Top-k queries: advanced techniques (1)

Information Systems M

Prof. Paolo Ciaccia
http://www-db.deis.unibo.it/courses/SI-M/

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)
- We will consider them later

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, in that R is spread over several sites, each providing information only on part of the objects
- Historically, the 2nd scenario, sometimes called the "middleware scenario", is the one that motivated the study of top-k (1-1) join queries, and which led to the introduction of the first non-trivial algorithms

The middleware scenario

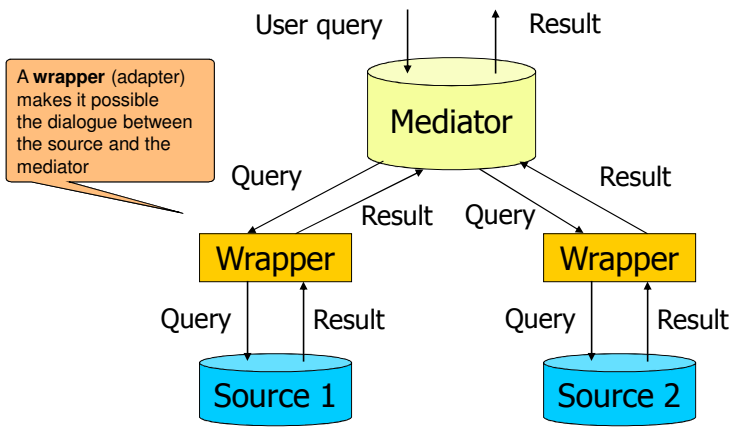
- The middleware scenario can be intuitively described as follows
  1. We have a number of "data sources"
  2. Our requests (queries) might involve several data sources at a time
  3. The result of our queries is obtained by "combining" in some way the results returned by the data sources
- These queries are collectively called "**middleware queries**", since they require the presence of a middleware whose role is to act as a "mediator" (also known as "information agent") between the user/client and the data sources/servers

Data sources

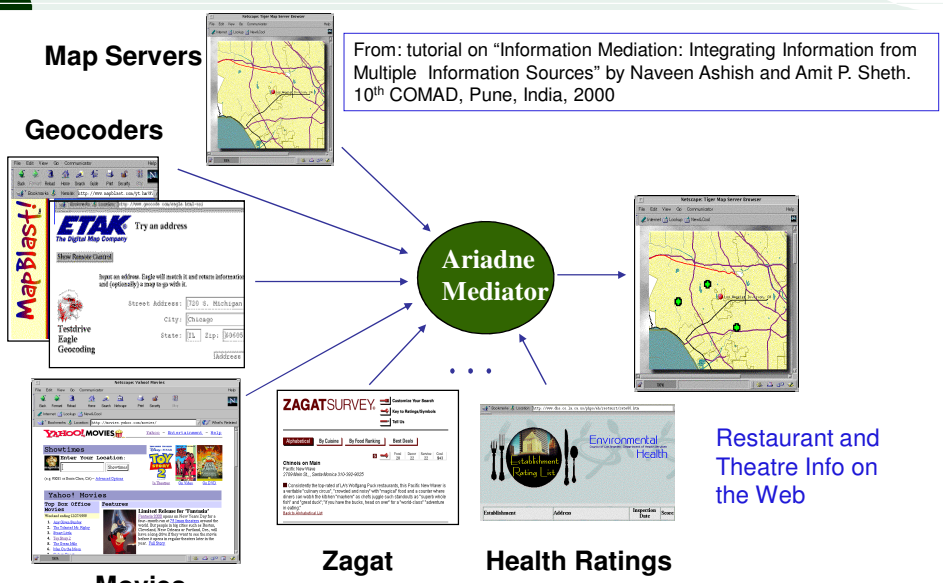
- Sources may be
  - databases (relational, object-relational, object-oriented, legacy, XML)
  - specialized servers (managing text, images, music, spatial data, ecc.)
  - web services
  - web sites
  - spreadsheets, e-mail archives
  - …
- In several cases, data sources are autonomous and heterogeneous
  - Different data models
  - Different data formats
  - Different query interfaces
  - Different semantics (same query, same data, yet different results)
  - …

The goal of a mediator is to hide all such differences to the user, so that all of them are perceived as a single source

The basic architecture



An example



Some links (projects/products/...)

