $τ_{k,S}$, which returns the k top-ranked tuples according to S
  - Unless otherwise specified, we assume that S has to be maximized

Later we will introduce a more powerful representation in which ranking (not just “limiting”) is a “first-class citizen”
Implementing Top: physical operators

How can the Top operator be evaluated?

2 relevant cases:

Top-Scan: the stream of tuples entering the Top operator is already sorted according to S: in this case it is sufficient to just read (consume) the first k tuples from the input.

- Top-Scan can work in pipeline: it can return a tuple as soon as it reads it!

Top-Sort: the input stream is not S-ordered; if k is not too large (which is the typical case), rather than sorting the whole input we can perform an in-memory sort.

- Top-Sort cannot work in pipeline: it has to read the whole input before returning the first tuple!
The Top-Sort physical operator: open method

- The idea of the Top-Sort method is to maintain in a main-memory buffer B only the best k tuples seen so far
  
  **RATIONALE**: if tuple t is not among the top-k tuples seen so far, then t cannot be part of the result

- A crucial issue is how to organize B so that the operations of lookup, insertion and removal can be performed efficiently

- Since B should act as a **priority queue** (the priority is given by the score), it can be implemented using a **heap**

**Method open**

**Input:** k, S

1. create a priority queue B of size k; // B can hold at most k tuples
   // B[i] is the current i-th best tuple, and B[i].score is its score
2. invoke open on the child node;
3. return.
The Top-Sort physical operator: next method

- The next method initially fills B with the first k tuples
  - For simplicity, the pseudocode does not consider the case when the input has less than k tuples
- Then, for each new read tuple t, it compares t with B[k], the worst tuple currently in B
  - If $S(t) > B[k].score$, then B[k] is dropped and t is inserted into B
  - If $S(t) < B[k].score$, t cannot be one of the top-k tuples
  - If $S(t) = B[k].score$, it is safe to discard t since Top has a non-deterministic semantics!

**Method next**

1. for $i=1$ to $k$: // fills B with the first k tuples
2. t := input_node.next(); ENQUEUE(B, t); // inserts t in B
3. while (input_node.has_next()) do:
4. t := input_node.next();
4. if $S(t) > B[k].score$ then: {DELETE(B, B[k]); ENQUEUE(B, t)};
5. return DEQUEUE(B). // returns the best tuple in B
Top-Sort: a simple example

- Let $k = 2$

**EMP**

<table>
<thead>
<tr>
<th>ENO</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>1000</td>
</tr>
<tr>
<td>E2</td>
<td>1200</td>
</tr>
<tr>
<td>E3</td>
<td>1400</td>
</tr>
<tr>
<td>E4</td>
<td>1100</td>
</tr>
<tr>
<td>E5</td>
<td>1500</td>
</tr>
</tbody>
</table>

**B**

<table>
<thead>
<tr>
<th>Order</th>
<th>ENO</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>E1</td>
<td>1000</td>
</tr>
<tr>
<td>1</td>
<td>E2</td>
<td>1200</td>
</tr>
</tbody>
</table>

**EMP**

<table>
<thead>
<tr>
<th>ENO</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>E3</td>
<td>1400</td>
</tr>
<tr>
<td>E2</td>
<td>1200</td>
</tr>
</tbody>
</table>

**B**

<table>
<thead>
<tr>
<th>Order</th>
<th>ENO</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>E3</td>
<td>1400</td>
</tr>
<tr>
<td>2</td>
<td>E2</td>
<td>1200</td>
</tr>
</tbody>
</table>

**EMP**

<table>
<thead>
<tr>
<th>ENO</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>E5</td>
<td>1500</td>
</tr>
</tbody>
</table>

**B**

<table>
<thead>
<tr>
<th>Order</th>
<th>ENO</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>E3</td>
<td>1400</td>
</tr>
<tr>
<td>1</td>
<td>E5</td>
<td>1500</td>
</tr>
</tbody>
</table>
Experimental results from [CK97] (1)

```
SELECT E.* FROM EMP E
ORDER BY E.Salary DESC
STOP AFTER N;
```

- TRADITIONAL = row blocking (about 500 tuples)
- TRAD(NRB) = no row blocking

In-memory sort

No Index available

The naïve method sorts ALL the tuples!

Figure 6: Resp. Time (log), Q1
No Index on Emp.salary.
Results from [CK97] (2)

SELECT E.* FROM EMP E
ORDER BY E.Salary DESC
STOP AFTER N;

If the DBMS ignores that we just need k tuples, it will not use the index: it will scan the EMP table and then sort ALL the N tuples!

Both TRADITIONAL and TRAD(NRB) still scan and sort the whole table.
SELECT E.* FROM EMP E
ORDER BY E.Salary DESC
STOP AFTER N;

Row blocking:
poor performance for small N values

No row blocking:
poor performance for large N values

Figure 4: Resp. Time (log), Q1
Clustered Index on Emp.salary.
Multi-dimensional top-k queries

- In the general case, the scoring function $S$ involves more than one attribute:
  
  ```sql
  SELECT *
  FROM USEDCARS
  WHERE Vehicle = 'Audi/A4'
  ORDER BY 0.8*Price + 0.2*Mileage
  STOP AFTER 5;
  ```

- If no index is available, we cannot do better than apply a Top-Sort operator by sequentially reading ALL the tuples 😞

- If an index is available on Vehicle the situation is better, yet it depends on how many Audi/A4 are in the DB 😐
  - Back to the 1st case if the WHERE clause is not present at all 😈

- Assume we have an index on the ranking attributes (i.e., Price and Mileage)
  - How can we use it to solve a top-k query?
  - What kind of index should we use?

- We first need to better understand the underlying geometry of the problem...
The “attribute space”: a geometric view

- Consider the 2-dimensional (2-dim) *attribute space* (Price,Mileage)

- Each tuple is represented by a 2-dim point \((p,m)\):
  - \(p\) is the *Price* value
  - \(m\) is the *Mileage* value

- Intuitively, minimizing
  \[ 0.8 \times \text{Price} + 0.2 \times \text{Mileage} \]
  is equivalent to look for points “close” to \((0,0)\)

- \((0,0)\) is our (ideal) “target value” (i.e., a free car with 0 km’s!)
The role of weights (preferences)

- Our preferences (e.g., 0.8 and 0.2) are essential to determine the result.

With preferences (0.8, 0.2) the best car is C6, then C5, etc.

In general, preferences are a way to determine, given points \((p_1, m_1)\) and \((p_2, m_2)\), which of them is “closer” to the target point \((0,0)\).

Consider the line \(l(v)\) of equation

\[
0.8 \times \text{Price} + 0.2 \times \text{Mileage} = v
\]

where \(v\) is a constant.

This can also be written as

\[
\text{Mileage} = -4 \times \text{Price} + 5 \times v
\]

from which we see that all the lines \(l(v)\) have a slope = -4.

By definition, all the points of \(l(v)\) are “equally good” to us.
Changing the weights

- Clearly, changing the weight values will likely lead to a different result.

- With $0.8 \cdot \text{Price} + 0.2 \cdot \text{Mileage}$, the best car is C6.

- With $0.5 \cdot \text{Price} + 0.5 \cdot \text{Mileage}$, the best cars are C5 and C11.

- On the other hand, if weights do not change too much, the results of two top-k queries will likely have a high degree of overlap.
Changing the target

- The target of a query is not necessarily (0,0), rather it can be any point \( q = (q_1, q_2) \) (\( q_i \) = query value for the \( i \)-th attribute)
- Example: assume you are looking for a house with a 1000 m\(^2\) garden and 3 bedrooms; then \( (1000, 3) \) is the target for your query

In general, in order to determine the “goodness” of a tuple \( t \), we compute its “distance” from the target point \( q \): The lower the distance from \( q \), the better \( t \) is

Note that distance values can always be converted into goodness “scores”, so that a higher score means a better match. Just change the sign and possibly add a constant,…
Top-k tuples = k-nearest neighbors

- In order to provide a homogeneous management of the problem when using an index, it is useful to consider distances rather than scores
  - since most indexes are “distance-based”

Therefore, the model is now:
- A D-dimensional (D≥1) attribute space \( A = (A_1,A_2,...,A_D) \) of ranking attributes
- A relation \( R(A_1,A_2,...,A_D,B_1,B_2,...) \), where \( B_1,B_2,... \) are other attributes
- A target (query) point \( q = (q_1,q_2,...,q_D) \), \( q \in A \)
- A function \( d: A \times A \rightarrow \mathbb{R} \), that measures the distance between points of \( A \) (e.g., \( d(t,q) \) is the distance between \( t \) and \( q \))

Under this model, a top-k query is transformed into a so-called

**k-Nearest Neighbors (k-NN) Query**
- **Given** a point \( q \), a relation \( R \), an integer \( k \geq 1 \), and a distance function \( d \)
- **Determine** the \( k \) tuples in \( R \) that are closest to \( q \) according to \( d \)
Some common distance functions

- The most commonly used distance functions are $L_p$-norms:

$$L_p(t, q) = \left( \sum_{i=1}^{D} |t_i - q_i|^p \right)^{1/p}$$

- Relevant cases are:

  - $L_2(t, q) = \sqrt{\sum_{i=1}^{D} |t_i - q_i|^2}$ Euclidean distance
  - $L_1(t, q) = \sum_{i=1}^{D} |t_i - q_i|$ Manhattan (city – block) distance
  - $L_{\infty}(t, q) = \max_i \{|t_i - q_i|\}$ Chebyshev (max) distance

Iso-distance (hyper-)surfaces
Shaping the attribute space

- Changing the distance function leads to a different shaping of the attribute space (each colored “stripe” in the figures corresponds to points with distance values between $v$ and $v+1$, $v$ integer)

$L_1; q=(7,12)$

$L_2; q=(7,12)$

Note that, for 2 tuples $t_1$ and $t_2$, it is possible to have $L_1(t_1,q) < L_1(t_2,q)$ and $L_2(t_2,q) < L_2(t_1,q)$

E.g.: $t_1=(13,12)$, $t_2=(12,10)$
Distance functions with weights

The use of weights just leads to “stretch” some of the coordinates:

\[ L_2(t,q; W) = \sqrt{\sum_{i=1}^{D} w_i |t_i - q_i|^2} \]

\[ L_1(t,q; W) = \sum_{i=1}^{D} w_i |t_i - q_i| \]

\[ L_\infty (t,q; W) = \max_i \{w_i |t_i - q_i|\} \]

Thus, the scoring function

\[ 0.8 \times \text{Price} + 0.2 \times \text{Mileage} \]

is just a particular case of weighted L1 distance.
The figures show the effects of using L1 with different weights.

