Top-k queries

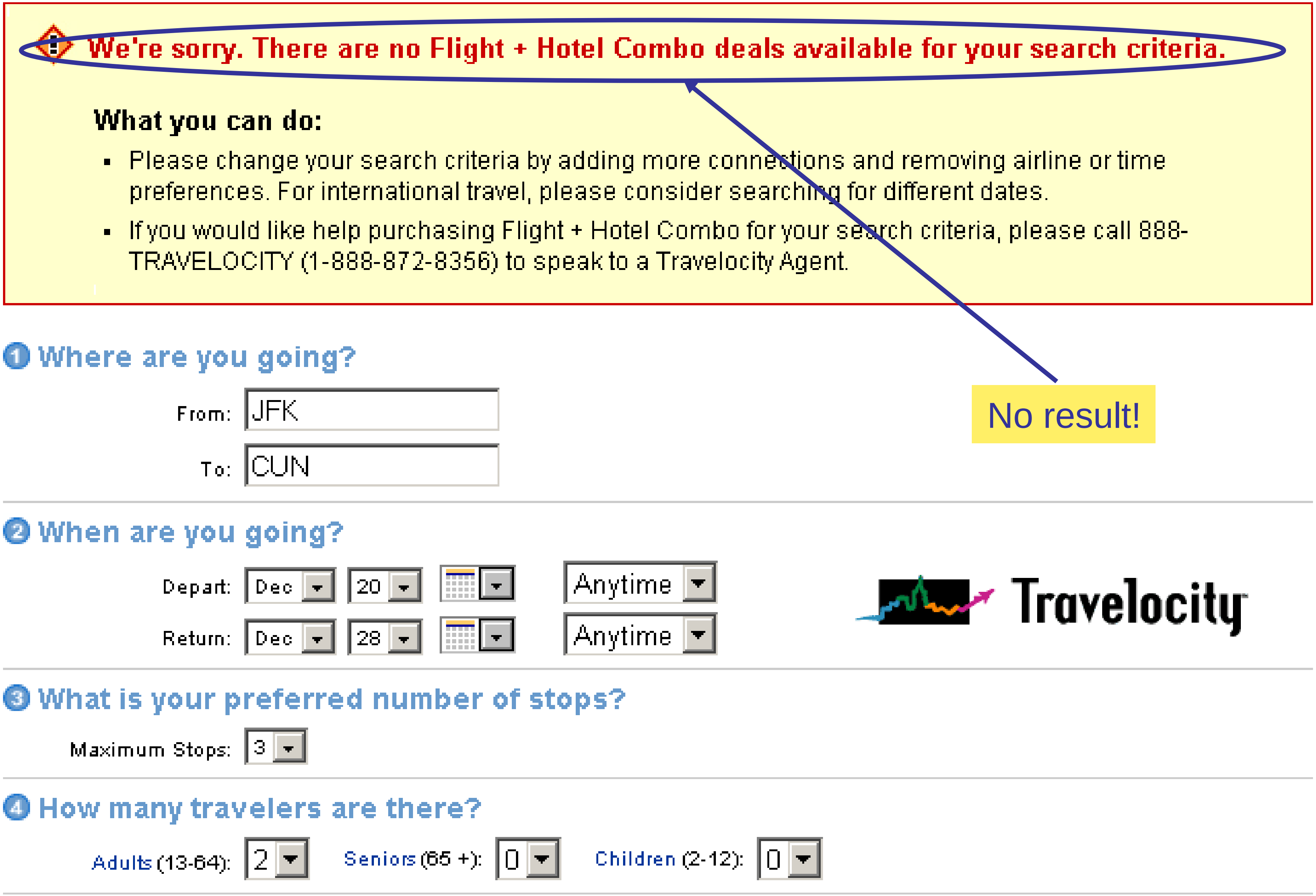
Tecnologie e Sistemi per la Gestione di Basi di Dati e Big Data M

Top-k queries: what and why

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

- Best = most important/interesting/relevant/…
- The need for suck 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 Anytime

Return: Dec 28 Anytime

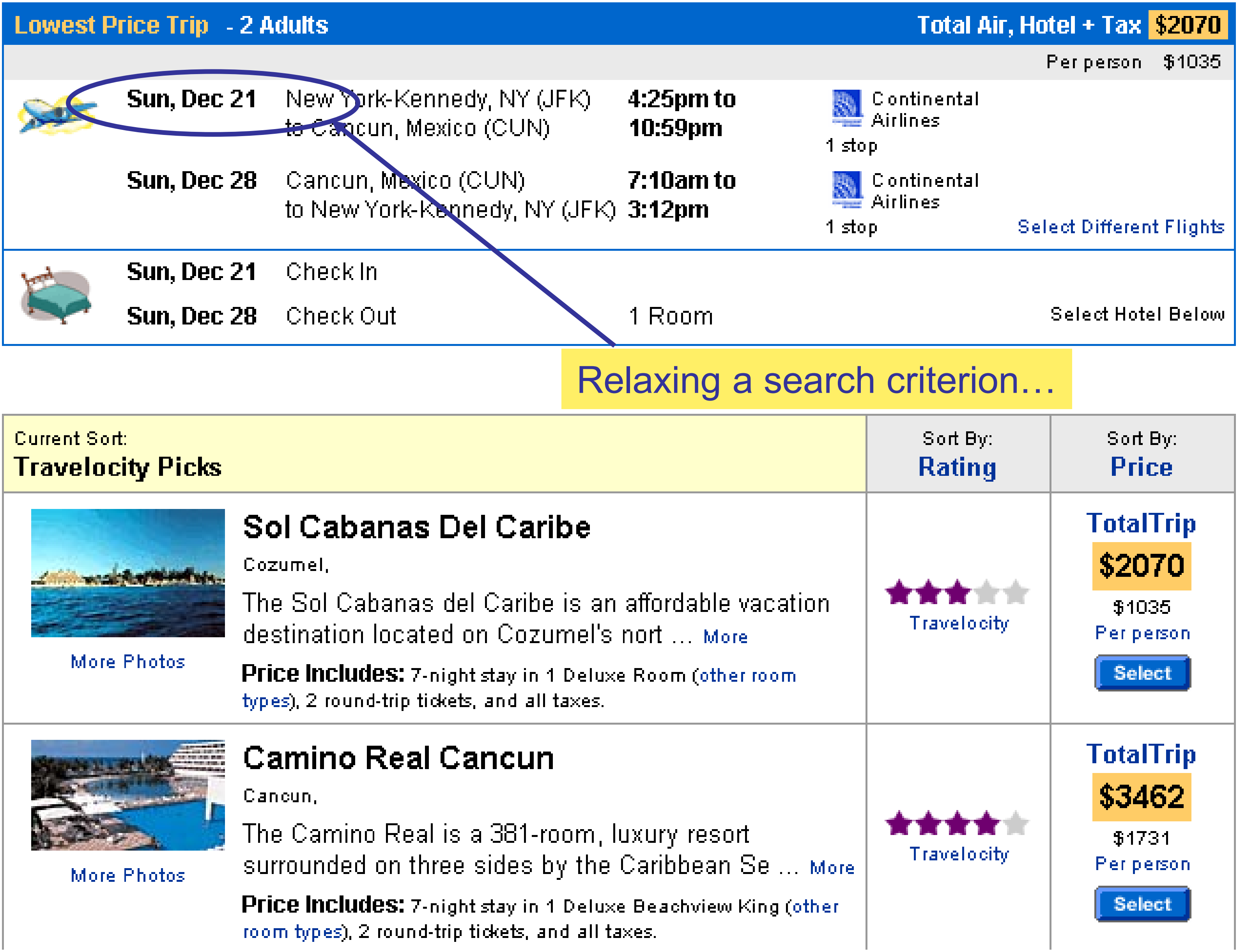
3 What is your preferred number of stops?

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 Total Air, Hotel + Tax $2070

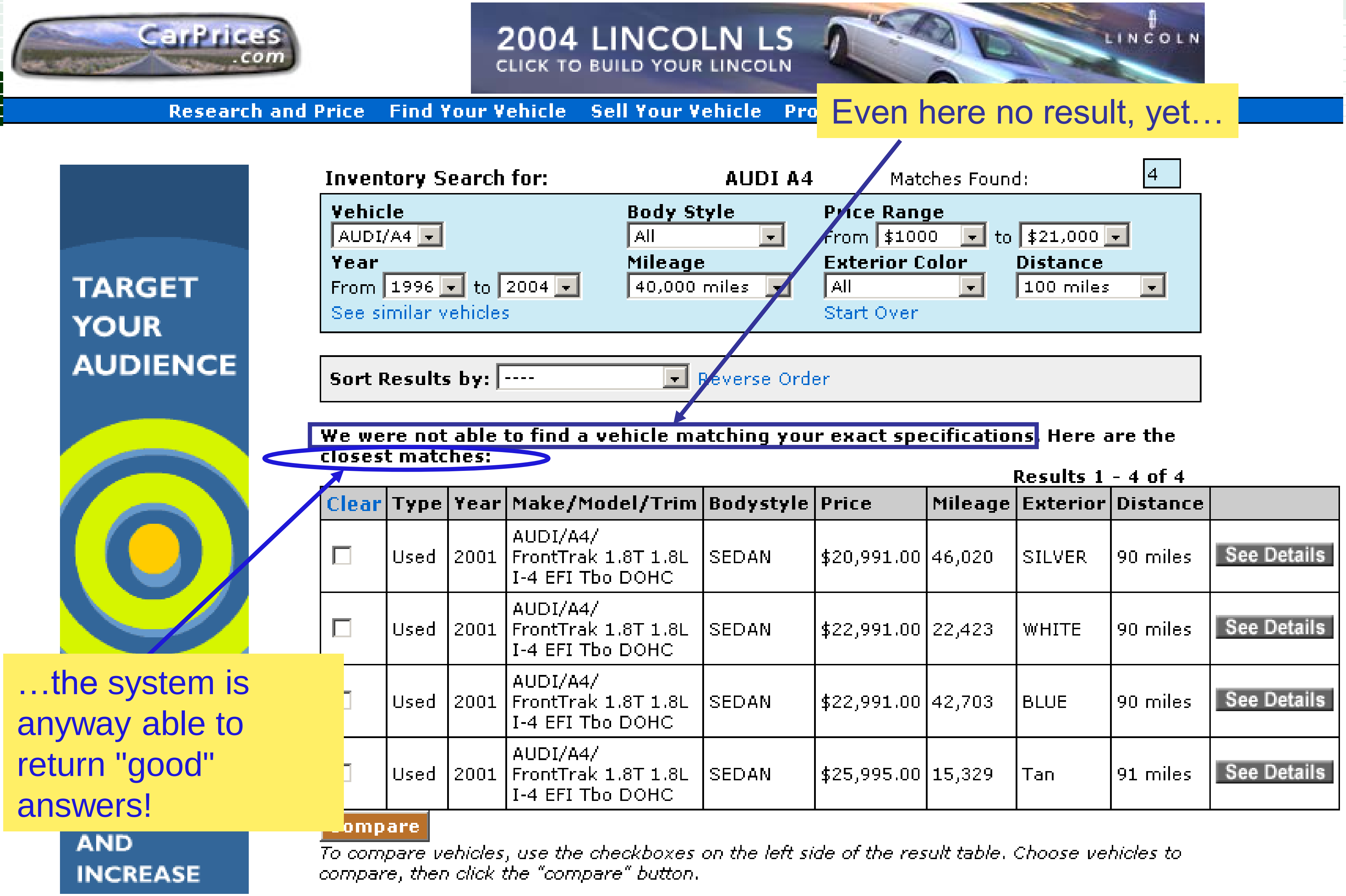
Per person $1035

Sun, Dec 21	New York-Kennedy, NY (JFK) to Cancun, Mexico (CUN)	4:25pm to 10:59pm	Continental Airlines 1 stop	
Sun, Dec 28	Cancun, Mexico (CUN) to New York-Kennedy, NY (JFK)	7:10am to 3:12pm	Continental Airlines 1 stop	Select Different Flights

Sun, Dec 21	Check In		
Sun, Dec 28	Check Out	1 Room	Select Hotel Below

Relaxing a search criterion…

Current Sort: Travelocity Picks	Sort By: Rating	Sort By: Price
Sol Cabanas Del Caribe Cozumel, The Sol Cabanas del Caribe is an affordable vacation destination located on Cozumel's nort … More More Photos **Price Includes:** 7-night stay in 1 Deluxe Room (other room types), 2 round-trip tickets, and all taxes.	★★★☆☆ Travelocity	TotalTrip $2070 $1035 Per person Select
Camino Real Cancun Cancun, The Camino Real is a 381-room, luxury resort surrounded on three sides by the Caribbean Se … More More Photos **Price Includes:** 7-night stay in 1 Deluxe Beachview King (other room types), 2 round-trip tickets, and all taxes.	★★★★☆ Travelocity	TotalTrip $3462 $1731 Per person Select



Research and Price Find Your Vehicle Sell Your Vehicle Pro

Inventory Search for: **AUDI A4** Matches Found: 4

Vehicle AUDI/A4	**Body Style** All	**Price Range** From $1000 to $21,000	
Year From 1996 to 2004	**Mileage** 40,000 miles	**Exterior Color** All	**Distance** 100 miles
See similar vehicles		Start Over	

Sort Results by: ---- Reverse Order

We were not able to find a vehicle matching your exact specifications. Here are the closest matches:

Results 1 - 4 of 4

Clear	Type	Year	Make/Model/Trim	Bodystyle	Price	Mileage	Exterior	Distance	
☐	Used	2001	AUDI/A4/ FrontTrak 1.8T 1.8L I-4 EFI Tbo DOHC	SEDAN	$20,991.00	46,020	SILVER	90 miles	See Details
☐	Used	2001	AUDI/A4/ FrontTrak 1.8T 1.8L I-4 EFI Tbo DOHC	SEDAN	$22,991.00	22,423	WHITE	90 miles	See Details
☐	Used	2001	AUDI/A4/ FrontTrak 1.8T 1.8L I-4 EFI Tbo DOHC	SEDAN	$22,991.00	42,703	BLUE	90 miles	See Details
☐	Used	2001	AUDI/A4/ FrontTrak 1.8T 1.8L I-4 EFI Tbo DOHC	SEDAN	$25,995.00	15,329	Tan	91 miles	See Details

ompare

To compare vehicles, use the checkboxes on the left side of the result table. Choose vehicles to compare, then click the "compare" button.

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.Points + t.Rebounds

Name	Points	Rebounds	…
Shaquille O'Neal	1669	760	…
Tracy McGrady	2003	484	…
Kobe Bryant	1819	392	…
Yao Ming	1465	669	…
Dwyane Wade	1854	397	…
Steve Nash	1165	249	…
…	…	…	…

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

$$S(t) = t.Points + t.Rebounds$$

Name	Points	…
Shaquille O'Neal	1669	…
Tracy McGrady	2003	…
Kobe Bryant	1819	…
Yao Ming	1465	…
Dwyane Wade	1854	…
Steve Nash	1165	…
…	…	…

Name	Rebounds	…
Shaquille O'Neal	760	…
Tracy McGrady	484	…
Kobe Bryant	392	…
Yao Ming	669	…
Dwyane Wade	397	…
Steve Nash	249	…
…	…	…

- 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)

Top-k queries in SQL (standard)

- 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:

```
SELECT     <some attributes>
FROM       R
WHERE      <Boolean conditions>
ORDER BY   S(…) [DESC]
```

Limits of the ORDER BY solution

- Consider the following queries:

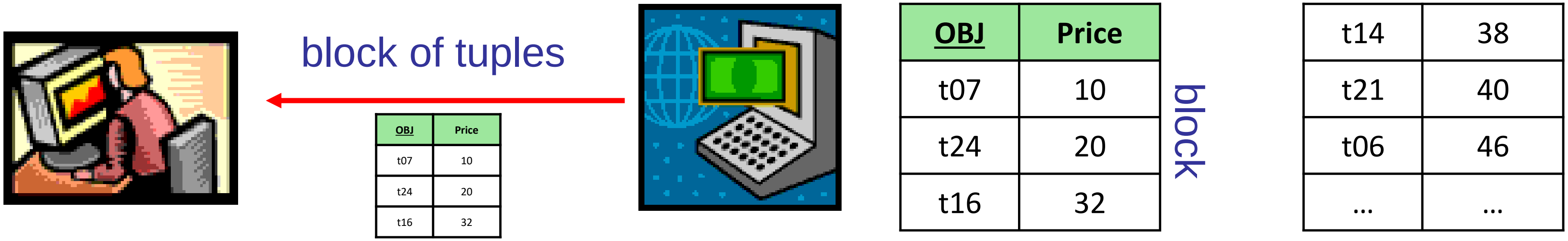
```
A) SELECT      *
   FROM        UsedCarsTable
   WHERE       Vehicle = 'Audi/A4' AND Price <= 21000
   ORDER BY    0.8*Price + 0.2*Mileage
B) SELECT      *
   FROM        UsedCarsTable
   WHERE       Vehicle = 'Audi/A4'
   ORDER BY    0.8*Price + 0.2*Mileage
```

  - The values 0.8 and 0.2, also called "weights", are a way to normalize ranges and/or express 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!!

ORDER BY solution & C/S architecture (1)

- 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



OBJ	Price
t07	10
t24	20
t16	32

OBJ	Price
t07	10
t24	20
t16	32

t14	38
t21	40
t06	46
…	…

ORDER BY solution & C/S architecture (2)

- 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*:

```
 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` or `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)

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

R

OBJ	Price
t15	50
t24	40
t26	30
t14	30
t21	40

OBJ	Price
t26	30
t14	30
t21	**40**

OBJ	Price
t26	30
t14	30
t24	**40**

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):

```
SELECT *
FROM   NBA
ORDER BY Points + Rebounds DESC
STOP AFTER 1
```

- The 2 cheapest chinese restaurants

```
SELECT *
FROM   RESTAURANTS
WHERE  Cuisine = `chinese’
ORDER BY Price
STOP AFTER 2
```

- The top-5% highest paid employees

```
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)

```
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

```
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

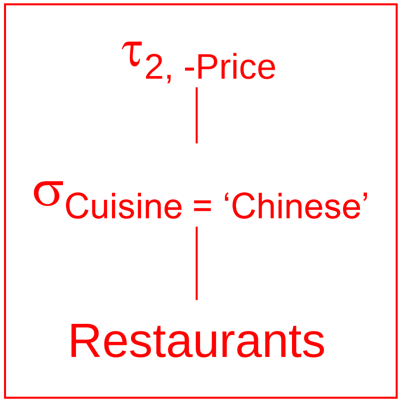
- Concerning evaluation, there are two basic aspects to consider:
  - query type: 1 relation, many relations, aggregate results, …
  - access paths: no index, indexes on all/some ranking attributes

- 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 $\tau_{k,S}$, which returns the k top-ranked tuples according to S
  - Unless otherwise specified, we assume that S has to be maximized



$\tau_{2,\ -Price}$

$\sigma_{Cuisine\ =\ \text{‘Chinese’}}$

Restaurants

  - 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
    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
    invoke open on the child node;
    return.
```

The Top-Sort physical operator: next method

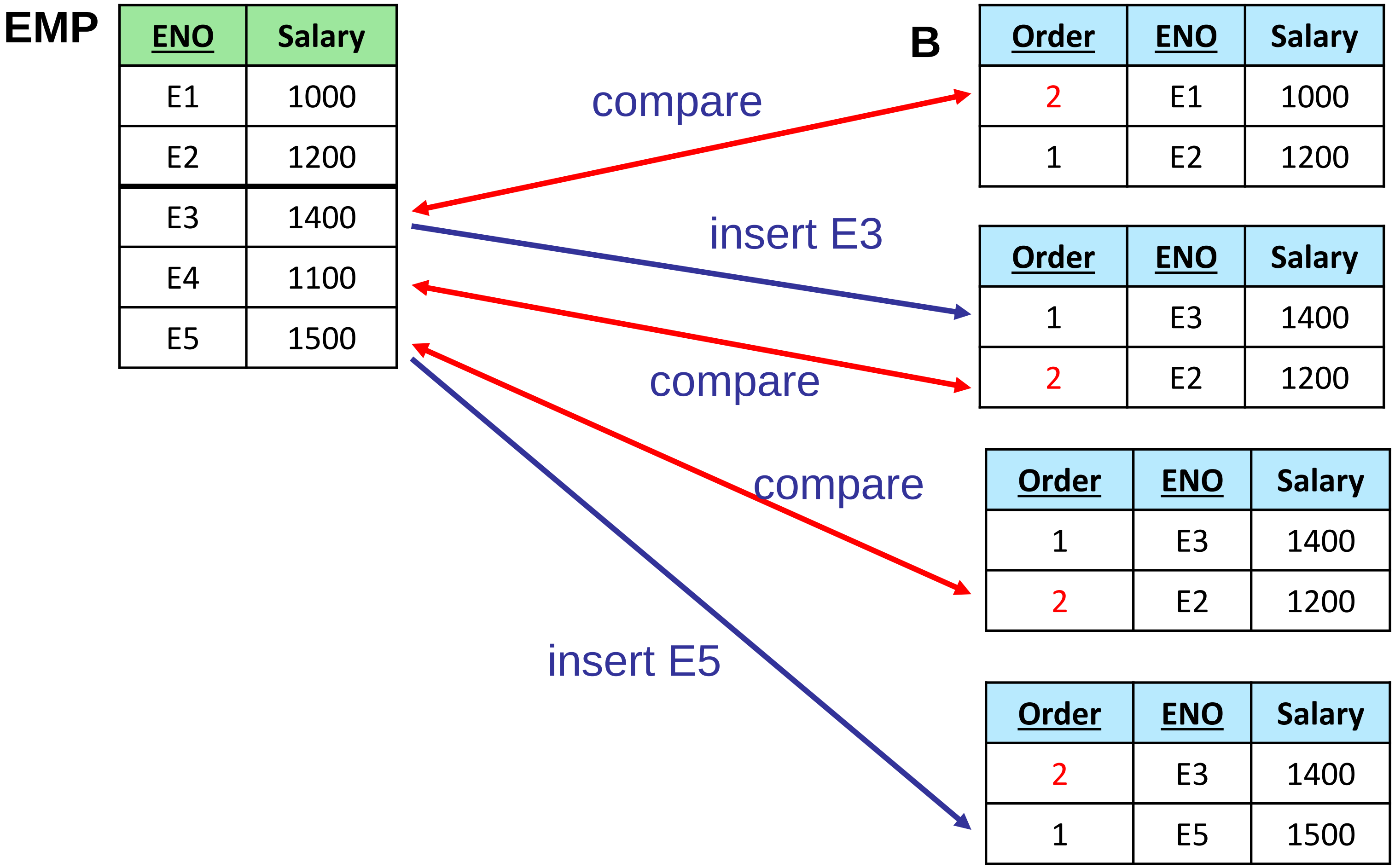
- The next method initially fills B with the first k tuples
  - For simplicity, the pseudocode does not consider the case when the input has less than k tuples
- Then, for each new read tuple t, it compares t with B[k], the worst tuple currently in B
  - If S(t) > B[k].score, then B[k] is dropped and 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

