06.Ranking Queries

The idea is that we have a lot of data stored in data lakes and the objective is to extract data and we want to extract the best result..

Multi-objective optimization

The idea is that we have data stored in tables with attributes, features of data describing them. A general problem formulation:

- Given N objects described by d attributes and some notion of “goodness” of an object. Find the best k objects
  If the numerical value have the possibilities to define a goodness criteria we can implement a criteria optimization.
  This problem is done each time you use a search engine, an e-commerce site,...
  Classical applications:
- Multi-criteria queries: example: ranking hotels by combining criteria about available facilities, driving distance, stars, …
- k-Nearest Neighbors: given N points in some metric (d-dimensional) space, and a query point q in the same space, find the k points closest to q. Used for classification in Machine Learning
  Main approaches:
- Ranking (aka top-k) queries
  - Top k objects according to a given scoring function
- Skyline queries
  - Set of non-dominated objects(not better than an object on one or more the dimension)

Rank aggregation

Rank aggregation is the problem of combining several ranked lists of objects in a robust way to produce a single consensus ranking of the objects. If everybody express a list of preference how can we aggregate these lists in a single list?
Borda’s and Condorcet’s proposals
Takes in account the position of the candidate in respect of the ranking: the lower the ranking the greatest the penalty and the one with the less penalty win(Borda's idea). Condorcet winner: a candidate who defeats every other candidate in pairwise majority rule election. Condorcet's winner may not exist as there could be a cycle in the preferences.
Approaches to rank aggregation
Axiomatic approach

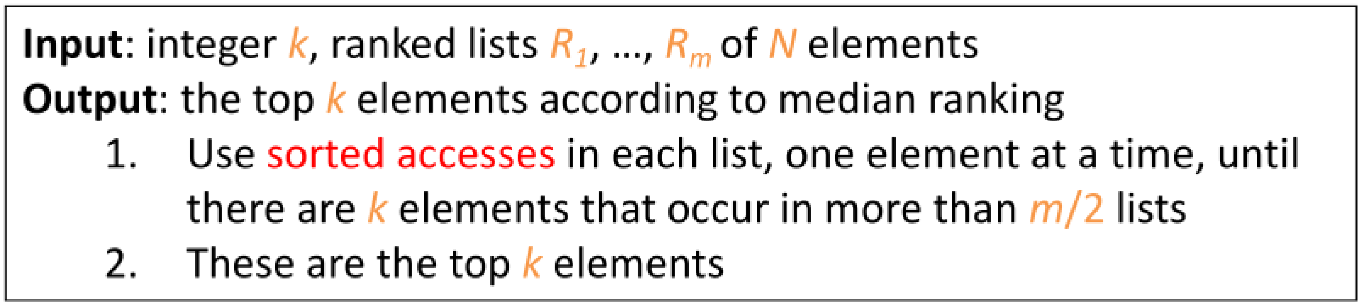
- Desiderata of aggregation formulated as “axioms”

- Arrow’s result: a small set of natural requirements cannot be simultaneously achieved by any nontrivial aggregation function
- Arrow’s paradox: no rank-order electoral system can be designed that always satisfies these "fairness" criteria:
  - No dictatorship (nobody determines, alone, the group’s preference)
  - If all prefer X to Y, then the group prefers X to Y
  - If, for all voters, the preference between X and Y is unchanged, then the group preference between X and Y is unchanged
  Metric approach
- Finding a new ranking R whose total distance to the initial rankings $R_1, \ldots, R_n$ is minimized
- Several ways to define a distance between rankings, e.g.:
- Kendall tau distance $K(R_1, R_2)$, defined as the number of exchanges in a bubble sort to convert R_1 to R_2
- Spearman’s footrule distance $F(R_1, R_2)$, which adds up the distance between the ranks of the same item in the two rankings
- Finding an exact solution is
  - computationally hard for Kendall tau (NP-complete)
  - tractable for Spearman’s footrule (PTIME)
- These distances are related: $K(R_1, R_2) \leq F(R_1, R_2) \leq 2K(R_1, R_2)$, and $F(R_1, R_2)$ admits efficient approximations (e.g., median ranking)

Combining opaque rankings

Techniques using only the position of the elements in the ranking (a.k.a. ranked list)

- no other associated score
  We review MedRank:
- Based on the notion of median, it provides a(n approximation of) Footrule-optimal aggregation


Input: integer k, ranked lists R_1, ..., R_m of N elements
Output: the top k elements according to median ranking

1. Use sorted accesses in each list, one element at a time, until there are k elements that occur in more than $m/2$ lists
2. These are the top k elements

The (maximum) number of sorted accesses made on each list is also called the depth reached by the algorithm. The higher the depth the higher the cost.
Optimality of MedRank
An algorithm is optimal if its execution cost is never worse than any other algorithm on any input. MedRank is not optimal. However, it is instance-optimal. Among the algorithms that

access the lists in sorted order, this is the best possible algorithm (up to a constant factor) on every input instance. In other words, its cost cannot be arbitrarily worse than any other algorithm on any problem instance.