- **L1; q=(7,12) W=(1,1)**

- **L1; q=(7,12) W=(0.6,1.4)**

Note that, if $w_2 > w_1$, then the hyper-romboids are more elongated along A1 (i.e., difference on A1 values is less important than an equal difference on A2 values).
Processing top-k queries with indexes

- Using a multi-attribute B+-tree, that organizes the tuples according to the order A1,A2,...,AD (e.g., first on Price, then on Mileage) is not going to perform well
  - Same problems as with window queries (poor spatial clustering)
  - It is however possible to use D single-attribute B+-trees, which we will cover when dealing with top-k join queries

- Much better is to consider a spatial index, like the R-tree...
**R-tree basic properties**

- The R-tree is a dynamic, height-balanced, and paged tree
- Each node stores a variable number of *entries*

**Leaf node:**
- An entry \( E = (\text{tuple-key}, \text{RID}) \), where tuple-key is the “spatial key” (position) of the tuple whose address is RID

**Internal node:**
- An entry \( E = (\text{MBR}, \text{PID}) \), where MBR (also: MBB) is the “Minimum Bounding Rectangle” (with sides parallel to the coordinate axes) of all the points reachable from (“under”) the child node whose address is PID

- We can uniform things by saying that each entry has the format \( E = (\text{key}, \text{ptr}) \)
- If N is the node pointed by E.ptr, then E.key is the “spatial key” of N, also denoted as \( \text{Reg}(N) \)
Search: range query

- We start with a query type simpler than k-NN queries, namely the

  **Range Query**
  - Given a point \( q \), a relation \( R \), a search radius \( r \geq 0 \), and a distance function \( d \),
  - Determine all the objects \( t \) in \( R \) such that \( d(t,q) \leq r \)

- The region of \( \mathcal{R}^D \) defined as \( \text{Reg}(q) = \{ p: p \in \mathcal{R}^D, d(p,q) \leq r \} \) is also called the query region (thus, the result is always contained in the query region)
  - For simplicity, both \( d \) and \( r \) are understood in the notation \( \text{Reg}(q) \)

- There are several variants of range queries, such as point queries (\( r = 0 \), look for a perfect match) and window queries (a special case of range queries obtained when the distance function is a weighted \( L_\infty \))

- The algorithm for processing a range query is extremely simple:
  - Start from the root and, for each entry \( E \) and corresponding node \( N \), check if \( \text{Reg}(N) \) intersects \( \text{Reg}(q) \)
  - On leaf nodes, check for each entry \( E \) if \( E.\text{key} \in \text{Reg}(q) \), that is, if \( d(E.\text{key},q) \leq r \).
Search: k-NN query (1)

- For a node N, let $d_{\text{MIN}}(q, \text{Reg}(N)) = \inf_p \{d(q, p) \mid p \in \text{Reg}(N)\}$ be the minimum possible distance between q and a point in Reg(N)

  - The “MinDist” $d_{\text{MIN}}(q, \text{Reg}(N))$ is a lower bound on the distances from q to any indexed point reachable from N

- We can make the following basic observation:
  \[
  \text{Reg}(q) \cap \text{Reg}(N) \neq \emptyset \iff d_{\text{MIN}}(q, \text{Reg}(N)) \leq r
  \]
We now present an algorithm, called kNNOptimal [BBK+97], for solving k-NN queries with an R-tree that is I/O-optimal.

- The algorithm also applies to many other index structures (e.g., the M-tree).
- We start with the basic case $k=1$.
- For a given query point $q$, let $t_{NN}(q)$ be the 1st nearest neighbor ($1$-NN = NN) of $q$ in $R$, and denote with $r_{NN} = d(q, t_{NN}(q))$ its distance from $q$.
  - Clearly, $r_{NN}$ is only known when the algorithm terminates.

**Theorem:**

Any correct algorithm for 1-NN queries must visit at least all the nodes $N$ whose MinDist is strictly less than $r_{NN}$, i.e., $d_{MIN}(q, Reg(N)) < r_{NN}$.

**Proof:** Assume that an algorithm $A$ stops by reporting as NN of $q$ a point $t$, and that $A$ does not read a node $N$ such that (s.t.) $d_{MIN}(q, Reg(N)) < d(q, t)$; then $Reg(N)$ might contain a point $t'$ s.t. $d(q, t') < d(q, t)$, thus contradicting the hypothesis that $t$ is the NN of $q$. 


The logic of the kNNOptimal Algorithm

- The kNNOptimal algorithm uses a priority queue PQ, whose elements are pairs \([\text{ptr}(N), d_{\text{MIN}}(q, \text{Reg}(N))]\).
- PQ is ordered by *increasing values of* \(d_{\text{MIN}}(q, \text{Reg}(N))\)
  - DEQUEUE(PQ) extracts from PQ the pair with minimal MinDist
  - ENQUEUE(PQ, \([\text{ptr}(N), d_{\text{MIN}}(q, \text{Reg}(N))]\)) performs an ordered insertion of the pair in the queue
- Pruning of the nodes is based on the following observation:
  - If, at a certain point of the execution of the algorithm, we have found a point \(t\) s.t. \(d(q, t) = r\),
  - Then, all the nodes \(N\) with \(d_{\text{MIN}}(q, \text{Reg}(N)) \geq r\) can be excluded from the search, since they cannot lead to an improvement of the result

- In the description of the algorithm, the pruning of pairs of PQ based on the above criterion is concisely denoted as UPDATE(PQ)
- With a slight abuse of terminology, we also say that “the node \(N\) is in PQ” meaning that the corresponding pair \([\text{ptr}(N), d_{\text{MIN}}(q, \text{Reg}(N))]\) is in PQ
- Intuitively, kNNOptimal performs a “range search with a variable (shrinking) search radius” until no improvement is possible anymore
The kNNOptimal Algorithm (case k=1)

**Input:** query point q, index tree with root node RN

**Output:** \( t_{NN}(q) \), the nearest neighbor of q, and \( r_{NN} = d(q, t_{NN}(q)) \)

1. Initialize PQ with [ptr(RN),0]; // starts from the root node
2. \( r_{NN} := \infty \); // this is the initial “search radius”
3. while PQ \( \neq \emptyset \): // until the queue is not empty...
4. \([\text{ptr}(N), d_{MIN}(q,\text{Reg}(N))] := \text{DEQUEUE}(PQ); \) // … get the closest pair...
5. \( \text{Read}(N); \) // … and reads the node
6. if N is a leaf then: for each point t in N:
   7. if \( d(q,t) < r_{NN} \) then: \( t_{NN}(q) := t; r_{NN} := d(q,t); \text{UPDATE}(PQ) \) // reduces the search radius and prunes nodes
8. else: for each child node Nc of N:
   9. if \( d_{MIN}(q,\text{Reg}(Nc)) < r_{NN} \) then:
10. \( \text{ENQUEUE}(PQ,[\text{ptr}(Nc), d_{MIN}(q,\text{Reg}(Nc))]); \)
11. return \( t_{NN}(q) \) and \( r_{NN} \);
12. end.
kNNOptimal in action

- Nodes are numbered following the order in which they are accessed.
- Objects are numbered as they are found to improve (reduce) the search radius.
- The accessed leaf nodes are shown in red.
kNNOptimal: The best used car

\[ d = 0.7 \times \text{Price} + 0.3 \times \text{Mileage} \]

Node | \( d_{\text{MIN}} \)  
--- | ---  
N1 | 16.4  
N2 | 19  
N3 | 16.4  
N4 | 22.9  
N5 | 19  
N6 | 26  

tuple | d  
--- | ---  
C5 | 20  
C6 | 19  
C11 | 24  
... | ...  

<table>
<thead>
<tr>
<th>Action</th>
<th>PQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(RN)</td>
<td>(N1,16.4)</td>
</tr>
</tbody>
</table>
| Read(N1)    | (N3,16.4)     | (N2,19)  
| Read(N3)    | (N2,19)       | (N4,22.9)  
| Read(N2)    | (N5,19)       |  
| Read(N5)    |               |  
| Return(C6,19)|              |  

Price vs Mileage graph with nodes and tuples.
Correctness and optimality of kNNOptimal

- The kNNOptimal algorithm is clearly correct
- To show that it is also I/O-optimal, that is, it reads the minimum number of nodes, it is sufficient to prove the following

**Theorem:**
The kNNOptimal algorithm for 1-NN queries never reads a node N whose MinDist is strictly larger than \( r_{\text{NN}} \), i.e., \( d_{\text{MIN}}(q, \text{Reg}(N)) > r_{\text{NN}} \)

**Proof:**
- Node N is read only if, at some execution step, it becomes the 1st element in PQ
- Let N1 be the node containing \( t_{\text{NN}}(q) \), N2 its parent node, N3 the parent node of N2, and so on, up to Nh = RN (h = height of the tree)
- Now observe that, by definition of MinDist, it is:
  \[ r_{\text{NN}} \geq d_{\text{MIN}}(q, \text{Reg}(N1)) \geq d_{\text{MIN}}(q, \text{Reg}(N2)) \geq \ldots \geq d_{\text{MIN}}(q, \text{Reg}(Nh)) \]
- At each time step before we find \( t_{\text{NN}}(q) \), one (and only one) of the nodes N1, N2, ..., Nh is in the priority queue
- It follows that N can never become the 1st element of PQ
What if $d_{\text{MIN}}(q, \text{Reg}(N)) = r_{\text{NN}}$?

Q: The optimality theorem says nothing about regions whose MinDist equals the NN distance. Why?

A: Because it cannot!

Note that all such regions tie, thus their relative ordering in PQ is undetermined. The possible cases are:

1. The NN is in a node whose region has MinDist < $r_{\text{NN}}$. In this case no node with $d_{\text{MIN}}(q, \text{Reg}(N)) = r_{\text{NN}}$ will be read.

2. The NN is in a node whose region has exactly MinDist = $r_{\text{NN}}$. Now everything depends on how ties are managed in PQ. In the worst case, all nodes with $d_{\text{MIN}}(q, \text{Reg}(N)) = r_{\text{NN}}$ will be read.
The general case ($k \geq 1$)

- The algorithm is easily extended to the case $k \geq 1$ by using:
  
  - a data structure, which we call Res, where we maintain the $k$ closest objects found so far, together with their distances from $q$
  - as “current search radius” the distance, $r_{k-NN}$, of the current $k$-th NN of $q$, that is, the $k$-th element of Res

<table>
<thead>
<tr>
<th>ObjectID</th>
<th>distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>t15</td>
<td>4</td>
</tr>
<tr>
<td>t24</td>
<td>8</td>
</tr>
<tr>
<td>t18</td>
<td>9</td>
</tr>
<tr>
<td>t4</td>
<td>12</td>
</tr>
<tr>
<td>t2</td>
<td>15</td>
</tr>
</tbody>
</table>

- The rest of the algorithm remains unchanged

$k = 5$

- No node with distance $\geq 15$ needs to be read
**The kNNOptimal Algorithm (case k ≥ 1)**

**Input:** query point q, integer k ≥ 1, index tree with root node RN

**Output:** the k nearest neighbors of q, together with their distances

1. Initialize PQ with [ptr(RN),0];
2. for i=1 to k: Res[i] := [null,∞]; r_{k-NN} := Res[k].dist;
3. while PQ ≠ Ø:
   4. [ptr(N), d_{MIN}(q,Reg(N))] := DEQUEUE(PQ);
   5. Read(N);
   6. if N is a leaf then: for each point t in N:
      7. if d(q,t) < r_{k-NN} then: { remove the element in ResultList[k];
         8. insert [t,d(q,t)] in ResultList;
         9. r_{k-NN} := Res[k].dist; UPDATE(PQ)}
   10. else: for each child node Nc of N:
      11. if d_{MIN}(q,Reg(Nc)) < r_{k-NN} then:
      12. ENQUEUE(PQ,[ptr(Nc), d_{MIN}(q,Reg(Nc))]);
13. return Res;
14. end.
Distance browsing

- Now we know how to solve top-k selection queries using a multi-dimensional index; but, what if our query is

```sql
SELECT *
FROM USED CARS
WHERE Vehicle = 'Audi/A4'
ORDER BY 0.8*Price + 0.2*Mileage
STOP AFTER 5;
```

and we have an R-tree on (Price,Mileage) built over ALL the cars?