```
for i=1 to k:                                // fills B with the first k tuples
      t := input_node.next(); ENQUEUE(B,t);  // inserts t in B
while (input_node.has_next()) do:
      t := input_node.next();
      if S(t) > B[k].score then: {DELETE(B,B[k]); ENQUEUE(B,t)};
return DEQUEUE(B).                           // returns the best tuple in B
```

Top-Sort: a simple example

- Let k = 2



EMP

ENO	Salary
E1	1000
E2	1200
E3	1400
E4	1100
E5	1500

B

Order	ENO	Salary
2	E1	1000
1	E2	1200

Order	ENO	Salary
1	E3	1400
2	E2	1200

Order	ENO	Salary
1	E3	1400
2	E2	1200

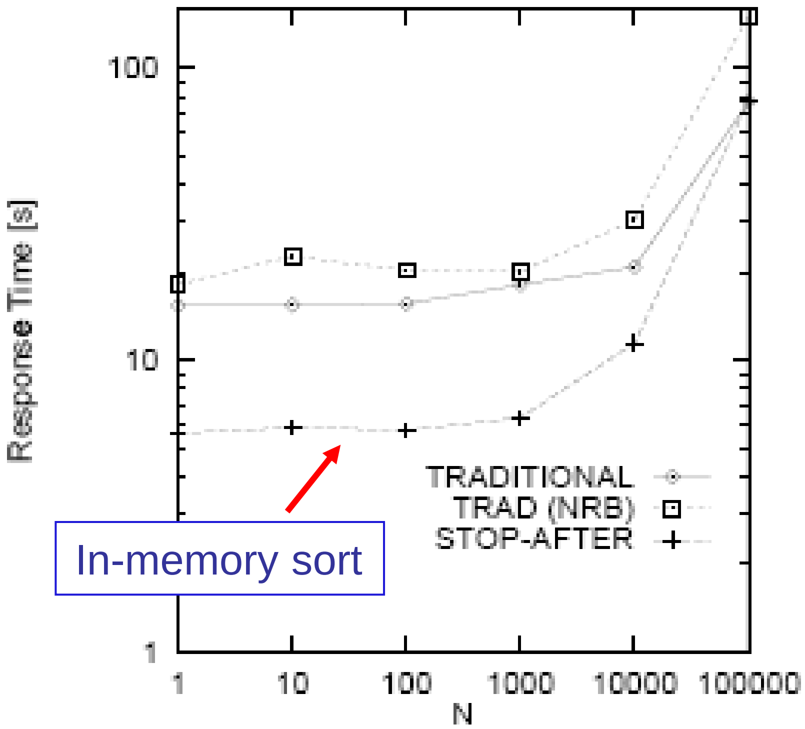
Order	ENO	Salary
2	E3	1400
1	E5	1500

Experimental results from [CK97] (1)

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

No Index available

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



In-memory sort

Figure 6: Resp. Time (log), Q1 No Index on Emp.salary.

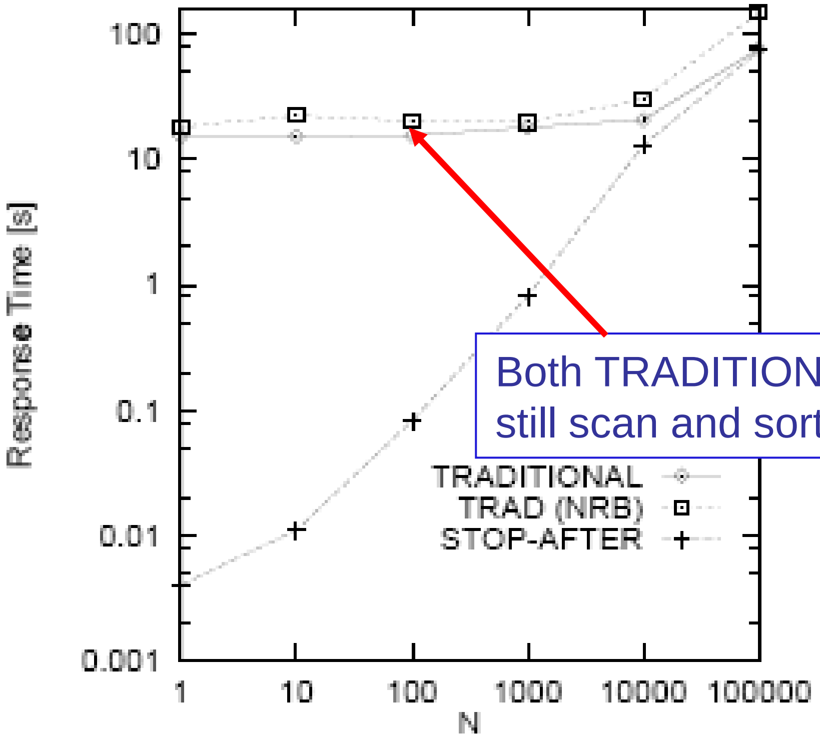
The naïve method sorts ALL the tuples!

Results from [CK97] (2)

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

Unclustered Index on Emp.Salary

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

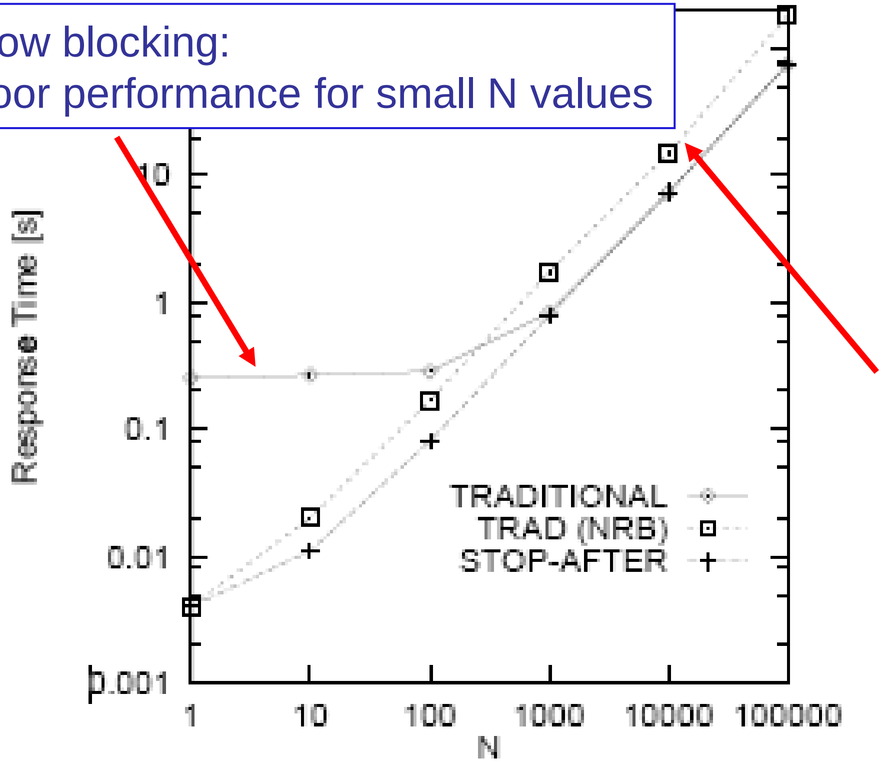
Figure 5: Resp. Time (log), Q1

Unclustered Index on Emp.salary.

Results from [CK97] (3)

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

Clustered Index on Emp.Salary



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:

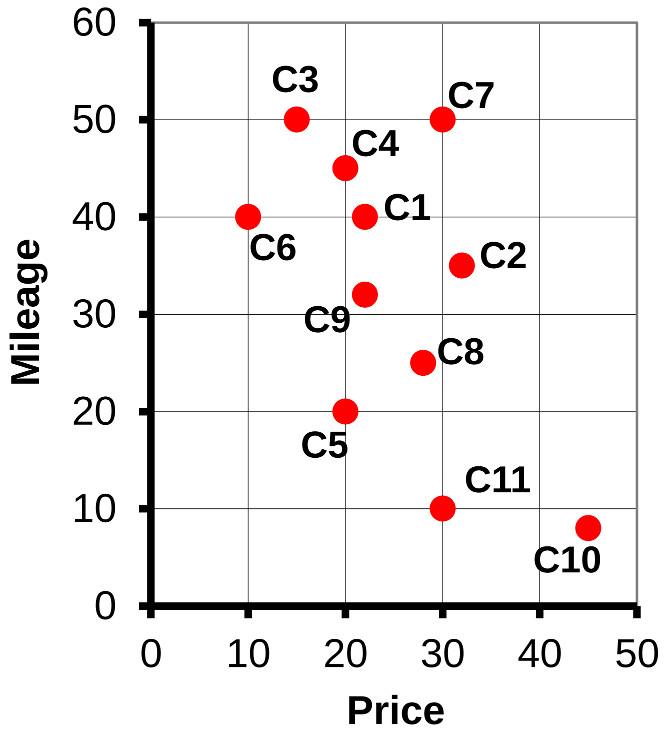
```
SELECT *
FROM   USEDCARS
WHERE  Vehicle = 'Audi/A4'
ORDER BY 0.8*Price + 0.2*Mileage
STOP AFTER 5;
```

- 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)



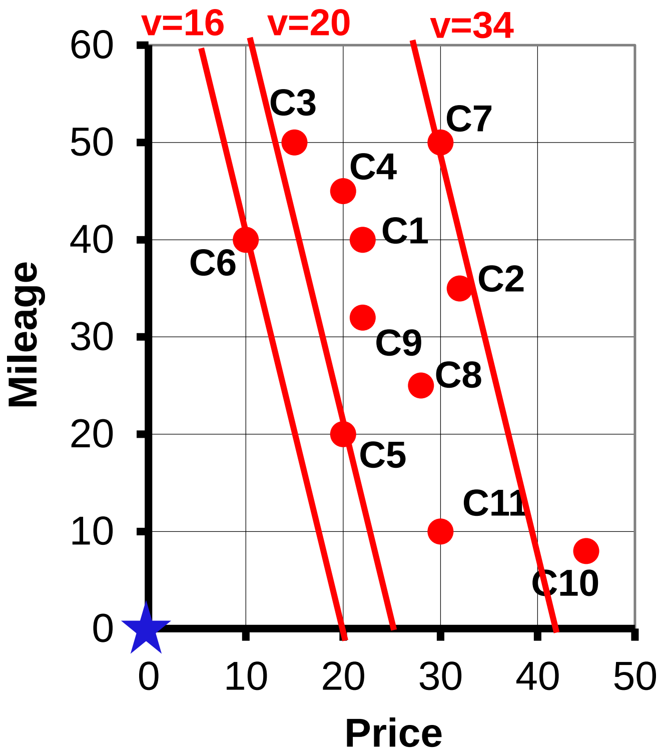
- Each tuple is represented by a 2-dim point (p,m):
  - p is the Price value
  - m is the Mileage value
- Intuitively, minimizing

  `0.8*Price + 0.2*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



- Consider the line l(v) of equation

  `0.8*Price + 0.2*Mileage = v`

  where v is a constant
- This can also be written as

  `Mileage = -4*Price + 5*v`

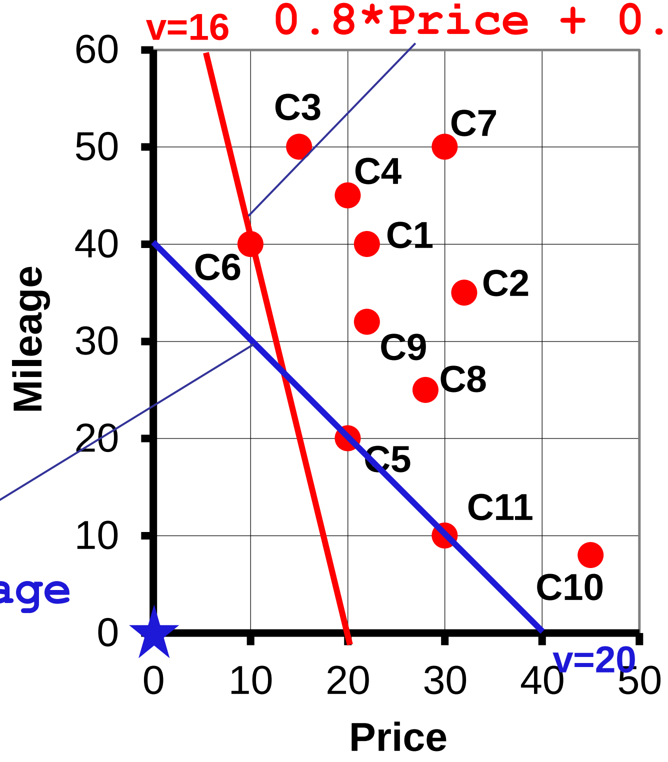
  from which we see that all the lines l(v) have a slope = -4
- By definition, all the points of l(v) are “equally good” to us

- With preferences (0.8,0.2) the best car is C6, then C5, etc.
- In general, preferences are a way to determine, given points (p1,m1) and (p2,m2), which of them is "closer" to the target point (0,0)

Changing the weights

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

- With
  0.8*Price + 0.2*Mileage
  the best car is C6
- With
  0.5*Price + 0.5*Mileage
  the best cars are C5 and C11



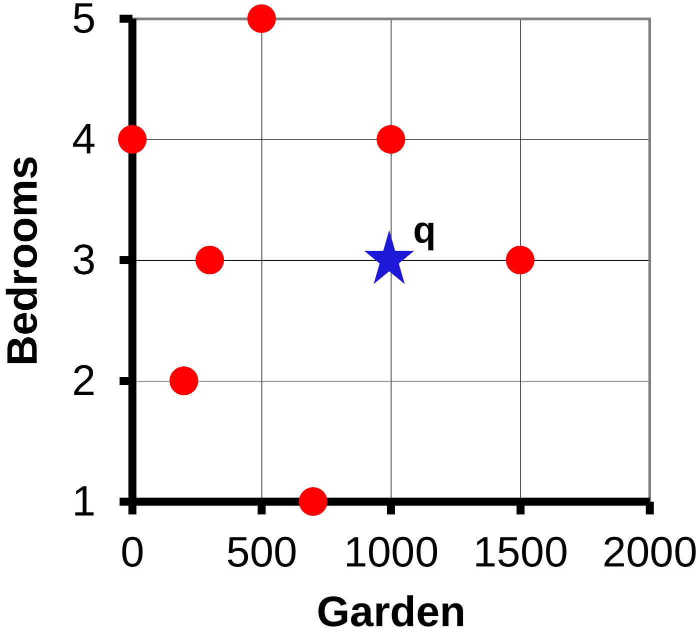
0.8*Price + 0.2*Mileage

0.5*Price + 0.5*Mileage

- 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=(q1,q2) (qi = 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 ($D \geq 1$) attribute space **A** = (A1,A2,…,AD) of ranking attributes
  - A relation R(A1,A2,…,AD,B1,B2,…), where B1,B2,… are other attributes
  - A target (query) point q = (q1,q2,…,qD), $q \in$ **A**
  - A function d: **A** x **A** $\rightarrow \Re$, 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 Lp-norms:

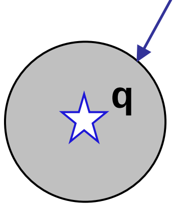
$$L_p(t,q) = \left(\sum_{i=1}^{D} |t_i - q_i|^p \right)^{1/p}$$

- Relevant cases are:

Iso-distance (hyper-)surfaces

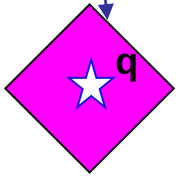
$$\mathbf{L_2(t,q)} = \sqrt{\sum_{i=1}^{D} |\mathbf{t_i} - \mathbf{q_i}|^2}$$

Euclidean distance



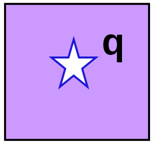
$$\mathbf{L_1(t,q)} = \sum_{i=1}^{D} |\mathbf{t_i} - \mathbf{q_i}|$$

Manhattan (city – block) distance



$$\mathbf{L_\infty(t,q)} = \mathbf{max}_i \{|\mathbf{t_i} - \mathbf{q_i}|\}$$

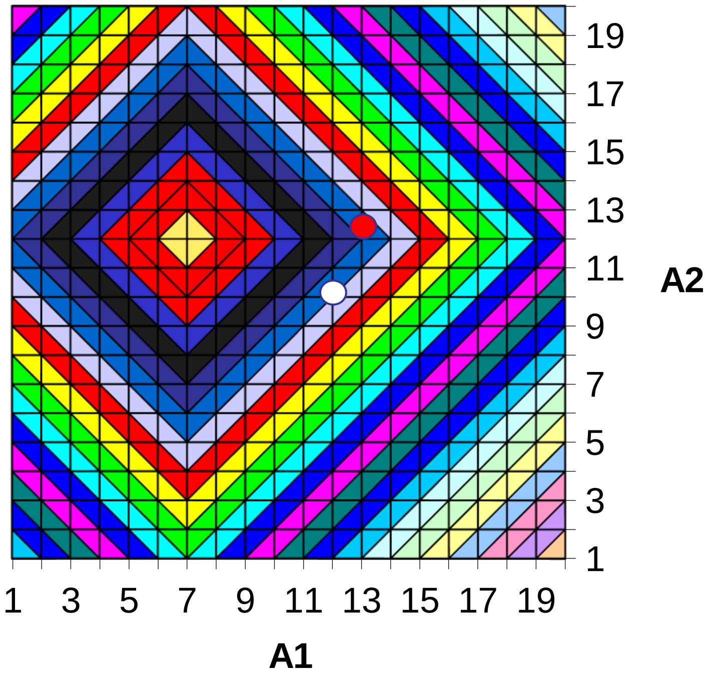
Chebyshev (max) distance



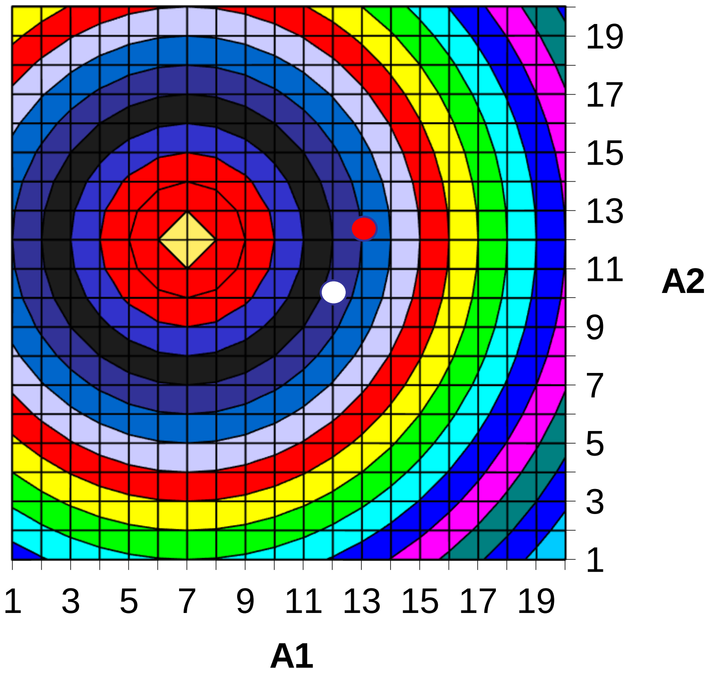
Shaping the attribute space

- 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)

L1; q=(7,12)



L2; q=(7,12)



Note that, for 2 tuples t1 and t2, it is possible to have L1(t1,q) < L1(t2,q) and L2(t2,q) < L2(t1,q)

E.g.: t1=(13,12) ●
t2=(12,10) ○

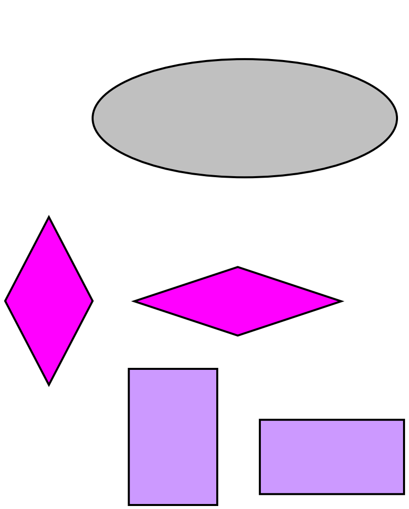
Distance functions with weights

- The use of weights just leads to "stretch" some of the coordinates:

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

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

$$L_\infty(t,q;W) = \max_i \left\{ w_i \left|t_i - q_i\right| \right\}$$



(hyper-)ellipsoids

(hyper-)romboids

(hyper-)rectangles

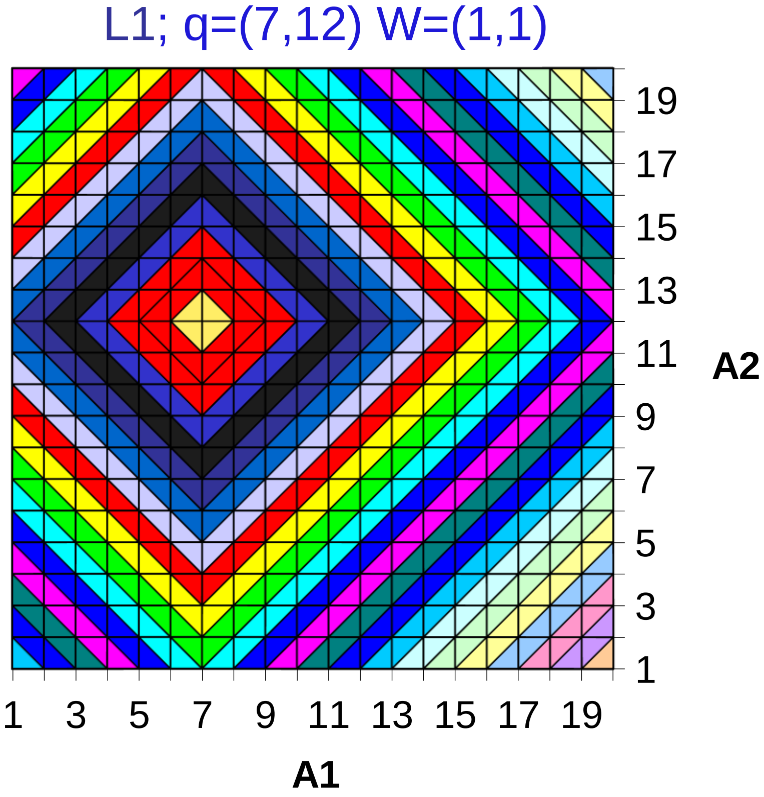
- Thus, the scoring function

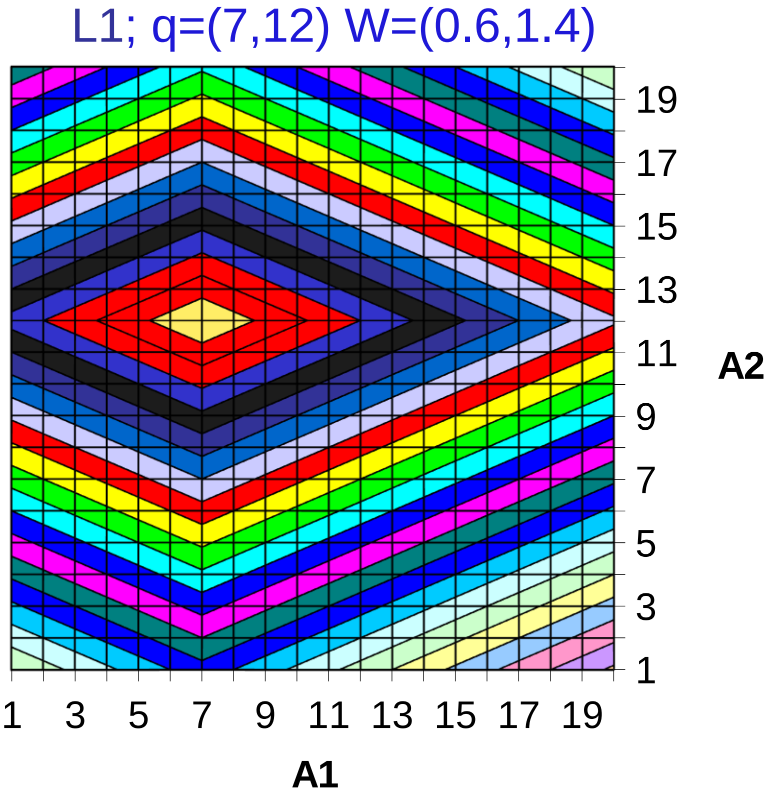
  `0.8*Price + 0.2*Mileage`

  is just a particular case of weighted L1distance

Shaping with weights the attribute space

- The figures show the effects of using L1 with different weights





- Note that, if w2 > w1, 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 has the form E=(tuple-key,RID), where tuple-key is the "spatial key" (position) of the tuple whose address is RID

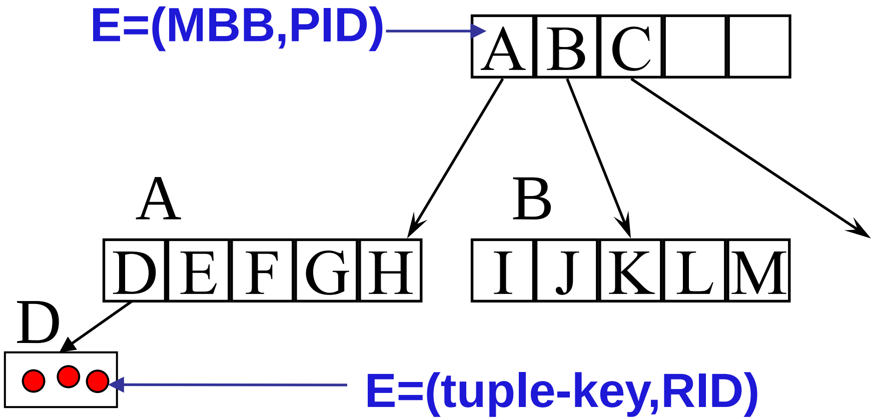
Internal node:

- An entry E has the form E=(MBB,PID), where MBB is the "Minimum Bounding Box" (i.e., 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=(key,ptr)

- If N is the node pointed by E.ptr, then E.key is the "spatial key" of N, also denoted as **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 $\Re^D$ defined as $Reg(q) = \{p: p \in \Re^D, d(p,q) \leq r\}$ 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 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_∞)
- 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 Reg(N) intersects Reg(q)
  - On leaf nodes, check for each entry E if E.key $\in$ Reg(q), that is, if $d(E.key,q) \leq r$.

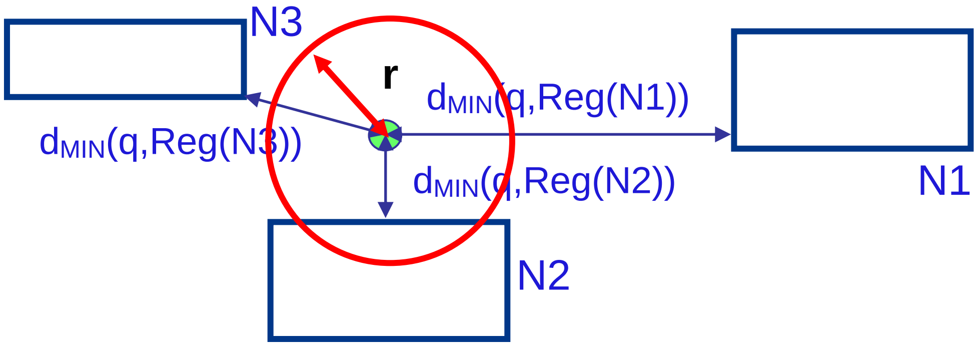
The d_{MIN} lower bound

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

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

- We can make the following basic observation:

$$\mathbf{Reg(q) \cap Reg(N) \neq \emptyset}$$
$$\Leftrightarrow$$
$$\mathbf{d_{MIN}(q,Reg(N)) \leq r}$$



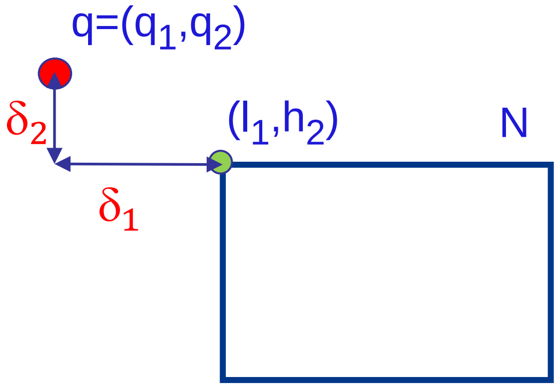
Computing d_{MIN} for weighted Lp norms

- Computing d_{MIN} for a weighted Lp norm has complexity O(D)
- Let the MBB of node N (i.e., Reg(N)) be: Reg(N) = $[l_1,h_1]x\ldots x[l_D,h_D]$
- For the i-th coordinate let us define the "offset" of query $q = (q_1,\ldots,q_D)$ with respect to Reg(N) as:

$$\delta_i = \begin{cases} q_i - h_i & \text{if } q_i \geq h_i \\ l_i - q_i & \text{if } l_i \geq q_i \\ 0 & \text{otherwise} \end{cases}$$

- Then, d_{MIN} is computed as:

$$L_{p,MIN}(q, Reg(N); W) = \left(\sum_{i=1}^{D} w_i \delta_i^p\right)^{1/p}$$



Search: k-NN query

- 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 [ptr(N), d_{MIN}(q,Reg(N))]
- PQ is ordered by *increasing values of* d_{MIN}(q,Reg(N))
  - DEQUEUE(PQ) extracts from PQ the pair with minimal MinDist
  - ENQUEUE(PQ, [ptr(N), d_{MIN}(q,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_{MIN}(q,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 [ptr(N), d_{MIN}(q,Reg(N))] is in PQ
- Intuitively, kNNOptimal performs a "*range search with a variable (shrinking) search radius*" until no improvement is possible anymore

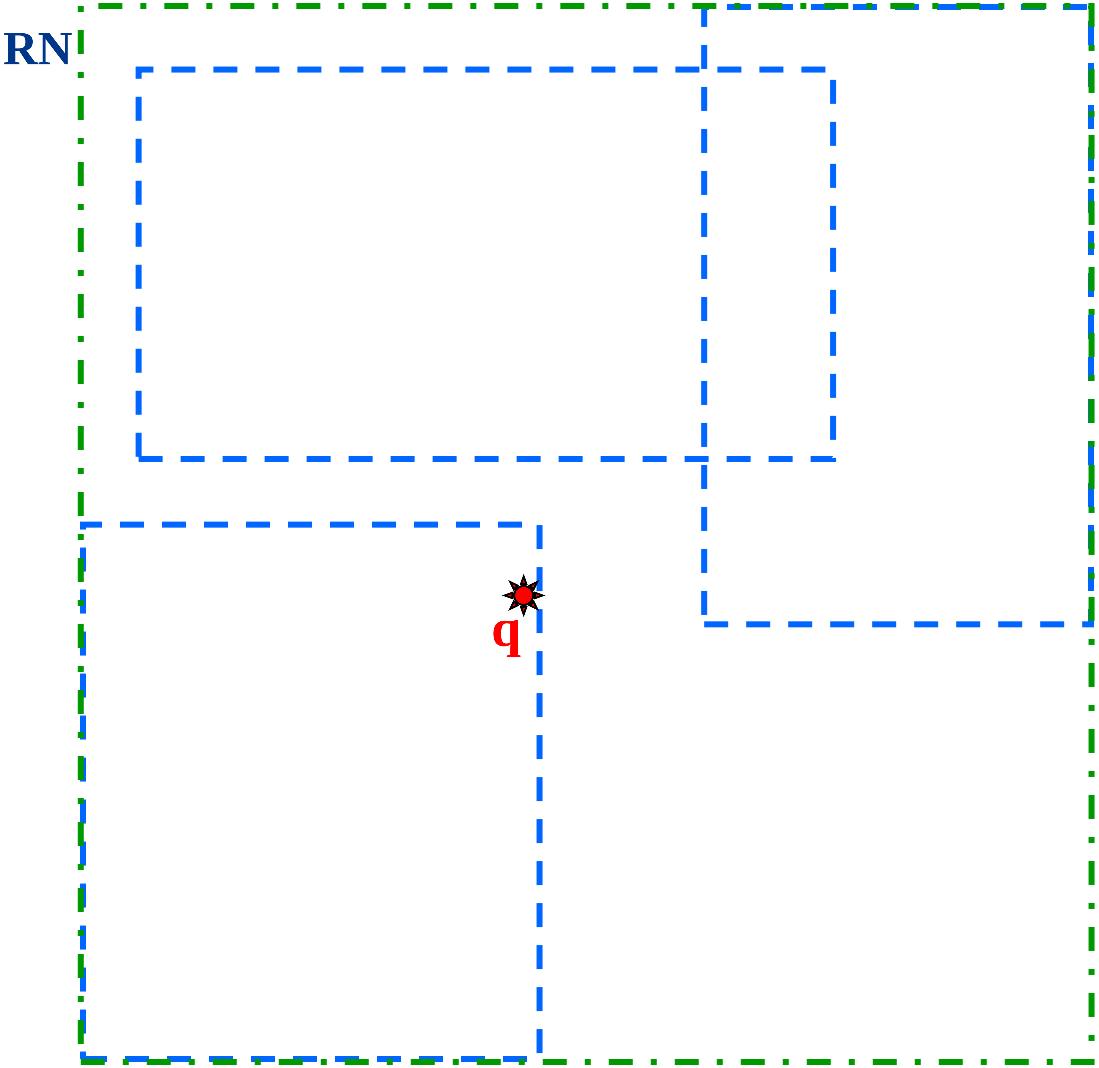
The kNNOptimal Algorithm (case k=1)

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

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

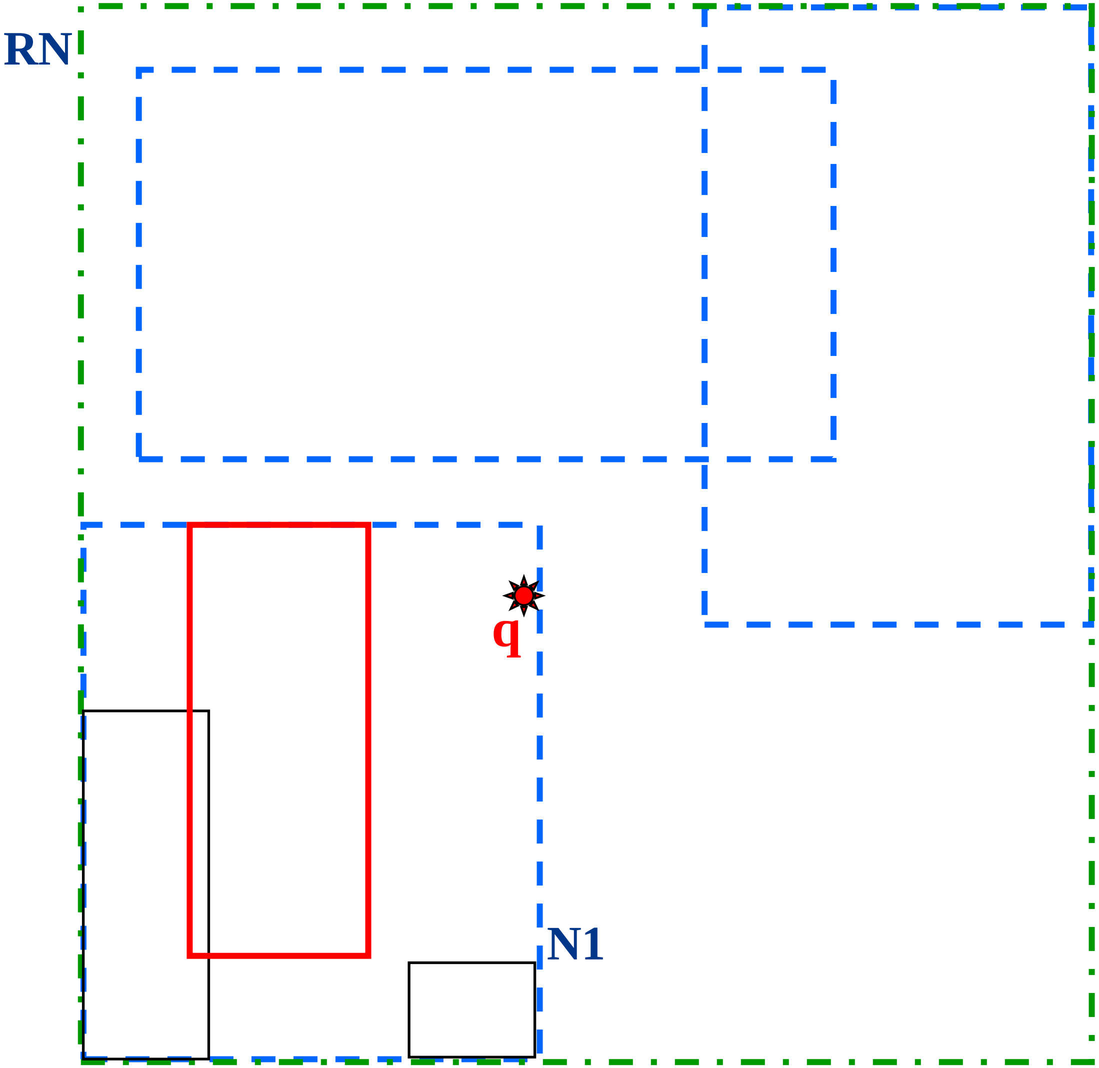
```
Initialize PQ with [ptr(RN),0];          // starts from the root node
$r_{NN} := \infty$;                      // this is the initial "search radius"
while PQ $\neq \emptyset$:                                 // until the queue is not empty…
    [ptr(N), $d_{MIN}$(q,Reg(N))] := DEQUEUE(PQ);           // … get the closest pair…
    Read(N);                                                // … and reads the node
    if N is a leaf then: for each point t in N:
        if d(q,t) < $r_{NN}$ then: {$t_{NN}$(q) := t; $r_{NN}$ := d(q,t); UPDATE(PQ)}
                                         // reduces the search radius and prunes nodes
    else: for each child node Nc of N:
        if $d_{MIN}$(q,Reg(Nc)) < $r_{NN}$ then:
            ENQUEUE(PQ,[ptr(Nc), $d_{MIN}$(q,Reg(Nc))]);
return $t_{NN}$(q) and $r_{NN}$;
end.
```

kNNOptimal in action



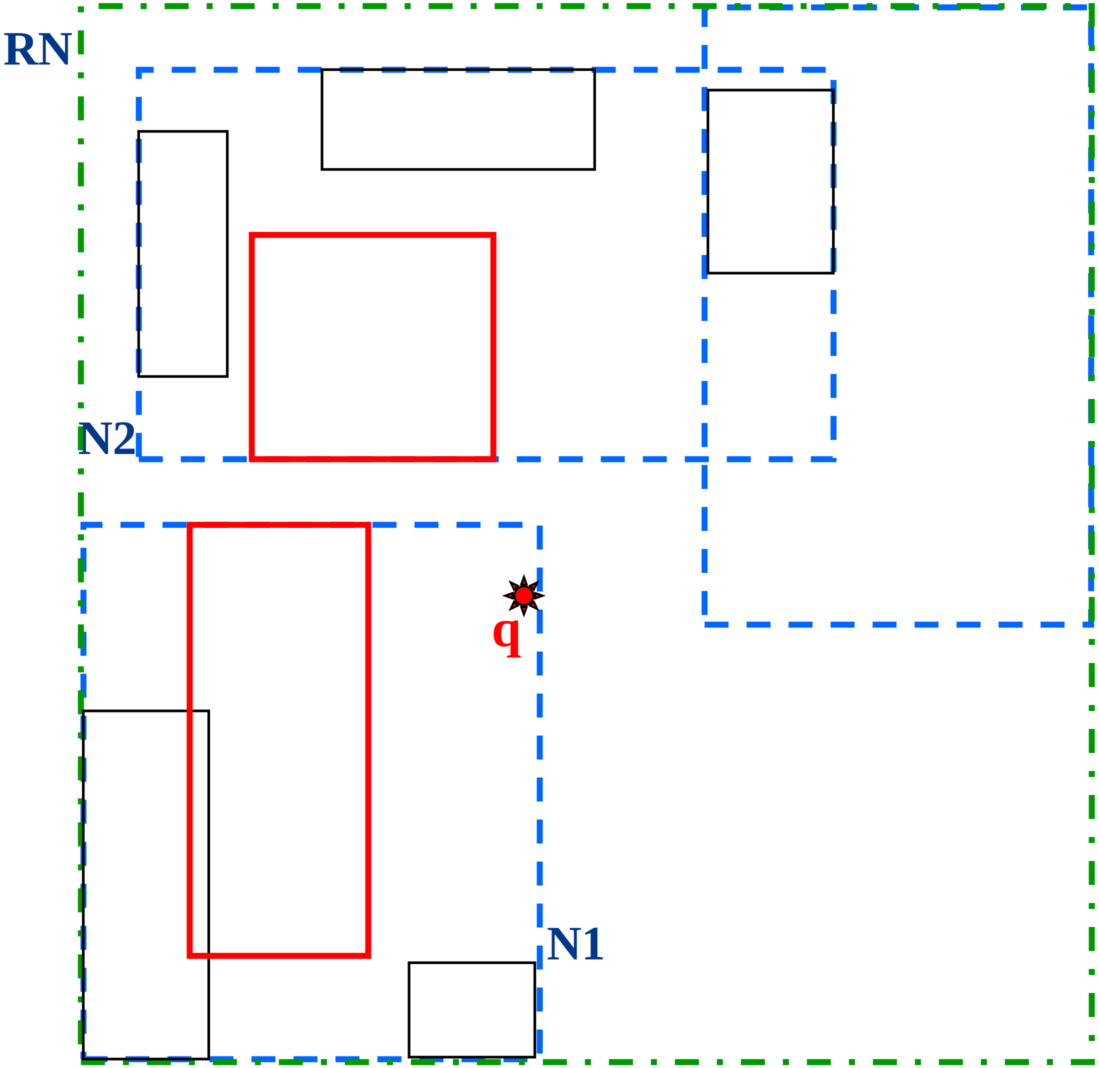
- Nodes are numbered following the order in which they are accessed
- Objects are numbered as they are found to improve (reduce) the search radius
- The accessed leaf nodes are shown in red

kNNOptimal in action



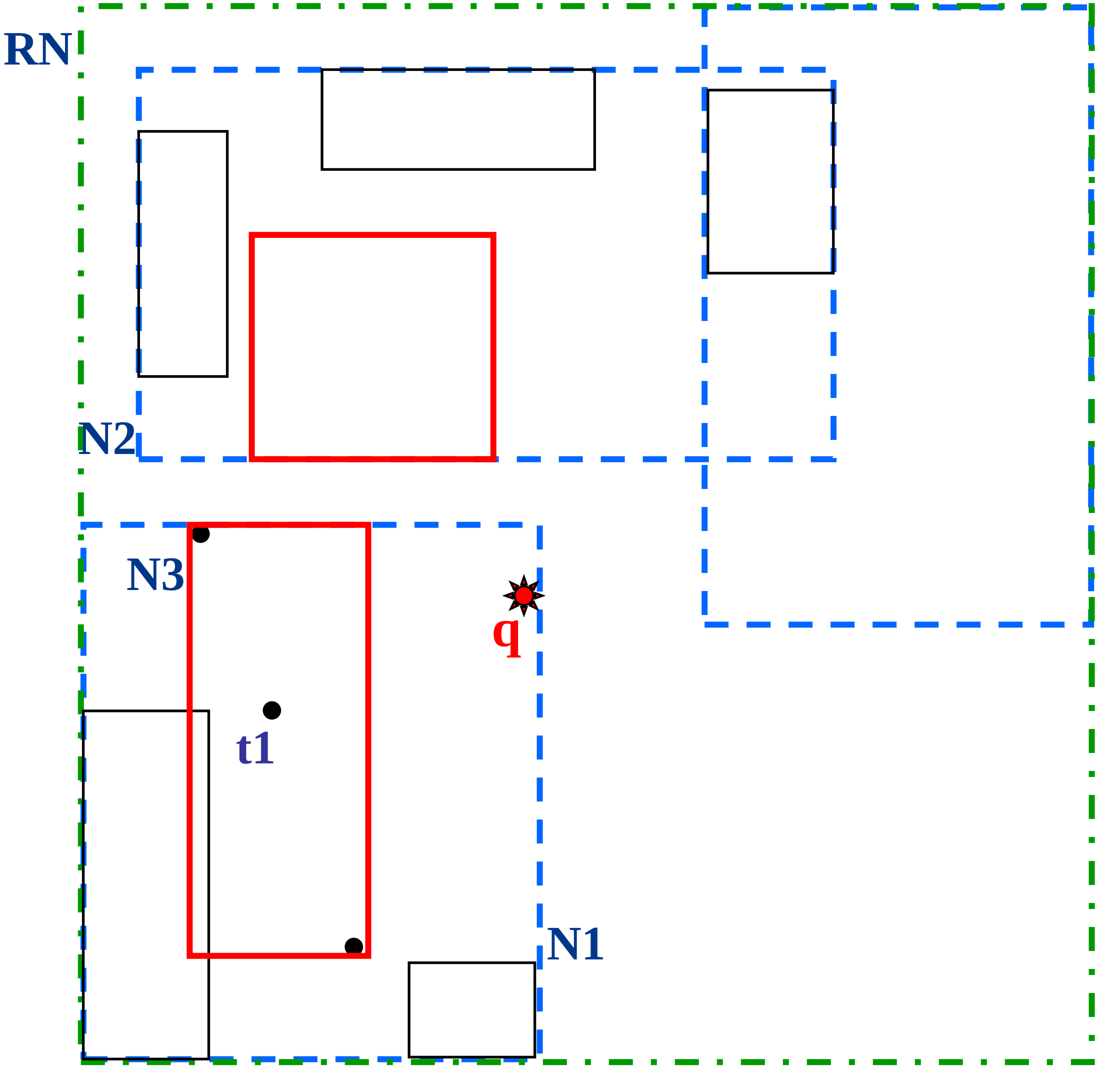
- Nodes are numbered following the order in which they are accessed
- Objects are numbered as they are found to improve (reduce) the search radius
- The accessed leaf nodes are shown in red

kNNOptimal in action



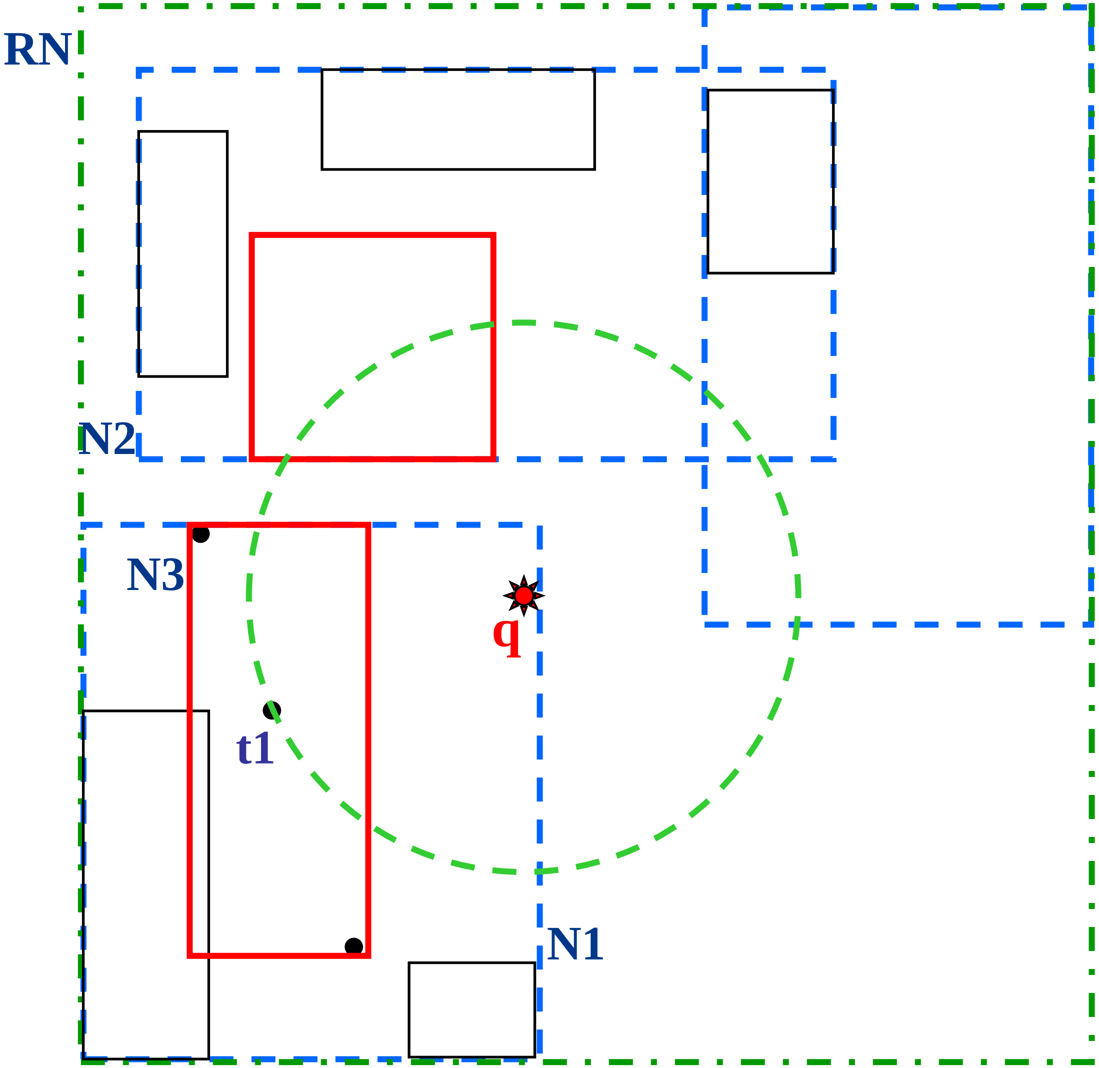
- Nodes are numbered following the order in which they are accessed
- Objects are numbered as they are found to improve (reduce) the search radius
- The accessed leaf nodes are shown in red

kNNOptimal in action



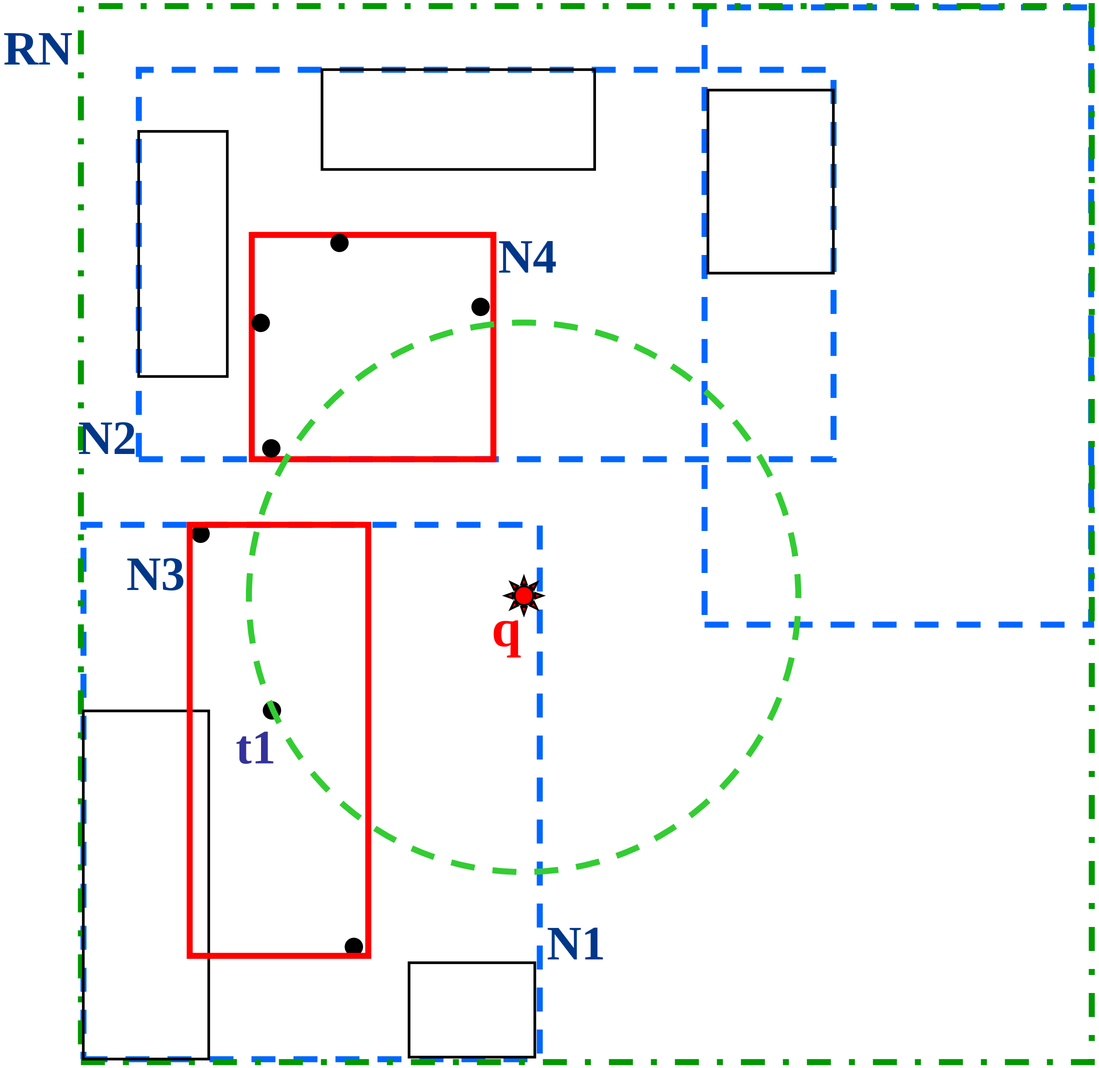
- Nodes are numbered following the order in which they are accessed
- Objects are numbered as they are found to improve (reduce) the search radius
- The accessed leaf nodes are shown in red

kNNOptimal in action



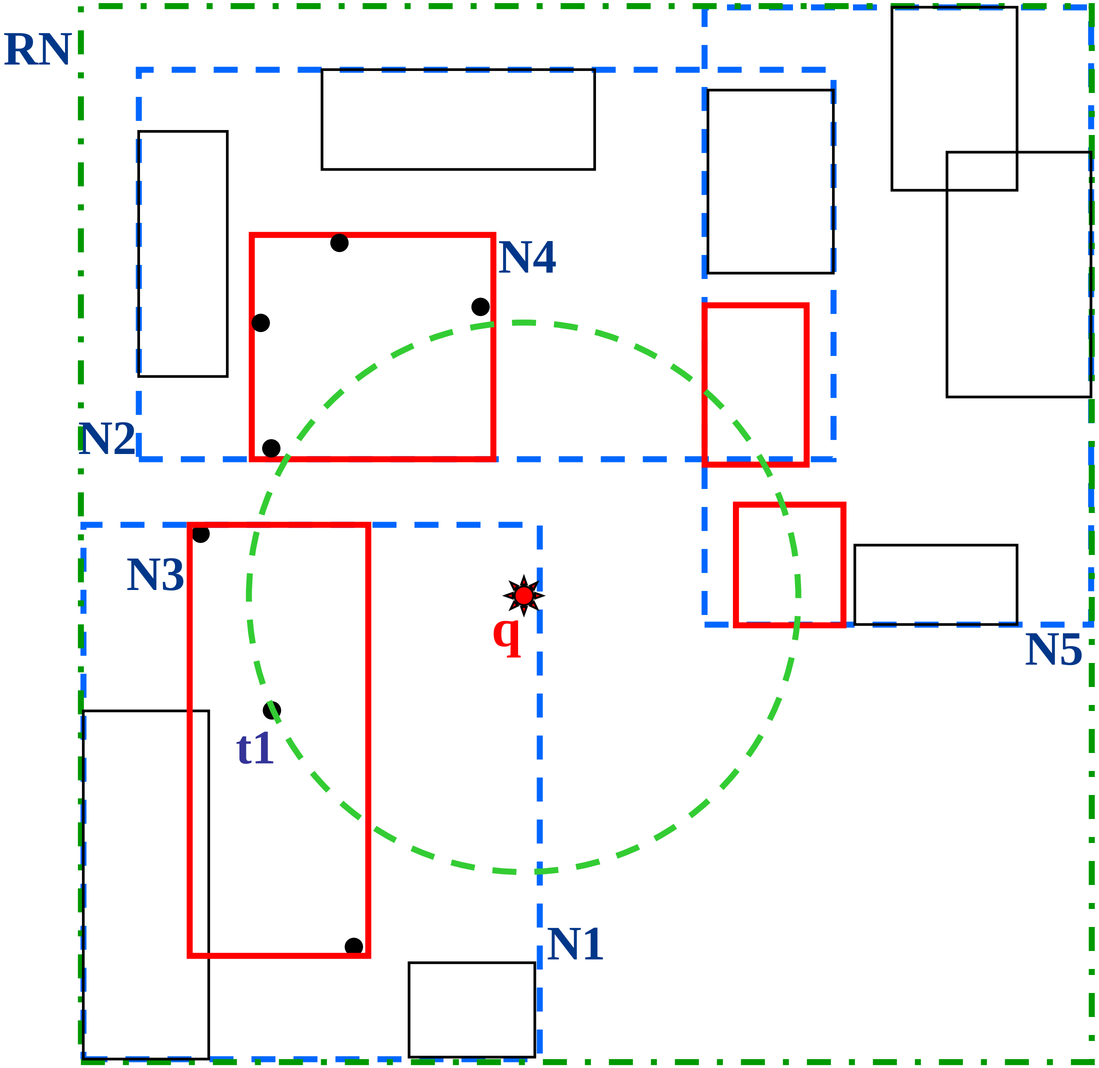
- Nodes are numbered following the order in which they are accessed
- Objects are numbered as they are found to improve (reduce) the search radius
- The accessed leaf nodes are shown in red

kNNOptimal in action



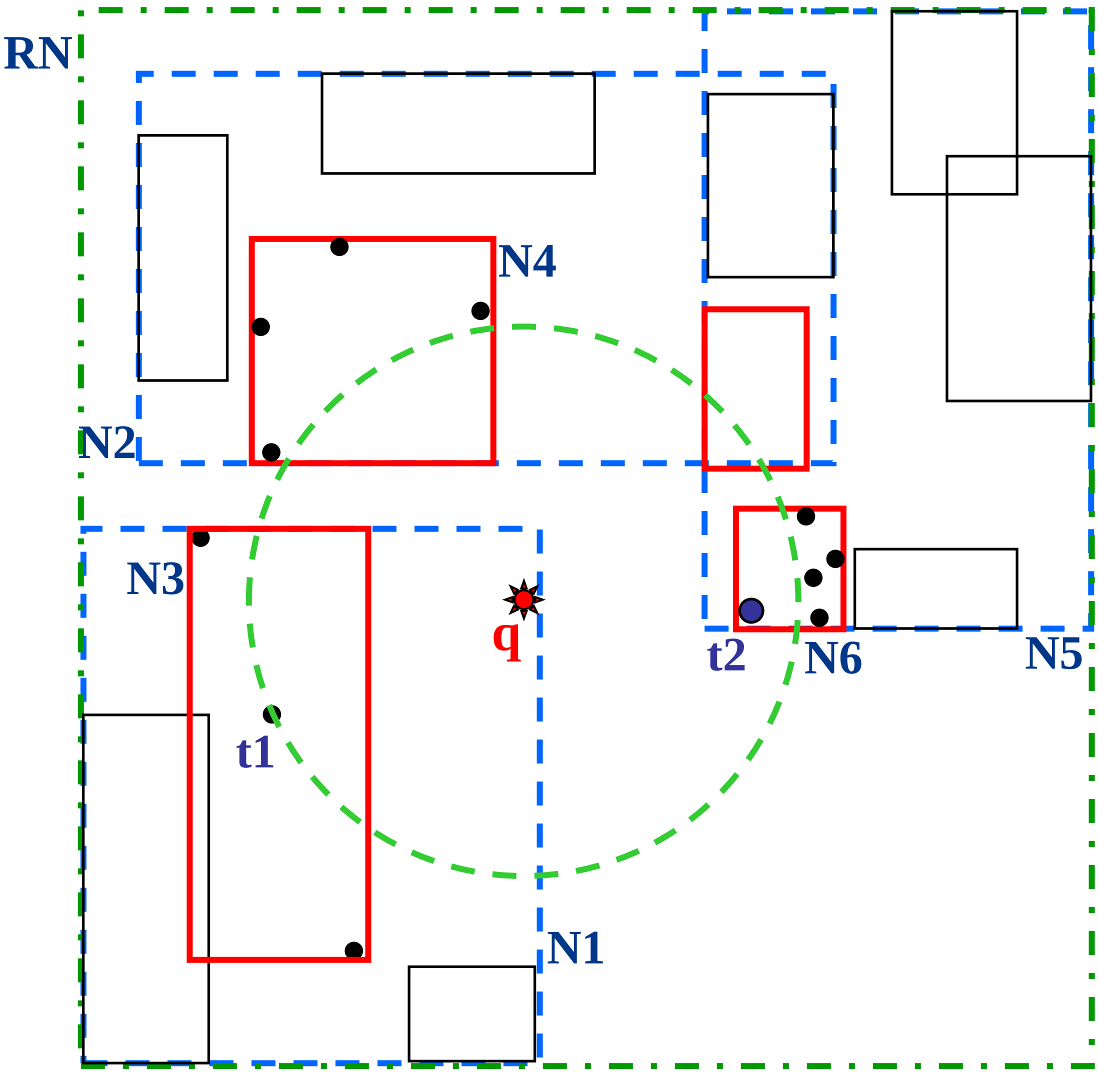
- Nodes are numbered following the order in which they are accessed
- Objects are numbered as they are found to improve (reduce) the search radius
- The accessed leaf nodes are shown in red

kNNOptimal in action



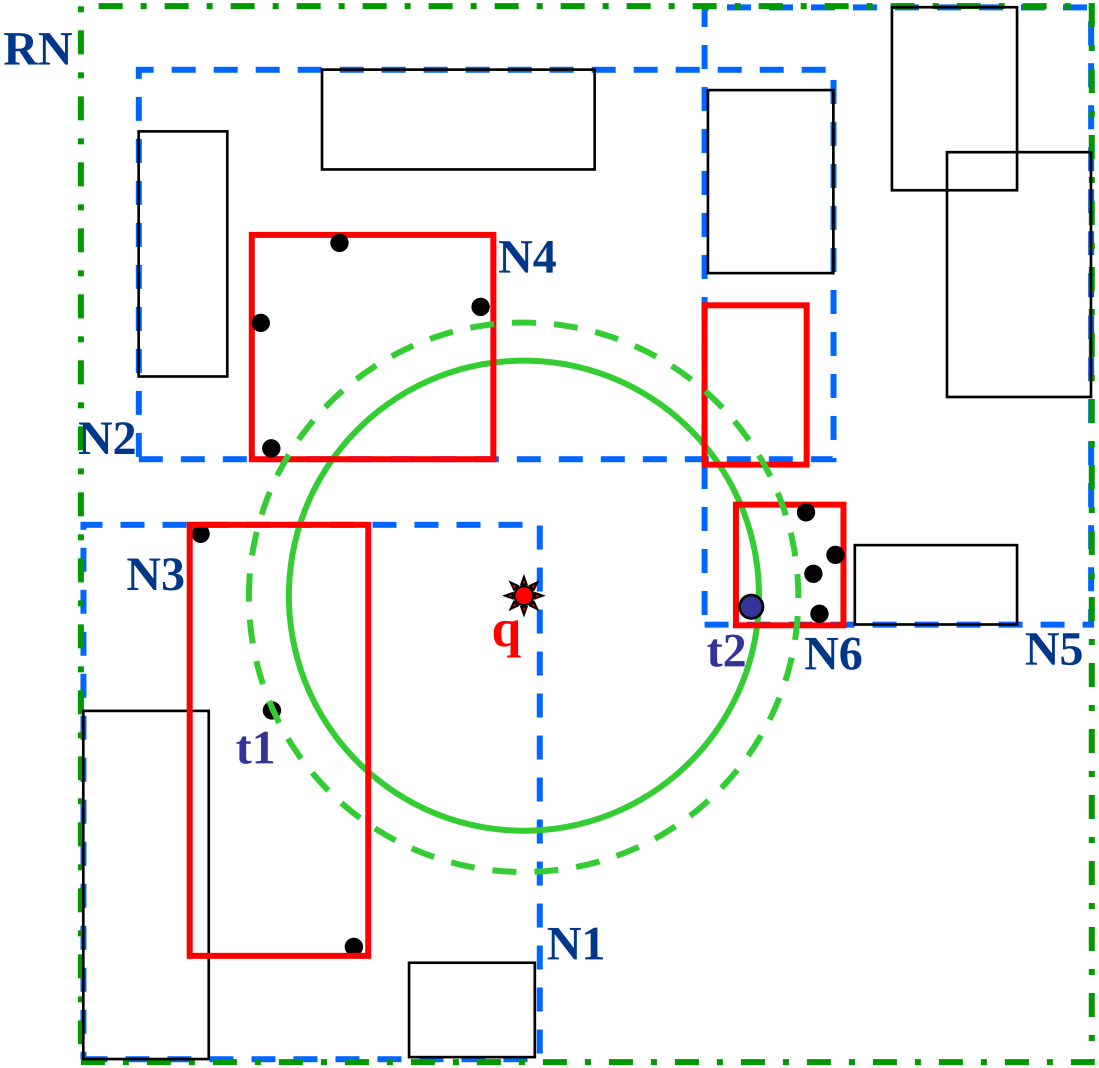
- Nodes are numbered following the order in which they are accessed
- Objects are numbered as they are found to improve (reduce) the search radius
- The accessed leaf nodes are shown in red

kNNOptimal in action



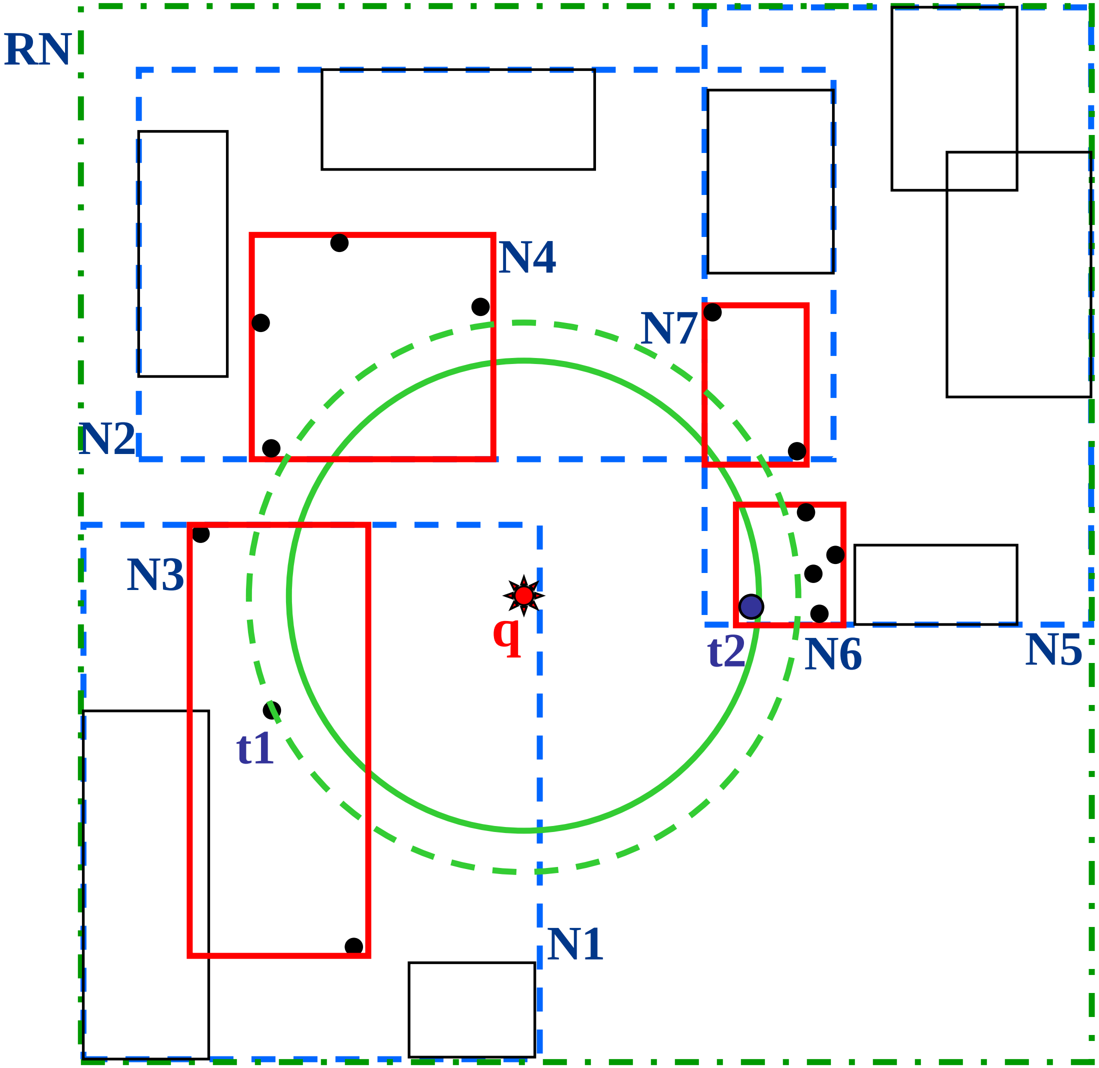
- Nodes are numbered following the order in which they are accessed
- Objects are numbered as they are found to improve (reduce) the search radius
- The accessed leaf nodes are shown in red

kNNOptimal 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

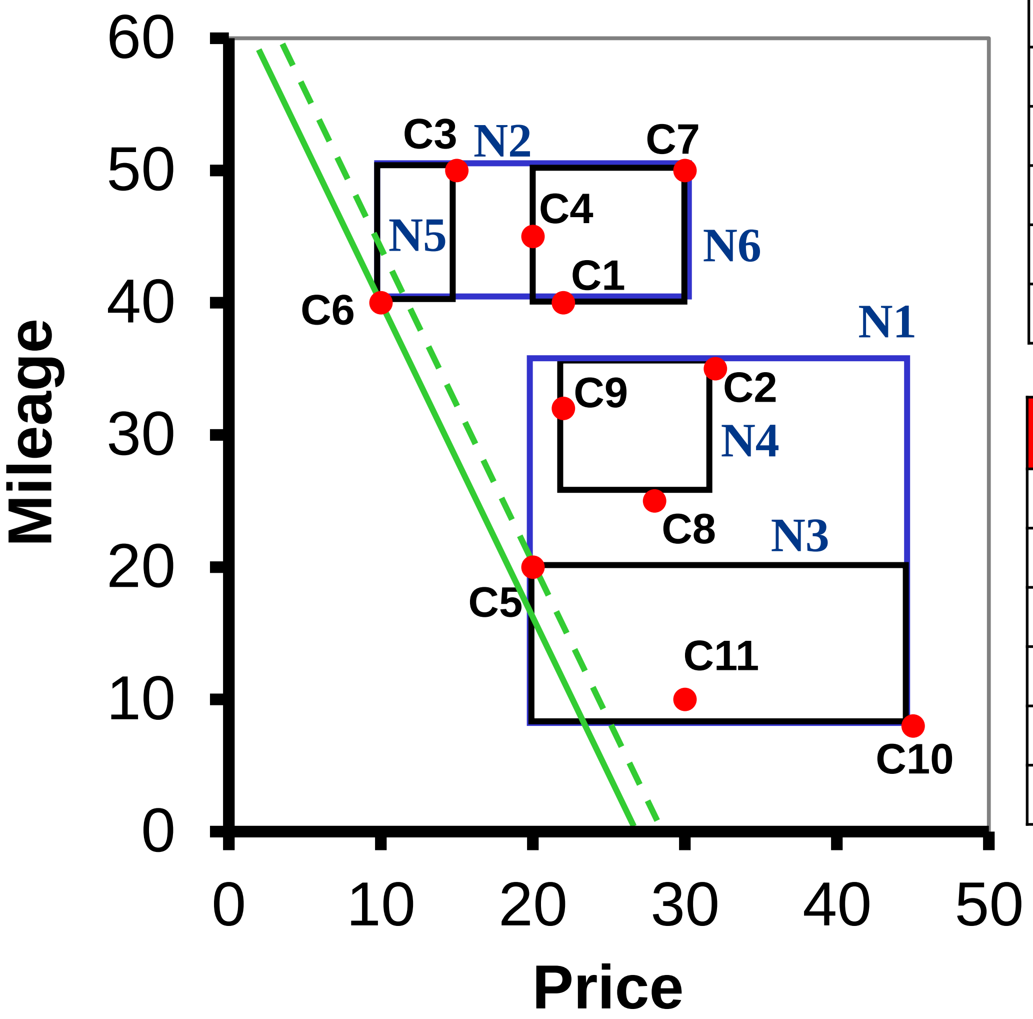
kNNOptimal in action



- 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*Price + 0.3*Mileage`



Node	d_{MIN}
N1	16.4
N2	19
N3	16.4
N4	22.9
N5	19
N6	26

tuple	d
C5	20
C6	19
C11	24
…	…

Action	PQ		
Read(RN)	(N1,16.4)	(N2,19)	
Read(N1)	(N3,16.4)	(N2,19)	(N4,22.9)
Read(N3)	(N2,19)		
Read(N2)	(N5,19)		
Read(N5)			
Return(C6,19)			

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_{NN}, i.e., $d_{MIN}(q,Reg(N)) > r_{NN}$

Proof:

- Node N is read only if, at some execution step, it becomes the 1st element in PQ
- Let N1 be the node containing $t_{NN}(q)$, 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:

$$r_{NN} \geq d_{MIN}(q,Reg(N1)) \geq d_{MIN}(q,Reg(N2)) \geq \ldots \geq d_{MIN}(q,Reg(Nh))$$

- At each time step before we find $t_{NN}(q)$, 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 $d_{MIN}(q,Reg(N)) = r_{NN}$?

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_{NN}.
   In this case no node with $d_{MIN}(q,Reg(N)) = r_{NN}$ will be read
2. The NN is in a node whose region has exactly MinDist = r_{NN}.
   Now everything depends on how ties are managed in PQ.
   In the worst case, all nodes with $d_{MIN}(q,Reg(N)) = r_{NN}$ will be read

The general case (k ≥ 1)

- The algorithm is easily extended to the case k ≥ 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, $r_{k\text{-NN}}$, of the current k-th NN of q, that is, the k-th element of Res

Res

ObjectID	distance
t15	4
t24	8
t18	9
t4	12
t2	**15**

k = 5

- No node with distance ≥ 15 needs to be read

- The rest of the algorithm remains unchanged

The kNNOptimal Algorithm (case $k \geq 1$)

Input: query point q, integer $k \geq 1$, index tree with root node RN

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