- TSIMMIS, Stanford University, http://www-db.stanford.edu/tsimmis/
- Garlic, IBM Almaden, http://www.almaden.ibm.com/projects/garlic.shtml
- MOMIS, U Modena e Reggio Emilia, http://dbgroup.unimo.it/Momis/
- Search Computing (SeCo), http://www.search-computing.it/
- IBM InfoSphere, http://www-01.ibm.com/software/data/integration/info_server/
- Fetch, http://www.fetch.com
- ...

Another (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')

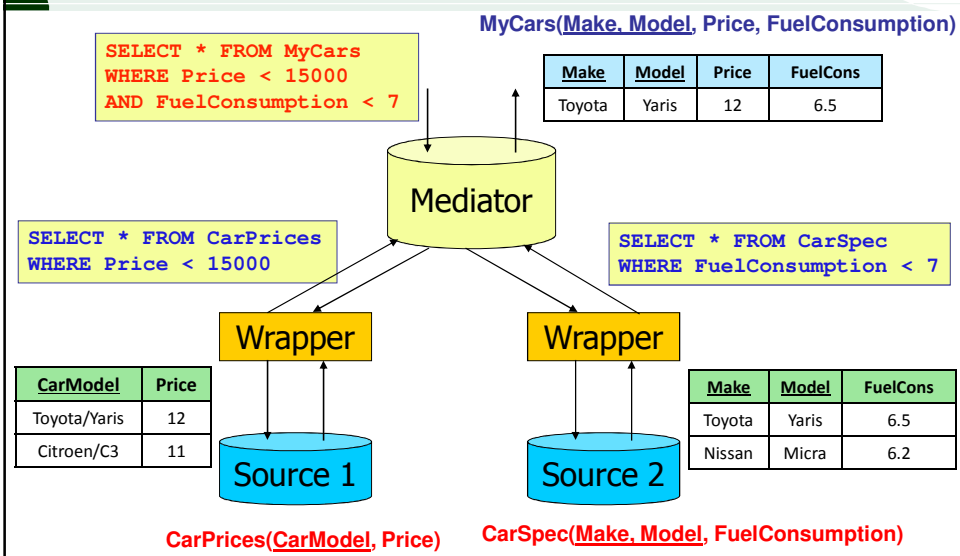
  we can now support queries on both Price and FuelConsumption, e.g.:

  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



A further 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:

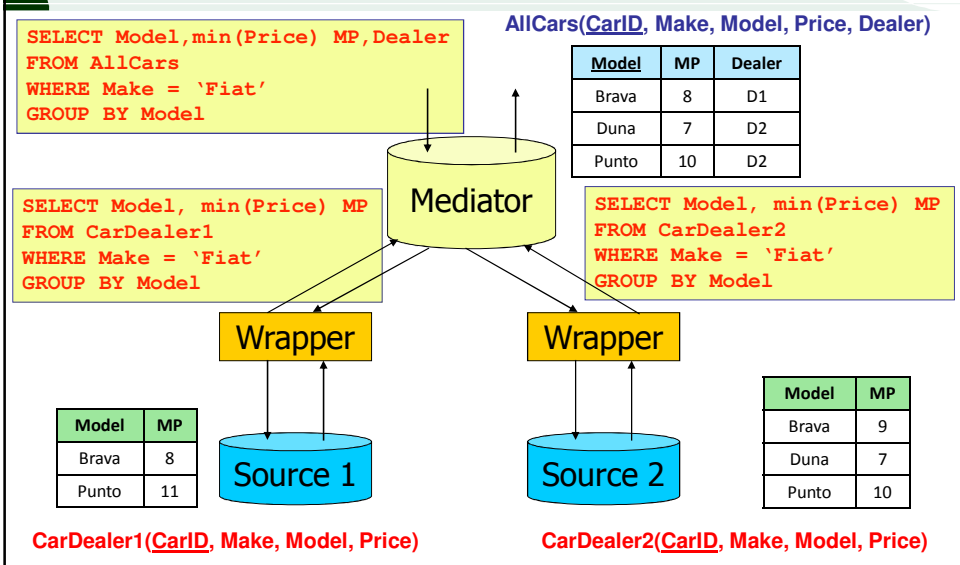
    For each FIAT model, which is the cheapest offer?