- The k = 5 best matches returned by the index will not necessarily be Audi/A4

- In this case we can use a variant of kNNoptimal, which supports the so-called “distance browsing” [HS99] or “incremental NN queries”

- For the case k = 1 the overall logic for using the index is:
  - get from the index the 1st NN
  - if it satisfies the query conditions (e.g., AUDI/A4) then stop,
    otherwise get the next (2nd) NN and do the same
  - until 1 object is found that satisfies the query conditions
The next\_NN algorithm

- In the queue \( PQ \) now we keep both tuples and nodes
  - If an entry of \( PQ \) is tuple \( t \) then its distance \( d(q,t) \) is written \( d_{\text{MIN}}(q,\text{Reg}(t)) \)
- Note that no pruning is possible (since we don’t know how many objects have to be returned before stopping)
- Before making the first call to the algorithm we initialize \( PQ \) with \([\text{ptr}(\text{RN}),0]\)
- When a tuple \( t \) becomes the 1st element of the queue the algorithm returns

**Input**: query point \( q \), index tree with root node \( \text{RN} \)

**Output**: the next nearest neighbor of \( q \), together with its distance

1. while \( PQ \neq \emptyset \):
2. \([\text{ptr}(\text{Elem}), d_{\text{MIN}}(q,\text{Reg}(\text{Elem}))]\) := DEQUEUE(PQ);
3. if \( \text{Elem} \) is a tuple \( t \) then: return \( t \) and its distance // no tuple can be better than \( t \)
4. else: if \( \text{N} \) is a leaf then: for each point \( t \) in \( \text{N} \): ENQUEUE(PQ,[\( t, d(q,t) \)])
5. else: for each child node \( \text{N}_c \) of \( \text{N} \):
6. \quad ENQUEUE(PQ,[\text{ptr}(\text{N}_c), d_{\text{MIN}}(q,\text{Reg}(\text{N}_c))]);
7. end.
Distance browsing: An example (1/2)

- q=(5,5), distance: L2

<table>
<thead>
<tr>
<th>Elem</th>
<th>(d_{MIN})</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>(\sqrt{5})</td>
</tr>
<tr>
<td>B</td>
<td>(\sqrt{13})</td>
</tr>
<tr>
<td>C</td>
<td>(\sqrt{18})</td>
</tr>
<tr>
<td>D</td>
<td>(\sqrt{13})</td>
</tr>
<tr>
<td>E</td>
<td>(\sqrt{13})</td>
</tr>
<tr>
<td>F</td>
<td>(\sqrt{10})</td>
</tr>
<tr>
<td>G</td>
<td>(\sqrt{13})</td>
</tr>
<tr>
<td>H</td>
<td>(\sqrt{2})</td>
</tr>
<tr>
<td>I</td>
<td>(\sqrt{10})</td>
</tr>
<tr>
<td>J</td>
<td>(\sqrt{10})</td>
</tr>
<tr>
<td>K</td>
<td>(\sqrt{17})</td>
</tr>
<tr>
<td>L</td>
<td>(\sqrt{25})</td>
</tr>
<tr>
<td>M</td>
<td>(\sqrt{20})</td>
</tr>
<tr>
<td>N1</td>
<td>(\sqrt{1})</td>
</tr>
<tr>
<td>N2</td>
<td>(\sqrt{2})</td>
</tr>
<tr>
<td>N3</td>
<td>(\sqrt{5})</td>
</tr>
<tr>
<td>N4</td>
<td>(\sqrt{5})</td>
</tr>
<tr>
<td>N5</td>
<td>(\sqrt{9})</td>
</tr>
<tr>
<td>N6</td>
<td>(\sqrt{2})</td>
</tr>
<tr>
<td>N7</td>
<td>(\sqrt{13})</td>
</tr>
</tbody>
</table>
Distance browsing: An example (2/2)

<table>
<thead>
<tr>
<th>Action</th>
<th>PQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(RN)</td>
<td>(N1, √1) (N2, √2)</td>
</tr>
<tr>
<td>Read(N1)</td>
<td>(N2, √2) (N3, √5) (N4, √5) (N5, √9)</td>
</tr>
<tr>
<td>Read(N2)</td>
<td>(N6, √2) (N3, √5) (N4, √5) (N5, √9) (N7, √13)</td>
</tr>
<tr>
<td>Read(N6)</td>
<td>(H, √2) (N3, √5) (N4, √5) (N5, √9) (I, √10) (G, √13) (N7, √13)</td>
</tr>
<tr>
<td>Return(H, √2)</td>
<td>(N3, √5) (N4, √5) (N5, √9) (I, √10) (G, √13) (N7, √13)</td>
</tr>
<tr>
<td>Read(N3)</td>
<td>(N4, √5) (N5, √9) (I, √10) (F, √10) (G, √13) (D, √13) (E, √13) (N7, √13)</td>
</tr>
<tr>
<td>Read(N4)</td>
<td>(A, √5) (N5, √9) (I, √10) (F, √10) (G, √13) (D, √13) (E, √13) (B, √13) (N7, √13) (C, √18)</td>
</tr>
<tr>
<td>Return(A, √5)</td>
<td>(N5, √9) (I, √10) (F, √10) (G, √13) (D, √13) (E, √13) (B, √13) (C, √18)</td>
</tr>
<tr>
<td>Read(N5)</td>
<td>... ... ...</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Indexes as iterators

- next_NN is just an implementation of the general next method for indexes that support incremental k-NN queries
- In practice, the specific query type (range, k-NN, incremental k-NN, etc.) is a parameter passed to the index with the `open` method, after that a simple `next()` suffices
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.:

$$
\text{SELECT} \quad \langle\text{some attributes}\rangle \\
\text{FROM} \quad R_1, R_2, \ldots, R_n \\
\text{WHERE} \quad \langle\text{join and local conditions}\rangle \\
\text{ORDER BY} \quad S(p_1, p_2, \ldots p_m) \ [\text{DESC}] \\
\text{STOP AFTER} \quad k
$$

where $p_1, p_2, \ldots p_m$ are scoring criteria (the "preferences")
Top-k join queries: examples

- The highest paid employee (compared to her dept budget):

  ```sql
  SELECT E.*
  FROM EMP E, DEPT D
  WHERE E.DNO = D.DNO
  ORDER BY E.Salary / D.Budget DESC
  STOP AFTER 1
  ```

- The 2 cheapest restaurant-hotel combinations in the same Italian city

  ```sql
  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
  ```
A (multidimensional) top-k selection query based on the scoring function $S(p_1,p_2,...,p_m)$ can also be viewed as a particular case of join query in which the input relation $R$ is virtually “partitioned” into $m$ parts, where the $j$-th part $R_i$ consists of the object id and of the attributes needed to compute $p_j$.

E.g. If $S$ is defined as:

$$S(t) = \frac{(t\cdot Price + t\cdot Mileage)}{(t\cdot Year - 1970)}$$

we can “partition” $USEDCARS$ as:

$UC1(CarID,Price), UC2(CarID,Mileage), UC3(CarID,Year)$

A top-k selection query would be equivalent to:

```sql
SELECT * 
FROM USEDCARS UC1, USEDCARS UC2, USEDCARS UC3 
STOP AFTER 2
```

In such cases the join is always 1-1 (PK-PK join)

Other partitioned cases can occur, e.g., $UC1(CarID,Price,Mileage), UC2(CarID,Year)$
Top-k 1-1 join queries

- The case in which all the joins are on a common key attribute(s) has been the first to be largely investigated.

- Its relevance is due to the following reasons:
  - It is the simplest case to deal with.
  - Its solution provides the basis for the more general case.
  - It occurs in many practical cases.

- The two scenarios in which 1-1 joins are worth considering are:
  - For each preference $p_j$ there is an index able to retrieve tuples according to that preference.
  - The “partitions” of $R$ are real, in that $R$ is spread over several sites, each providing information only on part of the objects.

- Historically, the 2nd scenario, sometimes called the “middleware scenario”, is the one that motivated the study of top-k (1-1) join queries, and which led to the introduction of the first non-trivial algorithms.
The middleware scenario

- The **middleware scenario** can be roughly described as follows:

  1. We have a number of “data sources”
  2. Our requests (queries) might involve several data sources at a time
  3. The result of our queries is obtained by “combining” in some way the results returned by the data sources

- These queries are collectively called “**middleware queries**”, since they require the presence of a middleware whose role is to act as a “**mediator**” between the user/client and the data sources/servers
A simplified example

- Assume you want to set up a web site that integrates the information of 2 sources:
  - The 1st source “exports” the following schema:
    \[
    \text{CarPrices(CarModel, Price)}
    \]
  - The schema exported by the 2nd source is:
    \[
    \text{CarSpec(Make, Model, FuelConsumption)}
    \]

- After a phase of “reconciliation”
  \[
  \text{CarModel} = \text{‘Audi/A4’} \Leftrightarrow (\text{Make,Model}) = (\text{‘Audi’,‘A4’})
  \]
  we can now support queries on both Price and FuelConsumption, e.g.:
  \[
  \text{find those cars whose consumption is less than 7 litres/100km and with a cost less than 15,000 €}
  \]

How?

1. send the (sub-)query on Price to the CarPrices source,
2. send the query on fuel consumption to the CarSpec source,
3. join the results
The details of query execution

SELECT * FROM MyCars
WHERE Price < 15000
AND FuelConsumption < 7

SELECT * FROM CarPrices
WHERE Price < 15000

SELECT * FROM CarSpec
WHERE FuelConsumption < 7

<table>
<thead>
<tr>
<th>Make</th>
<th>Model</th>
<th>Price</th>
<th>FuelCons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toyota</td>
<td>Yaris</td>
<td>12</td>
<td>6.5</td>
</tr>
<tr>
<td>Nissan</td>
<td>Micra</td>
<td></td>
<td>6.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CarModel</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toyota/Yaris</td>
<td>12</td>
</tr>
<tr>
<td>Citroen/C3</td>
<td>11</td>
</tr>
</tbody>
</table>
Another example

- We now want to build a site that integrates the information of (the sites of) m car dealers:
  - Each car dealer site CDj can give us the following information:
    \[ \text{CarDealer}_j(\text{CarID}, \text{Make}, \text{Model}, \text{Price}) \]
    and our goal is to provide our users with the cheapest available cars, that is, to support queries like:
    
    \[ \text{For each FIAT model, which is the cheapest offer?} \]

How?

1. send the same (sub-)query to the all the data sources,
2. take the union of the results,
3. for each model, get the best offer and the corresponding dealer

For queries of this kind, the mediator is also often called a “meta-broker” or “meta-search engine”
SELECT Model, min(Price) MP, Dealer
FROM AllCars
WHERE Make = 'Fiat'
GROUP BY Model

AllCars(CarID, Make, Model, Price, Dealer)

<table>
<thead>
<tr>
<th>Model</th>
<th>MP</th>
<th>Dealer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brava</td>
<td>8</td>
<td>D1</td>
</tr>
<tr>
<td>Duna</td>
<td>7</td>
<td>D2</td>
</tr>
<tr>
<td>Punto</td>
<td>10</td>
<td>D2</td>
</tr>
</tbody>
</table>

SELECT Model, min(Price) MP
FROM CarDealer1
WHERE Make = 'Fiat'
GROUP BY Model

CarDealer1(CarID, Make, Model, Price)

<table>
<thead>
<tr>
<th>Model</th>
<th>MP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brava</td>
<td>8</td>
</tr>
<tr>
<td>Punto</td>
<td>11</td>
</tr>
</tbody>
</table>

SELECT Model, min(Price) MP
FROM CarDealer2
WHERE Make = 'Fiat'
GROUP BY Model

CarDealer2(CarID, Make, Model, Price)

<table>
<thead>
<tr>
<th>Model</th>
<th>MP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brava</td>
<td>9</td>
</tr>
<tr>
<td>Duna</td>
<td>7</td>
</tr>
<tr>
<td>Punto</td>
<td>10</td>
</tr>
</tbody>
</table>
“Top-k middleware query” is just another name for top-k 1-1 join query, appropriate when the data to be 1-1 joined are distributed.