```
Initialize PQ with [ptr(RN),0];
for i=1 to k: Res[i] := [null,$\infty$]; $r_{k\text{-NN}}$ := Res[k].dist;
while PQ $\neq \emptyset$:
    [ptr(N), $d_{MIN}$(q,Reg(N))] := DEQUEUE(PQ);
    Read(N);
    if N is a leaf then: for each point t in N:
            if d(q,t) < $r_{k\text{-NN}}$ then: { remove the element in ResultList[k];
                                               insert [t,d(q,t)] in ResultList;
                                             $r_{k\text{-NN}}$ := Res[k].dist; UPDATE(PQ)}
        else: for each child node Nc of N:
            if $d_{MIN}$(q,Reg(Nc)) < $r_{k\text{-NN}}$ then:
                ENQUEUE(PQ,[ptr(Nc), $d_{MIN}$(q,Reg(Nc))]);
return Res;
end.
```

Distance browsing

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

  ```
  SELECT *
  FROM   USEDCARS
  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 $d_{MIN}(q,Reg(t))$
- 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 [ptr(RN),0]
- When a tuple t becomes the 1st element of the queue the algorithm returns

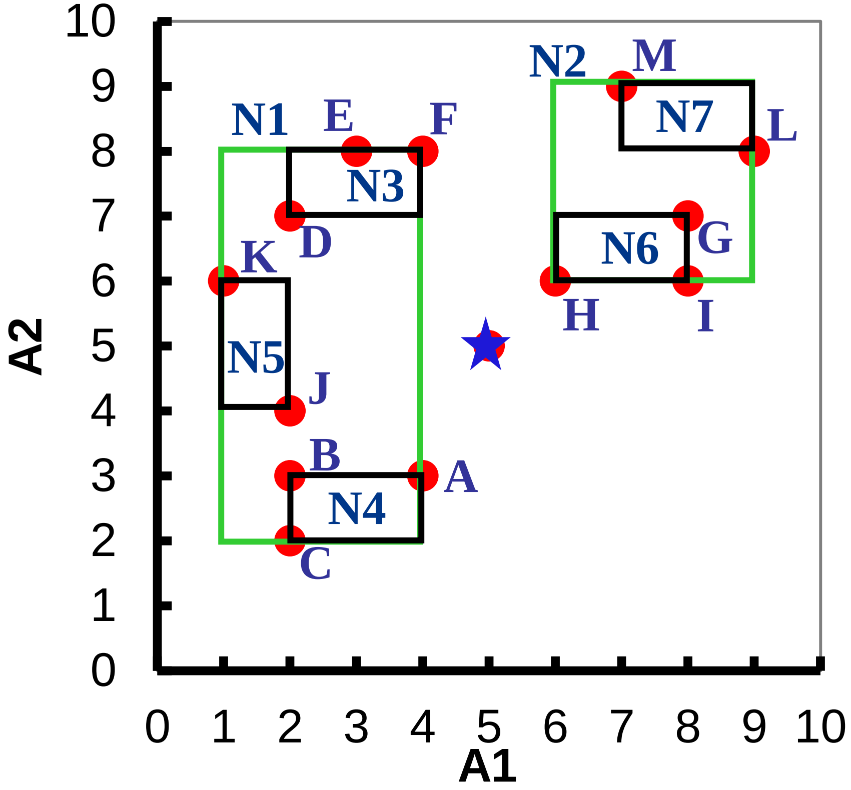
Input: query point q, index tree with root node RN

Output: the next nearest neighbor of q, together with its distance

```
while PQ ≠ ∅:
    [ptr(Elem), $d_{MIN}(q,Reg(Elem))$] := DEQUEUE(PQ);
    if Elem is a tuple t then: return t and its distance // no tuple can be better than t
            else: if N is a leaf then: for each point t in N: ENQUEUE(PQ,[t,d(q,t)])
                    else: for each child node Nc of N:
                        ENQUEUE(PQ,[ptr(Nc), $d_{MIN}(q,Reg(Nc))$]);
end.
```

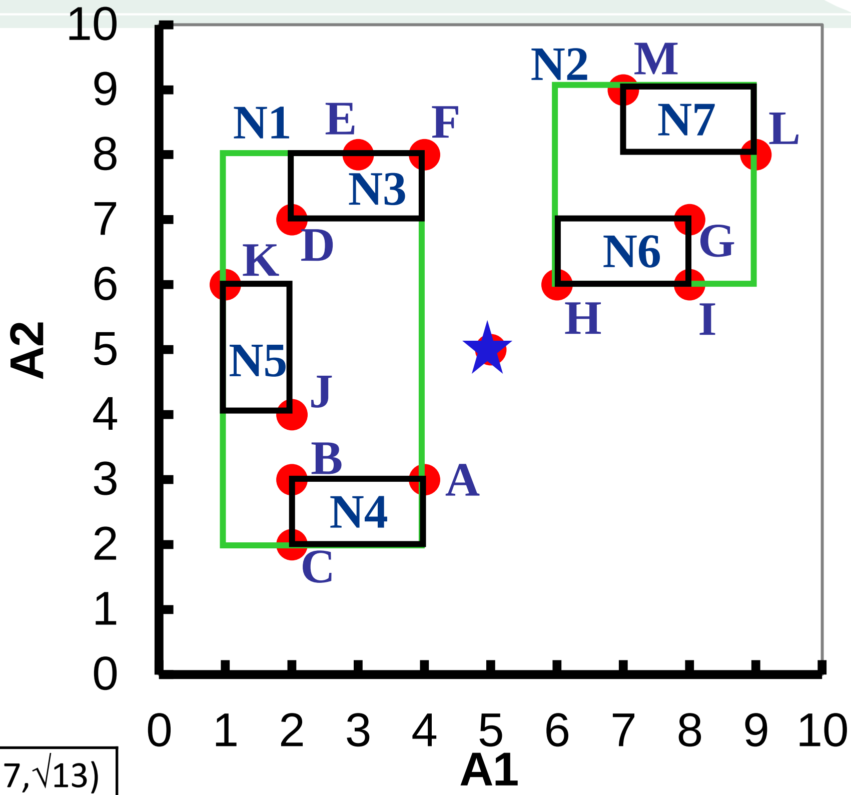
Distance browsing: An example (1/2)

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



Elem	d_{MIN}
A	√5
B	√13
C	√18
D	√13
E	√13
F	√10
G	√13
H	√2
I	√10
J	√10
K	√17
L	√25
M	√20
N1	√1
N2	√2
N3	√5
N4	√5
N5	√9
N6	√2
N7	√13

Distance browsing: An example (2/2)



Action	PQ									
Read(RN)	(N1,√1)	(N2,√2)								
Read(N1)	(N2,√2)	(N3,√5)	(N4,√5)	(N5,√9)						
Read(N2)	(N6,√2)	(N3,√5)	(N4,√5)	(N5,√9)	(N7,√13)					
Read(N6)	(H,√2)	(N3,√5)	(N4,√5)	(N5,√9)	(I,√10)	(G,√13)	(N7,√13)			
Return(H,√2)	(N3,√5)	(N4,√5)	(N5,√9)	(I,√10)	(G,√13)	(N7,√13)				
Read(N3)	(N4,√5)	(N5,√9)	(I,√10)	(F,√10)	(G,√13)	(D,√13)	(E,√13)	(N7,√13)		
Read(N4)	(A,√5)	(N5,√9)	(I,√10)	(F,√10)	(G,√13)	(D,√13)	(E,√13)	(B,√13)	(N7,√13)	(C,√18)
Return(A,√5)	(N5,√9)	(I,√10)	(F,√10)	(G,√13)	(D,√13)	(E,√13)	(B,√13)	(C,√18)		
Read(N5)	…	…	…							
…										

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

Top-k join queries

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

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

where p1,p2,…pm are scoring criteria (the "preferences")

Top-k join queries: examples

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

```
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

```
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
```

Top-k selection queries as top-k join queries

- A (multidimensional) top-k selection query based on the scoring function S(p1,p2,…,pm) 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 Ri consists of the object id and of the attributes needed to compute pj
- E.g. If S is defined as:

  S(t) = (t.Price + t.Mileage)/(t.Year - 1970)

  we can "partition" USEDCARS as:

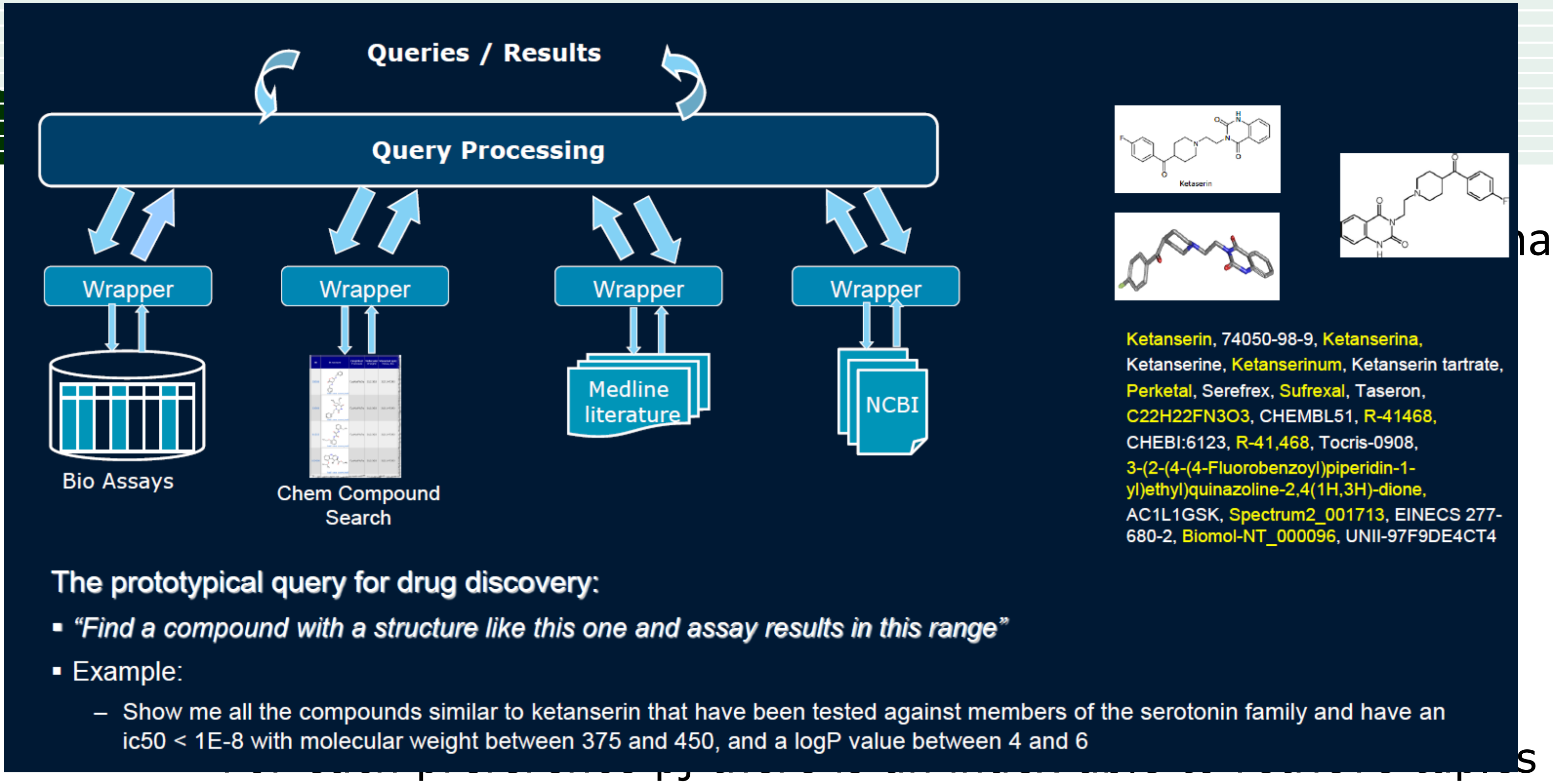
  UC1(CarID,Price), UC2(CarID,Mileage), UC3(CarID,Year)
- A top-k selection query would be equivalent to:

```
SELECT *
FROM   USEDCARS UC1, USEDCARS UC2, USEDCARS UC3
WHERE  UC1.CarID = UC2.CarID
AND    UC2.CarID = UC3.CarID
ORDER BY (UC1.Price + UC2.Mileage)/(UC3.Year-1970)
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 pj there is an index able to retrieve tuples according to that preference
  - The "partitions" of R are real, e.g., R is distributed 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



Ketanserin, 74050-98-9, Ketanserina, Ketanserine, Ketanserinum, Ketanserin tartrate, Perketal, Serefrex, Sufrexal, Taseron, C22H22FN3O3, CHEMBL51, R-41468, CHEBI:6123, R-41,468, Tocris-0908, 3-(2-(4-(4-Fluorobenzoyl)piperidin-1-yl)ethyl)quinazoline-2,4(1H,3H)-dione, AC1L1GSK, Spectrum2_001713, EINECS 277-680-2, Biomol-NT_000096, UNII-97F9DE4CT4

The prototypical query for drug discovery:

- *"Find a compound with a structure like this one and assay results in this range"*
- Example:
  - Show me all the compounds similar to ketanserin that have been tested against members of the serotonin family and have an ic50 < 1E-8 with molecular weight between 375 and 450, and a logP value between 4 and 6

as been

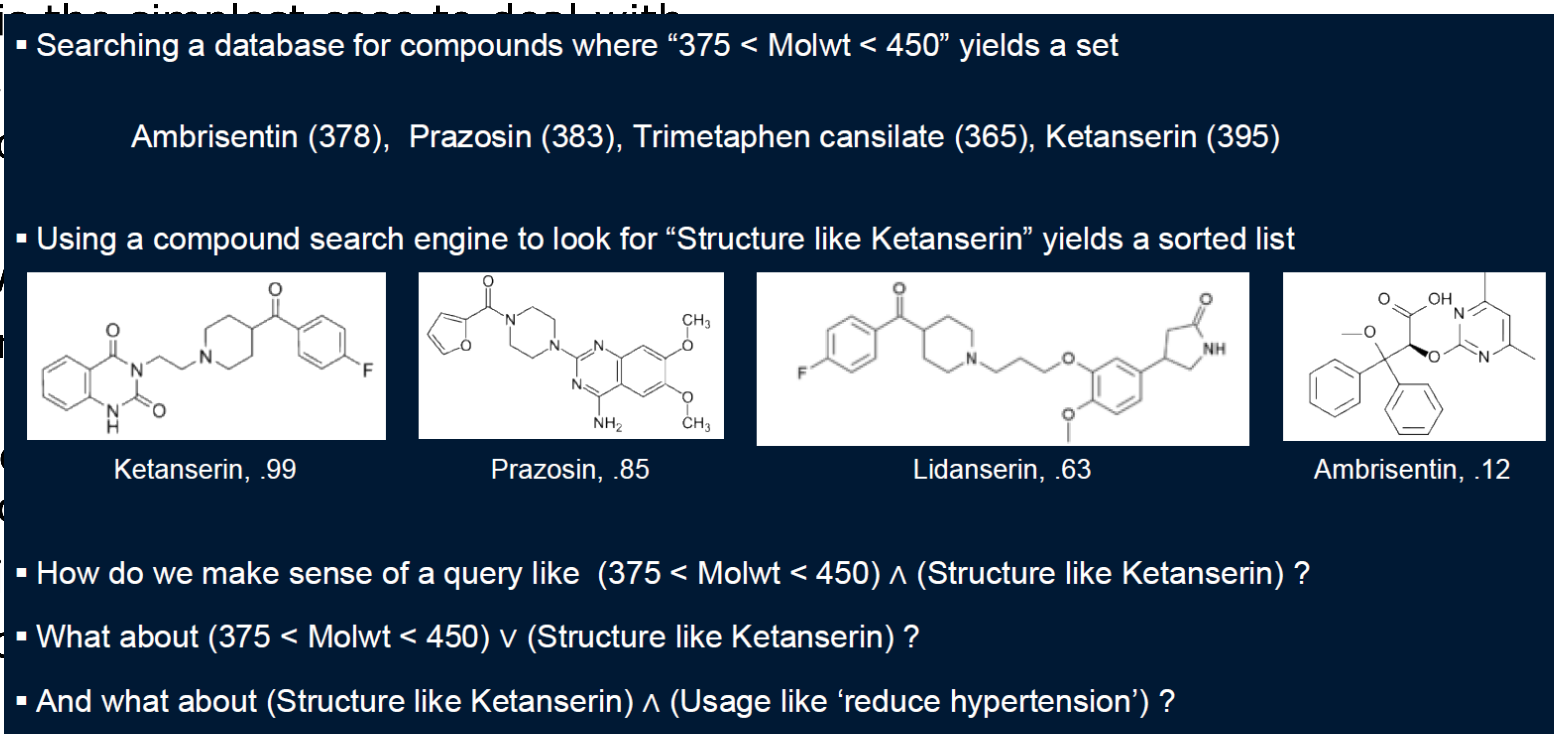
s according to that preference

- The "partitions" of R are real, e.g., R is distributed 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

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
  - It c
- The tw
  - For to
  - The eac
- Histori is the c to the



- Searching a database for compounds where “375 < Molwt < 450” yields a set

Ambrisentin (378), Prazosin (383), Trimetaphen cansilate (365), Ketanserin (395)

- Using a compound search engine to look for “Structure like Ketanserin” yields a sorted list

Ketanserin, .99

Prazosin, .85

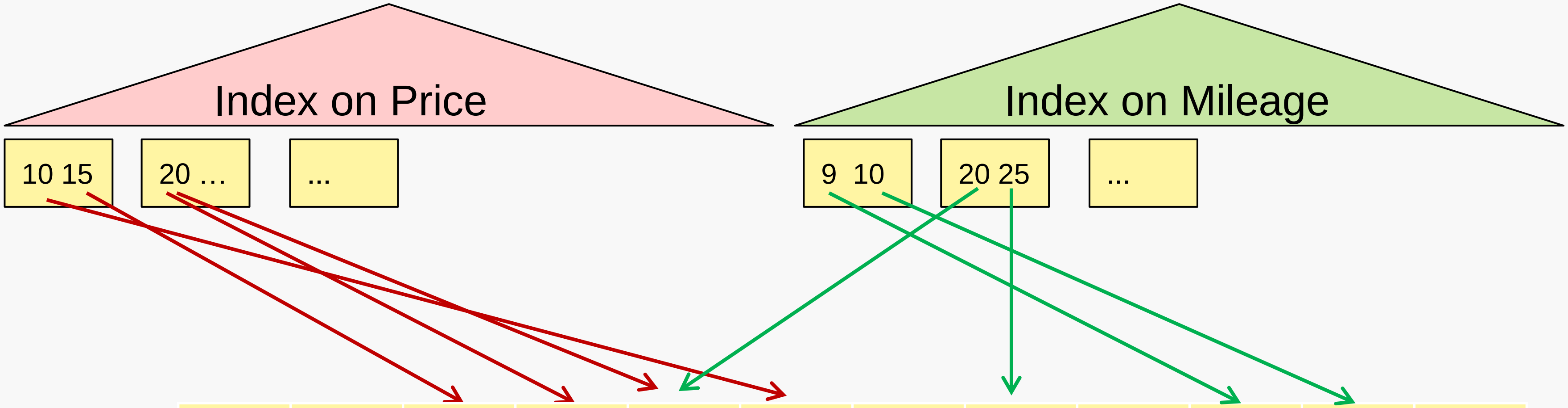
Lidanserin, .63

Ambrisentin, .12

- How do we make sense of a query like (375 < Molwt < 450) ∧ (Structure like Ketanserin) ?
- What about (375 < Molwt < 450) ∨ (Structure like Ketanserin) ?
- And what about (Structure like Ketanserin) ∧ (Usage like ‘reduce hypertension’) ?

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



Id	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	…
Price	22	32	15	20	20	10	30	28	22	45	30	…
Mileage	40	35	50	45	20	40	50	25	32	9	10	…

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 "integrating" in some way the results returned by the data sources

- These queries have been 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:

    CarPrices(CarModel, Price)
  - The schema exported by the 2nd source is:

    CarSpec(Make, Model, FuelConsumption)
- After a phase of "reconciliation"

  CarModel = ‘Audi/A4’ ⇔ (Make,Model) = (‘Audi’,‘A4’)

  we can now support queries on both Price and FuelConsumption, e.g.:

  find those cars whose consumption is less than 7 litres/100km and with a cost less than 15,000 €

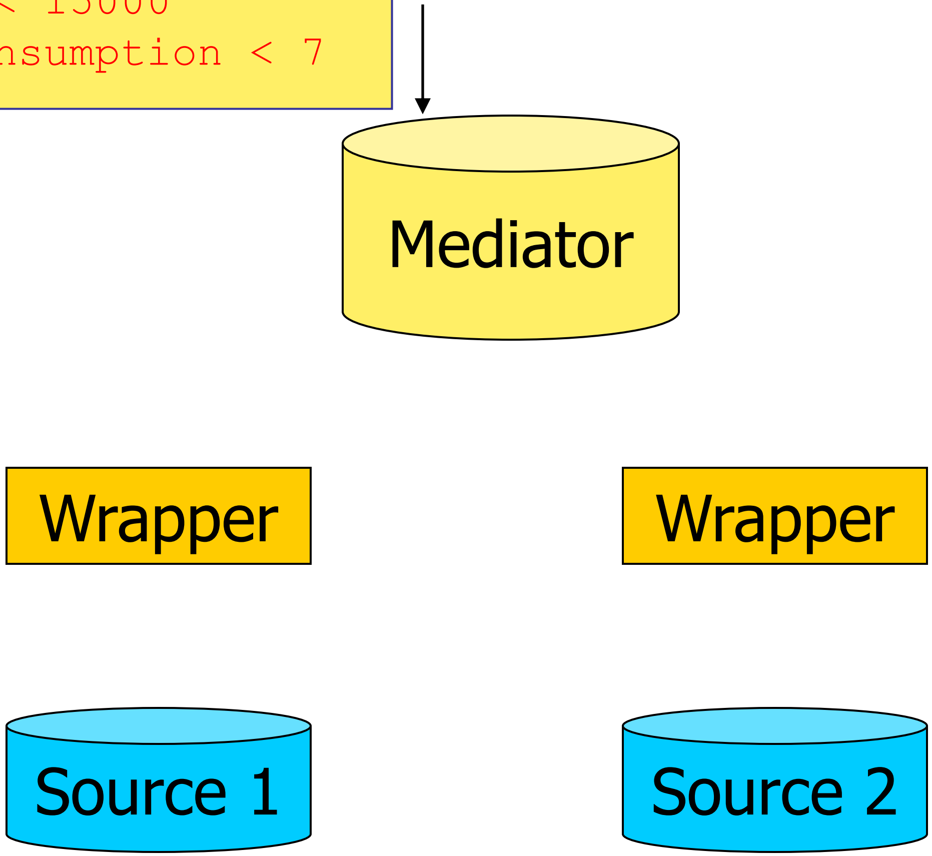
  How?

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

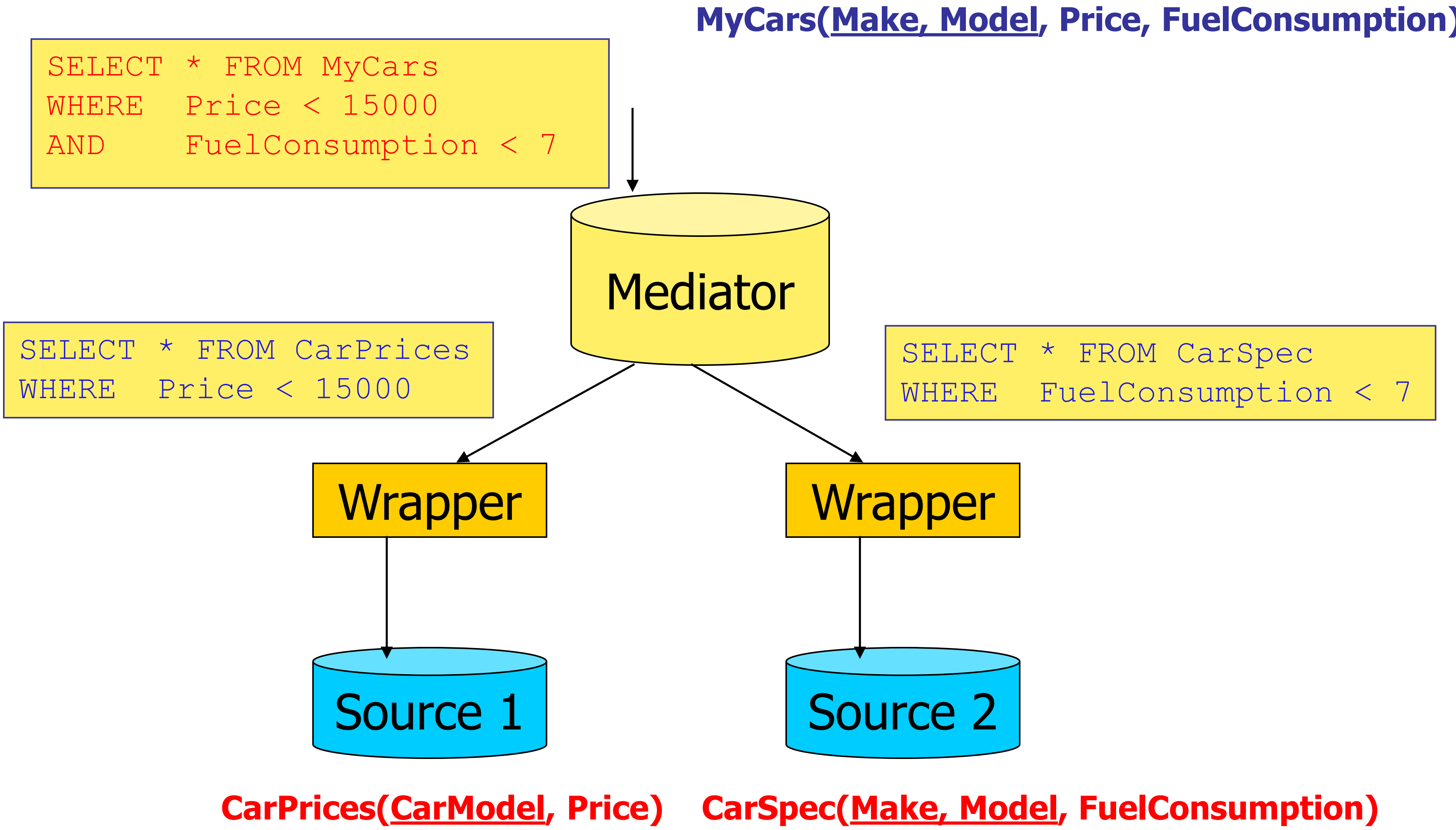
MyCars(Make, Model, Price, FuelConsumption)

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



CarPrices(CarModel, Price) **CarSpec(Make, Model, FuelConsumption)**

The details of query execution



MyCars(Make, Model, Price, FuelConsumption)

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

```
SELECT * FROM CarPrices
WHERE  Price < 15000
```

```
SELECT * FROM CarSpec
WHERE  FuelConsumption < 7
```

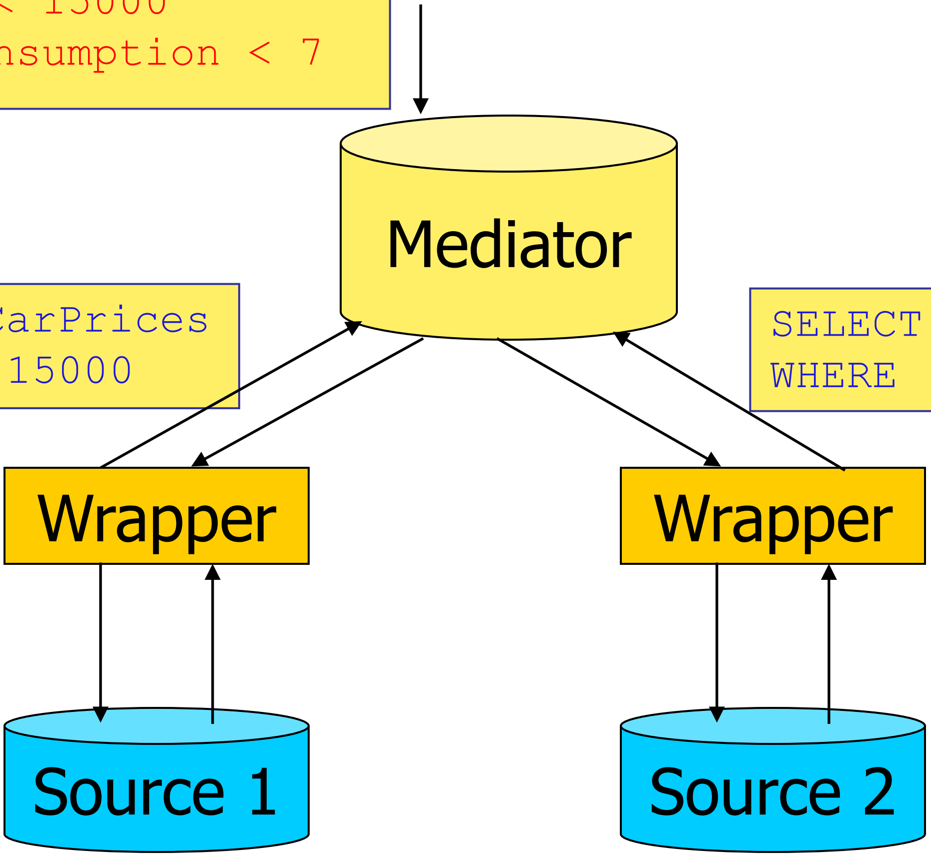
CarPrices(CarModel, Price)

CarSpec(Make, Model, FuelConsumption)

The details of query execution

MyCars(Make, Model, Price, FuelConsumption)

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



```
SELECT * FROM CarPrices
WHERE  Price < 15000
```

```
SELECT * FROM CarSpec
WHERE  FuelConsumption < 7
```

CarModel	Price
Toyota/Yaris	12
Citroen/C3	11

Make	Model	FuelCons
Toyota	Yaris	6.5
Nissan	Micra	6.2

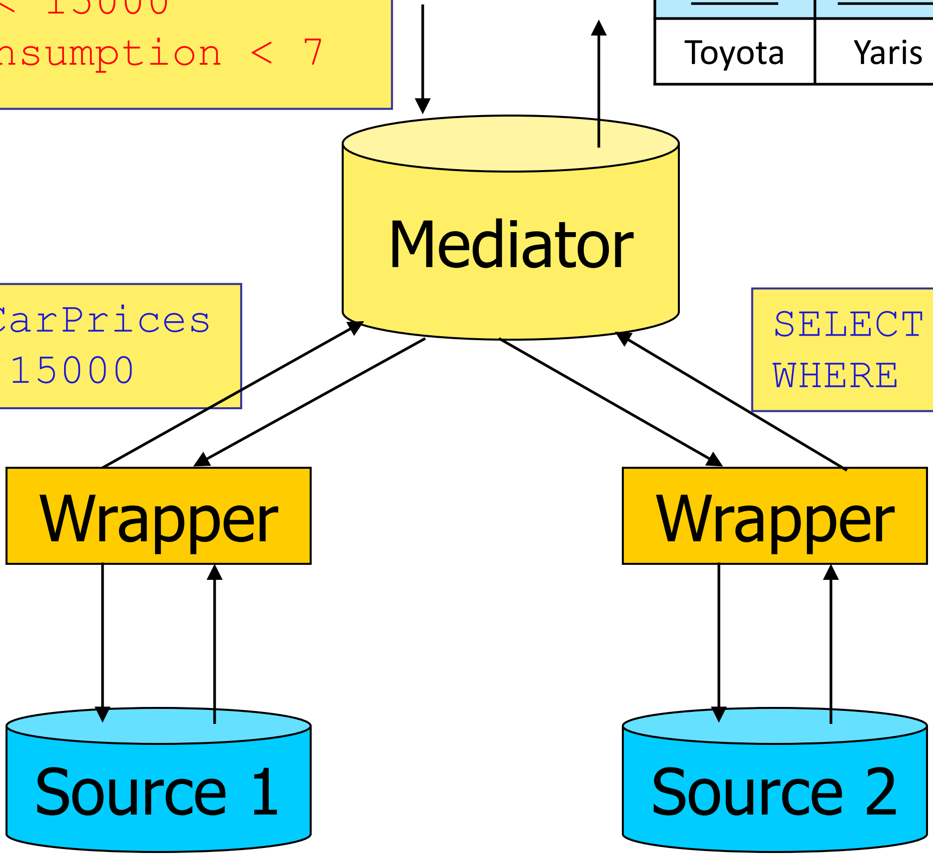
CarPrices(CarModel, Price) **CarSpec(Make, Model, FuelConsumption)**

The details of query execution

MyCars(<u>Make, Model</u>, Price, FuelConsumption)

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

<u>Make</u>	<u>Model</u>	Price	FuelCons
Toyota	Yaris	12	6.5



```
SELECT * FROM CarPrices
WHERE  Price < 15000
```

```
SELECT * FROM CarSpec
WHERE  FuelConsumption < 7
```

<u>CarModel</u>	Price
Toyota/Yaris	12
Citroen/C3	11

<u>Make</u>	<u>Model</u>	FuelCons
Toyota	Yaris	6.5
Nissan	Micra	6.2

CarPrices(<u>CarModel</u>, Price) **CarSpec(<u>Make, Model</u>, FuelConsumption)**

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:

    CarDealerj(CarID, Make, Model, Price)

    and our goal is to provide our users with the cheapest available cars, that is, to support queries like:

    For each FIAT model, which is the cheapest offer?

How?

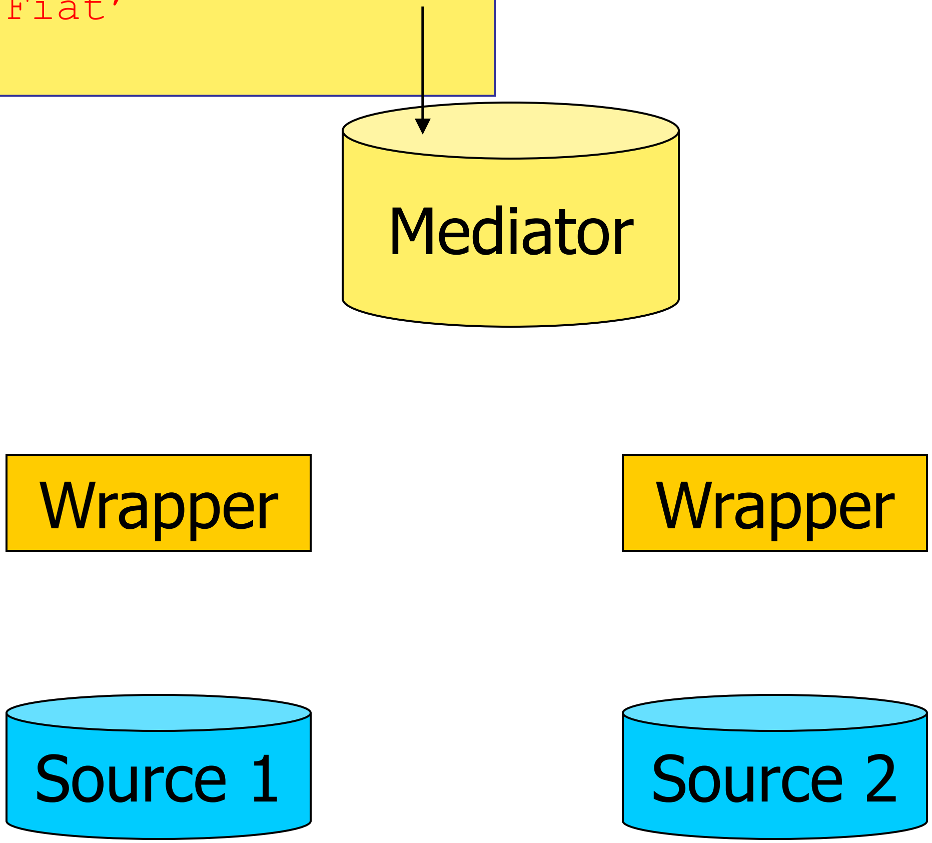
1. send the same (sub-)query to the all the data sources,
2. take the union of the results,
3. for each model, get the best offer and the corresponding dealer

➡ For queries of this kind, the mediator is also often called a "meta-broker" or "meta-search engine"

Query execution (some details omitted)

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

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



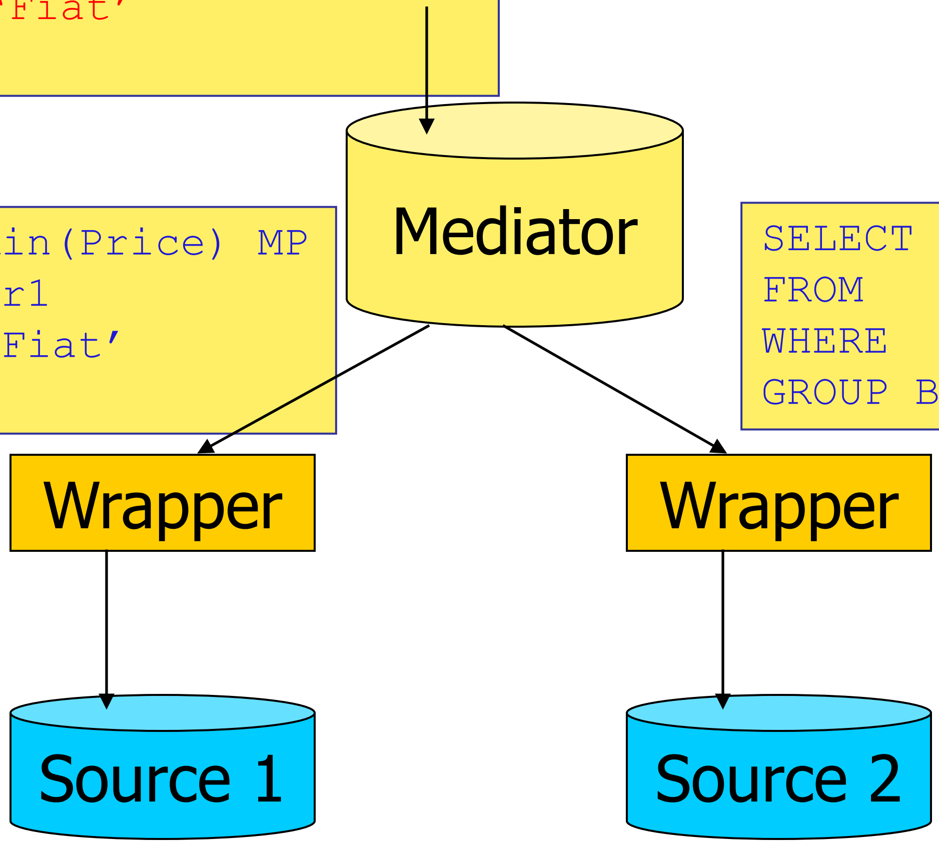
CarDealer1(CarID, Make, Model, Price)

CarDealer2(CarID, Make, Model, Price)

Query execution (some details omitted)

```
SELECT Model,min(Price) MP,Dealer
FROM   AllCars
WHERE  Make = ‘Fiat’
GROUP BY Model
```

AllCars(<u>CarID</u>, Make, Model, Price, Dealer)



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

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

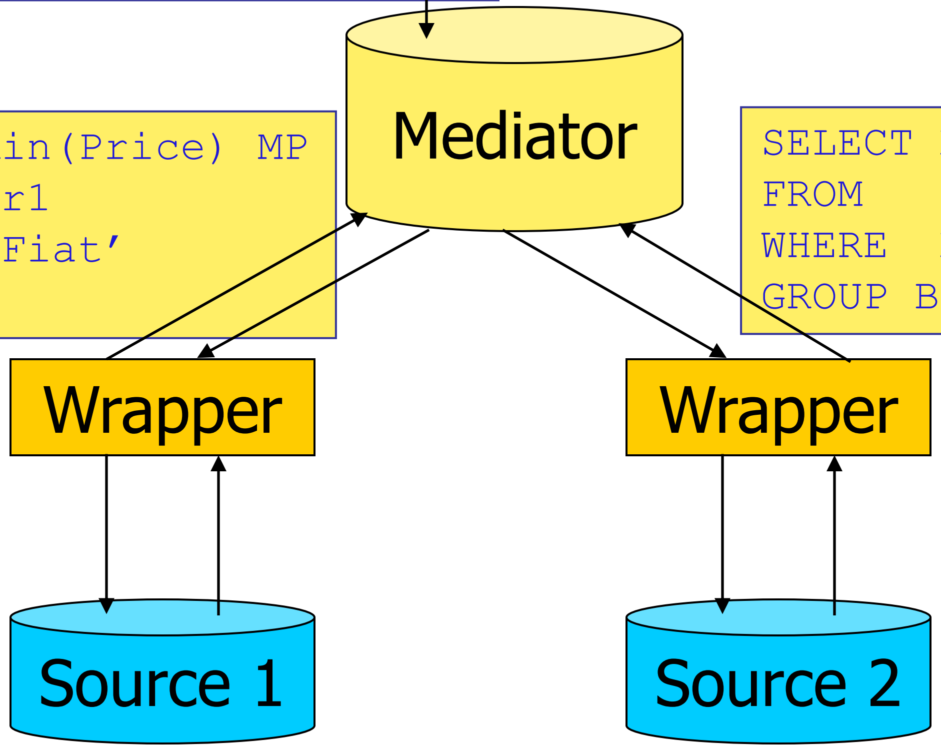
CarDealer1(<u>CarID</u>, Make, Model, Price)

CarDealer2(<u>CarID</u>, Make, Model, Price)

Query execution (some details omitted)

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

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



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

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

Model	MP
Brava	8
Punto	11

Model	MP
Brava	9
Duna	7
Punto	10

CarDealer1(CarID, Make, Model, Price)

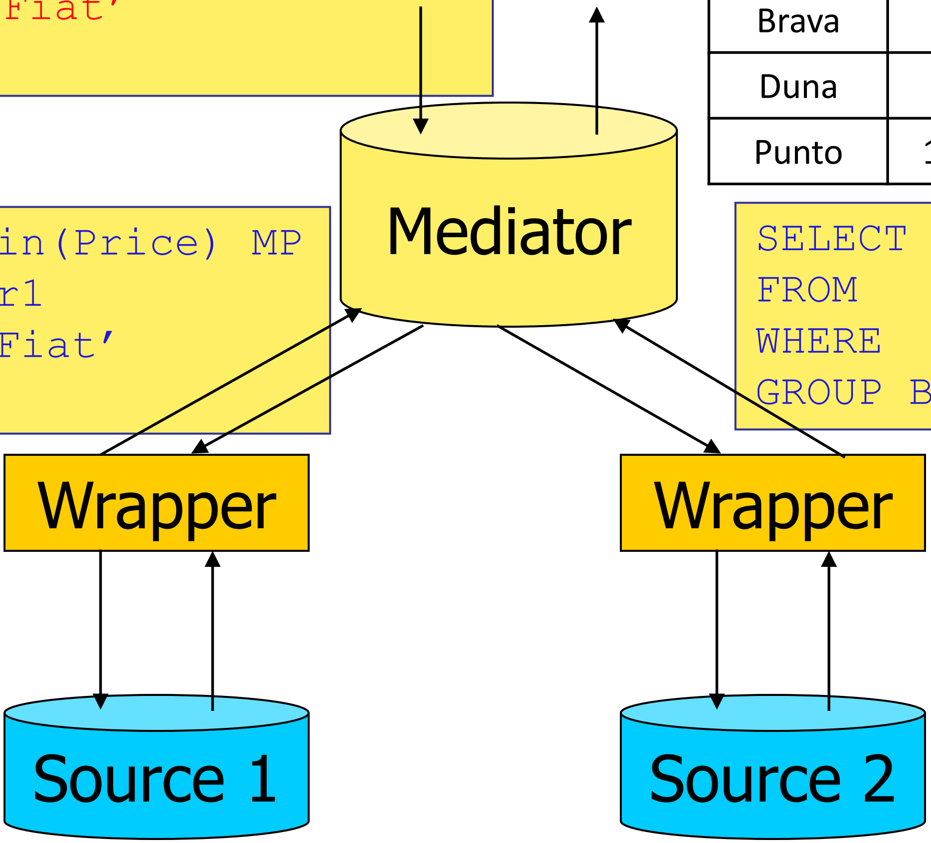
CarDealer2(CarID, Make, Model, Price)

Query execution (some details omitted)

```
SELECT Model,min(Price) MP,Dealer
FROM   AllCars
WHERE  Make = ‘Fiat’
GROUP BY Model
```

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

Model	MP	Dealer
Brava	8	D1
Duna	7	D2
Punto	10	D2



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

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

Model	MP
Brava	8
Punto	11

Model	MP
Brava	9
Duna	7
Punto	10

CarDealer1(CarID, Make, Model, Price)

CarDealer2(CarID, Make, Model, Price)

Top-k 1-1 join (middleware) queries

- "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 Lj
- 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 Lj supports a sorted access (s.a.) interface:

$\text{getNext}_{Lj}() \rightarrow (\text{OID},\text{Attributes},pj)$

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

Thus, **Lj** is a **ranked list**, which justifies its name ("L" stands for list)

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

$\text{getScore}_{Lj}(\text{OID}) \rightarrow pj$

➡ A random access gets the partial score of an object

Other assumptions

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
  - 4: provided no ranking attribute has `NULL` values
- 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

Name	Score
Al vecchio mulino	9.2
La tavernetta	9.0
Il desco	8.3
Da Gino	7.5
Tutti a tavola!	6.4
Le delizie del palato	5.5
Acqua in bocca	5.0

PaneeVino

Name	Score
Da Gino	9.0
Il desco	8.5
Al vecchio mulino	7.5
Le delizie del palato	7.5
La tavernetta	7.0
Acqua in bocca	6.5
Tutti a tavola!	6.0

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

Name	Global Score
Il desco	16.8
Al vecchio mulino	16.7
Da Gino	16.5
La tavernetta	16.0
Le delizie del palato	13.0
Tutti a tavola!	12.4
Acqua in bocca	11.5

Note: the winner is never the best locally!