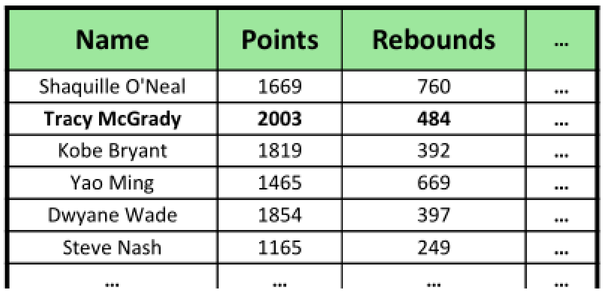
Definition of instance optimality

A form of optimality aimed at when standard optimality is unachievable

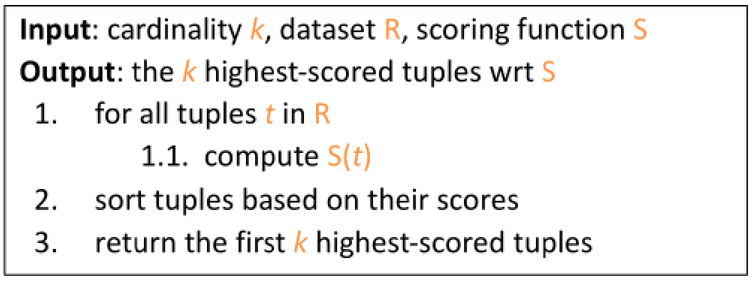
- Let A be a family of algorithms, I a set of problem instances
- Let cost be a cost metric applied to an algorithm-instance pair
- Algorithm A *is instance-optimal wrt. A and I for the cost metric cost if there exist constants* k_1 *and* k_2 *such that, for all A∈**A** and I∈**I**,* $cost(A, I) \le k_1 \cdot cost(A, I) + k_2$
  If A *is instance-optimal, then any algorithm can improve wrt A* by only a constant factor r, which is therefore called the optimality ratio of A*. Instance optimality is a much stronger notion than optimality in the average or worst case! (E.g., binary search is worst-case optimal, but not instance optimal) For each instance, a positive answer can be obtained in 1 probe, a negative answer in 2
  probes (<< log N)
  Example: MedRank:
- Optimality ratio = 2, additive constant = m
- If we want to can get rid of dependence on m, we get optimality ratio = 4

Ranking queries (a.k.a. top-k queries)

Aim: to retrieve only the k best answers from a potentially (very) large result set
Best = most important/interesting/relevant/…
Useful in a variety of scenarios, such as e-commerce, scientific DB's, Web search, multimedia systems, etc.
Needed: ability to "rank" objects (1st best, 2nd best, …)
Ranking = ordering objects based on their "relevance"
Top-k queries: naïve approach
Assume a scoring function S that assigns to each tuple t a numerical score for ranking tuples(E.g. S(t) = t.Points + t.Rebounds) Straightforward way to compute the result



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	...
...	...	...	...



Input: cardinality *k*, dataset R, scoring function S
Output: the *k* highest-scored tuples wrt S

```
1.   for all tuples t in R
        1.1. compute S(t)
2.   sort tuples based on their scores
3.   return the first k highest-scored tuples
```

Naïve approach expensive for large datasets: requires sorting a large amount of data, even worse if more than one relation is involved. S(t) = t.Points + t.Rebounds
Need to join all tuples, which is also costly. Here 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

Two abilities required:

1. Ordering the tuples according to their scores
2. Limiting the output cardinality to k tuples
   Ordering is taken care of by ORDER BY
   Limiting was not native in SQL until 2008
   FETCH FIRST k ROWS ONLY (SQL:2008)
   Supported by IBM DB2, PostgreSQL, Oracle, MS SQL Server, …
   Many non-standard syntaxes available:

- ROWNUM <= k (ORACLE)
- LIMIT k (PostgreSQL, MySQL)
- SELECT TOP k FROM (MS SQL Server)
  ORDER BY have limited power as can miss some relevant answers(near miss) or return too much informations(informations overload)