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

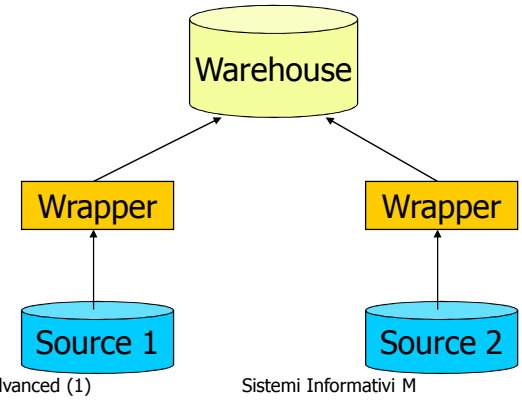
➡ For queries of this kind, the mediator is also often called a "meta-broker" or "meta-search engine"

Query execution (some details omitted)



Other possibilities

- With multiple data sources we can have other architectures as well
  - For instance, in a Data Warehouse (DW) all data from the sources are made "homogeneous" and loaded into the global schema of a centralized DW
    - Problems are quite different from the ones we are going to consider…
  - Peer-to-peer (P2P) systems are another relevant case…



The (many) omitted details

- Once one starts to consider a mediator-based architecture, several issues become relevant, e.g.:
- Which is a suitable query language? A suitable interchange format?
  - Nowadays the answer for the interchange format is: XML
- Which are the limitations posed by the interfaces of the data sources
  - Can we query using a predicate/filter on the price of cars? On their consumption? Can we formulate queries *at all*?
- Do we know, say, how a given source ranks objects?
  - E.g., which is the criterion used by Google? and by Altavista?
- Is there any cost charged by the data sources?
  - Free access? Pay-per-result? Pay-per-query?

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, for instance, 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:

$getNext_{Lj}() \rightarrow (OID, Attributes, pj)$

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:

$getScore_{Lj}(OID) \rightarrow pj$

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
- 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 also [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 get_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!

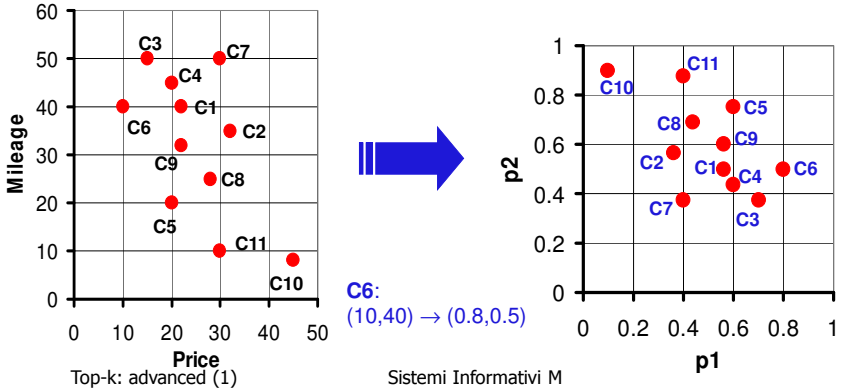
A homogeneous model for scoring functions

- 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), with pj(o) ∈ [0,1]
  - For convenience, scores are normalized, with higher scores being better
  - This can be relaxed; what matters is to know which is the best and 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)) ∈ $[0,1]^m$ 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:

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

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

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

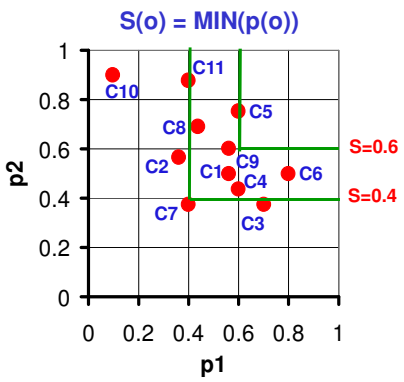
MAX (Maximum): just considers the best partial score

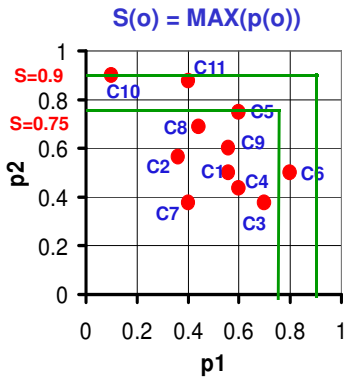
$$MAX(o) \equiv MAX(p(o)) = \max\{p1(o), p2(o), \ldots, pm(o)\}$$

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





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 B_0 algorithm

Input: ranked lists Lj (j=1,…,m), integer k ≥ 1
Output: the top-k objects according to the MAX scoring function

```
1.  B := ∅;                                  // B is a main-memory buffer
2.  for j = 1 to m:
3.      Obj(j) := ∅;                         // the set of objects "seen" on Lj
4.      for i = 1 to k:                      // get the best k objects from each list
5.          t := getNext_Lj();
6.          Obj(j) := Obj(j) ∪ {t.OID};
7.          if t.OID was not retrieved from other lists then: INSERT(B,t)  // adds t to the buffer
8.                      else: join t with the entry in B having the same OID;
9.  for each object o ∈ Obj := ∪_j Obj(j):   // for each object with at least one partial score…
10.     MAX(o) := max_j{pj(o): pj(o) is defined};  // …compute MAX using the available scores
11. return the k objects with maximum score;
12. 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

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

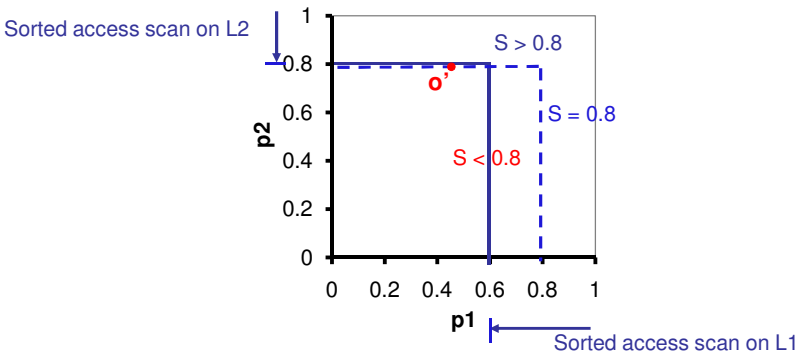
PaneeVino

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

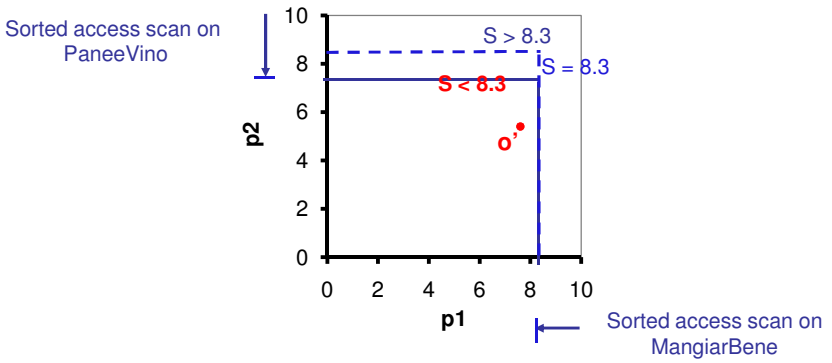
Why B_0 works: graphical intuition

- By hypothesis, in the figure below at least k objects o have S(o) ≥ 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' ∉ 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



B_0: A proof of correctness

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