Although the distributed and local scenario have different properties (e.g., communication costs, availability of sources, etc.), for both one can apply the same principles (and algorithms) to compute the result of a top-k query.

In particular, in both scenarios we reason in terms of “inputs”, which are “data sources” and “relations” in the distributed and local case, respectively.

For reasons that will be soon clear, the j-th input will be denoted $L_j$.

Also, in order to simplify the notation, we reason in terms of “objects” (rather than tuples) to be globally scored.

This is because it is fair to say that an object $o$ belongs to two different inputs, whereas it would be incorrect to say the same for a tuple.
Top-k 1-1 join queries

The following assumptions are quite standard if one has to avoid reading all the inputs:

1) Each input $L_j$ supports a sorted access (s.a.) interface:

\[
\text{getNext}_{L_j}(i) \rightarrow (\text{OID}, \text{Attributes}, p_j)
\]

A sorted access gets the id of the next best object, its partial score $p_j$, and possibly some attributes requested by the query.

Thus, $L_j$ is a ranked list, which justifies its name ("L" stands for list).

2) Each input $L_j$ also supports a random access (r.a.) interface:

\[
\text{getScore}_{L_j}(<\text{OID}> \rightarrow p_j
\]

A random access gets the partial score of an object.
3) The **OID is “global”**: a given object has the same identifier across the inputs

4) Each input consists of the same set of objects

- These two assumptions trivially holds if the top-k 1-1 join query is executed locally, since the lists to be joined are just different rankings of a same relation

- In a distributed environment, 3) rarely holds (e.g., see the previous example)
  - The challenge is to “match” the descriptions provided by the data sources (see, e.g., [WHT+99])

- If 4) does not hold, then some partial scores will be missing. The strategy to be taken depends on the specific scoring function (e.g., if Budget is undefined then Salary/Budget is undefined as well)

- In order to support sorted accesses, a possibility is to use **next_NN**

- For random accesses, a PK index is required
Top-k 1-1 join queries: example

- Aggregating reviews of restaurants

**MangiarBene**

<table>
<thead>
<tr>
<th>Name</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Al vecchio mulino</td>
<td>9.2</td>
</tr>
<tr>
<td>La tavernetta</td>
<td>9.0</td>
</tr>
<tr>
<td>Il desco</td>
<td>8.3</td>
</tr>
<tr>
<td>Da Gino</td>
<td>7.5</td>
</tr>
<tr>
<td>Tutti a tavola!</td>
<td>6.4</td>
</tr>
<tr>
<td>Le delizie del palato</td>
<td>5.5</td>
</tr>
<tr>
<td>Acqua in bocca</td>
<td>5.0</td>
</tr>
</tbody>
</table>

**PaneeVino**

<table>
<thead>
<tr>
<th>Name</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Da Gino</td>
<td>9.0</td>
</tr>
<tr>
<td>Il desco</td>
<td>8.5</td>
</tr>
<tr>
<td>Al vecchio mulino</td>
<td>7.5</td>
</tr>
<tr>
<td>Le delizie del palato</td>
<td>7.5</td>
</tr>
<tr>
<td>La tavernetta</td>
<td>7.0</td>
</tr>
<tr>
<td>Acqua in bocca</td>
<td>6.5</td>
</tr>
<tr>
<td>Tutti a tavola!</td>
<td>6.0</td>
</tr>
</tbody>
</table>

**SELECT** *
**FROM** MangiarBene MB, PaneeVino PV
**WHERE** MB.Name = PV.Name
**ORDER BY** MB.Score + PV.Score DESC
**STOP AFTER 1**

Note: the winner is never the best locally!
A homogeneous model for scoring functions

- In order to provide a unifying approach to the problem, we consider:

- A top-k 1-1 join query \( Q = (Q_1, Q_2, \ldots, Q_m) \)
  - \( Q_j \) is the sub-query sent to the \( j \)-th source/relation

- Each object \( o \) returned by the input \( L_j \) has an associated local/partial score \( p_j(o) \), \( p_j(o) \in [0,1] \) and dependent on \( Q_j \)
  - For convenience, scores are normalized, with higher scores being better
  - This can be easily relaxed; what matters is to know which is the best and the worst possible value of \( p_j \)
  - The hypercube \([0,1]^m\) is conveniently called the “score space”

- The point \( p(o) = (p_1(o), p_2(o), \ldots, p_m(o)) \in [0,1]^m \) is the map of \( o \) into the score space

- The global/overall score \( S(o) \) of \( o \) is computed by means of a scoring function (s.f.) \( S \) that combines in some way the local scores of \( o \):

\[
S : [0,1]^m \to \mathbb{R} \quad S(o) \equiv S(p(o)) = S(p_1(o), p_2(o), \ldots, p_m(o))
\]
The score space

- Consider the 2-dim attribute space \( A = (\text{Price}, \text{Mileage}) \)
- Let Q1 be the sub-query on Price, and Q2 the sub-query on Mileage
- We can set: \( p_1(o) = 1 - o.\text{Price}/\text{MaxP} \), \( p_2(o) = 1 - o.\text{Mileage}/\text{MaxM} \)
- Let’s take \( \text{MaxP} = 50,000 \) and \( \text{MaxM} = 80,000 \)
- Objects in A are mapped into the score space as in the figure on the right
  - The relative order on each coordinate (local ranking) remains unchanged

\[ \begin{align*}
\text{C6:} & \quad (10,40) \rightarrow (0.8,0.5) \\
\end{align*} \]
Some common scoring functions

**SUM (AVG):** used to equally weigh preferences

\[ SUM(o) \equiv SUM(p(o)) = p_1(o) + p_2(o) + \ldots + p_m(o) \]

**WSUM (Weighted sum):** to differently weigh the ranking attributes

\[ WSUM(o) \equiv WSUM(p(o)) = w_1*p_1(o) + w_2*p_2(o) + \ldots + w_m*p_m(o) \]

**MIN (Minimum):** just considers the worst partial score

\[ MIN(o) \equiv MIN(p(o)) = \min\{p_1(o), p_2(o), \ldots, p_m(o)\} \]

**MAX (Maximum):** just considers the best partial score

\[ MAX(o) \equiv MAX(p(o)) = \max\{p_1(o), p_2(o), \ldots, p_m(o)\} \]

Remind: (even with MIN) we always want to retrieve the k objects with the highest global scores
Equally scored objects

- Similarly to iso-distance curves in an attribute space, we can define iso-score curves in the score space, in order to highlight the sets of points with a same global score.

\[ S(o) = \text{MIN}(p(o)) \]

\[ S(o) = \text{MAX}(p(o)) \]
The simplest case: MAX

- We are now ready to ask “the big question”:

  How can we efficiently compute the result of a top-k 1-1 join query using a scoring function $S$? 

- For the particular case $S \equiv \text{MAX}$ the solution is really simple [Fag96]:
  
  You can use my algorithm $B_0$, which just retrieves the best $k$ objects from each source, that’s all!

  **Beware!** $B_0$ only works for MAX, other scoring functions require smarter, and more costly, algorithms.

Ronald Fagin
The $B_0$ algorithm

**Input:** ranked lists $L_j \ (j=1,\ldots,m)$, integer $k \geq 1$

**Output:** the top-$k$ objects according to the MAX scoring function

1. $B := \emptyset$; \hspace{1cm} // $B$ is a main-memory buffer
2. for $j = 1$ to $m$:
3. \hspace{0.5cm} $Obj(j) := \emptyset$; \hspace{1cm} // the set of objects “seen” on $L_j$
4. \hspace{0.5cm} for $i = 1$ to $k$:
\hspace{1cm} \hspace{0.5cm} // get the best $k$ objects from each list
5. \hspace{1cm} $t := \text{getNext}_{L_j}()$;
6. \hspace{1cm} $Obj(j) := Obj(j) \cup \{t.OID\}$;
7. \hspace{1cm} if $t.OID$ was not retrieved from other lists then: $\text{INSERT}(B, t)$ // adds $t$ to the buffer
\hspace{1cm} \hspace{0.5cm} else: join $t$ with the entry in $B$ having the same OID;
8. for each object $o \in Obj := \bigcup_j Obj(j)$: \hspace{1cm} // for each object with at least one partial score...
9. \hspace{1cm} $\text{MAX}(o) := \max_j \{p_j(o): p_j(o) \text{ is defined}\}$; // ...compute $\text{MAX}$ using the available scores
10. return the $k$ objects with maximum score;
11. end.

- Algorithm $B_0$ just performs $k$ sorted accesses on each list (k s.a. “rounds”), and then computes the result without the need to obtain missing partial scores (i.e., no random accesses are executed)
B₀ : examples

k = 2

<table>
<thead>
<tr>
<th>OID</th>
<th>p1</th>
<th>OID</th>
<th>p2</th>
<th>OID</th>
<th>p3</th>
</tr>
</thead>
<tbody>
<tr>
<td>o7</td>
<td>0.7</td>
<td>o2</td>
<td>0.9</td>
<td>o7</td>
<td>1.0</td>
</tr>
<tr>
<td>o3</td>
<td>0.65</td>
<td>o3</td>
<td>0.6</td>
<td>o2</td>
<td>0.8</td>
</tr>
<tr>
<td>o4</td>
<td>0.6</td>
<td>o7</td>
<td>0.4</td>
<td>o4</td>
<td>0.75</td>
</tr>
<tr>
<td>o2</td>
<td>0.5</td>
<td>o4</td>
<td>0.2</td>
<td>o3</td>
<td>0.7</td>
</tr>
</tbody>
</table>

k = 3

<table>
<thead>
<tr>
<th>Name</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Al vecchio mulino</td>
<td>9.2</td>
</tr>
<tr>
<td>La tavernetta</td>
<td>9.0</td>
</tr>
<tr>
<td>Il desco</td>
<td>8.3</td>
</tr>
<tr>
<td>Da Gino</td>
<td>7.5</td>
</tr>
<tr>
<td>Tutti a tavola!</td>
<td>6.4</td>
</tr>
<tr>
<td>Le delizie del palato</td>
<td>5.5</td>
</tr>
<tr>
<td>Acqua in bocca</td>
<td>5.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Da Gino</td>
<td>9.0</td>
</tr>
<tr>
<td>Il desco</td>
<td>8.5</td>
</tr>
<tr>
<td>Al vecchio mulino</td>
<td>7.5</td>
</tr>
<tr>
<td>Le delizie del palato</td>
<td>7.5</td>
</tr>
<tr>
<td>La tavernetta</td>
<td>7.0</td>
</tr>
<tr>
<td>Acqua in bocca</td>
<td>6.5</td>
</tr>
<tr>
<td>Tutti a tavola!</td>
<td>6.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OID</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>o7</td>
<td>1.0</td>
</tr>
<tr>
<td>o2</td>
<td>0.9</td>
</tr>
<tr>
<td>o3</td>
<td>0.65</td>
</tr>
</tbody>
</table>
Why $B_0$ works: graphical intuition

- By hypothesis, in the figure below at least $k$ objects $o$ have $S(o) \geq 0.8$
  - This holds because at least one sorted access scan (on L2, in the figure) stops after retrieving at the $k$-th round an object with local score $= 0.8$
- An object like $o'$, that has not been retrieved by any sorted access scan (thus $o' \not\in \text{Obj}$), cannot have a global score higher than 0.8!
How $B_0$ works on the restaurants example

- After 3 s.a. rounds it is guaranteed that there are at least 3 restaurants $o$ with $S(o) \geq 8.3$
- A restaurant like $o'$, that has not been retrieved by any sorted access scan (thus $o' \notin Obj$), cannot have a global score higher than 8.3
B₀: A proof of correctness