Semantics of top-k queries

Only the first k tuples become part of the result
If more than one set of k tuples satisfies the ORDER BY directive, any of such sets is a valid answer (non-deterministic semantics)

Evaluation of top-k queries

Two basic aspects to consider:

- query type: 1 relation, many relations, aggregate results, …
- access paths: no index, indexes on all/some ranking attributes
  Simplest case: top-k selection query with only 1 relation:
- If input sorted according to S: read only first k tuples. No need to scan the entire input
- Tuples not sorted: if k is not too large (which is the typical case), perform in-memory sort (through a heap). The entire input needs to be read (cost O(N log k)). Can be improved to O(N + k log N) or even O(N + k log k)

Top-k join queries

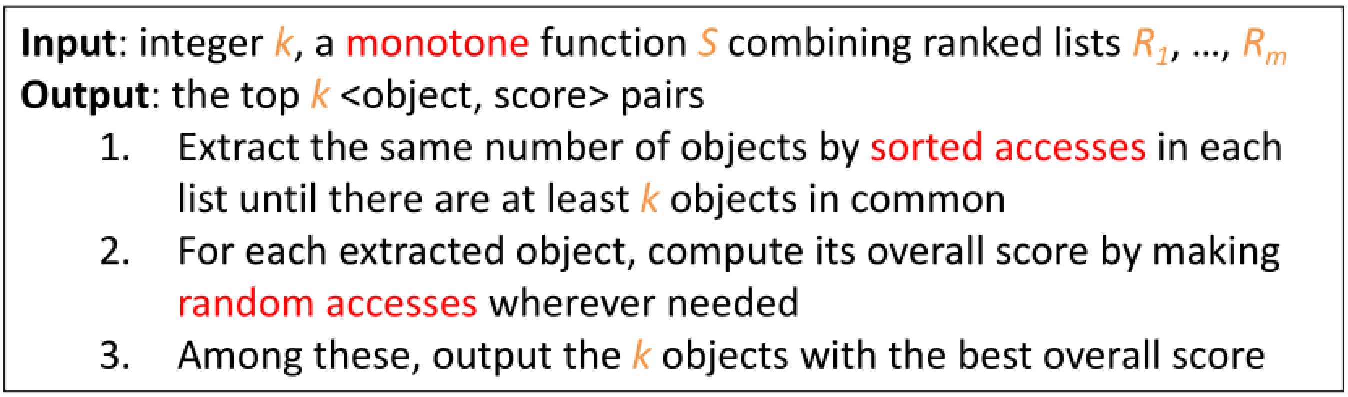
The scoring function takes some criteria to adapt to the result of the query with all the relations it has to respect as we have n>1 input relations
Pay attention to the convention you are using, common scoring functions:

- SUM (AVG): used to weigh preferences equally, SUM(o) º SUM(p(o)) = p1(o) + p2(o) + … + pm(o)

- WSUM (Weighted sum): to weigh the ranking attributes differently, WSUM(o) º WSUM(p(o)) = w1*p1(o) + w2*p2(o) + … + wm*pm(o)
- MIN (Minimum): just considers the worst partial score, MIN(o) º MIN(p(o)) = min{p1(o),p2(o), …, pm(o)}
- MAX (Maximum): just considers the best partial score, MAX(o) º MAX(p(o)) = max{p1(o),p2(o), …, pm(o)}

Fagin's Algorithm (FA)

Focus on monotoning functions.



Input: integer *k*, a monotone function *S* combining ranked lists R_1, …, R_m
Output: the top *k* <object, score> pairs