  if o ∈ Obj – Res, then S(o) is not guaranteed to be correct

  (e.g., see o3 in the 1st example)
- Thus, we have to show that this does not influence the result
- On the other hand, if o ∉ Obj we have just shown that o cannot be better than any object in Res

Theorem:
The 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 ∈ Res, then S(o) is correct**
  - We then show that **if o ∈ Obj - Res, then, even if its global score is not correct, the algorithm correctly determines the top-k objects**

Proof (1): if o ∈ Res, then S(o) is correct

- Let $SB_0(o)$ be the global score, as computed by B_0, for an object o ∈ 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 ∈ Res and assume by contradiction that $SB_0(o1) < S(o1)$
- This is to say that there exists Lj such that (s.t.): o1 ∉ Obj(j) and $S(o1) = pj(o1)$
- In turn this implies that there are k objects o ∈ 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 ∈ Res

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

- Consider an object, say o1, s.t. o1 ∈ 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 ∈ Res ■

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 to have S(o1) > S(o), o ∈ 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
...	...

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!

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

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 ≥ $\max_j\{p_j\}$

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 $\max_j\{p_j\}$. 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 < $\max_j\{p_j\}$. 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
...	...

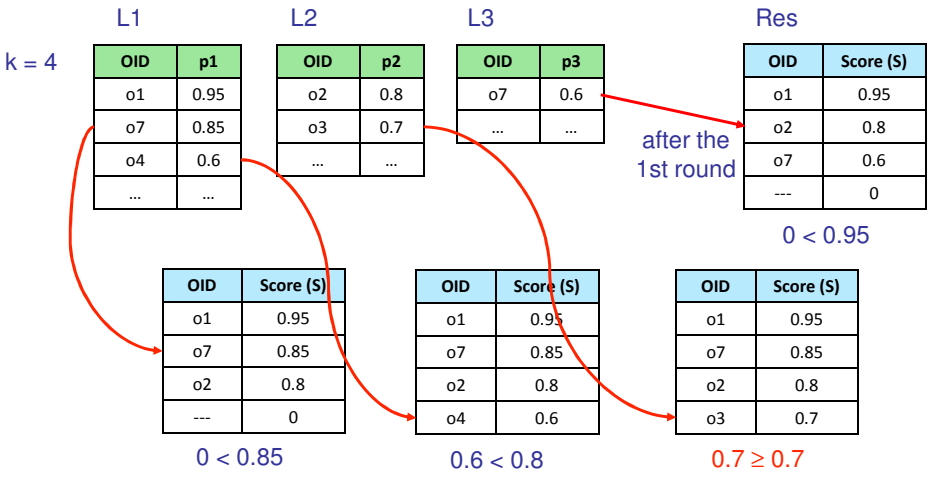
The MaxOptimal algorithm

- 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 $\underline{p_{j^*}}$ is maximum: $j^* = \text{argmax}_j\{\underline{p_j}\}$
  - 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