- Let Res be the result of B₀ (Res ⊆ Obj)
- The need for a formal proof of correctness is motivated by the following:
  
  if o ∈ Obj – Res, then S(o) is not guaranteed to be correct
  
  (e.g., see o3 in the 1st example)
- Thus, we have to show that this does not influence the result
- On the other hand, if o ∉ Obj we have just shown that o cannot be better than any object in Res

Theorem:
The B₀ algorithm correctly determines the top-k objects and their global scores

- We split the proof in two parts:
  - We first show that if o ∈ Res, then S(o) is correct
  - We then show that if o ∈ Obj - Res, then, even if its global score is not correct, the algorithm correctly determines the top-k objects
Proof (1): if \( o \in Res \), then \( S(o) \) is correct

- Let \( SB_0(o) \) be the global score, as computed by \( B_0 \), for an object \( o \in Obj \)
- By def. of MAX, it is: \( SB_0(o) \leq S(o) \) (e.g., \( SB_0(o_3) = 0.65 \leq S(o_3) = 0.7 \))
- Let \( o_1 \in Res \) and assume by contradiction that \( SB_0(o_1) < S(o_1) \)
- This is to say that there exists \( L_j \) such that (s.t.): \( o_1 \notin Obj(j) \) and \( S(o_1) = p_j(o_1) \)
- In turn this implies that there are \( k \) objects \( o \in Obj(j) \) s.t.
  \[
  SB_0(o_1) < S(o_1) = p_j(o_1) \leq p_j(o) \leq SB_0(o) \leq S(o) \quad \forall o \in Obj(j)
  \]
- Thus \( o_1 \) cannot belong to \( Res \), a contradiction

<table>
<thead>
<tr>
<th>OID</th>
<th>pj</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>( o )</td>
<td>( p_j(o) \leq SB_0(o) \leq S(o) )</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>( o_1 )</td>
<td>( SB_0(o_1) &lt; S(o_1) = p_j(o_1) )</td>
</tr>
</tbody>
</table>

\( Obj(j) \) contains \( k \) objects

?? Impossible if \( o_1 \in Res \)
Proof (2): Res contains the top-k objects

Consider an object, say o1, s.t. o1 ∈ Obj – Res

If SB₀(o₁) = S(o₁) then there is nothing to demonstrate 😊

Then, assume that at least one partial score of o₁, pj(o₁), is not available, and that SB₀(o₁) < S(o₁) = pj(o₁). Then

\[ SB₀(o₁) < S(o₁) = pj(o₁) \leq pj(o) \leq SB₀(o) \leq S(o) \quad \forall o \in \text{Obj(j)} \]

Since each object in Res has a global score at least equal to the lowest score seen on Lj, it follows that it is impossible to have S(o₁) > S(o) if o ∈ Res.

<table>
<thead>
<tr>
<th>OID</th>
<th>pj</th>
</tr>
</thead>
<tbody>
<tr>
<td>…..</td>
<td>…..</td>
</tr>
<tr>
<td>o</td>
<td>pj(o) ≤ SB₀(o) ≤ S(o)</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>o₁</td>
<td>SB₀(o₁) &lt; S(o₁) = pj(o₁)</td>
</tr>
</tbody>
</table>

Obj(j) contains k objects

Impossible to have S(o₁) > S(o), o ∈ Res
An interesting question is: can we compute the correct result using less than k rounds of sorted accesses?

k=1: 1 round is required
- Although some s.a.’s can be saved if we fetch an object with maximum score (10, in the example)

k=2: 2 rounds are required
- We can save 1 s.a. if we first access MangiarBene

k=3: Here we can stop after only 2 rounds!

k=4: 3 rounds are enough!

Which is the general rule?
Less than k rounds? Yes, sometimes

- For each list \( L_j \), let \( p_j \) denote the lowest score seen so far.
- Let \( \text{Res} \) denote the current set of top-\( k \) objects, ordered by their current \( \text{MAX} \) value; thus, \( \text{Res}[k].\text{score} \) is the lowest of such global scores.
- The following stopping condition is always verified after \( k \) s.a. rounds, but it might also hold earlier.

**Theorem:**
An algorithm for top-\( k \) 1-1 join queries using the MAX scoring function can be stopped iff \( \text{Res}[k].\text{score} \geq \max_j \{ p_j \} \)

**Proof:**

**(if)** Since each \( L_j \) is ordered by non-increasing values of \( p_j \), no unseen object on \( L_j \) can have a partial score higher than \( p_j \). Thus, no unseen object can have a score higher than \( \max_j \{ p_j \} \). It follows that \( \text{Res} \) is correct and that the scores of the objects in \( \text{Res} \) are correct as well.

**(only if)** Assume that the algorithm stops when \( \text{Res}[k].\text{score} < \max_j \{ p_j \} \). Then, a list \( L_j \), with \( p_j > \text{Res}[k].\text{score} \) might contain an object \( o \) s.t. \( p_j(o) > \text{Res}[k].\text{score} \).
Towards an optimal algorithm

- Besides minimizing the number of rounds, the theorem is also the key to minimize the overall number of sorted accesses.
- To this end, we do not insist in executing s.a.’s in a round-robin fashion.
  - Thus, we may also have situations like this:

<table>
<thead>
<tr>
<th>OID</th>
<th>p1</th>
</tr>
</thead>
<tbody>
<tr>
<td>o9</td>
<td>0.7</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OID</th>
<th>p2</th>
</tr>
</thead>
<tbody>
<tr>
<td>o2</td>
<td>0.9</td>
</tr>
<tr>
<td>o3</td>
<td>0.6</td>
</tr>
<tr>
<td>o5</td>
<td>0.4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OID</th>
<th>p3</th>
</tr>
</thead>
<tbody>
<tr>
<td>o1</td>
<td>0.8</td>
</tr>
<tr>
<td>o2</td>
<td>0.8</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
The new algorithm, called MaxOptimal, is essentially based on the same principles of kNNOptimal (!?).

- At each step it performs a sorted access on the “most promising” list Lj* for which \( p_j^* \) is maximum: \( j^* = \arg\max_j \{ p_j \} \)
- It only keeps in memory the k best objects found so far

**Input:** ranked lists \( L_j \) (\( j=1,\ldots,m \)), integer \( k \geq 1 \)

**Output:** the top-k objects according to the MAX scoring function

1. for \( i=1 \) to \( k \): \( \text{Res}[i] := [\text{null},0] \); // entry of type: \([\text{OID},\text{score}]\) (+ other attr.’s as needed)
2. for \( j = 1 \) to \( m \): \( p_j = 1 \); // the best possible score on \( L_j \)
3. while \( \text{Res}[k].\text{score} < \max_j \{ p_j \} \):
4. \hspace{1cm} \( j^* = \arg\max_j \{ p_j \} \); \( t := \text{getNext}_{L_j^*}() \); // get the next best object from list \( L_j^* \)
5. \hspace{1cm} if \( t.p_j^* > \text{Res}[k].\text{score} \) then:
6. \hspace{2cm} if \( t.\text{OID} \in \text{Res.OID} \) then: update the entry in \( \text{Res} \) having the same \( \text{OID} \)
7. \hspace{2cm} else: \{remove the object in \( \text{Res}[k] \); insert \([t.\text{OID},t.p_j^*]\) in \( \text{Res}\};
8. \hspace{1cm} return \( \text{Res} \);
9. end.
MaxOptimal: example

**k = 4**

<table>
<thead>
<tr>
<th>L1</th>
<th>p1</th>
<th>Score (S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>o1</td>
<td>0.95</td>
<td>0.95</td>
</tr>
<tr>
<td>o7</td>
<td>0.85</td>
<td>0.85</td>
</tr>
<tr>
<td>o4</td>
<td>0.6</td>
<td>0.6</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L2</th>
<th>p2</th>
<th>Score (S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>o2</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>o3</td>
<td>0.7</td>
<td>0.7</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L3</th>
<th>p3</th>
<th>Score (S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>o7</td>
<td>0.6</td>
<td>0.6</td>
</tr>
<tr>
<td>o2</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>o3</td>
<td>0.7</td>
<td>0.7</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Res</th>
<th>Score (S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>o1</td>
<td>0.95</td>
</tr>
<tr>
<td>o2</td>
<td>0.8</td>
</tr>
<tr>
<td>o7</td>
<td>0.6</td>
</tr>
<tr>
<td>...</td>
<td>0</td>
</tr>
</tbody>
</table>

---

0 < 0.95
0 < 0.85
0.6 < 0.8
0.7 ≥ 0.7

after the 1st round
The optimality of MaxOptimal

The reason why MaxOptimal minimizes the number of s.a.’s is similar to the one that applies to kNNOptimal

**Theorem:**
Let $\text{MAX}_k$ be the $k$-th highest global score in the dataset. Then, the MaxOptimal algorithm for top-$k$ 1-1 join queries never performs a sorted access on a list $L_j$ for which it is $p_j < \text{MAX}_k$

**Proof:** By contradiction, assume that MaxOptimal performs a s.a. on list $L_j^*$, with $p_j^* < \text{MAX}_k$. For this it has to be $p_j^* = \max_j \{p_j\}$, from which it is derived $\text{MAX}_k > p_j$, for each $j$. Before performing this s.a. it is $\text{Res}[k].\text{score} < \text{MAX}_k$, otherwise the algorithm would halt. But this implies that there exists an unseen object whose global score is $\text{MAX}_k$, which is impossible.
Why $B_0$ doesn’t work for other scoring f.’s

- Let $S \equiv \text{MIN}$ and $k = 1$

<table>
<thead>
<tr>
<th>OID</th>
<th>p1</th>
<th>OID</th>
<th>p2</th>
<th>OID</th>
<th>p3</th>
</tr>
</thead>
<tbody>
<tr>
<td>o7</td>
<td>0.9</td>
<td>o2</td>
<td>0.95</td>
<td>o7</td>
<td>1.0</td>
</tr>
<tr>
<td>o3</td>
<td>0.65</td>
<td>o3</td>
<td>0.7</td>
<td>o2</td>
<td>0.8</td>
</tr>
<tr>
<td>o2</td>
<td>0.6</td>
<td>o4</td>
<td>0.6</td>
<td>o4</td>
<td>0.75</td>
</tr>
<tr>
<td>o1</td>
<td>0.5</td>
<td>o1</td>
<td>0.5</td>
<td>o3</td>
<td>0.7</td>
</tr>
<tr>
<td>o4</td>
<td>0.4</td>
<td>o7</td>
<td>0.5</td>
<td>o1</td>
<td>0.6</td>
</tr>
</tbody>
</table>

- What if we consider ALL the partial scores of the objects in Obj (Obj = {o2,o7} in the figure)?

- After performing the necessary random accesses:
  
  \[
  \text{getScore}_{L1}(o2), \text{getScore}_{L3}(o2), \text{getScore}_{L2}(o7)
  \]

  we get:

<table>
<thead>
<tr>
<th>OID</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>o2</td>
<td>0.6</td>
</tr>
<tr>
<td>o7</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Wrong!!

Wrong!!?
Why $B_0$ doesn’t work: graphical intuition

- Let $S \equiv \text{MIN}$ and $k = 1$
- When the sorted accesses terminate, we don’t have any lower bound on the global scores of the retrieved objects (i.e., it might also be $S(o) = 0$)
- An object, like $o'$, that has not been retrieved by any sorted access scan can now be the winner!
- Note that, in this case, $o'$ would be the best match even for $S \equiv \text{SUM}$

![Graphical Intuition Diagram](image)
The FA algorithm: monotone scoring f.’s

- The FA (or $A_0$) algorithm [Fag96] can be used to solve top-k 1-1 join queries with any monotone scoring function $S$:

**Monotone scoring function:**
- An $m$-ary scoring function $S$ is monotone if
  \[ x_1 \leq y_1, \ x_2 \leq y_2, \ldots, \ x_m \leq y_m \ \Rightarrow \ S(x_1,x_2,\ldots,x_m) \leq S(y_1,y_2,\ldots,y_m) \]

- FA exploits the monotonicity property in order to understand when sorted accesses can be stopped.