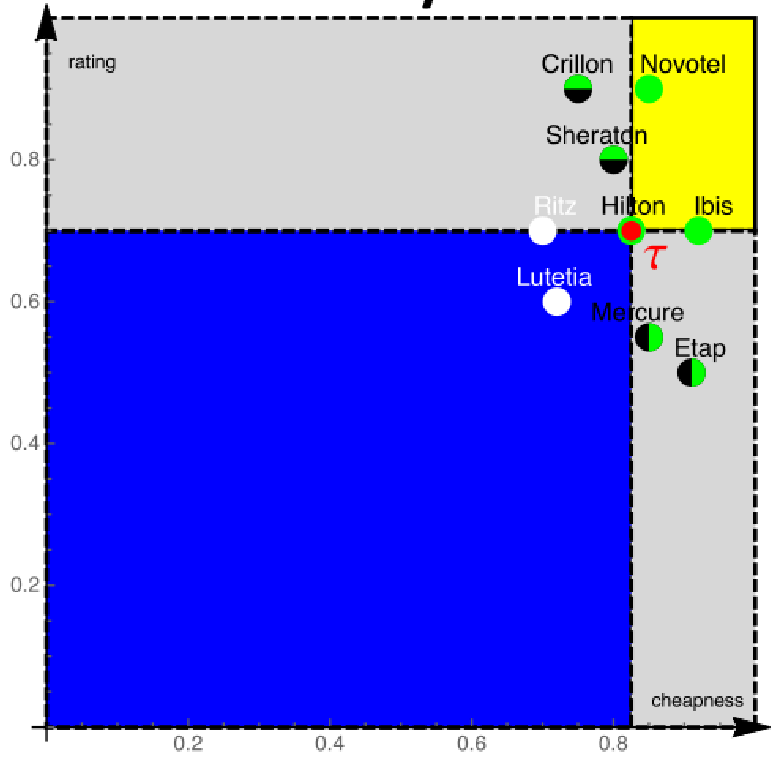
1. Extract the same number of objects by sorted accesses in each list until there are at least *k* objects in common
2. For each extracted object, compute its overall score by making random accesses wherever needed
3. Among these, output the *k* objects with the best overall score

- score is computed by the scoring function S

1. Fisrt phase sorted phase: we descend in the ranking for sorted accesses to find a group of k elements in common in all the rankings.
2. Retrieve phase: retrieve/compute the score for our objects that miss it with random accesses
3. Extraction phase: output the k objects with the best overall score
   Complexity is sub-linear in the number N of objects:

- Proportional to the square root of N when combining two lists (complexity is $O(N^{(m-1)/m}k^{1/m})$
- The stopping criterion is independent of the scoring function
- Not instance-optimal
  In step 1 we don't use the scoring function information so we can do better, for example reducing accesses, with algorithms that use this information.

Why does FA work?



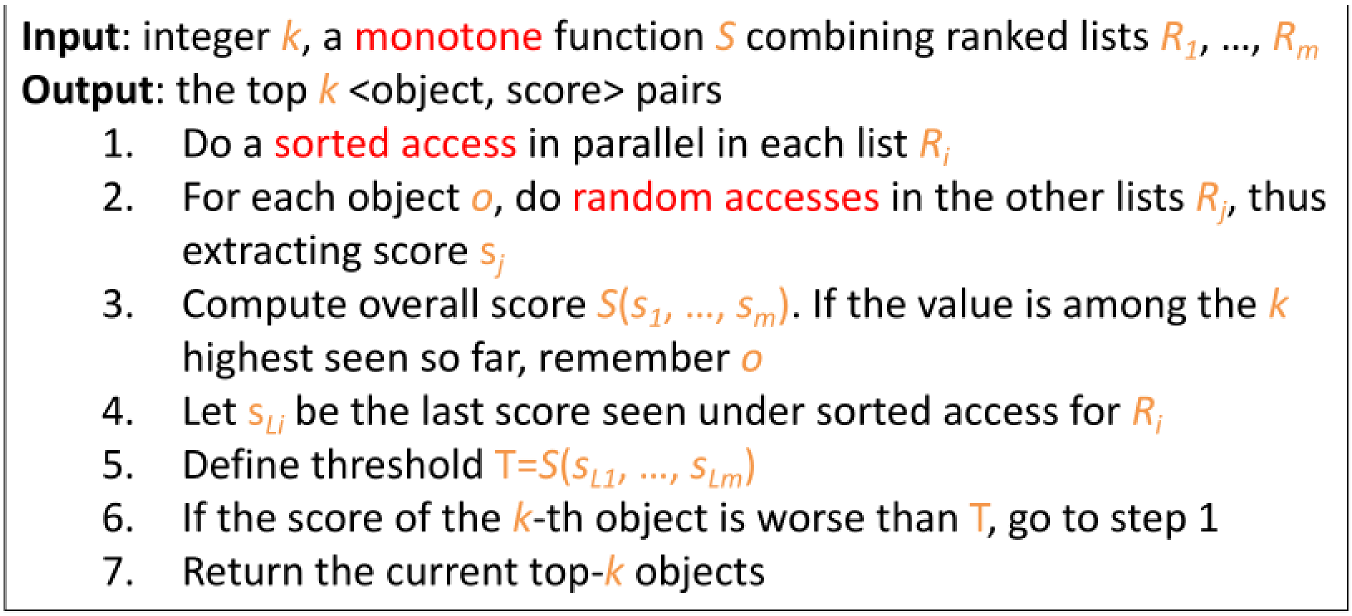
- The threshold point (τ) is the point with the smallest seen values on all lists in the sorted access phase
- FA stops when the yellow region (fully seen points) contains at least *k* points
- The gray regions contain the points seen in at least one ranking
- None of the points in the blue region (unseen points) can beat any point in the yellow region

We use the concept of dominance.