```
1. for i=1 to k: Res[i] := [null,0];        // entry of type: [OID,score] (+ other attr.'s as needed)
2. for j = 1 to m: pj = 1;                  // the best possible score on Lj
3. while Res[k].score < max_j{pj}:
4.       j* = argmax_j{pj}; t := getNext_Lj*();        // get the next best object from list Lj*
5.       if t.pj* > Res[k].score then:
6.             if t.OID ∈ Res.OID then: update the entry in Res having the same OID
7.                   else: {remove the object in Res[k]; insert [t.OID,t.pj*] in Res};
8.  return Res;
9.  end.
```

MaxOptimal: example



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 $\underline{p_j} < MAX_k$

Proof: By contradiction, assume that MaxOptimal performs a s.a. on list Lj*, with $\underline{p_j^*} < MAX_k$. For this it has to be $\underline{p_j^*} = \max_j\{\underline{p_j}\}$, from which it is derived $MAX_k > \underline{p_j}$, 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:

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

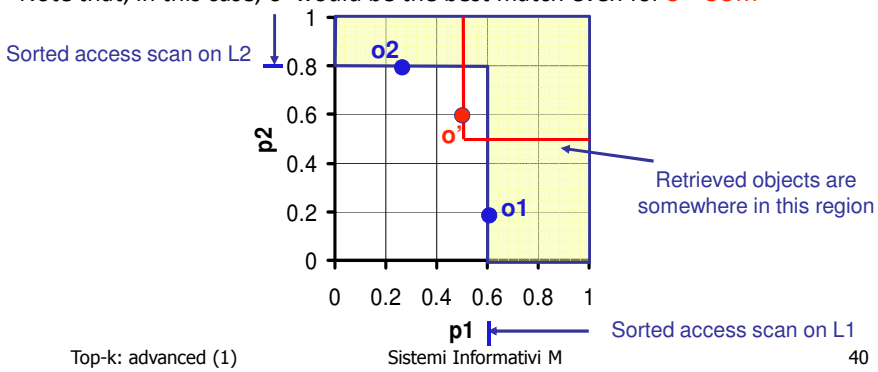
we get:

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



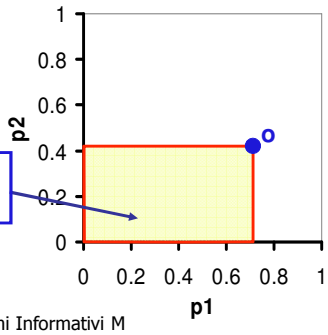
The FA algorithm: monotone scoring f.'s

- 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
  $x1 \le y1, x2 \le y2, \ldots, xm \le ym \Rightarrow S(x1,x2,\ldots,xm) \le S(y1,y2,\ldots,ym)$

- 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 $k \ge 1$, monotone scoring function S
Output: the top-k objects according to S
// 1st phase: sorted accesses
1. for j = 1 to m: Obj(j) := ∅; B := ∅; M := ∅;
2. while |M| < k:
4. for j = 1 to m:
5. t := $getNext_{Lj}$(); Obj(j) := Obj(j) ∪ {t.OID}; // get the next best object from list Lj
6. if t.OID was not retrieved from other lists then: INSERT(B,t)
7. else: join t with the entry in B having the same OID;
8. M := $\cap_j$ Obj(j); // the set of objects seen on all the m lists
// 2nd phase: random accesses
9. for each object o ∈ Obj := Obj(j): // for each object with at least one partial score…
perform random accesses to retrieve the missing partial scores for o;
// 3rd phase: score computation
10. for each object o ∈ Obj: compute S(o);
11. return the k objects with maximum score;
12. end.

How FA works

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

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

RIGHT!!

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

☺

S ≡ SUM

RIGHT!!

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

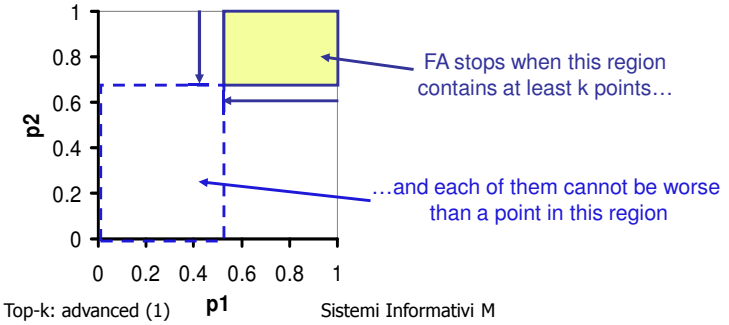
Why FA is correct: formal and intuitive

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' ∉ Obj, then o' cannot be better than any object o ∈ Res.

Let o be any object in Res. Then, there is at least one object o'' ∈ M (possibly coincident with o itself) such that S(o'') ≤ S(o), otherwise o would not be in Res.

Since o' ∉ Obj, for each Lj it is pj(o') ≤ pj(o''), and from the assumption of monotonicity of S it is S(o') ≤ S(o''); it follows that S(o') ≤ S(o) ■



FA: performance

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