A homogeneous model for scoring functions

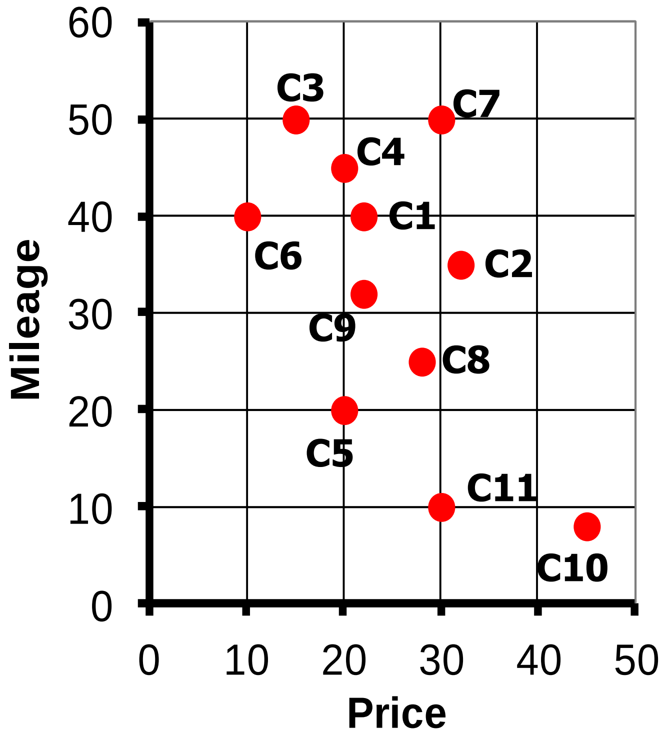
- In order to provide a unifying approach to the problem, we consider:

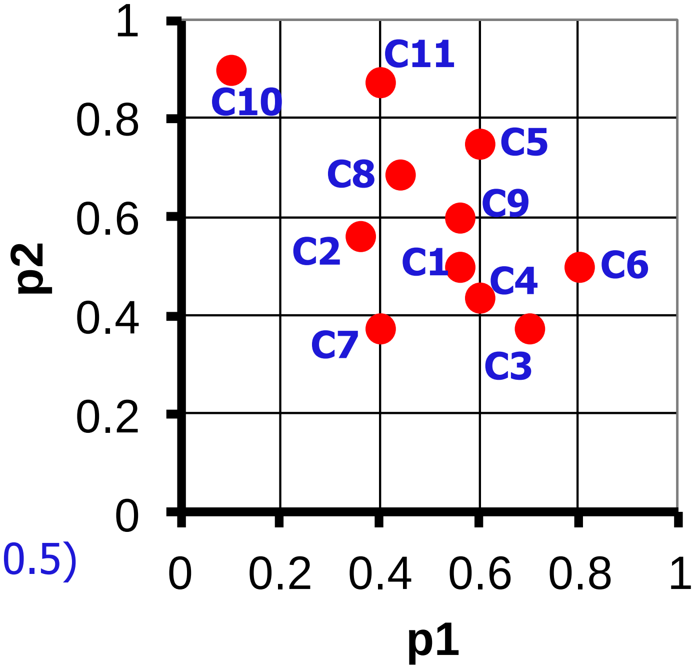
- A top-k 1-1 join query **Q** = (Q1,Q2,…,Qm)
  - Qj is the sub-query sent to the j-th source/relation
- Each object o returned by the input Lj has an associated local/partial score pj(o), pj(o) $\in$ [0,1] and dependent on Qj
  - For convenience, scores are normalized, with higher scores being better
  - This can be easly relaxed; what matters is to know which is the best and the worst possible value of pj
  - The hypercube $[0,1]^m$ is conveniently called the "score space"
- The point p(o) = (p1(o),p2(o),…,pm(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:

$$S : [0,1]^m \rightarrow \mathfrak{R} \qquad S(o) \equiv S(p(o)) = S(p1(o),p2(o),\ldots,pm(o))$$

The score space

- Consider the 2-dim attribute space **A** = (Price,Mileage)
- Let Q1 be the sub-query on Price, and Q2 the sub-query on Mileage
- We can set: p1(o) = 1 – o.Price/MaxP, p2(o) = 1 – o.Mileage/MaxM
- Let’s take MaxP = 50,000 and 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





C6: (10,40) → (0.8,0.5)

Some common scoring functions

SUM (AVG): used to equally weigh preferences

$$SUM(o) \equiv SUM(p(o)) = p1(o) + p2(o) + \ldots + pm(o)$$

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

$$WSUM(o) \equiv WSUM(p(o)) = w1*p1(o) + w2*p2(o) + \ldots + wm*pm(o)$$

MIN (Minimum): just considers the worst partial score

$$MIN(o) \equiv MIN(p(o)) = \min\{p1(o),p2(o), \ldots, pm(o)\}$$

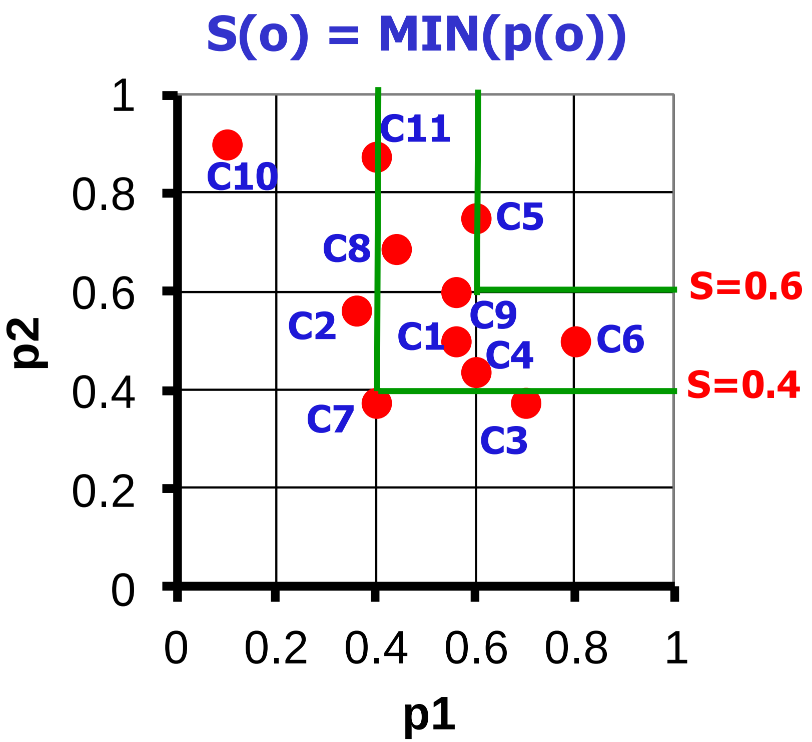
MAX (Maximum): just considers the best partial score

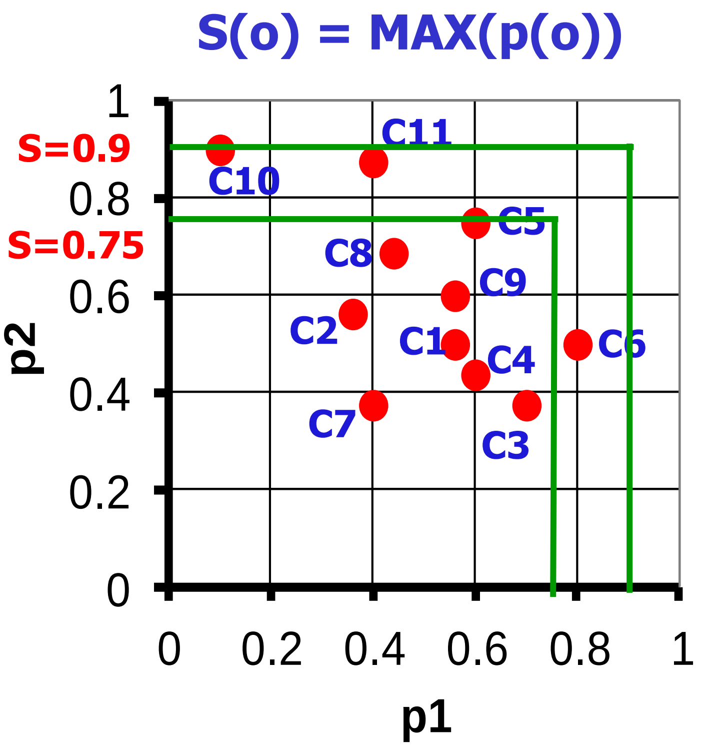
$$MAX(o) \equiv MAX(p(o)) = \max\{p1(o),p2(o), \ldots, pm(o)\}$$

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

Equallly 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) = MIN(p(o))


S(o) = 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 ≡ 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!

Ronald Fagin

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 ≡ 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!

Ronald Fagin

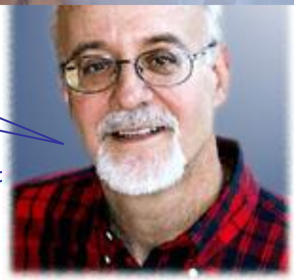
Beware! B_0 only works for MAX, other scoring functions require smarter, and more costly, algorithms

The simplest case: MAX

- We are now ready to ask "the big question":

- Fo

from each source, that's all!



Ronald Fagin

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

The B_0 algorithm

```
Input: ranked lists Lj (j=1,…,m), integer k $\geq$ 1
Output: the top-k objects according to the MAX scoring function
B := $\emptyset$;                                      // B is a main-memory buffer
for j = 1 to m:
    Obj(j) := $\emptyset$;                              // the set of objects "seen" on Lj
    for i = 1 to k:                                     // get the best k objects from each list
        t := getNext$_{Lj}$();
        Obj(j) := Obj(j) $\cup$ {t.OID};
        if t.OID was not retrieved from other lists then: INSERT(B,t)  // adds t to the buffer
                                  else: join t with the entry in B having the same OID;
for each object o $\in$ Obj := $\cup_j$ Obj(j):         // for each object with at least one partial score…
    MAX(o) := max$_j${pj(o): pj(o) is defined};         // …compute MAX using the available scores
return the k objects with maximum score;
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)

B_0 : examples

k = 2

OID	p1
o7	0.7
o3	0.65
o4	0.6
o2	0.5

OID	p2
o2	0.9
o3	0.6
o7	0.4
o4	0.2

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7

k = 3

MangiarBene

Name	Score
Al vecchio mulino	9.2
La tavernetta	9.0
Il desco	8.3
Da Gino	7.5
Tutti a tavola!	6.4
Le delizie del palato	5.5
Acqua in bocca	5.0

PaneeVino

Name	Score
Da Gino	9.0
Il desco	8.5
Al vecchio mulino	7.5
Le delizie del palato	7.5
La tavernetta	7.0
Acqua in bocca	6.5
Tutti a tavola!	6.0

B_0 : examples

k = 2

OID	p1
o7	0.7
o3	0.65
o4	0.6
o2	0.5

OID	p2
o2	0.9
o3	0.6
o7	0.4
o4	0.2

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7

k = 3

MangiarBene

Name	Score
Al vecchio mulino	9.2
La tavernetta	9.0
Il desco	8.3
Da Gino	7.5
Tutti a tavola!	6.4
Le delizie del palato	5.5
Acqua in bocca	5.0

PaneeVino

Name	Score
Da Gino	9.0
Il desco	8.5
Al vecchio mulino	7.5
Le delizie del palato	7.5
La tavernetta	7.0
Acqua in bocca	6.5
Tutti a tavola!	6.0

B_0 : examples

k = 2

OID	p1
o7	0.7
o3	0.65
o4	0.6
o2	0.5

OID	p2
o2	0.9
o3	0.6
o7	0.4
o4	0.2

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7

k = 3

MangiarBene

Name	Score
Al vecchio mulino	9.2
La tavernetta	9.0
Il desco	8.3
Da Gino	7.5
Tutti a tavola!	6.4
Le delizie del palato	5.5
Acqua in bocca	5.0

PaneeVino

Name	Score
Da Gino	9.0
Il desco	8.5
Al vecchio mulino	7.5
Le delizie del palato	7.5
La tavernetta	7.0
Acqua in bocca	6.5
Tutti a tavola!	6.0

B_0 : examples

k = 2

OID	p1
o7	0.7
o3	0.65
o4	0.6
o2	0.5

OID	p2
o2	0.9
o3	0.6
o7	0.4
o4	0.2

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7

OID	S
o7	**1.0**
o2	**0.9**
o3	0.65

k = 3

MangiarBene

Name	Score
Al vecchio mulino	9.2
La tavernetta	9.0
Il desco	8.3
Da Gino	7.5
Tutti a tavola!	6.4
Le delizie del palato	5.5
Acqua in bocca	5.0

PaneeVino

Name	Score
Da Gino	9.0
Il desco	8.5
Al vecchio mulino	7.5
Le delizie del palato	7.5
La tavernetta	7.0
Acqua in bocca	6.5
Tutti a tavola!	6.0

B_0 : examples

k = 2

OID	p1
o7	0.7
o3	0.65
o4	0.6
o2	0.5

OID	p2
o2	0.9
o3	0.6
o7	0.4
o4	0.2

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7

OID	S
o7	**1.0**
o2	**0.9**
o3	0.65

k = 3

MangiarBene

Name	Score
Al vecchio mulino	9.2
La tavernetta	9.0
Il desco	8.3
Da Gino	7.5
Tutti a tavola!	6.4
Le delizie del palato	5.5
Acqua in bocca	5.0

PaneeVino

Name	Score
Da Gino	9.0
Il desco	8.5
Al vecchio mulino	7.5
Le delizie del palato	7.5
La tavernetta	7.0
Acqua in bocca	6.5
Tutti a tavola!	6.0

B_0 : examples

k = 2

OID	p1
o7	0.7
o3	0.65
o4	0.6
o2	0.5

OID	p2
o2	0.9
o3	0.6
o7	0.4
o4	0.2

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7

OID	S
o7	**1.0**
o2	**0.9**
o3	0.65

k = 3

MangiarBene

Name	Score
Al vecchio mulino	9.2
La tavernetta	9.0
Il desco	8.3
Da Gino	7.5
Tutti a tavola!	6.4
Le delizie del palato	5.5
Acqua in bocca	5.0

PaneeVino

Name	Score
Da Gino	9.0
Il desco	8.5
Al vecchio mulino	7.5
Le delizie del palato	7.5
La tavernetta	7.0
Acqua in bocca	6.5
Tutti a tavola!	6.0

B_0 : examples

k = 2

OID	p1
o7	0.7
o3	0.65
o4	0.6
o2	0.5

OID	p2
o2	0.9
o3	0.6
o7	0.4
o4	0.2

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7

OID	S
o7	**1.0**
o2	**0.9**
o3	0.65

k = 3

MangiarBene

Name	Score
Al vecchio mulino	9.2
La tavernetta	9.0
Il desco	8.3
Da Gino	7.5
Tutti a tavola!	6.4
Le delizie del palato	5.5
Acqua in bocca	5.0

PaneeVino

Name	Score
Da Gino	9.0
Il desco	8.5
Al vecchio mulino	7.5
Le delizie del palato	7.5
La tavernetta	7.0
Acqua in bocca	6.5
Tutti a tavola!	6.0

B_0 : examples

k = 2

OID	p1
o7	0.7
o3	0.65
o4	0.6
o2	0.5

OID	p2
o2	0.9
o3	0.6
o7	0.4
o4	0.2

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7

OID	S
o7	**1.0**
o2	**0.9**
o3	0.65

k = 3

MangiarBene

Name	Score
Al vecchio mulino	9.2
La tavernetta	9.0
Il desco	8.3
Da Gino	7.5
Tutti a tavola!	6.4
Le delizie del palato	5.5
Acqua in bocca	5.0

PaneeVino

Name	Score
Da Gino	9.0
Il desco	8.5
Al vecchio mulino	7.5
Le delizie del palato	7.5
La tavernetta	7.0
Acqua in bocca	6.5
Tutti a tavola!	6.0

B_0 : examples

k = 2

OID	p1
o7	0.7
o3	0.65
o4	0.6
o2	0.5

OID	p2
o2	0.9
o3	0.6
o7	0.4
o4	0.2

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7

OID	S
o7	**1.0**
o2	**0.9**
o3	0.65

k = 3

MangiarBene

Name	Score
Al vecchio mulino	9.2
La tavernetta	9.0
Il desco	8.3
Da Gino	7.5
Tutti a tavola!	6.4
Le delizie del palato	5.5
Acqua in bocca	5.0

PaneeVino

Name	Score
Da Gino	9.0
Il desco	8.5
Al vecchio mulino	7.5
Le delizie del palato	7.5
La tavernetta	7.0
Acqua in bocca	6.5
Tutti a tavola!	6.0

Name	S
Al vecchio mulino	**9.2**
Da Gino	**9.0**
La tavernetta	**9.0**
Il desco	8.5

B_0 : examples

k = 2

OID	p1
o7	0.7
o3	0.65
o4	0.6
o2	0.5

OID	p2
o2	0.9
o3	0.6
o7	0.4
o4	0.2

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7

OID	S
o7	**1.0**
o2	**0.9**
o3	0.65

k = 3

MangiarBene

Name	Score
Al vecchio mulino	9.2
La tavernetta	9.0
Il desco	8.3
Da Gino	7.5
Tutti a tavola!	6.4
Le delizie del palato	5.5
Acqua in bocca	5.0

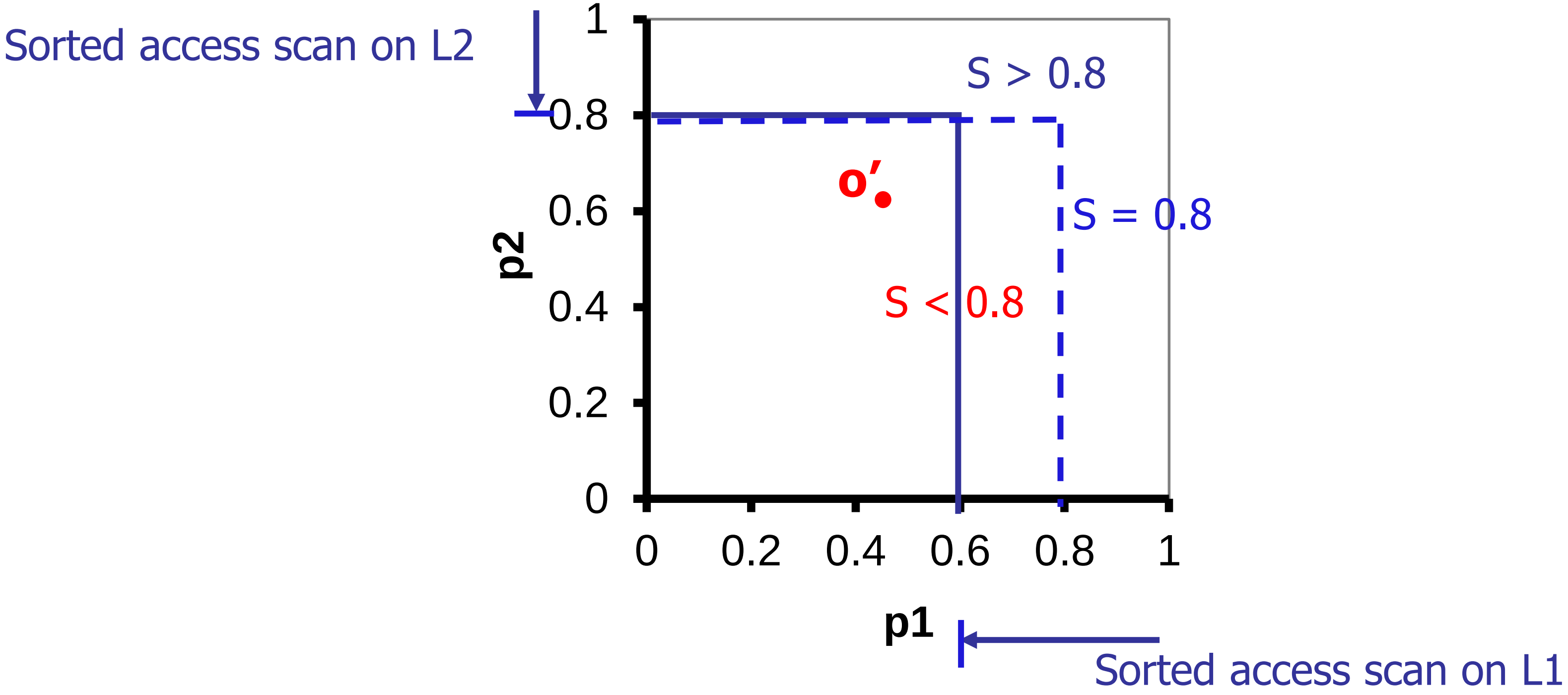
PaneeVino

Name	Score
Da Gino	9.0
Il desco	8.5
Al vecchio mulino	7.5
Le delizie del palato	7.5
La tavernetta	7.0
Acqua in bocca	6.5
Tutti a tavola!	6.0

Name	S
Al vecchio mulino	**9.2**
Da Gino	**9.0**
La tavernetta	**9.0**
Il desco	8.5

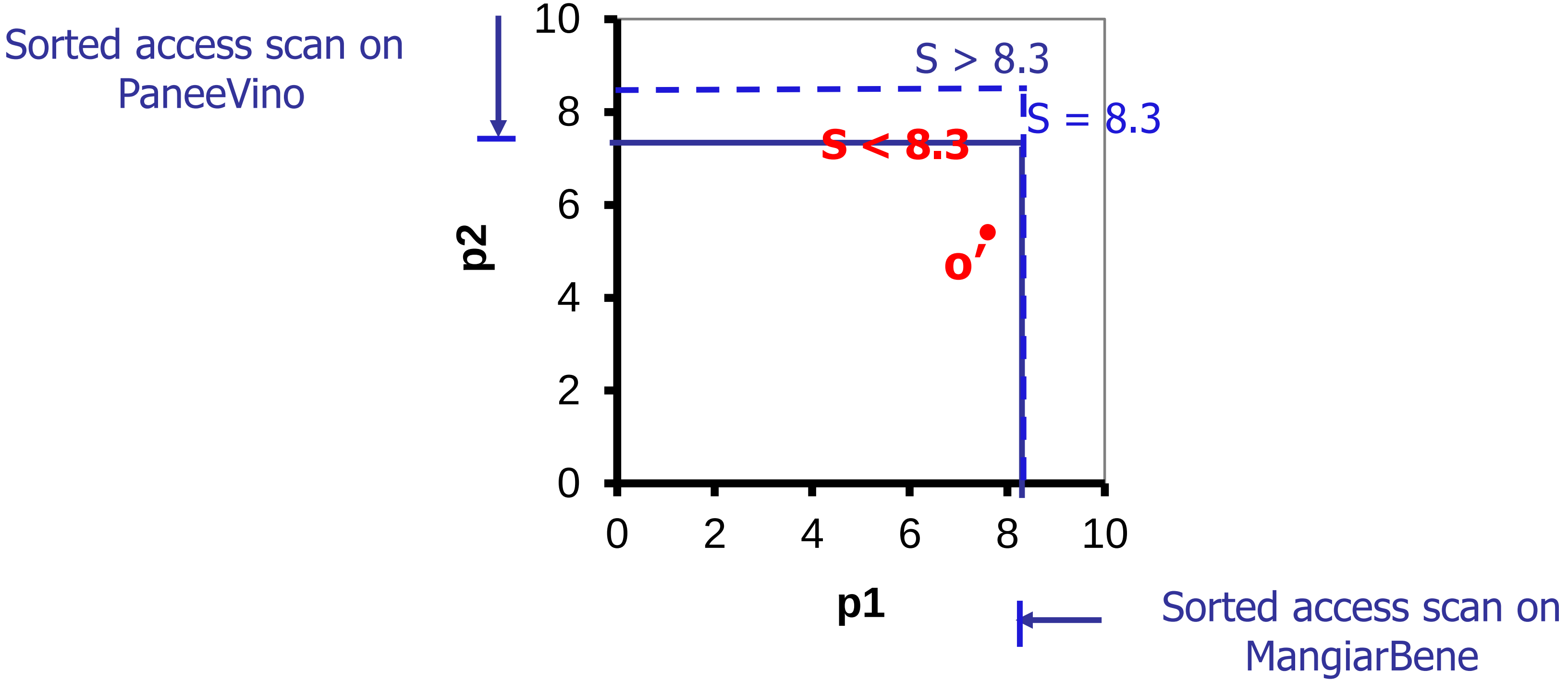
Why B_0 works: graphical intuition

- 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 $o' \notin Obj$), 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) ≥ 8.3
- A restaurant like o’, that has not been retrieved by any sorted access scan (thus o’ ∉ Obj), cannot have a global score higher than 8.3



Sorted access scan on PaneeVino
10
8
6
4
2
0
p2
S > 8.3
S = 8.3
S < 8.3
o’
0
2
4
6
8
10
p1
Sorted access scan on MangiarBene

B_0: A proof of correctness

- Let Res be the result of B_0 ($Res \subseteq Obj$)
- The need for a formal proof of correctness is motivated by the following:

  if $o \in Obj - Res$, 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 $o \notin Obj$ we have just shown that o cannot be better than any object in Res

Theorem:

The B_0 algorithm correctly determines the top-k objects and their global scores

- We split the proof in two parts:
  - We first show that **if $o \in Res$, then $S(o)$ is correct**
  - We then show that **if $o \in Obj - Res$, 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., $SB_0(o3) = 0.65 \leq S(o3) = 0.7$)
- Let o1 $\in$ Res and assume by contradiction that $SB_0(o1) < S(o1)$
- This is to say that there exists Lj such that (s.t.): o1 $\notin$ Obj(j) and S(o1) = pj(o1)
- In turn this implies that there are k objects o $\in$ Obj(j) s.t.

$$\mathbf{SB_0(o1) <} S(o1) = pj(o1) \leq pj(o) \leq \mathbf{SB_0(o)} \leq S(o) \qquad \forall o \in Obj(j)$$

- Thus o1 cannot belong to Res, a contradiction ■

OID	pj
….	….
o	**pj(o)** $\leq SB_0(o) \leq S(o)$
…	…
…	…
o1	$SB_0(o1) < S(o1) =$ **pj(o1)**

Obj(j) contains k objects

?? Impossible if o1 $\in$ Res

Proof (2): Res contains the top-k objects

- Consider an object, say o1, s.t. $o1 \in Obj - Res$
- If $SB_0(o1) = S(o1)$ then there is nothing to demonstrate ☺

- Then, assume that at least one partial score of o1, pj(o1), is not available, and that $SB_0(o1) < S(o1) = pj(o1)$. Then

$$SB_0(o1) < \mathbf{S(o1)} = pj(o1) \leq pj(o) \leq \mathbf{SB_0(o)} \leq \mathbf{S(o)} \qquad \forall o \in Obj(j)$$

- 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 $o \in Res$ ■

OID	pj
….	….
o	$\mathbf{pj(o)} \leq SB_0(o) \leq S(o)$
…	…
…	…
o1	$SB_0(o1) < S(o1) = \mathbf{pj(o1)}$

Obj(j) contains k objects

Impossible to have $S(o1) > S(o)$, $o \in Res$

Less than k rounds?

- An interesting question is: can we compute the correct result using less than k rounds of sorted accesses?

MangiarBene

Name	Score
Al vecchio mulino	9.2
La tavernetta	9.0
Il desco	8.3
…	…

PaneeVino

Name	Score
Da Gino	9.0
Il desco	8.5
Al vecchio mulino	7.5
…	…

Name	S
Al vecchio mulino	9.2
Da Gino	9.0
La tavernetta	9.0
Il desco	8.5

k=1: 1 round is required

- Althoug 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 Lj, let pj denote the lowest score seen so far
- Let Res denote the current set of top-k objects, ordered by their current MAX value; thus, Res[k].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 Res[k].score ≥ $\max_j\{pj\}$

Proof:

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

(only if) Assume that the algorithm stops when Res[k].score < maxj{pj}. Then, a list Lj, with pj > Res[k].score might contain an object o s.t. pj(o) > Res[k].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:

OID	p1
o9	0.7
…	…

OID	p2
o2	0.9
o3	0.6
o5	0.4
…	…

OID	p3
o1	0.8
o2	0.8
…	…

The MaxOptimal algorithm

- 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 $\underline{pj^*}$ is maximum: $j^* = \text{argmax}_j\{\underline{pj}\}$
  - It only keeps in memory the k best objects found so far

Input: ranked lists Lj (j=1,…,m), integer k $\geq$ 1

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

```
for i=1 to k: Res[i] := [null,0];          // entry of type: [OID,score] (+ other attr.'s as needed)
for j = 1 to m: $\underline{pj}$ = 1;                 // the best possible score on Lj
while Res[k].score < $\max_j\{\underline{pj}\}$:
     j* = $\text{argmax}_j\{\underline{pj}\}$; t := $\text{getNext}_{Lj^*}$();        // get the next best object from list Lj*
     if t.pj* > Res[k].score then:
          if t.OID $\in$ Res.OID then: update the entry in Res having the same OID
               else: {remove the object in Res[k]; insert [t.OID,t.pj*] in Res};
return Res;
end.
```

MaxOptimal: example

k = 4

L1

OID	p1
o1	0.95
o7	0.85
o4	0.6
…	…

L2

OID	p2
o2	0.8
o3	0.7
…	…

L3

OID	p3
o7	0.6
…	…

MaxOptimal: example

k = 4

L1

OID	p1
o1	0.95
o7	0.85
o4	0.6
…	…

L2

OID	p2
o2	0.8
o3	0.7
…	…

L3

OID	p3
o7	0.6
…	…

after the 1st round

Res

OID	Score (S)
o1	0.95
o2	0.8
o7	0.6
---	0

$0 < 0.95$

MaxOptimal: example

k = 4

L1

OID	p1
o1	0.95
o7	0.85
o4	0.6
…	…

L2

OID	p2
o2	0.8
o3	0.7
…	…

L3

OID	p3
o7	0.6
…	…

Res

after the 1st round

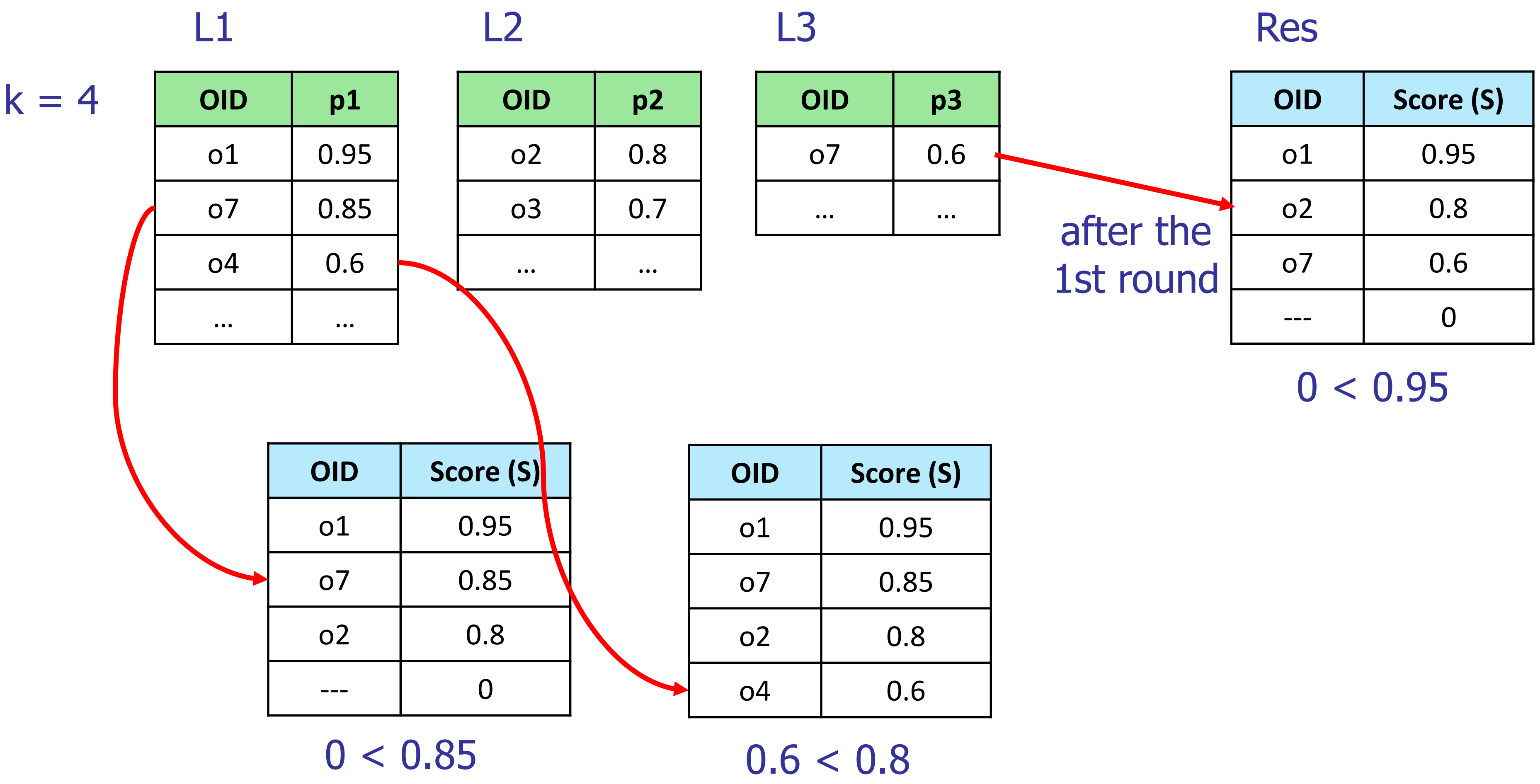
OID	Score (S)
o1	0.95
o2	0.8
o7	0.6
---	0

0 < 0.95

OID	Score (S)
o1	0.95
o7	0.85
o2	0.8
---	0

0 < 0.85

MaxOptimal: example



k = 4

L1

OID	p1
o1	0.95
o7	0.85
o4	0.6
…	…

L2

OID	p2
o2	0.8
o3	0.7
…	…

L3

OID	p3
o7	0.6
…	…

after the 1st round

Res

OID	Score (S)
o1	0.95
o2	0.8
o7	0.6
---	0

$0 < 0.95$

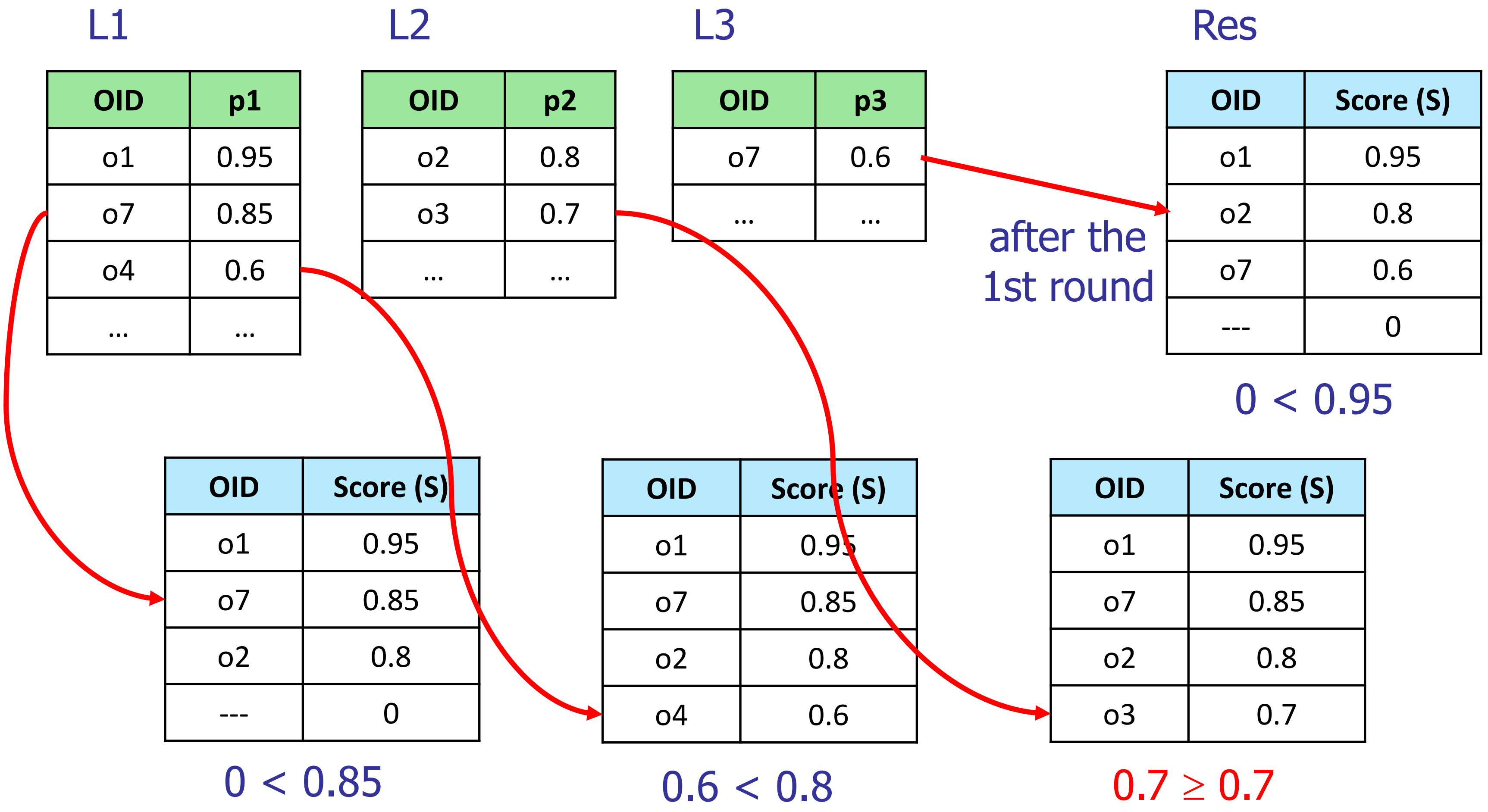
OID	Score (S)
o1	0.95
o7	0.85
o2	0.8
---	0

$0 < 0.85$

OID	Score (S)
o1	0.95
o7	0.85
o2	0.8
o4	0.6

$0.6 < 0.8$

MaxOptimal: example



k = 4

L1

OID	p1
o1	0.95
o7	0.85
o4	0.6
…	…

L2

OID	p2
o2	0.8
o3	0.7
…	…

L3

OID	p3
o7	0.6
…	…

Res

OID	Score (S)
o1	0.95
o2	0.8
o7	0.6
---	0

after the 1st round

$0 < 0.95$

OID	Score (S)
o1	0.95
o7	0.85
o2	0.8
---	0

$0 < 0.85$

OID	Score (S)
o1	0.95
o7	0.85
o2	0.8
o4	0.6

$0.6 < 0.8$

OID	Score (S)
o1	0.95
o7	0.85
o2	0.8
o3	0.7

$0.7 \geq 0.7$

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 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 Lj for which it is $\underline{pj} < MAX_k$

Proof: By contradiction, assume that MaxOptimal performs a s.a. on list Lj*, with $\underline{pj*} < MAX_k$. For this it has to be $\underline{pj*} = max_j\{\underline{pj}\}$, from which it is derived $MAX_k > \underline{pj}$, for each j. Before performing this s.a. it is Res[k].score < MAX_k, otherwise the algorithm would halt. But this implies that there exists an unseen object whose global score is MAX_k, which is impossible. ■

Why B_0 **doesn't** work for other scoring f.'s

- Let S ≡ MIN and k = 1

OID	p1
o7	0.9
o3	0.65
o2	0.6
o1	0.5
o4	0.4

OID	p2
o2	0.95
o3	0.7
o4	0.6
o1	0.5
o7	0.5

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7
o1	0.6

OID	S
o2	**0.95**
o7	0.9

WRONG!!

- 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:

$getScore_{L1}(o2), getScore_{L3}(o2), getScore_{L2}(o7)$

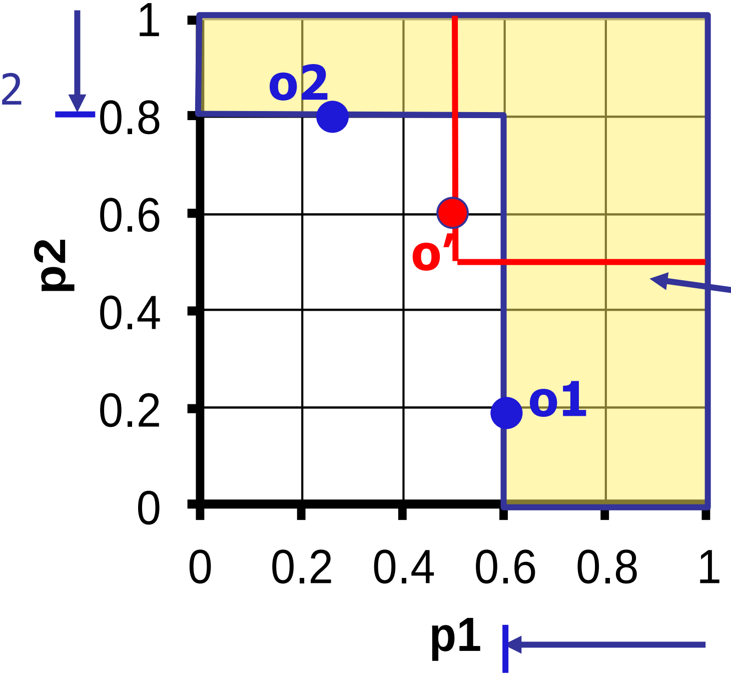
we get:

OID	S
o2	**0.6**
o7	0.5

STILL WRONG!!? ☹

Why B_0 **doesn't** work: graphical intuition

- Let S ≡ 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 ≡ SUM



Sorted access scan on L2

Retrieved objects are somewhere in this region

Sorted access scan on L1

The FA algorithm: monotone scoring f.’s

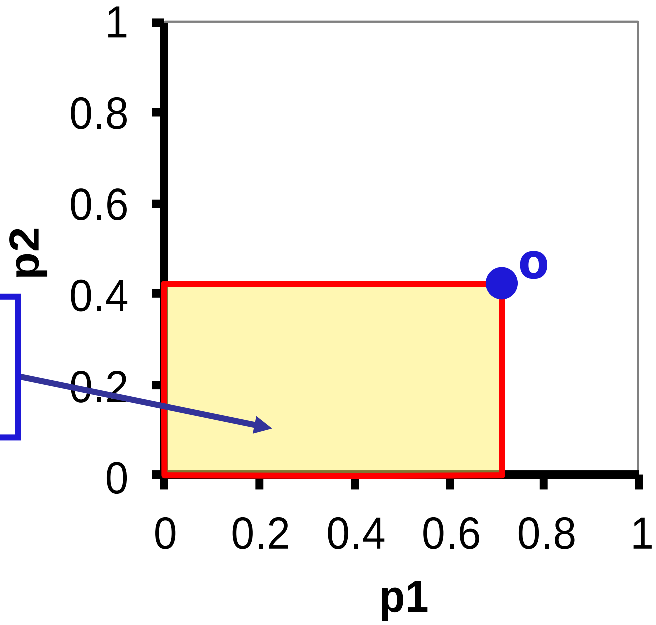
- The FA (or A_0) algorithm [Fag96] can be used to solve top-k 1-1 join queries with **any** monotone scoring function S:

Monotone scoring function:

- An m-ary scoring function S is monotone if

$$x1 \leq y1,\ x2 \leq y2,\ \ldots,\ xm \leq ym \quad \Rightarrow \quad S(x1,x2,\ldots,xm) \leq S(y1,y2,\ldots,ym)$$

- 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 Lj (j=1,…,m), integer $k \geq 1$, monotone scoring function S

Output: the top-k objects according to S