Limits of FA

Drawback: specific scoring function not exploited at all, in particular, the sorted+random access cost of FA is independent of the scoring function!
Memory requirements can become prohibitive: FA has to buffer all the objects accessed through sorted access
Small improvements are possible(E.g., by interleaving random accesses and score computation, which might save some random access)
Significant improvements require changing the stopping condition

Threshold Algorithm (TA)

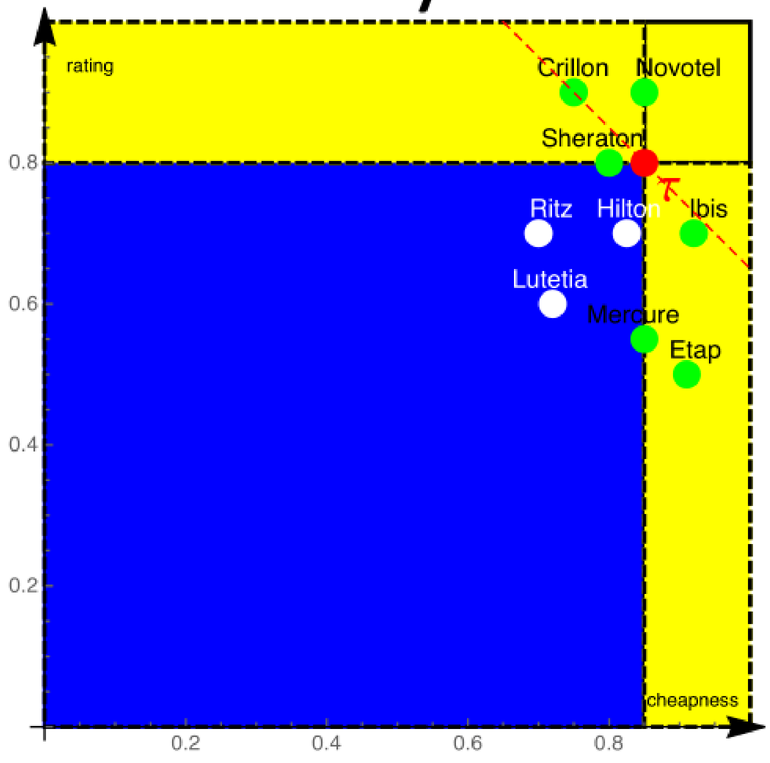


Input: integer k, a monotone function S combining ranked lists $R_1, \ldots, R_m$
Output: the top k <object, score> pairs

```
1. Do a sorted access in parallel in each list $R_i$
2. For each object $o$, do random accesses in the other lists $R_j$, thus
   extracting score $s_j$
3. Compute overall score $S(s_1, \ldots, s_m)$. If the value is among the $k$
   highest seen so far, remember $o$
4. Let $s_{Li}$ be the last score seen under sorted access for $R_i$
5. Define threshold $T=S(s_{L1}, \ldots, s_{Lm})$
6. If the score of the $k$-th object is worse than $T$, go to step 1
7. Return the current top-$k$ objects
```

TA is instance-optimal among all algorithms that use random and sorted accesses (FA is not). The stopping criterion depends on the scoring function. As soon we discover an item with random access we extract its score. For all these objects we compute the score with the scoring function. Now we already have the score and retain the object only if its score is in the top k ⇒ we need a buffer for only k object. We also retain the best score for the ranking. We also define the threshold for each point we have seen. The threshold is the value that cannot be exceeded by the objects not already seen. We can safely stop when we have at least k objects with a score that is greater or equal than the threshold. The depth is lower.

Why does TA work?