No object in this closed (hyper-)rectangle can be better than $o$!
The FA algorithm

**Input:** ranked lists $L_j$ ($j=1,...,m$), integer $k \geq 1$, monotone scoring function $S$

**Output:** the top-$k$ objects according to $S$

// 1st phase: sorted accesses
1. for $j = 1$ to $m$: $\text{Obj}(j) := \emptyset$; $B := \emptyset$; $M := \emptyset$;
2. while $|M| < k$:
4. for $j = 1$ to $m$:
5. $t := \text{getNext}_{L_j}();$ $\text{Obj}(j) := \text{Obj}(j) \cup \{t.\text{OID}\};$ // get the next best object from list $L_j$
6. if $t.\text{OID}$ was not retrieved from other lists then: INSERT($B,t$)
7. else: join $t$ with the entry in $B$ having the same OID;
8. $M := \bigcap_j \text{Obj}(j)$; // the set of objects seen on all the $m$ lists

// 2nd phase: random accesses
9. for each object $o \in \text{Obj} := \bigcup_j \text{Obj}(j)$: // for each object with at least one partial score...
   perform random accesses to retrieve the missing partial scores for $o$;

// 3rd phase: score computation
10. for each object $o \in \text{Obj}$: compute $S(o)$;
11. return the $k$ objects with maximum score;
12. end.
How FA works

- Let’s take \( k = 1 \). We apply FA to the following data:

<table>
<thead>
<tr>
<th>OID</th>
<th>( p_1 )</th>
<th>OID</th>
<th>( p_2 )</th>
<th>OID</th>
<th>( p_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>o7</td>
<td>0.9</td>
<td>o2</td>
<td>0.95</td>
<td>o7</td>
<td>1.0</td>
</tr>
<tr>
<td>o3</td>
<td>0.65</td>
<td>o3</td>
<td>0.7</td>
<td>o2</td>
<td>0.8</td>
</tr>
<tr>
<td>o2</td>
<td>0.6</td>
<td>o4</td>
<td>0.6</td>
<td>o4</td>
<td>0.75</td>
</tr>
<tr>
<td>o1</td>
<td>0.5</td>
<td>o1</td>
<td>0.5</td>
<td>o3</td>
<td>0.7</td>
</tr>
<tr>
<td>o4</td>
<td>0.4</td>
<td>o7</td>
<td>0.5</td>
<td>o1</td>
<td>0.6</td>
</tr>
</tbody>
</table>

and after the sorted accesses we obtain:

\[
M = \{o2\} \\
\text{Obj} = \{o2, o3, o4, o7\}
\]

- After performing the needed random accesses we get:

\[
\begin{align*}
S & \equiv \text{MIN} \\
\begin{array}{|c|c|}
\hline
\text{OID} & S \\
\hline
o3 & 0.65 \\
\hline
\end{array} & \quad \begin{array}{|c|c|}
\hline
\text{OID} & S \\
\hline
o2 & 0.6 \\
\hline
o7 & 0.5 \\
\hline
o4 & 0.4 \\
\hline
\end{array}
\end{align*}
\]

\[
S \equiv \text{SUM} \\
\begin{array}{|c|c|}
\hline
\text{OID} & S \\
\hline
o7 & 2.4 \\
\hline
o2 & 2.35 \\
\hline
o3 & 2.05 \\
\hline
o4 & 1.75 \\
\hline
\end{array}
\]

RIGHT!!
Why FA is correct: formal and intuitive

**Theorem:**
The FA algorithm is correct for any monotone scoring function S

**Proof:** Let Res be the set of objects returned by FA. It is sufficient to show that if $o' \notin \text{Obj}$, then $o'$ cannot be better than any object $o \in \text{Res}$.

Let $o$ be any object in Res. Then, there is at least one object $o'' \in M$ (possibly coincident with $o$ itself) such that $S(o'') \leq S(o)$, otherwise $o$ would not be in Res. Since $o' \notin \text{Obj}$, for each $L_j$ it is $p_j(o') \leq p_j(o'')$, and from the assumption of monotonicity of $S$ it is $S(o') \leq S(o'')$; it follows that $S(o') \leq S(o)$.

FA stops when this region contains at least $k$ points...

...and each of them cannot be worse than a point in this region.
FA: performance

- When the sub-queries are independent (i.e., each local ranking is independent of the others) it can be proved that the cost of FA (no. of sorted and random accesses) for a DB of N objects is, with arbitrarily high probability, and assuming m constant:

\[
O\left(\frac{N^{(m-1)/m} k^{1/m}}{m} \right)
\]

Proof intuition: Since FA executes the s.a.’s using a round-robin strategy, on each list it executes, say, X s.a.’s. The expected size of the intersection of m random subsets, each of cardinality X, taken from a set with N elements is

\[
N \times \left(\frac{X}{N}\right)^m = \frac{X^m}{N^{m-1}}
\]

By equating to k and solving it is obtained: \(X = N^{(m-1)/m} k^{1/m}\)

The result follows after observing that the number of s.a.’s is \(m^*X\) and the number of r.a.’s is at most \((m^*X - m^*k)(m-1)\)
Limits of FA

- The major drawback of the algorithm is that it does not exploit at all the specific scoring function $S$.

- In particular, since $S$ is used only in the third step (when global scores are computed), for a given DB the s.a.+r.a. cost of FA is independent of $S$!

- Further, the memory requirements of FA can become prohibitive (since FA has to buffer all the objects accessed through sorted access).

- Although some amelioration is possible (e.g., interleaving random accesses and score computation, which might save some r.a.), a major improvement is possible only by changing the stopping condition, which in FA is based only on the local rankings of the objects.
The TA algorithm

- TA (Threshold Algorithm) [FLN01, FLN03] differs from FA in that:
  - it interleaves sorted and random accesses
  - it is based on a numerical stopping rule
- In particular, TA uses a threshold $T$, which is an upper bound to the scores of all unseen objects

**Input:** ranked lists $L_j$ ($j=1,...,m$), integer $k \geq 1$, monotone scoring function $S$

**Output:** the top-$k$ objects according to $S$

1. for $i = 1$ to $k$: $Res[i] := [null,0]$;
2. for $j = 1$ to $m$: $p_j := 1$;
3. while $Res[k].score < T := S(p_1,p_2,...,p_m)$:  // $T$ is the “threshold”
4.   for $j = 1$ to $m$:
5.      $t := \text{getNext}_{L_j}();$ $o := t.OID$;
6.      perform random accesses to retrieve the missing partial scores for $o$;
7.      if $S(o) := S(p_1(o),...,p_m(o)) > Res[k].score$ then:
8.         {remove the object in $Res[k]$; insert $[o,S(o)]$ in $Res$};
9.   return $Res$;
10. end.
Why TA is correct: formal and intuitive

**Theorem:**
The TA algorithm is correct for any monotone scoring function $S$

**Proof:** Consider an object $o'$ that has not been seen under sorted access. Thus, for each $j$ it is $p_j(o') \leq p_j$. Due to the monotonicity of $S$ this implies $S(o') \leq T$. By definition of $\text{Res}$, for each object $o \in \text{Res}$ it is $S(o) \geq T$, thus $S(o') \leq S(o)$.

![Graph showing TA stops when this region contains k points at least as good as T and no object here can be better than T!](image)
How TA works

- Let’s take $S \equiv \text{MIN}$ and $k = 1$

<table>
<thead>
<tr>
<th>OID</th>
<th>p1</th>
<th>OID</th>
<th>p2</th>
<th>OID</th>
<th>p3</th>
</tr>
</thead>
<tbody>
<tr>
<td>o7</td>
<td>0.9</td>
<td>o2</td>
<td>0.95</td>
<td>o7</td>
<td>1.0</td>
</tr>
<tr>
<td>o3</td>
<td>0.65</td>
<td>o3</td>
<td>0.7</td>
<td>o2</td>
<td>0.8</td>
</tr>
<tr>
<td>o2</td>
<td>0.6</td>
<td>o4</td>
<td>0.6</td>
<td>o4</td>
<td>0.75</td>
</tr>
<tr>
<td>o1</td>
<td>0.5</td>
<td>o1</td>
<td>0.5</td>
<td>o3</td>
<td>0.7</td>
</tr>
<tr>
<td>o4</td>
<td>0.4</td>
<td>o7</td>
<td>0.5</td>
<td>o1</td>
<td>0.6</td>
</tr>
</tbody>
</table>

$S(o2) = 0.6; T = 0.9$
$S(o3) = 0.65; T = 0.65$

- Let’s take $S \equiv \text{SUM}$ and $k = 2$

<table>
<thead>
<tr>
<th>OID</th>
<th>p1</th>
<th>OID</th>
<th>p2</th>
<th>OID</th>
<th>p3</th>
</tr>
</thead>
<tbody>
<tr>
<td>o7</td>
<td>0.9</td>
<td>o2</td>
<td>0.95</td>
<td>o7</td>
<td>1.0</td>
</tr>
<tr>
<td>o3</td>
<td>0.65</td>
<td>o3</td>
<td>0.7</td>
<td>o2</td>
<td>0.8</td>
</tr>
<tr>
<td>o2</td>
<td>0.6</td>
<td>o4</td>
<td>0.6</td>
<td>o4</td>
<td>0.75</td>
</tr>
<tr>
<td>o1</td>
<td>0.5</td>
<td>o1</td>
<td>0.5</td>
<td>o3</td>
<td>0.7</td>
</tr>
<tr>
<td>o4</td>
<td>0.4</td>
<td>o7</td>
<td>0.5</td>
<td>o1</td>
<td>0.6</td>
</tr>
</tbody>
</table>

$S(o7) = 2.4; S(o2) = 2.35; T = 2.85$
$S(o7) = 2.4; S(o2) = 2.35; T = 2.15$
Let’s take $S \equiv \text{SUM}$ and $k = 2$

$S(o_3) > S(o_1) \geq T$

Stop!
Performance of TA: Cost model

- In general, TA performs much better than FA, since it can “adapt” to the specific scoring function $S$.