```
// 1st phase: sorted accesses
for j = 1 to m: Obj(j) := $\emptyset$; B := $\emptyset$; M := $\emptyset$;
while $|M| < k$:
    for j = 1 to m:
        t := getNext$_{Lj}$(); Obj(j) := Obj(j) $\cup$ {t.OID};  // get the next best object from list Lj
        if t.OID was not retrieved from other lists then: INSERT(B,t)
                                        else: join t with the entry in B having the same OID;
    M := $\cap_j$ Obj(j);       // the set of objects seen on all the m lists
// 2nd phase: random accesses
for each object o $\in$ Obj := $\cup_j$ 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
for each object o $\in$ Obj: compute S(o);
return the k objects with maximum score;
end.
```

How FA works

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

OID	p1
o7	0.9
o3	0.65
o2	0.6
o1	0.5
o4	0.4

OID	p2
o2	0.95
o3	0.7
o4	0.6
o1	0.5
o7	0.5

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7
o1	0.6

and after the sorted accesses we obtain:

How FA works

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

OID	p1
o7	0.9
o3	0.65
o2	0.6
o1	0.5
o4	0.4

OID	p2
o2	0.95
o3	0.7
o4	0.6
o1	0.5
o7	0.5

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7
o1	0.6

and after the sorted accesses we obtain:

M = {o2}
Obj = {o2,o3,o4,o7}

How FA works

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

OID	p1
o7	0.9
o3	0.65
o2	0.6
o1	0.5
o4	0.4

OID	p2
o2	0.95
o3	0.7
o4	0.6
o1	0.5
o7	0.5

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7
o1	0.6

and after the sorted accesses we obtain:

M = {o2}
Obj = {o2,o3,o4,o7}

- After performing the needed random accesses we get:

S ≡ MIN

OID	S
o3	**0.65**
o2	0.6
o7	0.5
o4	0.4

RIGHT!!

How FA works

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

OID	p1
o7	0.9
o3	0.65
o2	0.6
o1	0.5
o4	0.4

OID	p2
o2	0.95
o3	0.7
o4	0.6
o1	0.5
o7	0.5

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7
o1	0.6

and after the sorted accesses we obtain:

M = {o2}
Obj = {o2,o3,o4,o7}

- After performing the needed random accesses we get:

S ≡ MIN

OID	S
o3	**0.65**
o2	0.6
o7	0.5
o4	0.4

RIGHT!!

☺

S ≡ SUM

OID	S
o7	**2.4**
o2	2.35
o3	2.05
o4	1.75

RIGHT!!

Why FA is correct: formal and intuitive

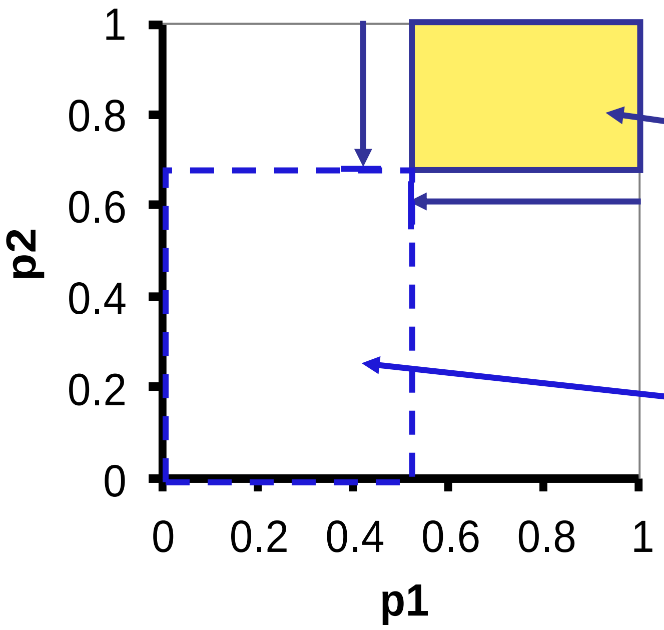
Theorem:

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’ $\notin$ Obj, then o’ cannot be better than any object o $\in$ 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’ $\notin$ Obj, for each Lj it is pj(o’) $\leq$ pj(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:

$$\boxed{O\left(N^{(m-1)/m} k^{1/m}\right)}$$

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] difffers 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 Lj (j=1,…,m), integer k ≥ 1, monotone scoring function S

Output: the top-k objects according to S

```
for i = 1 to k: Res[i] := [null,0];
for j = 1 to m: pj := 1;
while Res[k].score < T := S(p1,p2,…,pm):          // T is the "threshold"
  for j = 1 to m:
      t := getNext$_{Lj}$(); o := t.OID;
      perform random accesses to retrieve the missing partial scores for o;
      if S(o) := S(p1(o),…,pm(o)) > Res[k].score then:
            {remove the object in Res[k]; insert [o,S(o)] in Res};
  return Res;
  end.
```

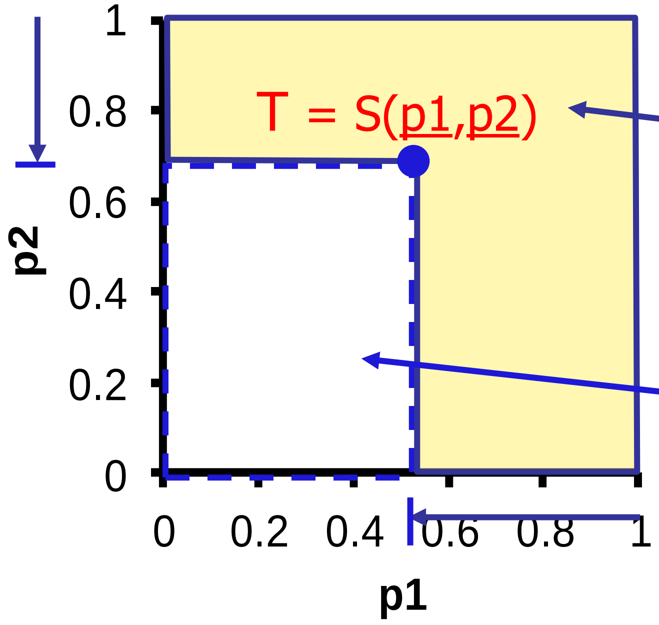
Why TA is correct: formal and intuitive

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 pj(o') ≤ pj. Due to the monotonicity of S this implies S(o') ≤ T.

By definition of Res, for each object o ∈ Res it is S(o) ≥ T, thus S(o') ≤ S(o) ■



TA stops when this region contains k points at least as good as T…

…and no object here can be better than T!

How TA works

- Let’s take S ≡ MIN and k = 1

OID	p1
o7	0.9
o3	0.65
o2	0.6
o1	0.5
o4	0.4

OID	p2
o2	0.95
o3	0.7
o4	0.6
o1	0.5
o7	0.5

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7
o1	0.6

OID	p1
o7	0.9
o3	0.65
o2	0.6
o1	0.5
o4	0.4

OID	p2
o2	0.95
o3	0.7
o4	0.6
o1	0.5
o7	0.5

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7
o1	0.6

How TA works

- Let’s take S ≡ MIN and k = 1

OID	p1
o7	0.9
o3	0.65
o2	0.6
o1	0.5
o4	0.4

OID	p2
o2	0.95
o3	0.7
o4	0.6
o1	0.5
o7	0.5

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7
o1	0.6

S(o2) = 0.6. T = 0.9

OID	p1
o7	0.9
o3	0.65
o2	0.6
o1	0.5
o4	0.4

OID	p2
o2	0.95
o3	0.7
o4	0.6
o1	0.5
o7	0.5

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7
o1	0.6

How TA works

- Let’s take S ≡ MIN and k = 1

OID	p1
o7	0.9
o3	0.65
o2	0.6
o1	0.5
o4	0.4

OID	p2
o2	0.95
o3	0.7
o4	0.6
o1	0.5
o7	0.5

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7
o1	0.6

S(o2) = 0.6. T = 0.9

S(o3) = 0.65. T = 0.65

OID	p1
o7	0.9
o3	0.65
o2	0.6
o1	0.5
o4	0.4

OID	p2
o2	0.95
o3	0.7
o4	0.6
o1	0.5
o7	0.5

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7
o1	0.6

How TA works

- Let’s take S ≡ MIN and k = 1

OID	p1	OID	p2	OID	p3
o7	0.9	o2	0.95	o7	1.0
o3	0.65	o3	0.7	o2	0.8
o2	0.6	o4	0.6	o4	0.75
o1	0.5	o1	0.5	o3	0.7
o4	0.4	o7	0.5	o1	0.6

S(o2) = 0.6. T = 0.9

S(o3) = 0.65. T = 0.65

- Let’s take S ≡ SUM and k = 2

OID	p1	OID	p2	OID	p3
o7	0.9	o2	0.95	o7	1.0
o3	0.65	o3	0.7	o2	0.8
o2	0.6	o4	0.6	o4	0.75
o1	0.5	o1	0.5	o3	0.7
o4	0.4	o7	0.5	o1	0.6

How TA works

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

OID	p1	OID	p2	OID	p3
o7	0.9	o2	0.95	o7	1.0
o3	0.65	o3	0.7	o2	0.8
o2	0.6	o4	0.6	o4	0.75
o1	0.5	o1	0.5	o3	0.7
o4	0.4	o7	0.5	o1	0.6

S(o2) = 0.6. T = 0.9

S(o3) = 0.65. T = 0.65

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

OID	p1	OID	p2	OID	p3
o7	0.9	o2	0.95	o7	1.0
o3	0.65	o3	0.7	o2	0.8
o2	0.6	o4	0.6	o4	0.75
o1	0.5	o1	0.5	o3	0.7
o4	0.4	o7	0.5	o1	0.6

S(o7) = 2.4; S(o2) = 2.35. T = 2.85

How TA works

- Let’s take S ≡ MIN and k = 1

OID	p1	OID	p2	OID	p3
o7	0.9	o2	0.95	o7	1.0
o3	0.65	o3	0.7	o2	0.8
o2	0.6	o4	0.6	o4	0.75
o1	0.5	o1	0.5	o3	0.7
o4	0.4	o7	0.5	o1	0.6

S(o2) = 0.6. T = 0.9

S(o3) = 0.65. T = 0.65

- Let’s take S ≡ SUM and k = 2

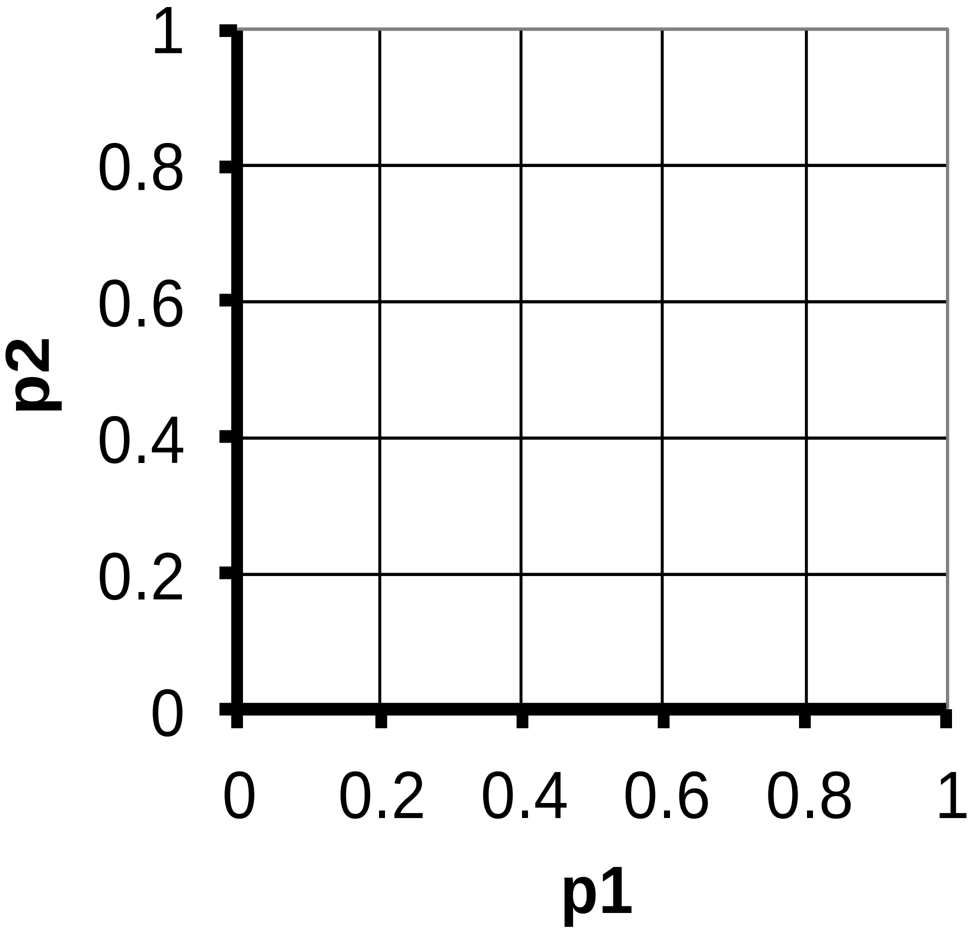
OID	p1	OID	p2	OID	p3
o7	0.9	o2	0.95	o7	1.0
o3	0.65	o3	0.7	o2	0.8
o2	0.6	o4	0.6	o4	0.75
o1	0.5	o1	0.5	o3	0.7
o4	0.4	o7	0.5	o1	0.6

S(o7) = 2.4; S(o2) = 2.35. T = 2.85

S(o7) = 2.4; S(o2) = 2.35. T = 2.15

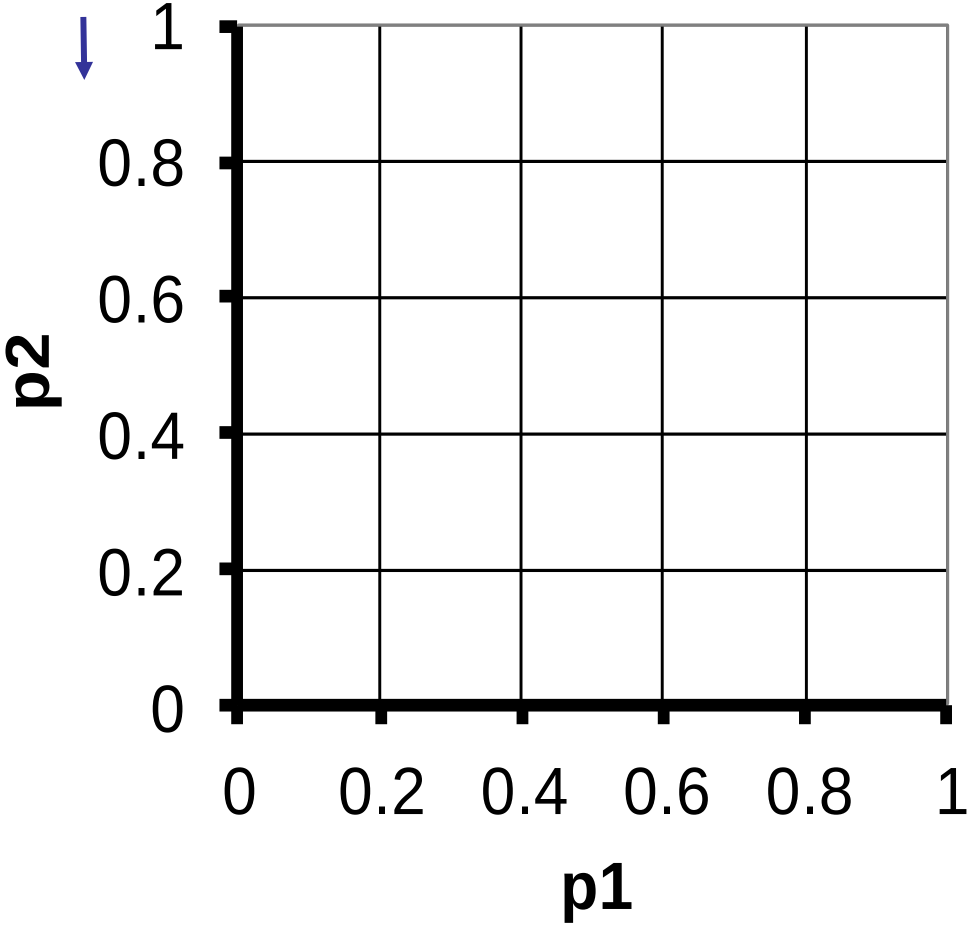
The geometric view

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



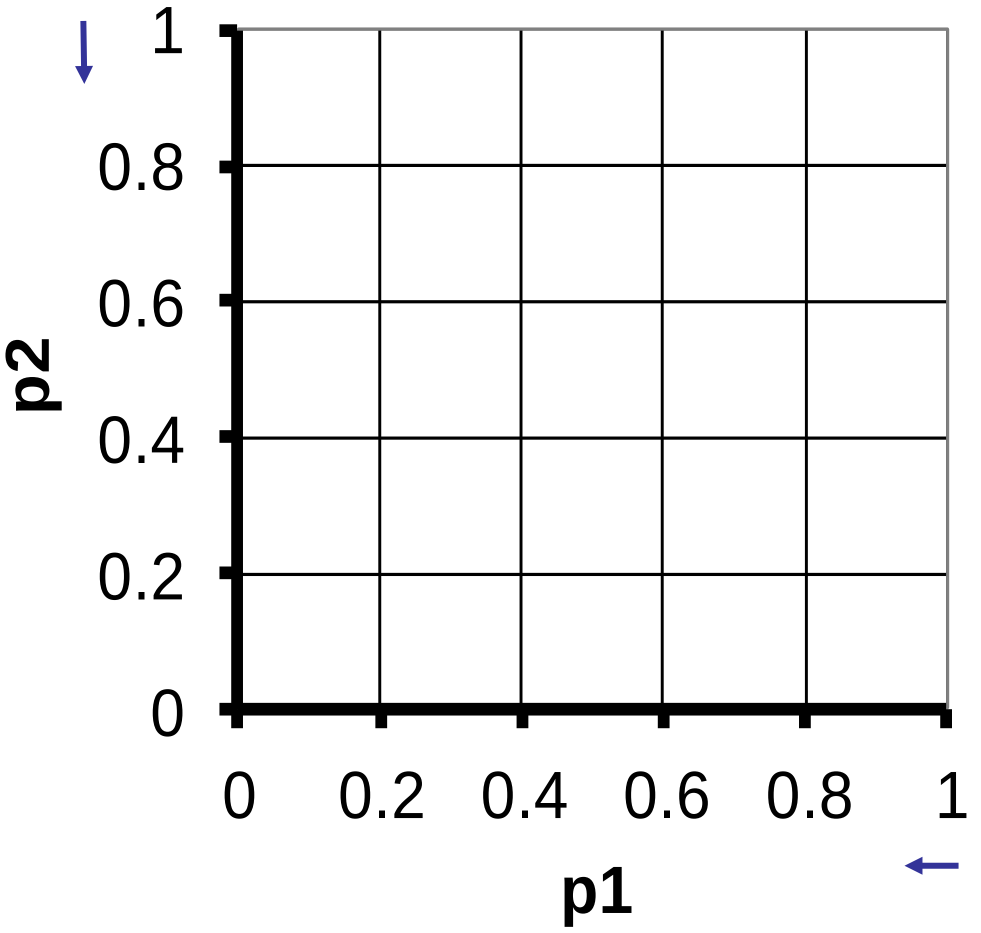
The geometric view

- Let’s take S ≡ SUM and $k = 2$



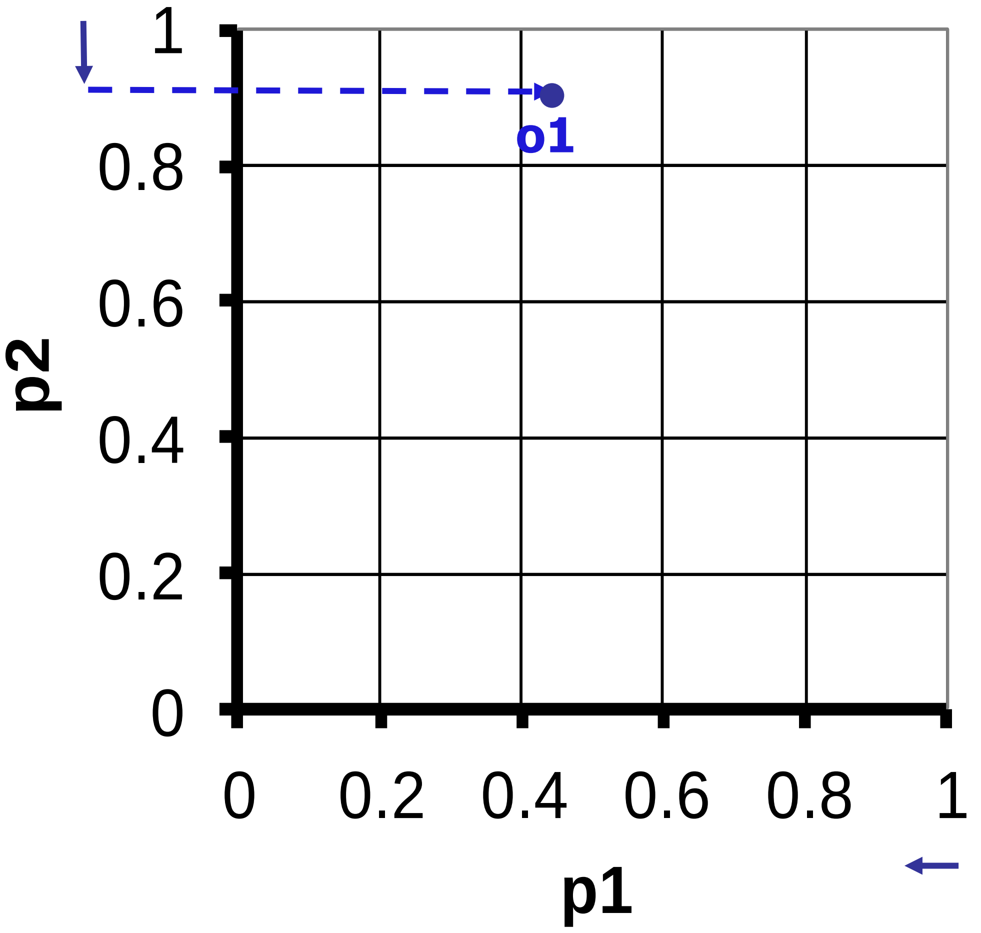
The geometric view

- Let’s take S ≡ SUM and k = 2



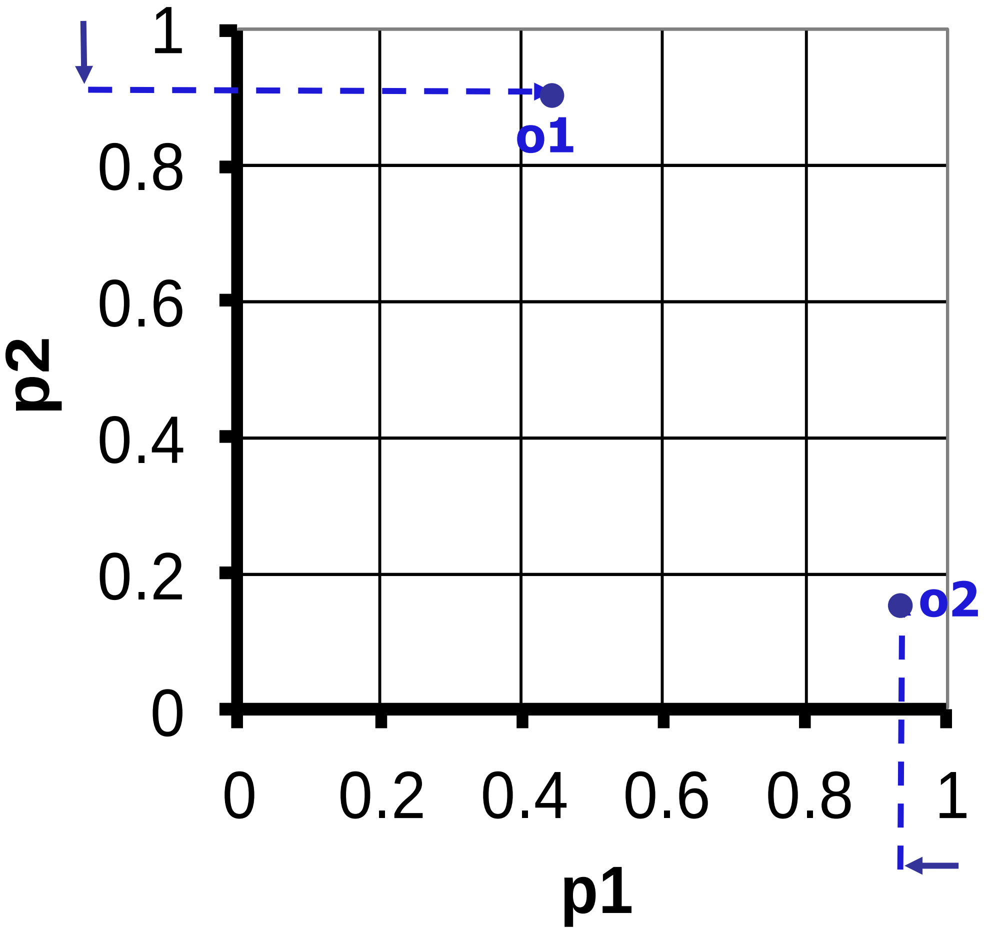
The geometric view

- Let’s take S ≡ SUM and k = 2



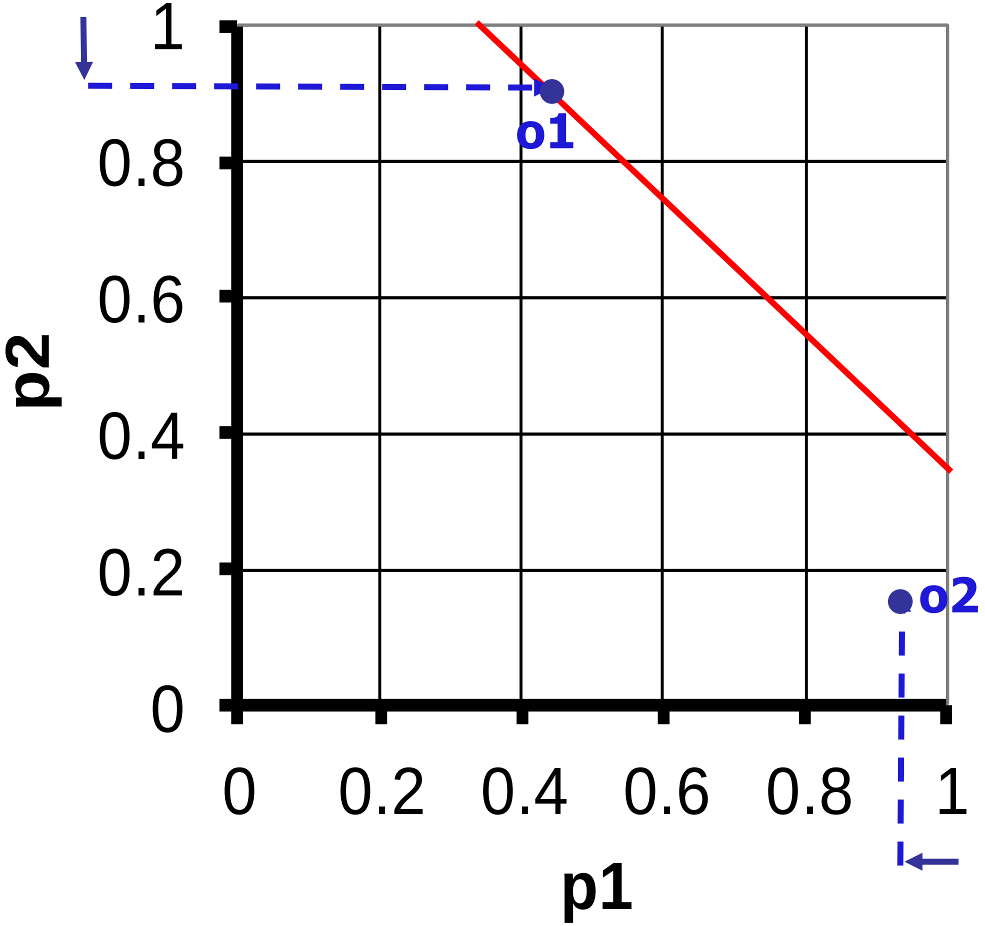
The geometric view

- Let’s take S ≡ SUM and k = 2



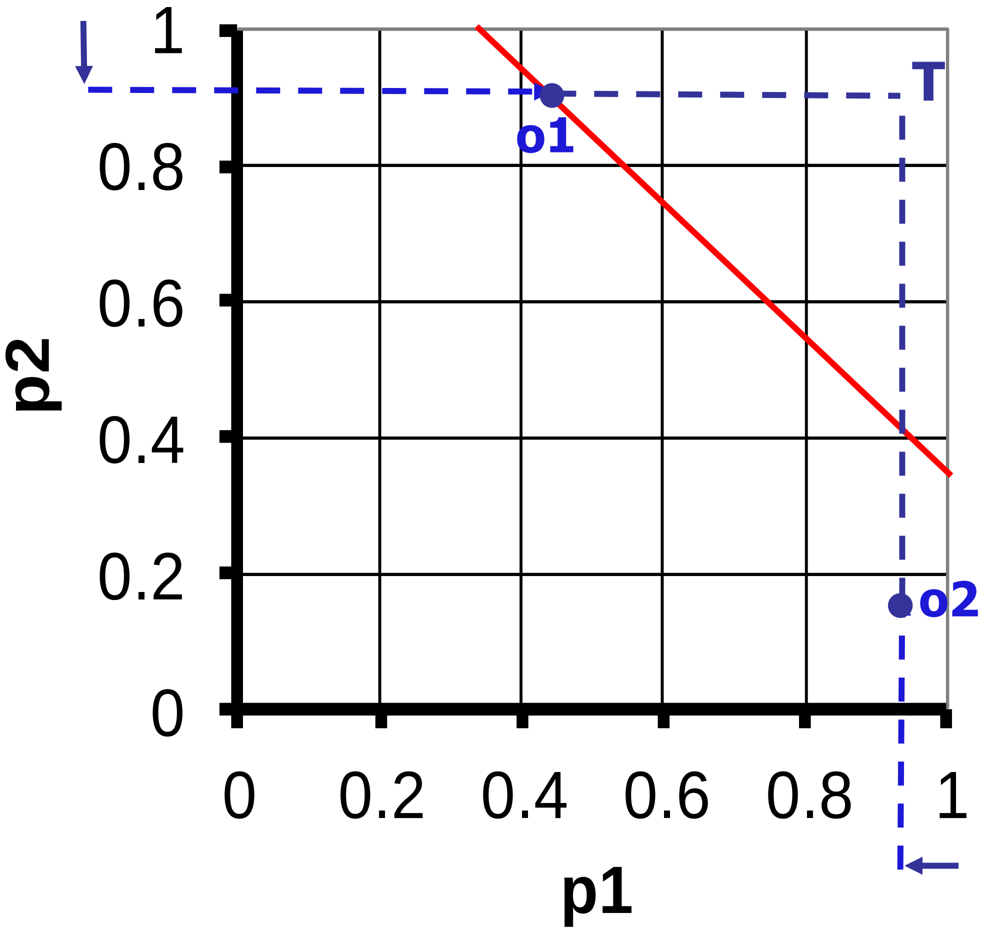
The geometric view

- Let’s take S ≡ SUM and k = 2



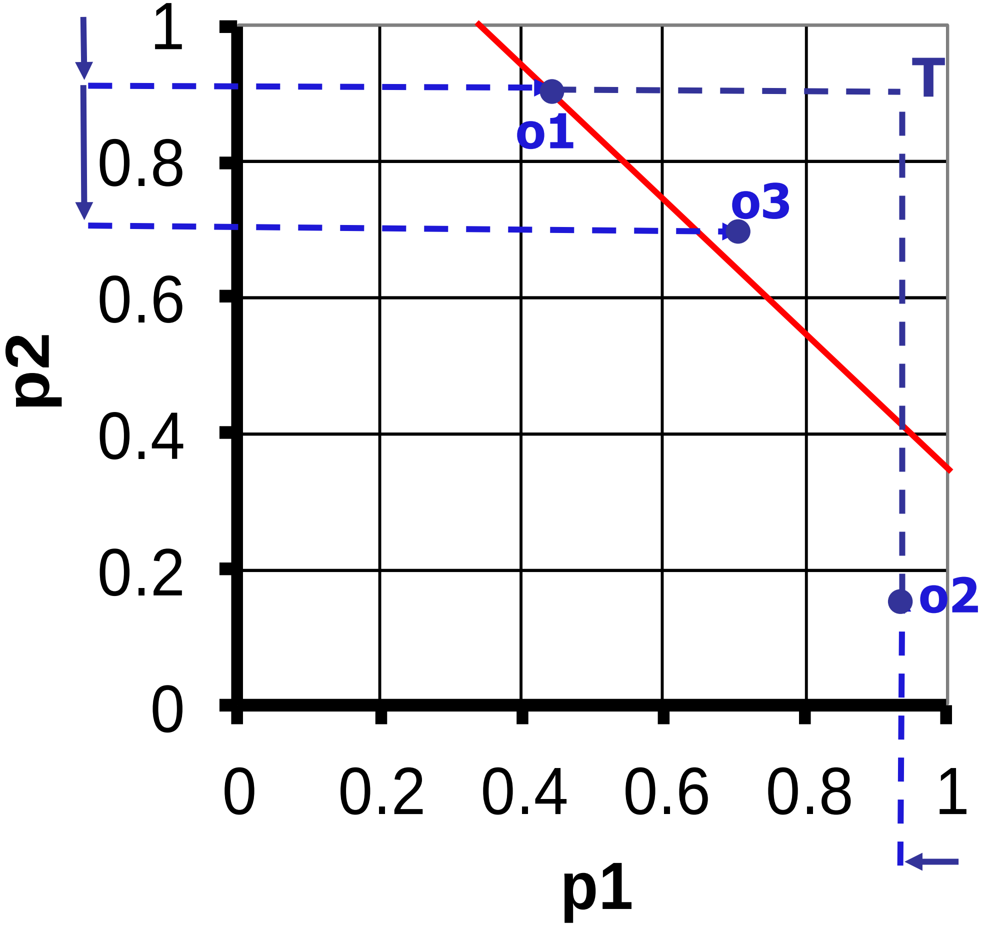
The geometric view

- Let’s take S ≡ SUM and k = 2



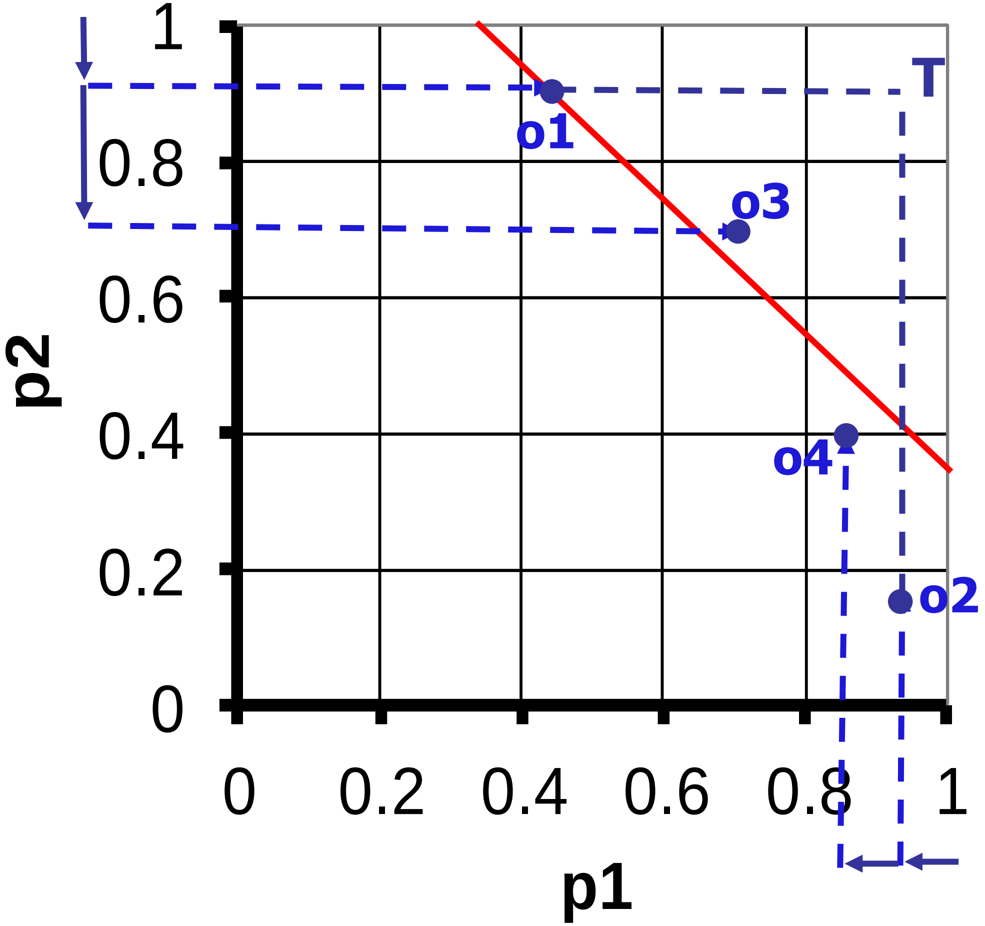
The geometric view

- Let’s take S ≡ SUM and k = 2



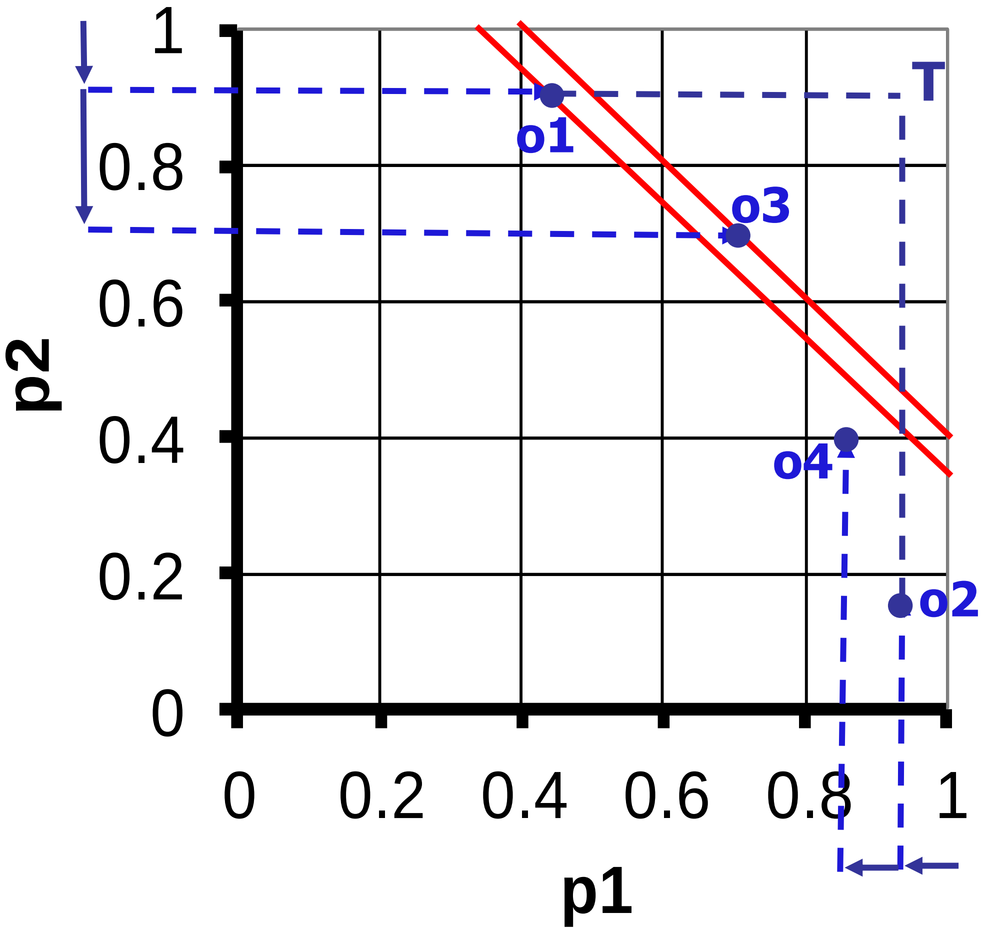
The geometric view

- Let’s take S ≡ SUM and k = 2



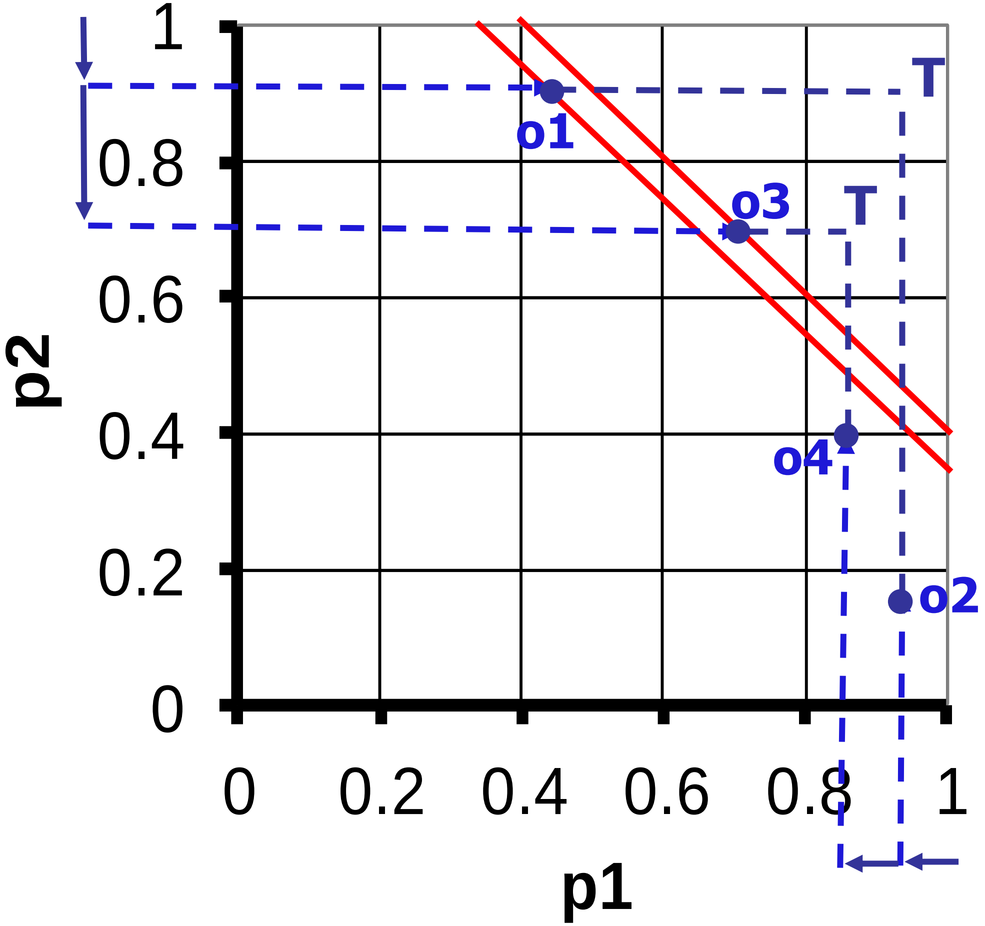
The geometric view

- Let’s take S ≡ SUM and k = 2



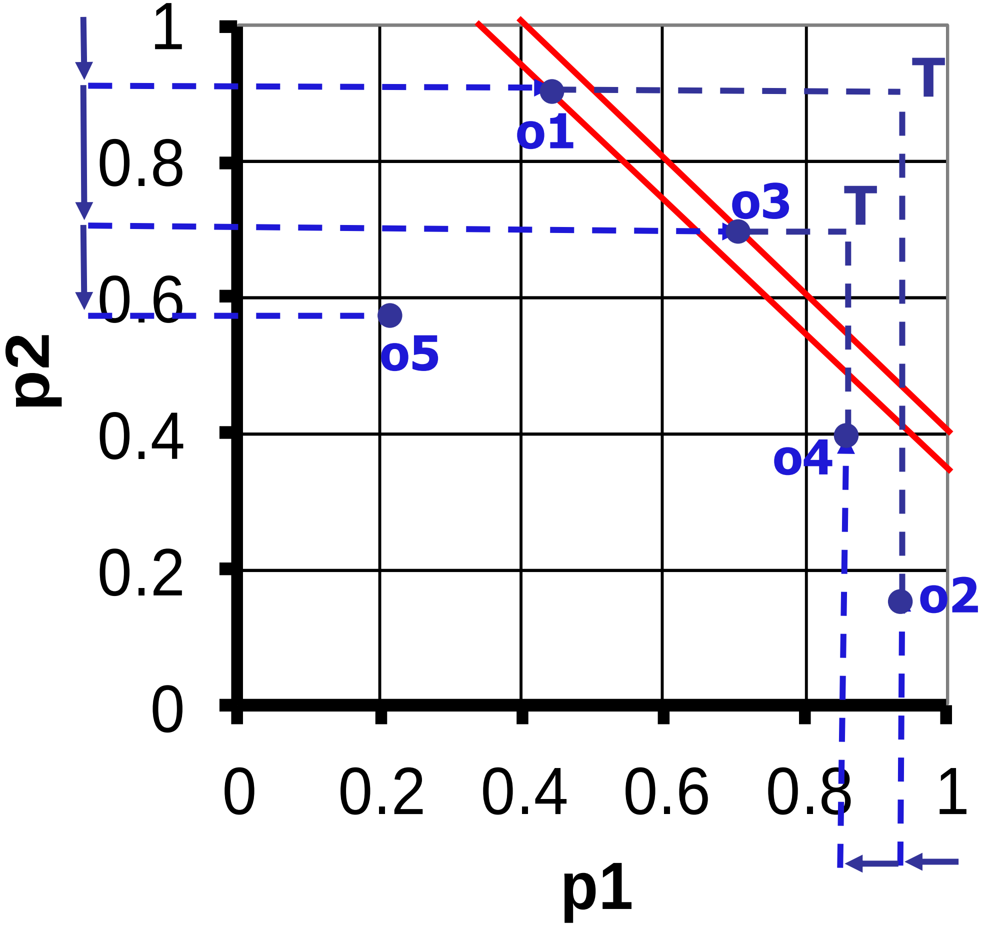
The geometric view

- Let’s take S ≡ SUM and k = 2



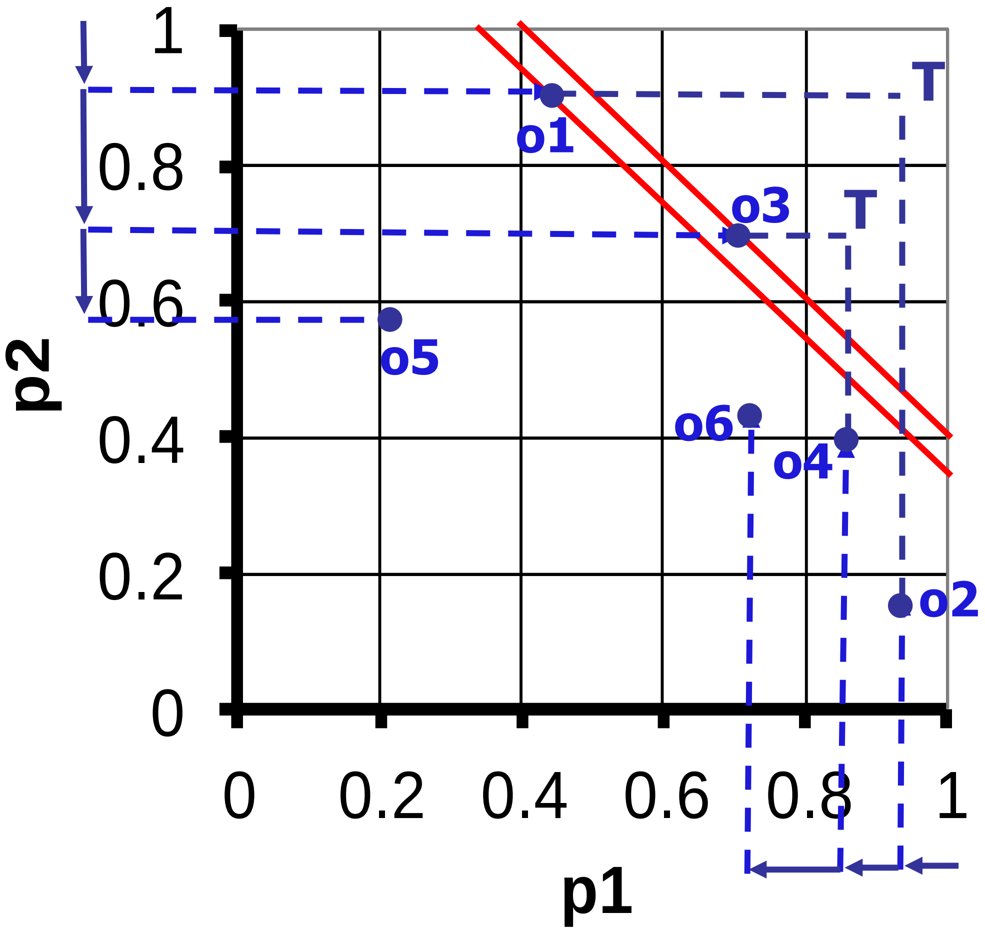
The geometric view

- Let’s take S ≡ SUM and k = 2



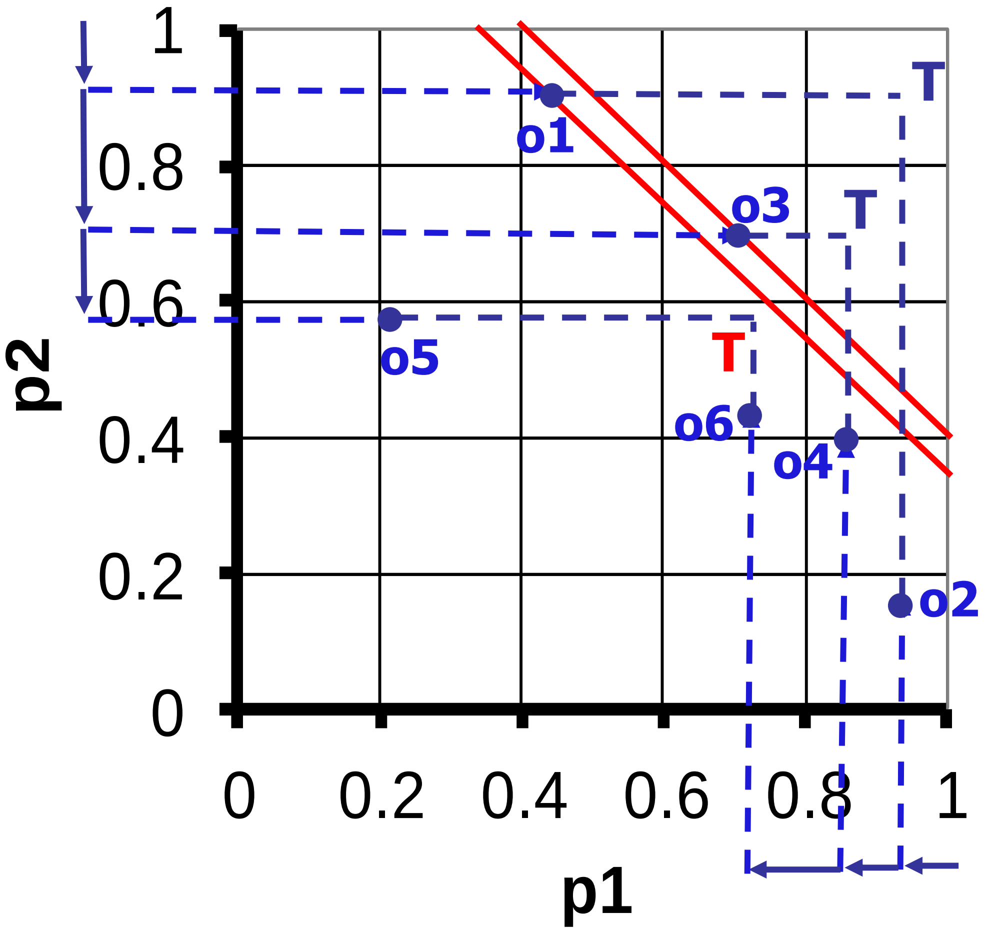
The geometric view

- Let’s take S ≡ SUM and k = 2



The geometric view

- Let's take S ≡ SUM and k = 2



S(o3) > S(o1) ≥ 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: $cost = SA*c_{SA} + RA*c_{RA}$, 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 **A** and a class **D** of DB's (inputs of the algorithms), an algorithm A $\in$ **A** is instance-optimal over **A** and **D**, for a given cost measure, if for every B $\in$ **A** and every DB $\in$ **D** it is

$$cost(A,DB) = O(cost(B,DB))$$

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

$$cost(A,DB) \leq c*cost(B,DB) + c'$$

- 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 Lj 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) ≥ X, thus its cost is ≥ 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

  $$cost(TA,DB) = m*X*c_{SA} + m*X*(m-1)*c_{RA}$$

  since in the worst case TA retrieves m*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

  $$cost(B,DB) \geq X*c_{SA}$$
- 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 pj(o) = 0 (or the minimal possible value of pj) if o has not been seen on Lj
  - $S^+(o)$ is obtained by setting pj(o) = $\underline{pj}$ if o has not been seen on Lj
- NRA uses a buffer B with unlimited capacity, which is kept sorted according to decreasing lbscore values

S ≡ SUM

L1

OID	p1
o1	1.0
o7	0.9
…	…

L2

OID	p2
o2	0.8
o3	0.75
…	…

L3

OID	p3
o7	0.6
o2	0.6
…	…

B

OID	lbscore	ubscore
o7	1.5	2.25
o2	1.4	2.3
o1	1.0	2.35
o3	0.75	2.25

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

$$S^{+}(o') \leq S^{-}(o) \qquad \forall\ o' \notin Res,\ o \in Res$$

- 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 $\geq$ 1, monotone scoring function S

Output: the top-k objects according to S