- τ is the threshold point
- FA stops when the yellow region (fully seen points) contains at least k points at least as good as τ
- None of the points in the blue region (unseen points) can beat τ
- The dashed red line separates the region of points with a higher score than τ from the rest
  - Now, Crillon is as good as τ and Novotel is better

Performance of TA: Cost model

In general, TA performs much better than FA, since it can "adapt" to the specific scoring function
In order to characterize the performance of TA, we consider the so-called middleware cost:

$$cost = SA * c_{SA} + RA * c_{RA}$$

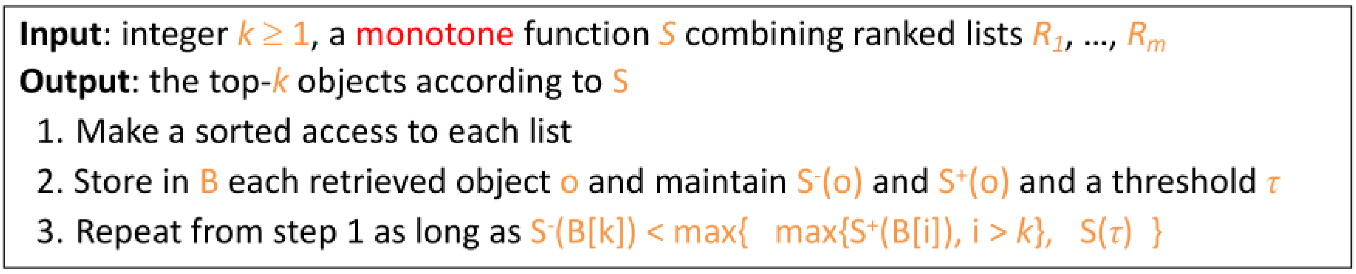
where:

- 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, $c_{SA} = c_{RA}$ (=1, for simplicity)
  In other cases, base costs may differ:
- For web sources, usually $c_{RA} > c_{SA}$
  - limit case $c_{RA} = \infty$ (random access is impossible)
- Some sources might not be accessible through sorted access
  - $c_{SA} = \infty$ (for instance, we do not have an index to process p_j)

The NRA algorithm

The idea is we descend per random accesses and use an upper bound and lower bound as we need to make estimate of best and worst case of an item.
NRA (No Random Access) applies when random accesses cannot be executed_

- It returns the top-k objects, but their scores might be uncertain
  - This is to limit the cost of the algorithm
  Idea: maintain, for each object o retrieved by sorted access, a lower bound $S^-(o)$ and an upper bound $S^+(o)$ on its score. NRA uses a buffer B with unlimited capacity, which is kept sorted according to decreasing lower bound values
  The idea is we do sorted accesses until we have k objects whose lower bounds are better than all the upper bounds of the objects we have seen and than the threshold we can stop.



Input: integer $k \geq 1$, a monotone function S combining ranked lists R_1, ..., R_m
Output: the top-k objects according to S

```
1. Make a sorted access to each list
2. Store in B each retrieved object o and maintain $S^-(o)$ and $S^+(o)$ and a threshold $\tau$
3. Repeat from step 1 as long as $S^-(B[k]) < \max\{ \max\{S^+(B[i]), i > k\}, S(\tau) \}$
```

Maintain a set Res of current best k objects. Halt when no object o' not in Res can do better than any of the objects in Res. $\forall o' \notin Res, \forall o \in Res : S^+(o') \leq S^-(o)$
I.e., when $S^-(o)$ beats both the max value of $S^+(o')$ among the objects in B-Res and the score S(τ) of the threshold point τ (upper bound to unseen objects)
We need to do two checks: is the lower bound better than the threshold and of all the upper bound seen before.

NRA observations

NRA’s 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!
NRA is instance-optimal among all algorithms that do not make random accesses: optimality ratio is m
There is a quirk in the fact that it may be easier to find the k = 2 items instead of k = 1 objects. Top 2 is easier to compute than top 1

The overall picture

Algorithm	Scoring Function	Data Access	Notes
B_0	MAX	sorted	instance-optimal
FA	monotone	sorted and random	cost independent of scoring function
TA	monotone	sorted and random	instance-optimal
NRA	monotone	sorted	instance-optimal, no exact scores

Summary on top-k 1-1 join queries

There are several algorithms to process a top-k 1-1 join query. Common assumption: the scoring function S is monotone. Simplest case: MAX
FA’s stopping condition does not use the scoring function
TA’s stopping condition is based on a threshold T, which provides an upper bound to the scores of all unseen objects
TA is instance-optimal, FA isn’t
NRA does not execute random accesses at all
A scoring function is a way to express preferences. We want to control the output cardinality.