- In order to characterize the performance of TA, we consider the so-called “middleware” (or “access”) cost: $\text{cost} = SA \cdot c_{SA} + RA \cdot c_{RA}$, where:
  - $SA$ (RA) is the total number of sorted (random) accesses
  - $c_{SA}$ ($c_{RA}$) is the unitary (base) cost of a sorted (random) access

- In the basic setting, it is $c_{SA} = c_{RA}$ (=1, for simplicity)

- In other cases, base costs may widely differ
  - E.g. for web sources it is usually the case $c_{RA} > (>>) c_{SA}$, with the limit case $c_{RA} = \infty$ in which r.a.’s are impossible
  - On the other hand, some sources might not be accessible through sorted access, in which case it is $c_{SA} = \infty$ (for instance, we do not have an index to process pj)
Performance of TA: Instance optimality

- A fundamental concept needed to understand in which sense “TA performs well” is that of

**Instance optimality:**

- Given a class of algorithms \( \mathcal{A} \) and a class \( \mathcal{D} \) of DB’s (inputs of the algorithms), an algorithm \( A \in \mathcal{A} \) is instance-optimal over \( \mathcal{A} \) and \( \mathcal{D} \), for a given cost measure, if for every \( B \in \mathcal{A} \) and every \( DB \in \mathcal{D} \) it is

\[
\text{cost}(A, DB) = O(\text{cost}(B, DB))
\]

- This is to say that there are constants \( c \) and \( c' \) such that:

\[
\text{cost}(A, DB) \leq c \times \text{cost}(B, DB) + c'
\]

- If \( A \) is instance-optimal, then any algorithm can improve on the cost of \( A \) by only a constant factor \( c \), which is therefore called the optimality ratio of \( A \)

- Observe that instance optimality is a much stronger notion than optimality in the average or worst case
  - E.g., binary search is optimal in the worst case, but it is not instance-optimal
Performance of TA: Main fact

- TA is instance-optimal over all DB’s and over all algorithms that do not make “wild guesses”, and its optimality ratio is $m$ when $c_{RA} = 0$

An algorithm A makes wild guesses if it makes a random access for object o without having seen before o under sorted access

- Note that algorithms making wild guesses are only of theoretical interest

Proof: Assume TA stops after having executed $X$ s.a. rounds (at “depth $X$”), thus $m*X$ s.a.’s. Consider any correct algorithm B and assume that, on each list, B executes strictly less than $X$ s.a.’s. Thus, on list $L_j$ it reaches $Depth(B,j) < X$. Let $MaxDepth(B) = \max_j\{Depth(B,j)\}$. Consider now executing TA for $MaxDepth(B)$ rounds. Since these include all the s.a.’s (and corresponding r.a.’s) done by B, and B is assumed to be correct, then TA could halt at depth $MaxDepth(B) < X$, a contradiction. It follows that any correct algorithm B has to have $MaxDepth(B) \geq X$, thus its cost is $\geq X$. Thus, for each DB it is:

$$cost(TA,DB) = m*X \leq m*cost(B,DB)$$
Adapting to scenarios with different costs

- When $c_{RA} > 0$ (the real case) the cost of TA halting at depth $X$ is at most
  \[ \text{cost}(TA, DB) = m \times X \times c_{SA} + m \times X \times (m - 1) \times c_{RA} \]
  since in the worst case TA retrieves $m \times X$ distinct objects, and for each of them executes $(m-1)$ random accesses.

- As seen, any other algorithm $B$ will pay at least a cost
  \[ \text{cost}(B, DB) \geq X \times c_{SA} \]

- Thus, the optimality ratio is now:
  \[ \frac{m \times X \times c_{SA} + m \times (m - 1) \times X \times c_{RA}}{X \times c_{SA}} = m + m \times (m - 1) \times \frac{c_{RA}}{c_{SA}} \]

- The above is quite bad when $c_{RA} > (>>) c_{SA}$, since the cost of random accesses will prevail.
  - E.g., with $c_{RA}/c_{SA} = 10$, and $m = 3$, the optimality ratio is 63.
The NRA algorithm: preliminaries

- NRA (No Random Access Algorithm) [FLN01,FLN03] is an algorithm that applies when r.a.’s cannot be executed (or their cost is really prohibitive)
- It correctly returns the top-k objects, but their scores might be wrong
  - This is to limit the cost of the algorithm
- The idea of NRA is to maintain, for each object o retrieved by s.a., a lower bound (lbscore), $S^-(o)$, and an upper bound (ubscore), $S^+(o)$, on its score
  - $S^-(o)$ is obtained by setting $p_j(o) = 0$ (or the minimal possible value of $p_j$) if o has not been seen on $L_j$
  - $S^+(o)$ is obtained by setting $p_j(o) = p_j$ if o has not been seen on $L_j$
- NRA uses a buffer B with unlimited capacity, which is kept sorted according to decreasing lbscore values
The NRA algorithm

- Let Res denote the first k positions of B, Res = {B[1],...,B[k]};
- The idea of the algorithm is to halt when no object o’ not in Res can do better than any of the objects in Res, i.e., when
  \[ S^+(o') \leq S^-(o) \quad \forall \ o' \not\in \text{Res}, \ o \in \text{Res} \]
- To check this, it is sufficient to consider the maximum value of \( S^+(o') \) among the objects in B-Res and the threshold (the latter providing an upper bound to unseen objects)

**Input:** ranked lists Lj (j=1,...,m), integer k ≥ 1, monotone scoring function S

**Output:** the top-k objects according to S

1. B := \emptyset; // entry of type: [OID,lbscore,ubscore]; B is ordered by decreasing lbscore values
2. for j = 1 to m: pj := 1;
3. while B[k].lbscore < max{max{B[i].ubscore, i > k},S(p1,p2,...,pm)}:
4.   for j = 1 to m:
5.     t := getNext_{L_j}(); o := t.OID; insert [o,S^-(o),S^+(o)] in B;
6. return {B[1],...,B[k]};
7. end.
NRA: example

L1

<table>
<thead>
<tr>
<th>OID</th>
<th>p1</th>
</tr>
</thead>
<tbody>
<tr>
<td>o1</td>
<td>1.0</td>
</tr>
<tr>
<td>o7</td>
<td>0.9</td>
</tr>
<tr>
<td>o2</td>
<td>0.7</td>
</tr>
<tr>
<td>o6</td>
<td>0.2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

L2

<table>
<thead>
<tr>
<th>OID</th>
<th>p2</th>
</tr>
</thead>
<tbody>
<tr>
<td>o2</td>
<td>0.8</td>
</tr>
<tr>
<td>o3</td>
<td>0.75</td>
</tr>
<tr>
<td>o4</td>
<td>0.5</td>
</tr>
<tr>
<td>o1</td>
<td>0.4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

L3

<table>
<thead>
<tr>
<th>OID</th>
<th>p3</th>
</tr>
</thead>
<tbody>
<tr>
<td>o7</td>
<td>0.6</td>
</tr>
<tr>
<td>o2</td>
<td>0.6</td>
</tr>
<tr>
<td>o3</td>
<td>0.5</td>
</tr>
<tr>
<td>o1</td>
<td>0.4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

S \equiv \text{SUM} k = 2

<table>
<thead>
<tr>
<th>OID</th>
<th>lbscore</th>
<th>ubscore</th>
</tr>
</thead>
<tbody>
<tr>
<td>o1</td>
<td>1.0</td>
<td>2.4</td>
</tr>
<tr>
<td>o2</td>
<td>0.8</td>
<td>2.4</td>
</tr>
<tr>
<td>o7</td>
<td>0.6</td>
<td>2.4</td>
</tr>
</tbody>
</table>

B (1st round)

0.8 < \text{max}\{2.4,2.4\}

B (2nd round)

<table>
<thead>
<tr>
<th>OID</th>
<th>lbscore</th>
<th>ubscore</th>
</tr>
</thead>
<tbody>
<tr>
<td>o7</td>
<td>1.5</td>
<td>2.25</td>
</tr>
<tr>
<td>o2</td>
<td>1.4</td>
<td>2.3</td>
</tr>
<tr>
<td>o1</td>
<td>1.0</td>
<td>2.35</td>
</tr>
<tr>
<td>o3</td>
<td>0.75</td>
<td>2.25</td>
</tr>
</tbody>
</table>

1.4 < \text{max}\{2.35,2.25\}

B (3rd round)

<table>
<thead>
<tr>
<th>OID</th>
<th>lbscore</th>
<th>ubscore</th>
</tr>
</thead>
<tbody>
<tr>
<td>o2</td>
<td>2.1</td>
<td>2.1</td>
</tr>
<tr>
<td>o7</td>
<td>1.5</td>
<td>2.0</td>
</tr>
<tr>
<td>o3</td>
<td>1.25</td>
<td>1.95</td>
</tr>
<tr>
<td>o1</td>
<td>1.0</td>
<td>2.0</td>
</tr>
<tr>
<td>o4</td>
<td>0.5</td>
<td>1.7</td>
</tr>
</tbody>
</table>

1.5 < \text{max}\{2.0,1.7\}

B (4th round)

<table>
<thead>
<tr>
<th>OID</th>
<th>lbscore</th>
<th>ubscore</th>
</tr>
</thead>
<tbody>
<tr>
<td>o2</td>
<td>2.1</td>
<td>2.1</td>
</tr>
<tr>
<td>o7</td>
<td>1.5</td>
<td>1.9</td>
</tr>
<tr>
<td>o1</td>
<td>1.4</td>
<td>1.5</td>
</tr>
<tr>
<td>o3</td>
<td>1.25</td>
<td>1.45</td>
</tr>
<tr>
<td>o4</td>
<td>0.5</td>
<td>0.8</td>
</tr>
</tbody>
</table>

1.5 \geq \text{max}\{1.5,0.7\}
NRA: observations

- An interesting observation about NRA is that its cost does not grow monotonically with $k$, i.e., it might be cheaper to look for the top-$k$ objects rather than for the top-$(k-1)$ ones!

Example:
- $k = 1$: the winner is $o_2$ ($S(o_2) = 1.2$), since the score of $o_1$ is $S(o_1) = 1.0$ and that of all other objects is $0.6$. NRA has to reach depth $N-1$ to halt.
- $k = 2$: to discover that the top-2 objects are $o_1$ and $o_2$ only 3 rounds are needed.

Concerning instance optimality, it can be shown that NRA is instance-optimal over all DB’s and all algorithms that do not execute random accesses, and its optimality ratio is $m$ (i.e., if NRA halts at depth $X$, then any other algorithm $B$ must read $X$ objects from at least one list).
If exact scores for the top-k objects are needed, the algorithm, which we call NRA*, will work as follows:

1) Run NRA until the top-k objects are determined (i.e., Res is stable)
2) Perform as many sorted accesses as needed until all the partial scores for the objects in Res are retrieved

Note that NRA* will perform at least as many s.a.’s rounds as FA, and possibly many more

FA halts, at depth X, after having seen k objects in all the lists (those in M)
NRA* cannot halt at a depth < X (since for at least one object in Res the exact score is not known); further, there is no guarantee that objects in M are also in Res

It can be easily shown that NRA* is instance-optimal over all DB’s and all algorithms that compute the exact scores and do not execute random accesses, and its optimality ratio is m
The CA algorithm

- CA (Combined Algorithm) [FLN01,FLN03] is an attempt to reduce the negative influence of high r.a. costs
- The idea of CA is simple: rather than performing r.a.’s at each round, just do them only every \( \lceil \frac{c_{RA}}{c_{SA}} \rceil \) rounds (more precisely: \( \lceil \frac{c_{RA}}{c_{SA}} \rceil \))
- In practice CA behaves as NRA (and as NRA keeps lower and upper bounds on objects’ scores), but every \( \lfloor \frac{c_{RA}}{c_{SA}} \rfloor \) s.a. rounds it performs random accesses
- The key point is for which object(s) such r.a.’s have to be invoked
- Not surprisingly, these are done for the object \( o \) that misses some partial scores and for which \( S^+(o) \) is maximum
- Compared to TA, CA will execute more s.a.’s but less r.a.’s
- It can be proved that CA is instance-optimal, with an optimality ratio independent of \( \frac{c_{RA}}{c_{SA}} \), but only if
  1) on each list scores are all distinct (no two objects tie on \( p_j \))
  2) \( S = \text{MIN} \) or \( S \) is strictly monotone in each argument: whenever one \( p_j \) is increased and the others stay unchanged, then the value of the \( S \) increases as well (e.g., \( \text{SUM} \))
The overall picture

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>scoring f. S</th>
<th>Data access</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>B₀</td>
<td>MAX</td>
<td>sorted</td>
<td>instance-optimal</td>
</tr>
<tr>
<td>MaxOptimal</td>
<td>MAX</td>
<td>sorted</td>
<td>instance-optimal</td>
</tr>
<tr>
<td>FA</td>
<td>monotone</td>
<td>sorted and random</td>
<td>cost independent of ( S )</td>
</tr>
<tr>
<td>TA</td>
<td>monotone</td>
<td>sorted and random</td>
<td>instance-optimal</td>
</tr>
<tr>
<td>NRA</td>
<td>monotone</td>
<td>sorted</td>
<td>instance-optimal, wrong scores</td>
</tr>
<tr>
<td>NRA*</td>
<td>monotone</td>
<td>sorted</td>
<td>instance-optimal, exact scores</td>
</tr>
<tr>
<td>CA</td>
<td>monotone</td>
<td>sorted and random</td>
<td>instance-optimal, optimality ratio independent of ( c_{RA}/c_{SA} ) in some cases</td>
</tr>
</tbody>
</table>


Summary on top-k 1-1 join queries

- There are several algorithms to process a top-k 1-1 join query, all of which are based on the assumption that the scoring function $S$ is monotone.