```
B := ∅; // entry of type: [OID,lbscore,ubscore]; B is ordered by decreasing lbscore values
for j = 1 to m: $\underline{pj}$ := 1;
 while B[k].lbscore < max{max{B[i].ubscore, i > k},S($\underline{p1}$,$\underline{p2}$,…,$\underline{pm}$)}:
   for j = 1 to m:
        t := getNext$_{Lj}$(); o := t.OID; insert [o,$S^{-}$(o),$S^{+}$(o)] in B;
return {B[1],…,B[k]};
end.
```

NRA: example

L1

OID	p1
o1	1.0
o7	0.9
o2	0.7
o6	0.2
…	…

L2

OID	p2
o2	0.8
o3	0.75
o4	0.5
o1	0.4
…	…

L3

OID	p3
o7	0.6
o2	0.6
o3	0.5
o5	0.1
…	…

S ≡ SUM
k = 2

NRA: example

L1

OID	p1
o1	1.0
o7	0.9
o2	0.7
o6	0.2
…	…

L2

OID	p2
o2	0.8
o3	0.75
o4	0.5
o1	0.4
…	…

L3

OID	p3
o7	0.6
o2	0.6
o3	0.5
o5	0.1
…	…

S ≡ SUM
k = 2

B (1st round)

OID	lbscore	ubscore
o1	1.0	2.4
o2	0.8	2.4
o7	0.6	2.4

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

NRA: example

L1

OID	p1
o1	1.0
o7	0.9
o2	0.7
o6	0.2
…	…

L2

OID	p2
o2	0.8
o3	0.75
o4	0.5
o1	0.4
…	…

L3

OID	p3
o7	0.6
o2	0.6
o3	0.5
o5	0.1
…	…

$S \equiv \text{SUM}$
$k = 2$

B (1st round)

OID	lbscore	ubscore
o1	1.0	2.4
o2	0.8	2.4
o7	0.6	2.4

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

B (2nd round)

OID	lbscore	ubscore
o7	1.5	2.25
o2	1.4	2.3
o1	1.0	2.35
o3	0.75	2.25

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

NRA: example

L1

OID	p1
o1	1.0
o7	0.9
o2	0.7
o6	0.2
…	…

L2

OID	p2
o2	0.8
o3	0.75
o4	0.5
o1	0.4
…	…

L3

OID	p3
o7	0.6
o2	0.6
o3	0.5
o5	0.1
…	…

$S \equiv$ SUM
$k = 2$

B (1st round)

OID	lbscore	ubscore
o1	1.0	2.4
o2	0.8	2.4
o7	0.6	2.4

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

B (2nd round)

OID	lbscore	ubscore
o7	1.5	2.25
o2	1.4	2.3
o1	1.0	2.35
o3	0.75	2.25

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

B (3rd round)

OID	lbscore	ubscore
o2	2.1	2.1
o7	1.5	2.0
o3	1.25	1.95
o1	1.0	2.0
o4	0.5	1.7

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

NRA: example

L1

OID	p1
o1	1.0
o7	0.9
o2	0.7
o6	0.2
…	…

L2

OID	p2
o2	0.8
o3	0.75
o4	0.5
o1	0.4
…	…

L3

OID	p3
o7	0.6
o2	0.6
o3	0.5
o5	0.1
…	…

$S \equiv SUM$
$k = 2$

B (1st round)

OID	lbscore	ubscore
o1	1.0	2.4
o2	0.8	2.4
o7	0.6	2.4

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

B (2nd round)

OID	lbscore	ubscore
o7	1.5	2.25
o2	1.4	2.3
o1	1.0	2.35
o3	0.75	2.25

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

B (3rd round)

OID	lbscore	ubscore
o2	2.1	2.1
o7	1.5	2.0
o3	1.25	1.95
o1	1.0	2.0
o4	0.5	1.7

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

B (4th round)

OID	lbscore	ubscore
o2	2.1	2.1
o7	1.5	1.9
o1	1.4	1.5
o3	1.25	1.45
o4	0.5	0.8

$1.5 \geq \max\{1.5,0.7\}$

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 o2 (S(o2) = 1.2), since the score of o1 is S(o1) = 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 o1 and o2 only 3 rounds are needed

S ≡ SUM

L1

OID	p1
o1	1.0
o2	1.0
…	0.3
…	…
…	0.3

L2

OID	p2
…	0.3
…	…
…	0.3
o2	0.2
o1	0

- 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)

NRA*: computing the exact scores

- 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 c_{RA}/c_{SA} rounds (more precisely: $\lfloor c_{RA}/c_{SA} \rfloor$)
- In practice CA behaves as NRA (and as NRA keeps lower and upper bounds on objects' scores), but every $\lfloor c_{RA}/c_{SA} \rfloor$ 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 c_{RA}/c_{SA}, but only if
  1) on each list scores are all distinct (no two objects tie on pj)
  2) $S \equiv$ MIN or S is *strictly monotone in each argument*:whenever one pj is increased and the others stay unchanged, then the value of the S increases as well (e.g., SUM)

The overall picture

Algorithm	scoring f. S	Data access	Notes
B_0	MAX	sorted	instance-optimal
MaxOptimal	MAX	sorted	instance-optimal
FA	monotone	sorted and random	cost independent of S
TA	monotone	sorted and random	instance-optimal
NRA	monotone	sorted	instance-optimal, wrong scores
NRA*	monotone	sorted	instance-optimal, exact scores
CA	monotone	sorted and random	instance-optimal, optimality ratio independent of c_{RA}/c_{SA} in some cases

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 $S \equiv MAX$: 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; 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.:

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

  where p1,p2,…pm are scoring criteria (the "preferences")

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

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

- Notice that the join is not anymore on the relations' PK's

Top-k M-N join queries: example

RName	City	Price
Al vecchio mulino	Bologna	25
La tavernetta	Roma	30
Tutti a tavola!	Bologna	40
Le delizie del palato	Milano	50
Acqua in bocca	Roma	70

Restaurants

HName	City	Price
La pensioncina	Milano	40
Dormi Bene!	Milano	50
RonfRonf	Roma	60
La Cascina	Bologna	80
La Quiete	Bologna	85
Il Riposino	Roma	90
CheapSleep	Bologna	100

Hotels

RName	Hname	City	TotPrice
Al vecchio mulino	La Cascina	Bologna	105
Al vecchio mulino	La Quiete	Bologna	110
Al vecchio mulino	CheapSleep	Bologna	125
La tavernetta	RonfRonf	Roma	90
La tavernetta	Il Riposino	Roma	120
Tutti a tavola!	La Cascina	Bologna	115
Tutti a tavola!	La Quiete	Bologna	120
Tutti a tavola!	CheapSleep	Bologna	140
Le delizie del palato	La pensioncina	Milano	90
Le delizie del palato	Dormi Bene!	Milano	110
Acqua in bocca	RonfRonf	Roma	130
Acqua in bocca	Il Riposino	Roma	160

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

RName	City	Price
Al vecchio mulino	Bologna	25
La tavernetta	Roma	30
Tutti a tavola!	Bologna	40
Le delizie del palato	Milano	50
Acqua in bocca	Roma	70

HName	City	Price
La pensioncina	Milano	40
Dormi Bene!	Milano	50
RonfRonf	Roma	60
La Cascina	Bologna	80
La Quiete	Bologna	85
Il Riposino	Roma	90
CheapSleep	Bologna	100

$\boxed{k = 2}$

The easy case: example

RName	City	Price
Al vecchio mulino	Bologna	25
La tavernetta	Roma	30
Tutti a tavola!	Bologna	40
Le delizie del palato	Milano	50
Acqua in bocca	Roma	70

HName	City	Price
La pensioncina	Milano	40
Dormi Bene!	Milano	50
RonfRonf	Roma	60
La Cascina	Bologna	80
La Quiete	Bologna	85
Il Riposino	Roma	90
CheapSleep	Bologna	100

RName	Hname	City	TotPrice
Al vecchio mulino	La Cascina	Bologna	105
Al vecchio mulino	La Quiete	Bologna	110

k = 2

The easy case: example

RName	City	Price
Al vecchio mulino	Bologna	25
La tavernetta	Roma	30
Tutti a tavola!	Bologna	40
Le delizie del palato	Milano	50
Acqua in bocca	Roma	70

HName	City	Price
La pensioncina	Milano	40
Dormi Bene!	Milano	50
RonfRonf	Roma	60
La Cascina	Bologna	80
La Quiete	Bologna	85
Il Riposino	Roma	90
CheapSleep	Bologna	100

RName	Hname	City	TotPrice
Al vecchio mulino	La Cascina	Bologna	105
Al vecchio mulino	La Quiete	Bologna	110

k = 2

RName	Hname	City	TotPrice
Le delizie del palato	La pensioncina	Milano	90
Al vecchio mulino	La Cascina	Bologna	105

The easy case: example

RName	City	Price
Al vecchio mulino	Bologna	25
La tavernetta	Roma	30
Tutti a tavola!	Bologna	40
Le delizie del palato	Milano	50
Acqua in bocca	Roma	70

HName	City	Price
La pensioncina	Milano	40
Dormi Bene!	Milano	50
RonfRonf	Roma	60
La Cascina	Bologna	80
La Quiete	Bologna	85
Il Riposino	Roma	90
CheapSleep	Bologna	100

RName	Hname	City	TotPrice
Al vecchio mulino	La Cascina	Bologna	105
Al vecchio mulino	La Quiete	Bologna	110

k = 2

RName	Hname	City	TotPrice
Le delizie del palato	La pensioncina	Milano	90
Al vecchio mulino	La Cascina	Bologna	105

RName	Hname	City	TotPrice
Le delizie del palato	La pensioncina	Milano	90
La tavernetta	RonfRonf	Roma	90

The easy case: example

RName	City	Price
Al vecchio mulino	Bologna	25
La tavernetta	Roma	30
Tutti a tavola!	Bologna	40
Le delizie del palato	Milano	50
Acqua in bocca	Roma	70

HName	City	Price
La pensioncina	Milano	40
Dormi Bene!	Milano	50
RonfRonf	Roma	60
La Cascina	Bologna	80
La Quiete	Bologna	85
Il Riposino	Roma	90
CheapSleep	Bologna	100

RName	Hname	City	TotPrice
Al vecchio mulino	La Cascina	Bologna	105
Al vecchio mulino	La Quiete	Bologna	110

k = 2

RName	Hname	City	TotPrice
Le delizie del palato	La pensioncina	Milano	90
Al vecchio mulino	La Cascina	Bologna	105

RName	Hname	City	TotPrice
Le delizie del palato	La pensioncina	Milano	90
La tavernetta	RonfRonf	Roma	90

112.5

The easy case: example

RName	City	Price
Al vecchio mulino	Bologna	25
La tavernetta	Roma	30
Tutti a tavola!	Bologna	40
Le delizie del palato	Milano	50
Acqua in bocca	Roma	70

HName	City	Price
La pensioncina	Milano	40
Dormi Bene!	Milano	50
RonfRonf	Roma	60
La Cascina	Bologna	80
La Quiete	Bologna	85
Il Riposino	Roma	90
CheapSleep	Bologna	100

RName	Hname	City	TotPrice
Al vecchio mulino	La Cascina	Bologna	105
Al vecchio mulino	La Quiete	Bologna	110

k = 2

RName	Hname	City	TotPrice
Le delizie del palato	La pensioncina	Milano	90
Al vecchio mulino	La Cascina	Bologna	105

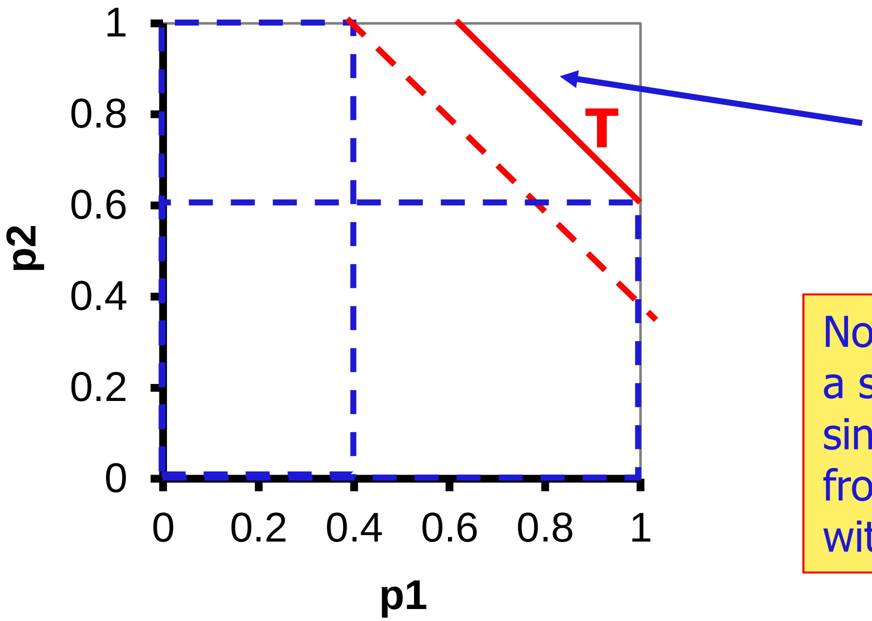
RName	Hname	City	TotPrice
Le delizie del palato	La pensioncina	Milano	90
La tavernetta	RonfRonf	Roma	90

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 Lj, let pj^{max} denote the first (highest) score seen on Lj
    - Let T be the maximum among the following m values:
      $S(\underline{p1},p2^{max},\dots,pm^{max}),\ S(p1^{max},\underline{p2},\dots,pm^{max}),\dots,\ S(p1^{max},p2^{max},\dots,\underline{pm})$
    - This is also called the "corner bound"
- Let j denote a generic join combination, i.e., j = (t1,t2,…,tm)
- The (at this point, obvious) observation is that one can halt when there are k join combinations j such that S(j) ≥ T
- S.a.’s can be executed using a round-robin strategy, by accessing the list for which $\underline{pj}$ si maximum, etc.

Visualizing the corner bound

- For simplicity, assume $p1^{max} = p2^{max} = 1$, and S ≡ SUM
- In the figure it is: <u>p1</u> = 0.4 and <u>p2</u> = 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 L2 will match a tuple from L1 with score = 1

Is the corner bound the best possible one?

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

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

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

Rank-Join: example of non optimality

S ≡ SUM