Ranking queries – main aspects

Very effective in identifying the best objects wrt. a specific scoring function
Excellent control of the cardinality of the result, k is an input parameter of a top-k query
For a user, it is difficult to specify a scoring function E.g. the weights of a weighted sum
Computation is very efficient:

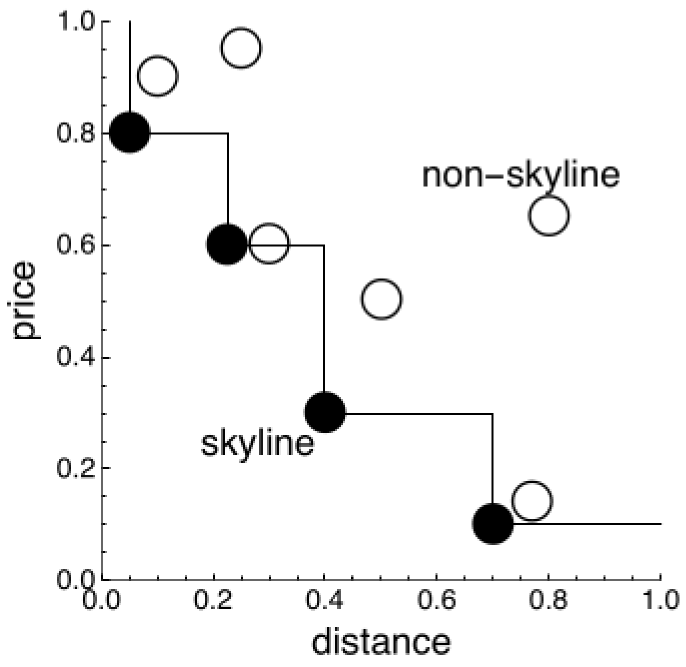
- E.g., O(N log k) for local, unordered datasets of N elements
- Instance optimality in some settings
  Easy to express the relative importance of attributes

Skyline queries

Quite recent in research in databases queries. Tries to address a limitation in top-k queries, don't use a scoring function as they focus on an overview of the interesting elements in a

dataset. Use the notion of dominance:
Tuple t dominates tuple s, indicated t $\prec$ s, iff

- ∀i. 1≤i≤m → t[A_i] ≤ s[A_i] (t is nowhere worse than s)
- ∃j. 1≤j≤m ∧ t[A_j] < s[A_j] (and better at least once)
  Typically, lower values are considered better. Opposite convention wrt. top-k queries, **but it's just a convention that can be changed!**
  Dominance means no way worse and at least once better.
  The skyline of a relation is the set of its non-dominated tuples. Aka:
- Maximal vectors problem (computational geometry)
- Pareto-optimal solutions (multi-objective optimization)



Skylines: Properties

A tuple t is in the skyline iff it is the top-1 result w.r.t. at least one monotone scoring function i.e., the skyline is the set of potentially optimal tuples!
Skyline ≠ Top-k query: there is no scoring function that, on all possible instances, yields in the first k positions the skyline points. Based on the notion of dominance

Skyline queries in SQL

Only a proposed syntax (not part of the standard):

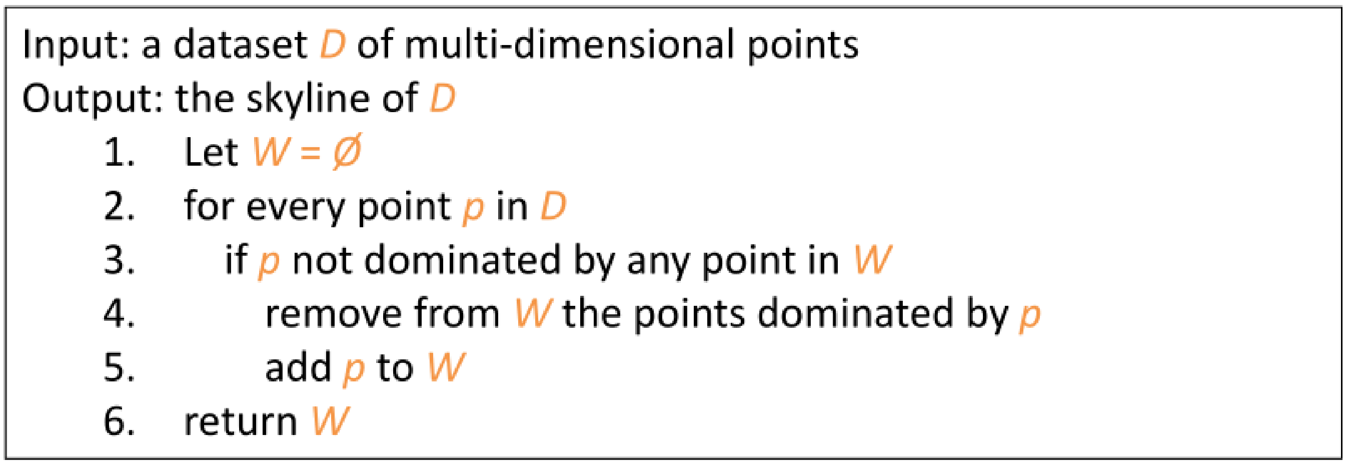
```
SELECT ... FROM ... WHERE ...
GROUP BY ... HAVING ...
SKYLINE OF [DISTINCT] d1 [MIN | MAX | DIFF],
                                    ..., dm [MIN | MAX | DIFF]
ORDER BY ...

Example
```

```
SELECT * FROM Hotels WHERE city = 'Paris'
SKYLINE OF price MIN, distance MIN

Can be easily translated to actual SQL:
SELECT * FROM Hotels h
WHERE h.city = 'Paris' AND NOT EXISTS (
SELECT * FROM Hotels h1
WHERE h1.city = h.city AND
h1.distance <= h.distance AND
h1.price
<= h.price AND
(h1.distance < h.distance OR
h1.price
< h.price))
```