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

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

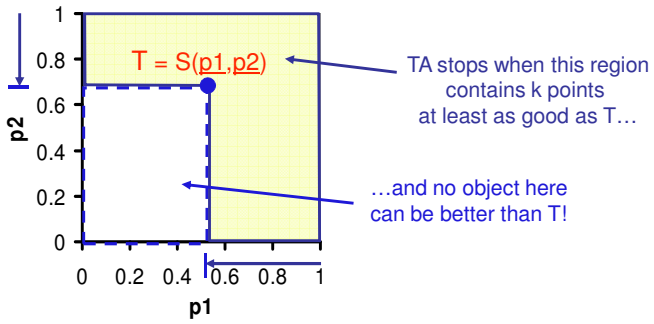
Input: ranked lists Lj (j=1,…,m), integer k ≥ 1, monotone scoring function S
Output: the top-k objects according to S

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

Why TA is correct: formal and intuitive

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

Proof: Consider an object o' that has not been seen under sorted access. Thus, for each j it is 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) ■



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

- Let's take S ≡ SUM and k = 2

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

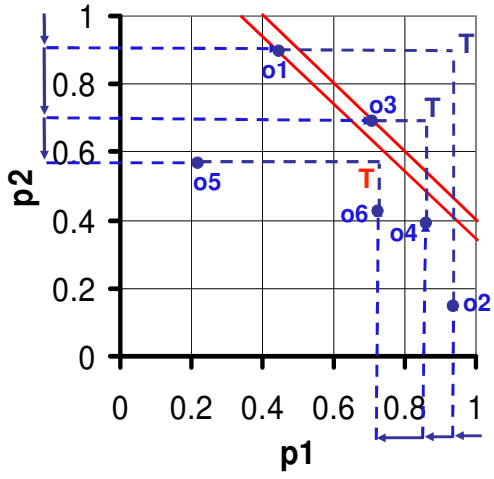
OID	p2
o2	0.95
o3	0.7
o4	0.6
o1	0.5
o7	0.5

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

S(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 ≡ SUM and k = 2



S(o3) > S(o1) ≥ T
Stop!

Performance of TA: Cost model

- In general, TA performs much better than FA, since TA 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) = $\max_j\{\text{Depth}(B,j)\}$. 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:

$$\text{cost}(TA,DB) = m*X \leq m*\text{cost}(B,DB)$$ ■

Adapting to scenarios with different costs

- When $c_{RA} > 0$ (the real case) the cost of TA halting at depth X is at most

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

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

$$\text{cost}(B,DB) \geq X*c_{SA}$$

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

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
...	...

S ≡ SUM

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

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

- To check this, it is sufficient to consider the maximum value of $S^+(o')$ among the objects in B-Res and the threshold (the latter providing an upper bound to unseen objects)

Input: ranked lists L_j (j=1,…,m), integer $k \geq 1$, monotone scoring function S
Output: the top-k objects according to S

1. B := ∅; // entry of type: [OID,lbscore,ubscore]; B is ordered by decreasing lbscore values
2. for j = 1 to m: $\underline{p_j}$:= 1;
3. while B[k].lbscore < max{max{B[i].ubscore, i > k},S($\underline{p_1}$,$\underline{p_2}$,…,$\underline{p_m}$)}:
4. for j = 1 to m:
5. t := getNext$_{L_j}$(); o := t.OID; insert [o,S^-(o),S^+(o)] in B;
6. return {B[1],…,B[k]};
7. end.

NRA: example

L1

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}

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	0.7

1.5 ≥ 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

L1

OID	p1
o1	1.0
o2	1.0
...	0.3
...	...
...	0.3

L2

S ≡ SUM

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 $\lfloor c_{RA}/c_{SA} \rfloor$ 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 ≡ 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

Recap

- 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, and it is instance optimal (with optimality ratio inpendent of c_{RA}/c_{SA}) only for a subset of scoring functions S and a subset of DB's