```
SELECT *
FROM   R1,R2,R3
WHERE  R1.A = R2.A
AND    R1.A = R3.A
ORDER BY p1 + p2 + p3 DESC
STOP AFTER 1
```

A	p1
a	1.0
x	0.95
x	0.9
…	…
…	…
w	0.5

A	p2
y	1.0
a	0.7
y	0.4
…	…

A	p3
z	1.0
a	0.8
z	0.4
…	…

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

Rank-Join: example of non optimality

S ≡ SUM

```
SELECT *
FROM   R1,R2,R3
WHERE  R1.A = R2.A
AND    R1.A = R3.A
ORDER BY p1 + p2 + p3 DESC
STOP AFTER 1
```

A	p1
a	1.0
x	0.95
x	0.9
…	…
…	…
w	0.5

A	p2
y	1.0
a	0.7
y	0.4
…	…

A	p3
z	1.0
a	0.8
z	0.4
…	…

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

Rank-Join: example of non optimality

S ≡ SUM

```
SELECT *
FROM   R1,R2,R3
WHERE  R1.A = R2.A
AND    R1.A = R3.A
ORDER BY p1 + p2 + p3 DESC
STOP AFTER 1
```

A	p1
a	1.0
x	0.95
x	0.9
…	…
…	…
w	0.5

A	p2
y	1.0
a	0.7
y	0.4
…	…

A	p3
z	1.0
a	0.8
z	0.4
…	…

j	S
(a,a,a)	**2.5**

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

Rank-Join: example of non optimality

S ≡ SUM

```
SELECT *
FROM   R1,R2,R3
WHERE  R1.A = R2.A
AND    R1.A = R3.A
ORDER BY p1 + p2 + p3 DESC
STOP AFTER 1
```

A	p1
a	1.0
x	0.95
x	0.9
…	…
…	…
w	0.5

A	p2
y	1.0
a	0.7
y	0.4
…	…

A	p3
z	1.0
a	0.8
z	0.4
…	…

Res

j	S
(a,a,a)	**2.5**

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

Rank-Join: example of non optimality

S ≡ SUM

```
SELECT *
FROM   R1,R2,R3
WHERE  R1.A = R2.A
AND    R1.A = R3.A
ORDER BY p1 + p2 + p3 DESC
STOP AFTER 1
```

A	p1
a	1.0
x	0.95
x	0.9
…	…
…	…
w	0.5

A	p2
y	1.0
a	0.7
y	0.4
…	…

A	p3
z	1.0
a	0.8
z	0.4
…	…

Res

j	S
(a,a,a)	**2.5**

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

A tight bounding scheme: results

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

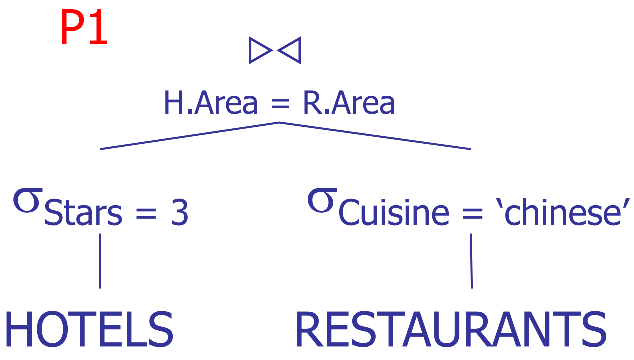
- We sketch the basic concepts of RankSQL, in particular:
  - The "splitting and interleaving" requirements
  - The concept of "rank-relation" and the "ranking principle"
  - The "rank algebra" for rank-relations

Splitting and interleaving

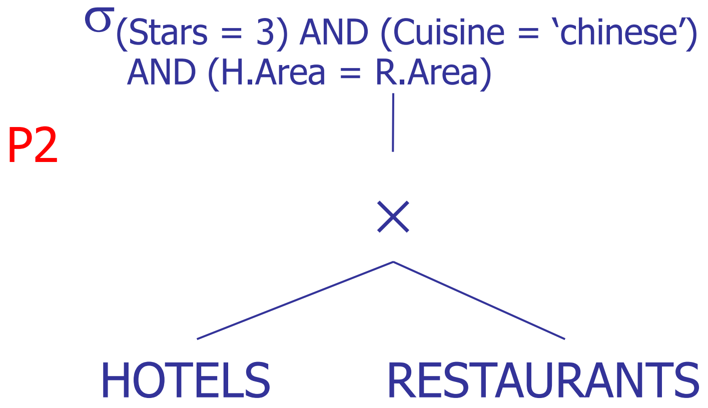
- Consider the following query SQL:

```
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:


$\sigma_{(Stars = 3) AND (Cuisine = 'chinese') AND (H.Area = R.Area)}$
P2
×
HOTELS
RESTAURANTS

We can transform P2 into P1 by:

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

The same should be done with the ranking function!

Rank-relations and the ranking principle

- To make query optimizers "rank-aware" it is necessary to introduce the concept of rank-relation, i.e., a relation that consists of tuples with scores
- The definition is tightly related to the ranking principle, which formalizes the now-well-known fact that the "most promising" tuples should be processed first

Rank-relation:

Given a relation R and a monotone scoring function S(p1,p2,…,pm), the rank-relation **R_P**, with **$P \subseteq$ {p1,p2,…,pm}**, 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 substituing in S the values of pj(t), if pj $\in$P, otherwise 1 (or the maximum possible value for pj)
- Tuples in R_P are ranked by **decreasing values of S_P^+(t)** (ranking principle)

- Notice that $S_P^+(t) = S(t)$ when P = {p1,p2,…,pm}, and that $R_{\varnothing} \equiv R$

The RankSQL algebra

- The RankSQL algebra extends the semantics of RA to rank-relations
- It introduces a new rank operator, μ, which applies to a rank-relation **R_P** a not-evaluated-yet preference p ($p \not\in P$), yielding the new rank-relation **$R_{P\cup\{p\}}$**
- The μ operator is the basis to **split** the scoring function S(p1,p2,…,pm), since:

  $$\mathbf{R_{\{p1,p2,\ldots,pm\}} = \mu_{p1}(\mu_{p2}(\ldots(\mu_{pm}(R))))}$$

  i.e., the final ranking can be obtained by applying one-by-one the preferences
- Any order of evaluation is admissible, since $\mu_{p1}(\mu_{p2}(R_P)) = \mu_{p2}(\mu_{p1}(R_P))$
- **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 μ_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 μ_p fetches a tuple t'' such that $S_{P\cup\{p\}}^+(t) \geq S_P^+(t'')$
- Index (and sequential) scans are also treated as operators, since they could be used in place of μ to rank tuples according to a preference p

The rank operator in action

Hotel	…	p1	p2	p3	score
h1		0.7	0.8	0.9	2.4
h2		0.9	0.85	0.8	2.55
h3		0.5	0.45	0.75	1.7
h4		0.4	0.7	0.95	2.05
…		…	…	…	…

$H_{p1,\ p2,\ p3}$

Hotel	$S_{\{p1,p2,p3\}}^{+} = S$

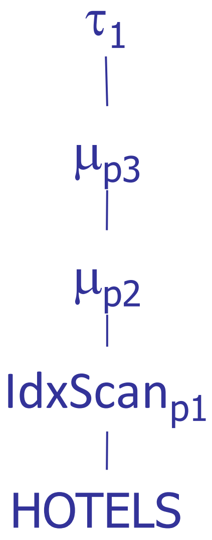
$H_{p1,\ p2}$

Hotel	$S_{\{p1,p2\}}^{+}$

H_{p1}

Hotel	$S_{\{p1\}}^{+}$

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



τ_1

μ_{p3}

μ_{p2}

IdxScan_{p1}

HOTELS

The rank operator in action

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

$H_{p1, p2, p3}$

Hotel	S^+

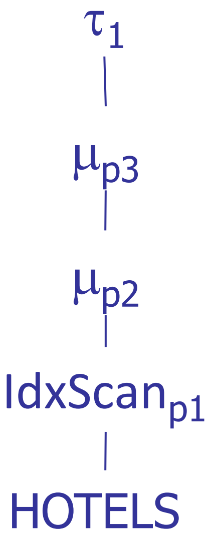
$H_{p1, p2}$

Hotel	S^+

H_{p1}

Hotel	S^+
h2	2.9

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



τ_1

μ_{p3}

μ_{p2}

IdxScan_{p1}

HOTELS

The rank operator in action

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

$H_{p1, p2, p3}$

Hotel	S^+

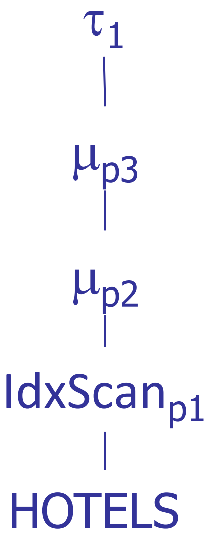
$H_{p1, p2}$

Hotel	S^+
h2	2.9

H_{p1}

Hotel	S^+
h2	2.9

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



τ_1

μ_{p3}

μ_{p2}

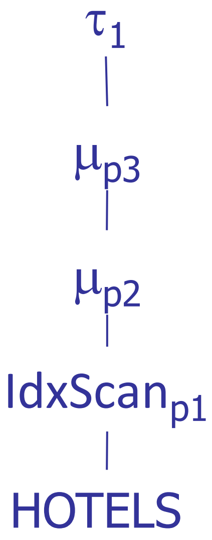
$IdxScan_{p1}$

HOTELS

The rank operator in action

Hotel	…	p1	p2	p3	score
h1		0.9	1.0	1.0	2.9
h2		0.9	0.85	1.0	2.75
h3		0.9	1.0	1.0	2.9
h4		0.9	1.0	1.0	2.9
…		0.9	1.0	1.0	2.9

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



$H_{p1, p2, p3}$

Hotel	S^+

$H_{p1, p2}$

Hotel	S^+
h2	2.75

H_{p1}

Hotel	S^+
h2	2.9

The rank operator in action

Hotel	…	p1	p2	p3	score
h1		0.7	1.0	1.0	2.7
h2		0.9	0.85	1.0	2.75
h3		0.7	1.0	1.0	2.7
h4		0.7	1.0	1.0	2.7
…		0.7	1.0	1.0	2.7

$H_{p1, p2, p3}$

Hotel	S^+

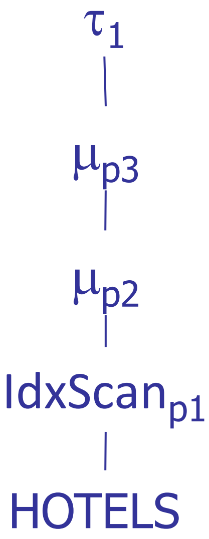
$H_{p1, p2}$

Hotel	S^+
h2	2.75

H_{p1}

Hotel	S^+
h2	2.9
h1	2.7

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



τ_1

μ_{p3}

μ_{p2}

$\mathrm{IdxScan}_{p1}$

HOTELS

The rank operator in action

Hotel	…	p1	p2	p3	score
h1		0.7	0.8	1.0	2.5
h2		0.9	0.85	1.0	2.75
h3		0.7	1.0	1.0	2.7
h4		0.7	1.0	1.0	2.7
…		0.7	1.0	1.0	2.7

$H_{p1,\ p2,\ p3}$

Hotel	S^+

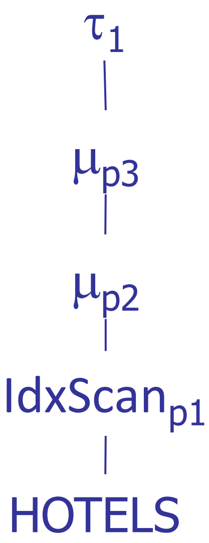
$H_{p1,\ p2}$

Hotel	S^+
h2	2.75
h1	2.5 (*)

H_{p1}

Hotel	S^+
h2	2.9
h1	2.7

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



τ_1

μ_{p3}

μ_{p2}

IdxScan_{p1}

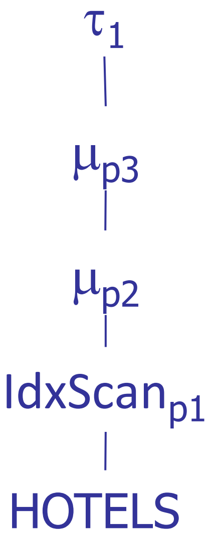
HOTELS

(*) Evaluation of p2 can be delayed

The rank operator in action

Hotel	…	p1	p2	p3	score
h1		0.7	0.8	1.0	2.5
h2		0.9	0.85	0.8	2.55
h3		0.7	1.0	1.0	2.7
h4		0.7	1.0	1.0	2.7
…		0.7	1.0	1.0	2.7

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



τ_1

μ_{p3}

μ_{p2}

IdxScan_{p1}

HOTELS

$H_{p1, p2, p3}$

Hotel	S^+
h2	2.55

$H_{p1, p2}$

Hotel	S^+
h2	2.75
h1	2.5

H_{p1}

Hotel	S^+
h2	2.9
h1	2.7

The rank operator in action

Hotel	…	p1	p2	p3	score
h1		0.7	0.8	1.0	2.5
h2		0.9	0.85	0.8	2.55
h3		0.5	1.0	1.0	2.5
h4		0.5	1.0	1.0	2.5
…		0.5	1.0	1.0	2.5

$H_{p1, p2, p3}$

Hotel	S^+
h2	2.55

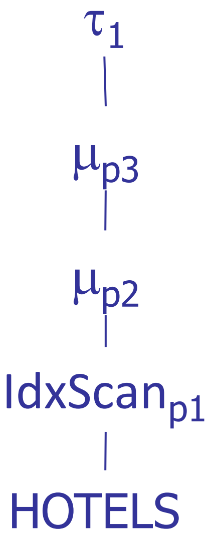
$H_{p1, p2}$

Hotel	S^+
h2	2.75
h1	2.5

H_{p1}

Hotel	S^+
h2	2.9
h1	2.7
h3	2.5

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



The rank operator in action

Hotel	…	p1	p2	p3	score
h1		0.7	0.8	1.0	2.5
h2		0.9	0.85	0.8	2.55
h3		0.5	0.45	1.0	1.95
h4		0.5	1.0	1.0	2.5
…		0.5	1.0	1.0	2.5

$H_{p1,\ p2,\ p3}$

Hotel	S^+
h2	2.55

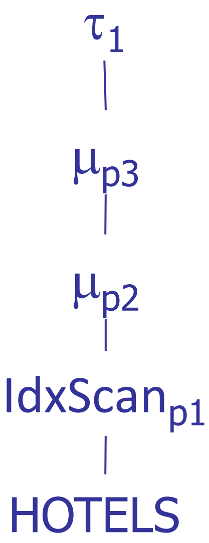
$H_{p1,\ p2}$

Hotel	S^+
h2	2.75
h1	2.5
h3	1.95 (*)

H_{p1}

Hotel	S^+
h2	2.9
h1	2.7
h3	2.5

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



τ_1

μ_{p3}

μ_{p2}

IdxScan_{p1}

HOTELS

(*) Evaluation of p2 can be delayed

The rank operator in action

Hotel	…	p1	p2	p3	score
h1		0.7	0.8	0.9	2.4
h2		0.9	0.85	0.8	2.55
h3		0.5	0.45	1.0	1.95
h4		0.5	1.0	1.0	2.5
…		0.5	1.0	1.0	2.5

$H_{p1,\ p2,\ p3}$

Hotel	S^+
h2	2.55
h1	2.4 (*)

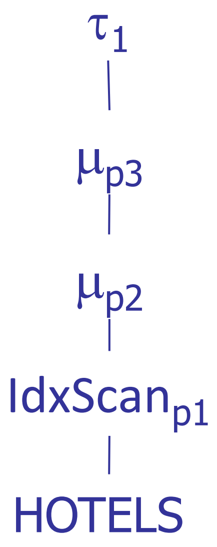
$H_{p1,\ p2}$

Hotel	S^+
h2	2.75
h1	2.5
h3	1.95

H_{p1}

Hotel	S^+
h2	2.9
h1	2.7
h3	2.5

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



τ_1

μ_{p3}

μ_{p2}

IdxScan_{p1}

HOTELS

(*) Evaluation of p3 can be omitted

The rank operator in action

Hotel	…	p1	p2	p3	score
h1		0.7	0.8	0.9	2.4
h2		0.9	0.85	0.8	2.55
h3		0.5	0.45	1.0	1.95
h4		0.5	1.0	1.0	2.5
…		0.5	1.0	1.0	2.5

$H_{p1,\ p2,\ p3}$

Hotel	S^+
h2	2.55
h1	2.4

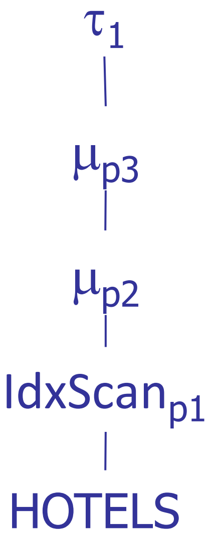
$H_{p1,\ p2}$

Hotel	S^+
h2	2.75
h1	2.5
h3	1.95

H_{p1}

Hotel	S^+
h2	2.9
h1	2.7
h3	2.5

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



τ_1

μ_{p3}

μ_{p2}

IdxScan_{p1}

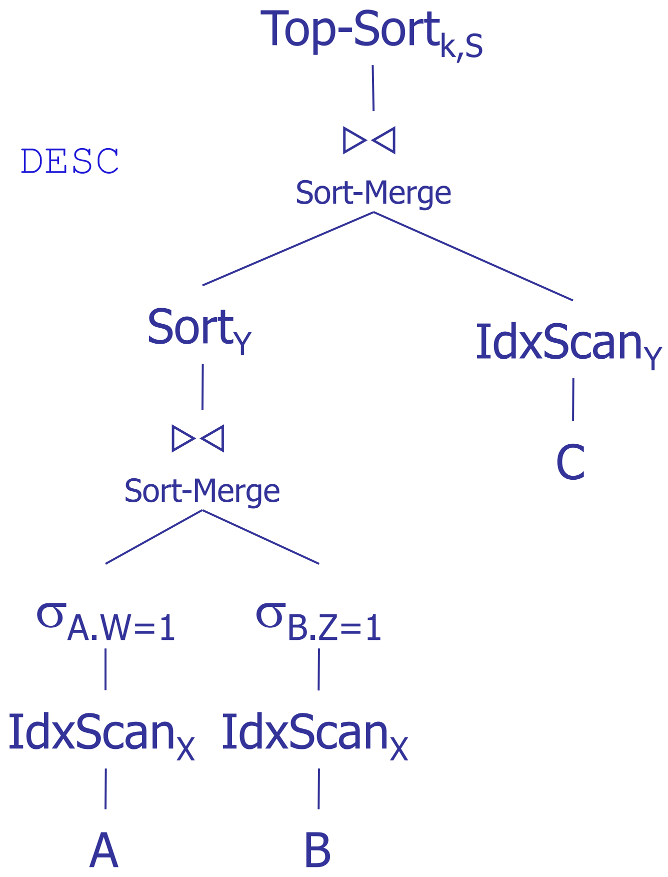
HOTELS

RankSQL: an example (1)

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

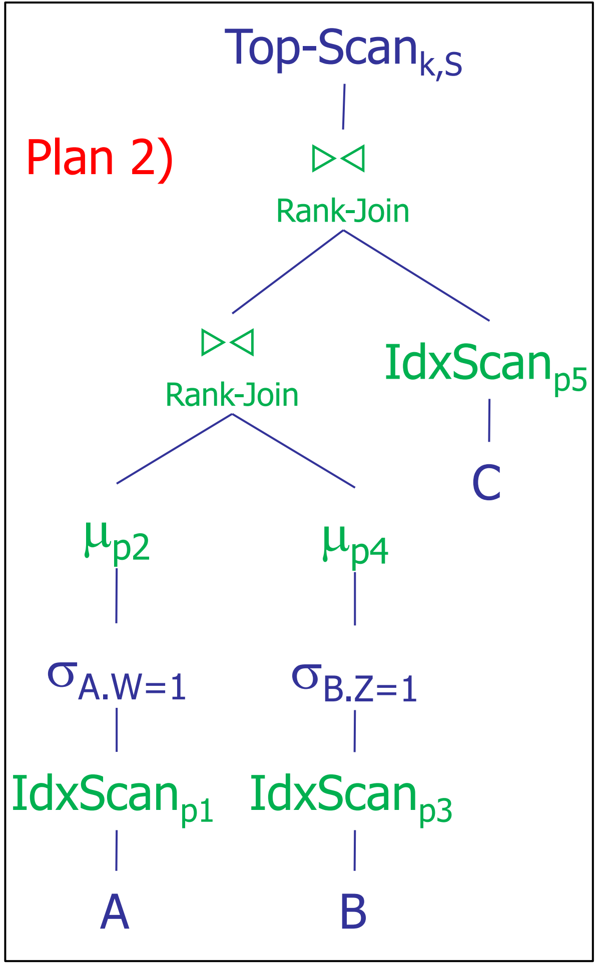
- A traditional (not rank-aware) plan for processing the query could just exploit a Top-Sort operator to avoid sorting all the tuples produced by the second (top-most) join

Plan 1 (traditional)



RankSQL: an example (2)

- With rank-aware operators several alternative plans are possible



Plan 2)

$\text{Top-Scan}_{k,S}$

▷◁ Rank-Join

▷◁ Rank-Join

IdxScan_{p5}

C

μ_{p2}

μ_{p4}

$\sigma_{A.W=1}$

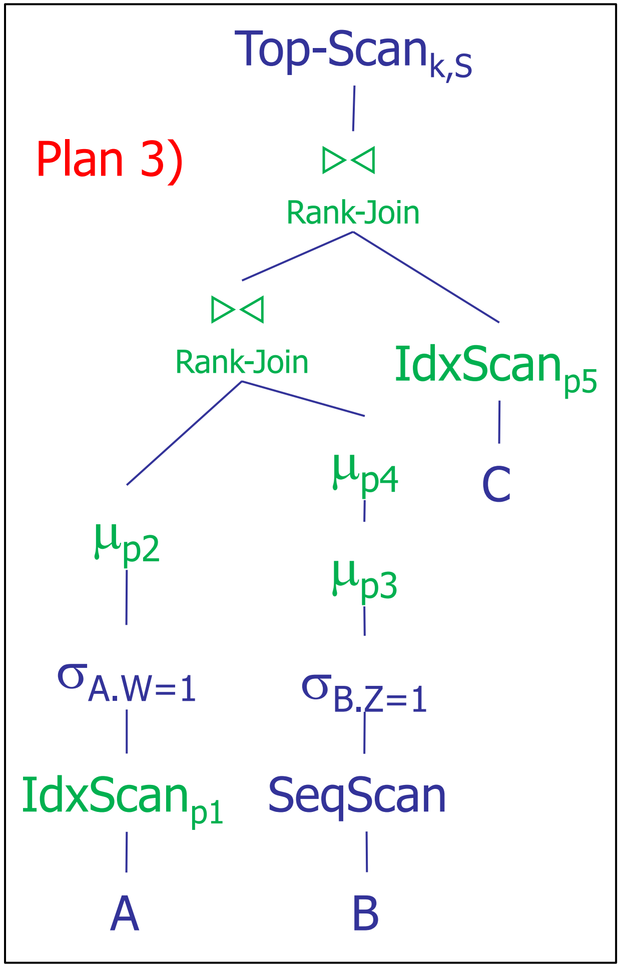
$\sigma_{B.Z=1}$

IdxScan_{p1}

IdxScan_{p3}

A

B



Plan 3)

$\text{Top-Scan}_{k,S}$

▷◁ Rank-Join

▷◁ Rank-Join

IdxScan_{p5}

μ_{p4}

C

μ_{p2}

μ_{p3}

$\sigma_{A.W=1}$

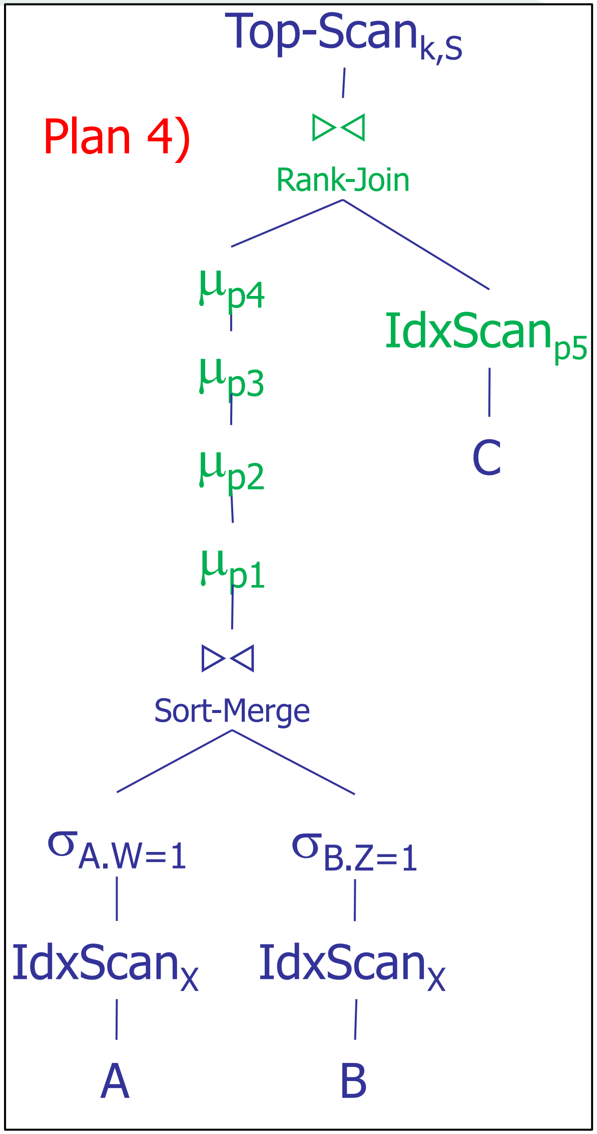
$\sigma_{B.Z=1}$

IdxScan_{p1}

SeqScan

A

B



$\text{Top-Scan}_{k,S}$

Plan 4)

▷◁ Rank-Join

μ_{p4}

IdxScan_{p5}

μ_{p3}

C

μ_{p2}

μ_{p1}

▷◁ Sort-Merge

$\sigma_{A.W=1}$

$\sigma_{B.Z=1}$

IdxScan_{X}

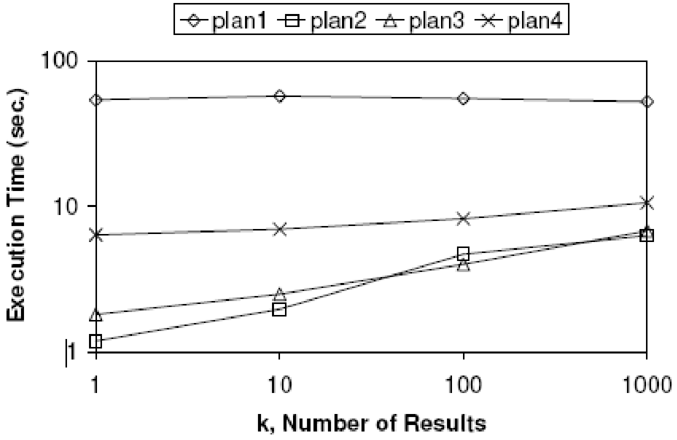
IdxScan_{X}

A

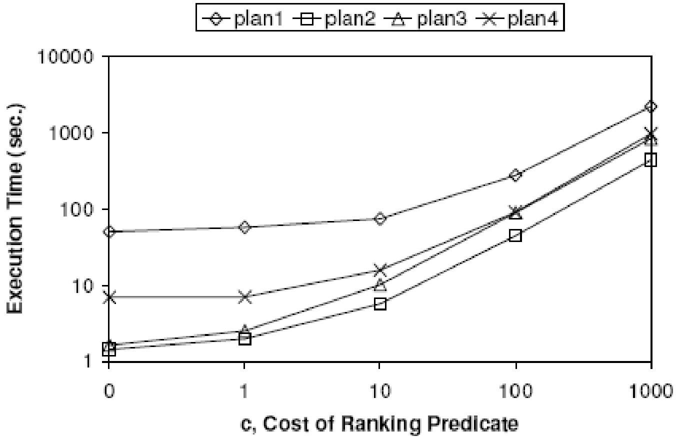
B

Experimental results (from [LCI+05])

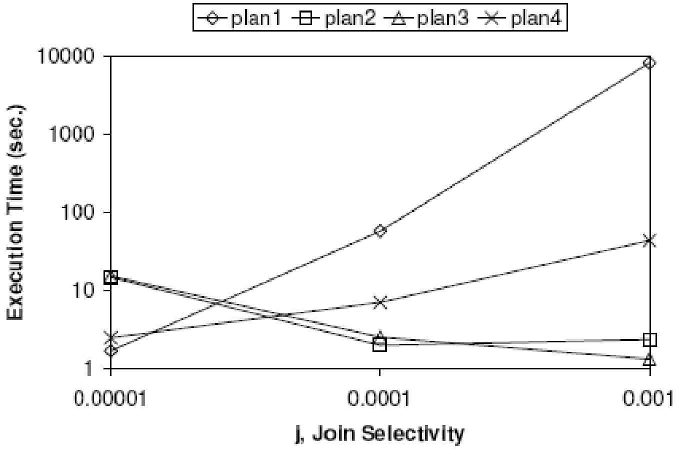
- None of the plans is always the best → need for optimization
- The (optimal) traditional plan is almost never the best one



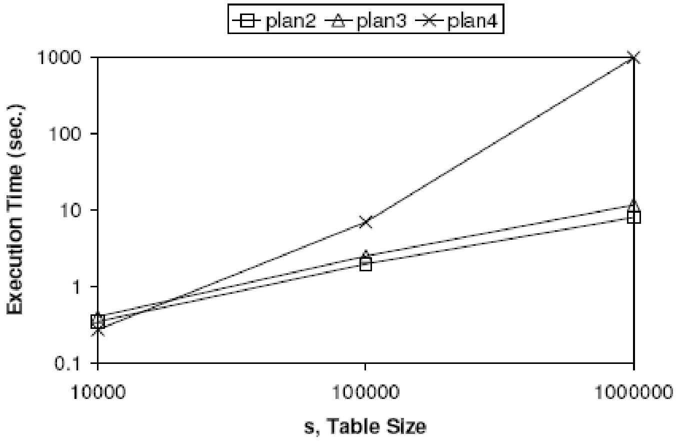
(a) $s = 100,000,\ j = 0.0001,\ c = 1$



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



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

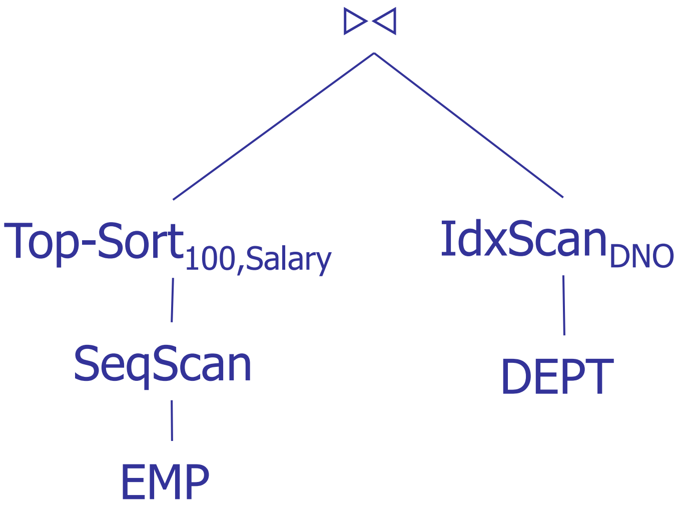


(d) $k = 10,\ j = 0.0001,\ c = 1$

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
  - Similar issues when pushing-down a `GROUP BY`
  - This can be verified by looking at:
  - DB integrity constraints (FK,PK,NOT NULL,…), and
  - The query predicates that remain to be evaluated after the Top operator is executed

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

- Here the Top operator 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

[BBK+97] Stefan Berchtold, Christian Böhm, Daniel A. Keim, Hans-Peter Kriegel: A Cost Model For Nearest Neighbor Search in High-Dimensional Data Space. PODS 1997: 78-86

[CK97] Michael J. Carey, Donald Kossmann: On Saying "Enough Already!" in SQL. SIGMOD Conference 1997: 219-230

[Fag96] Ronald Fagin: Combining Fuzzy Information from Multiple Systems. PODS 1996: 216-226

[FLN01] Ronald Fagin, Amnon Lotem, Moni Naor: Optimal Aggregation Algorithms for Middleware. PODS 2001

[FLN03] Ronald Fagin, Amnon Lotem, Moni Naor: Optimal Aggregation Algorithms for Middleware. J. Comput. Syst. Sci. 66(4): 614-656 (2003)

[HS99] Gísli R. Hjaltason, Hanan Samet: Distance Browsing in Spatial Databases. ACM Trans. Database Syst. 24 (2): 265-318 (1999)

[IAE03] Ihab F. Ilyas, Walid G. Aref, Ahmed K. Elmagarmid: Supporting Top-k Join Queries in Relational Databases. VLDB 2003: 754-765

[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

[WHT+99] Edward L. Wimmers, Laura M. Haas, Mary Tork Roth, Christoph Braendli: Using Fagin's Algorithm for Merging Ranked Results in Multimedia Middleware. CoopIS 1999: 267-278