- The simplest case is when $S \equiv \text{MAX}$: the basic $B_0$ algorithm by Fagin can be improved (MaxOptimal) by exploiting principles similar to those applied for k-NN search.

- Algorithm FA is the only one whose stopping condition just considers the local rankings of the objects rather than their partial scores.

- The stopping condition of TA is based on a threshold $T$, which provides an upper bound to the scores of all unseen objects.

- TA is instance-optimal, yet its optimality ratio depends on $c_{RA}/c_{SA}$, the ratio of random access to sorted access costs.

- NRA does not execute random accesses at all.

- CA is a combination of TA and NRA, and it is instance-optimal (with optimality ratio independent of $c_{RA}/c_{SA}$) only for a subset of scoring functions $S$ and a subset of DB’s.
Top-k join queries: the general case

- 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.:

```sql
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.:

```sql
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**

<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>

**Restaurants**

<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>
<tr>
<td>Al vecchio mulino</td>
<td>CheapSleep</td>
<td>Bologna</td>
<td>125</td>
</tr>
<tr>
<td>La tavernetta</td>
<td>RonfRonf</td>
<td>Roma</td>
<td>90</td>
</tr>
<tr>
<td>La tavernetta</td>
<td>Il Riposino</td>
<td>Roma</td>
<td>120</td>
</tr>
<tr>
<td>Tutti a tavola!</td>
<td>La Cascina</td>
<td>Bologna</td>
<td>115</td>
</tr>
<tr>
<td>Tutti a tavola!</td>
<td>La Quiete</td>
<td>Bologna</td>
<td>120</td>
</tr>
<tr>
<td>Tutti a tavola!</td>
<td>CheapSleep</td>
<td>Bologna</td>
<td>140</td>
</tr>
<tr>
<td>Le delizie del palato</td>
<td>La pensioncina</td>
<td>Milano</td>
<td>90</td>
</tr>
<tr>
<td>Le delizie del palato</td>
<td>Dormi Bene!</td>
<td>Milano</td>
<td>110</td>
</tr>
<tr>
<td>Acqua in bocca</td>
<td>RonfRonf</td>
<td>Roma</td>
<td>130</td>
</tr>
<tr>
<td>Acqua in bocca</td>
<td>Il Riposino</td>
<td>Roma</td>
<td>160</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HName</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**

$k = 5$
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:
     2.1) using the index on the join attributes, do r.a.’s to retrieve all the matches on other inputs
     2.2) keep only the best k so-resulting join combinations
     2.3) 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>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>

**k = 2**
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:
    
    $S(p_1^{\text{max}}, p_2^{\text{max}}, \ldots, p_m^{\text{max}}), S(p_1^{\text{max}}, p_2, \ldots, p_m^{\text{max}}), \ldots, S(p_1, p_2^{\text{max}}, \ldots, p_m^{\text{max}})$
  
  - 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.
For simplicity, assume $p_1^{\text{max}} = p_2^{\text{max}} = 1$, and $S \equiv \text{SUM}$.

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

- After the first 3 s.a.’s rounds, the corner bound yields $T = 0.9 + 1 + 1 = 2.9$
- However, no unseen tuple from $L_1$ 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 $L_1$ between $(x,0.9)$ and $(w,0.5)$ is unbounded

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

```
<table>
<thead>
<tr>
<th>A</th>
<th>p1</th>
<th>A</th>
<th>p2</th>
<th>A</th>
<th>p3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1.0</td>
<td>y</td>
<td>1.0</td>
<td>z</td>
<td>1.0</td>
</tr>
<tr>
<td>x</td>
<td>0.95</td>
<td>a</td>
<td>0.7</td>
<td>a</td>
<td>0.8</td>
</tr>
<tr>
<td>x</td>
<td>0.9</td>
<td>y</td>
<td>0.4</td>
<td>z</td>
<td>0.4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>w</td>
<td>0.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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

$S \equiv \text{SUM}$

$S(j) \leq 2.5$
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 dominance”, which is at the core of skyline queries.
Ranking as a first-class concept in a DBMS

- 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/](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:

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

```sql
σ(Stars = 3) AND (Cuisine = 'chinese')
AND (H.Area = R.Area)
```

P2

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!
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 \( S(p_1,p_2,...,p_m) \), the rank-relation \( R_P \), with \( P \subseteq \{p_1,p_2,...,p_m\} \), 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 \( P = \{p_1,p_2,...,p_m\} \), 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 $R_{p \cup \{p\}}$.
- 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:
  \[
  \mu_p(\sigma_c(R_P)) = \sigma_c(\mu_p(R_P))
  \]
- Because of the ranking principle, a $\mu_p$ operator can return a tuple $t$ iff it is guaranteed that there is no unseen tuple $t'$ such that $S_{P \cup \{p\}}^+(t') > S_{P \cup \{p\}}^+(t)$.
- This can be done as soon as $\mu_p$ fetches a tuple $t''$ such that $S_{P \cup \{p\}}^+(t) \geq S_{P \cup \{p\}}^+(t'')$.
- 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 $p$. 

The rank operator in action

<table>
<thead>
<tr>
<th>Hotel</th>
<th>...</th>
<th>p1</th>
<th>p2</th>
<th>p3</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>h1</td>
<td></td>
<td>0.7</td>
<td>0.8</td>
<td>0.9</td>
<td>2.4</td>
</tr>
<tr>
<td>h2</td>
<td></td>
<td>0.9</td>
<td>0.85</td>
<td>0.8</td>
<td>2.55</td>
</tr>
<tr>
<td>h3</td>
<td></td>
<td>0.5</td>
<td>0.45</td>
<td>0.75</td>
<td>1.7</td>
</tr>
<tr>
<td>h4</td>
<td></td>
<td>0.4</td>
<td>0.7</td>
<td>0.95</td>
<td>2.05</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Query:**

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

**Diagram:**

\[ H_{p1, p2, p3} \]

\[ H_{p1, p2} \]

\[ H_{p1} \]

\[ \text{IdxScan}_{p1} \]

\[ \text{STOP AFTER 1} \]

\[ S_{\{p1,p2,p3\}^+} = S \]

\[ S_{\{p1,p2\}^+} \]

\[ S_{\{p1\}^+} \]

\[ \tau_1 \]

\[ \mu_{p3} \]

\[ \mu_{p2} \]
The rank operator in action

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

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>0.9</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>

\[
H_{p1, p2, p3}
\]

\[
H_{p1, p2}
\]

\[
H_{p1}
\]

\[
\text{SELECT * FROM HOTELS H ORDER BY } p1+p2+p3 \text{ DESC STOP AFTER 1}
\]
The rank operator in action

```
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>...</th>
<th>p1</th>
<th>p2</th>
<th>p3</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>h1</td>
<td></td>
<td>0.7</td>
<td>0.8</td>
<td>1.0</td>
<td>2.5</td>
</tr>
<tr>
<td>h2</td>
<td></td>
<td>0.9</td>
<td>0.85</td>
<td>1.0</td>
<td>2.75</td>
</tr>
<tr>
<td>h3</td>
<td></td>
<td>0.7</td>
<td>1.0</td>
<td>1.0</td>
<td>2.7</td>
</tr>
<tr>
<td>h4</td>
<td></td>
<td>0.7</td>
<td>1.0</td>
<td>1.0</td>
<td>2.7</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>0.7</td>
<td>1.0</td>
<td>1.0</td>
<td>2.7</td>
</tr>
</tbody>
</table>

\[ \text{SELECT * FROM HOTELS H ORDER BY p1+p2+p3 DESC STOP AFTER 1} \]

\[ \text{IdxScan}_{p1} \]

\[ \text{\( H_{p1, p2, p3} \)} \]

\[ \text{H_{p1, p2}} \]

\[ \text{H_{p1}} \]

\[ \text{\( \tau_1 \)} \]

\[ \text{\( \mu_{p3} \)} \]

\[ \text{\( \mu_{p2} \)} \]

\[ \text{\( \text{S}^+ \)} \]

\( h2 \)

\( h1 \text{ 2.5 (*)} \)

\( h2 \text{ 2.9} \)

\( h1 \text{ 2.7} \)

(* Evaluation of p2 can be delayed
The rank operator in action

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

\[
\begin{align*}
\tau_1 & \leftarrow \mu_{p3} \\
\mu_{p2} & \leftarrow \mu_{p3} \\
\text{IdxScan}_{p1} & \leftarrow \text{HOTELS}
\end{align*}
\]

```
SELECT * 
FROM HOTELS H
ORDER BY p1+p2+p3 DESC
STOP AFTER 1
```
### The rank operator in action

**Table:**

<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>

**Diagram:**

\[
\begin{align*}
\text{H}_{p1, p2, p3} &: \begin{cases} 
    \text{Hotel} & S^+ \\
    h2 & 2.55 \\
\end{cases} \\
\text{H}_{p1, p2} &: \begin{cases} 
    \text{Hotel} & S^+ \\
    h2 & 2.75 \\
    h1 & 2.5 \\
\end{cases} \\
\text{H}_{p1} &: \begin{cases} 
    \text{Hotel} & S^+ \\
    h2 & 2.9 \\
    h1 & 2.7 \\
    h3 & 2.5 \\
\end{cases}
\end{align*}
\]

**SQL:**

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

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

(*) 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>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>

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

(∗) Evaluation of p3 can be omitted
The rank operator in action

<table>
<thead>
<tr>
<th>Hotel</th>
<th>...</th>
<th>p1</th>
<th>p2</th>
<th>p3</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>h1</td>
<td></td>
<td>0.7</td>
<td>0.8</td>
<td>0.9</td>
<td>2.4</td>
</tr>
<tr>
<td>h2</td>
<td></td>
<td>0.9</td>
<td>0.85</td>
<td>0.8</td>
<td>2.55</td>
</tr>
<tr>
<td>h3</td>
<td></td>
<td>0.5</td>
<td>0.45</td>
<td>1.0</td>
<td>1.95</td>
</tr>
<tr>
<td>h4</td>
<td></td>
<td>0.5</td>
<td>1.0</td>
<td>1.0</td>
<td>2.5</td>
</tr>
<tr>
<td>...</td>
<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
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.
With rank-aware operators several alternative plans are possible.
Experimental 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

(a) \( s = 100,000, j = 0.0001, c = 1 \)

(b) \( k = 10, s = 100,000, j = 0.0001 \)

(c) \( k = 10, s = 100,000, c = 1 \)

(d) \( k = 10, j = 0.00001, c = 1 \)
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:

```
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.

```sql
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.
Summary on top-k M-N join queries

- 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 intearleaving” 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.
References


[LCI+05] Chengkai Li, Kevin Chen-Chuan Chang, Ihab F. Ilyas, Sumin Song: RankSQL: Query Algebra and Optimization for Relational Top-k Queries. SIGMOD Conference 2005: 131-142

[SP08] Karl Schnaitter, Neoklis Polyzotis: Evaluating rank joins with optimal cost. PODS 2008: 43-52