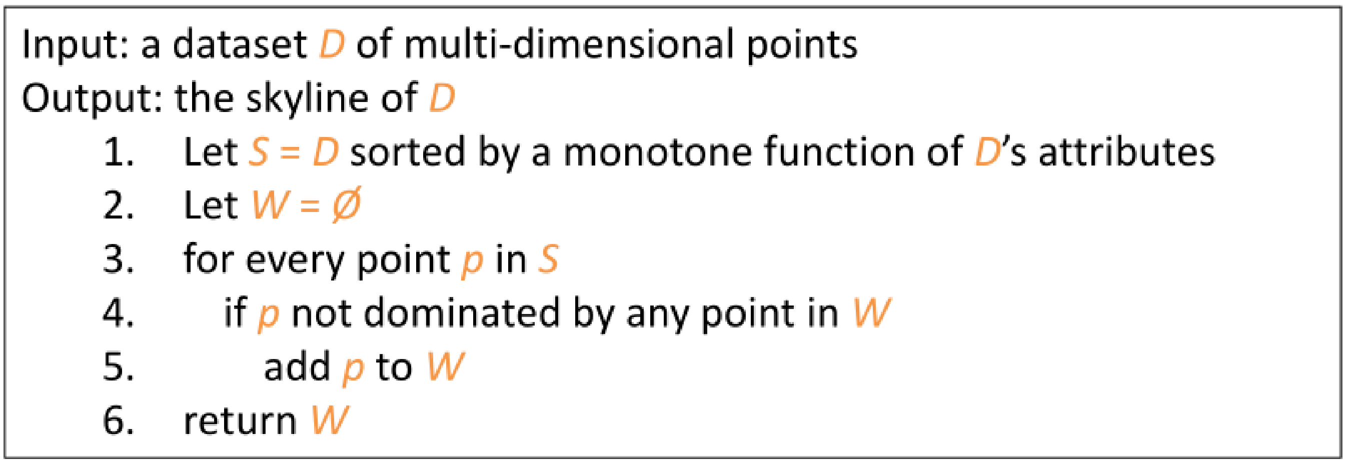
Skylines – Block Nested Loop (BNL)



```
Input: a dataset $D$ of multi-dimensional points
Output: the skyline of $D$
    1.  Let $W = Ø$
    2.  for every point $p$ in $D$
    3.      if $p$ not dominated by any point in $W$
    4.          remove from $W$ the points dominated by $p$
    5.          add $p$ to $W$
    6.  return $W$
```

Computation is $O(n^2)$ where $n = |D|$. Very inefficient for large datasets

Skylines – Sort-Filter-Skyline (SFS)



```
Input: a dataset $D$ of multi-dimensional points
Output: the skyline of $D$
    1.  Let $S = D$ sorted by a monotone function of $D$'s attributes
    2.  Let $W = Ø$
    3.  for every point $p$ in $S$
    4.      if $p$ not dominated by any point in $W$
    5.          add $p$ to $W$
    6.  return $W$
```

Pre-sorting pays off for large datasets, thus SFS performs much better than BNL. If the input is sorted, then a later tuple cannot dominate any previous tuple!

- Will never compare two non-skyline points

- Can immediately output any points in W as part of the result
  But still $O(n^2)$
  You can only dominate points that arrive later than you in the (sorted)dataset.

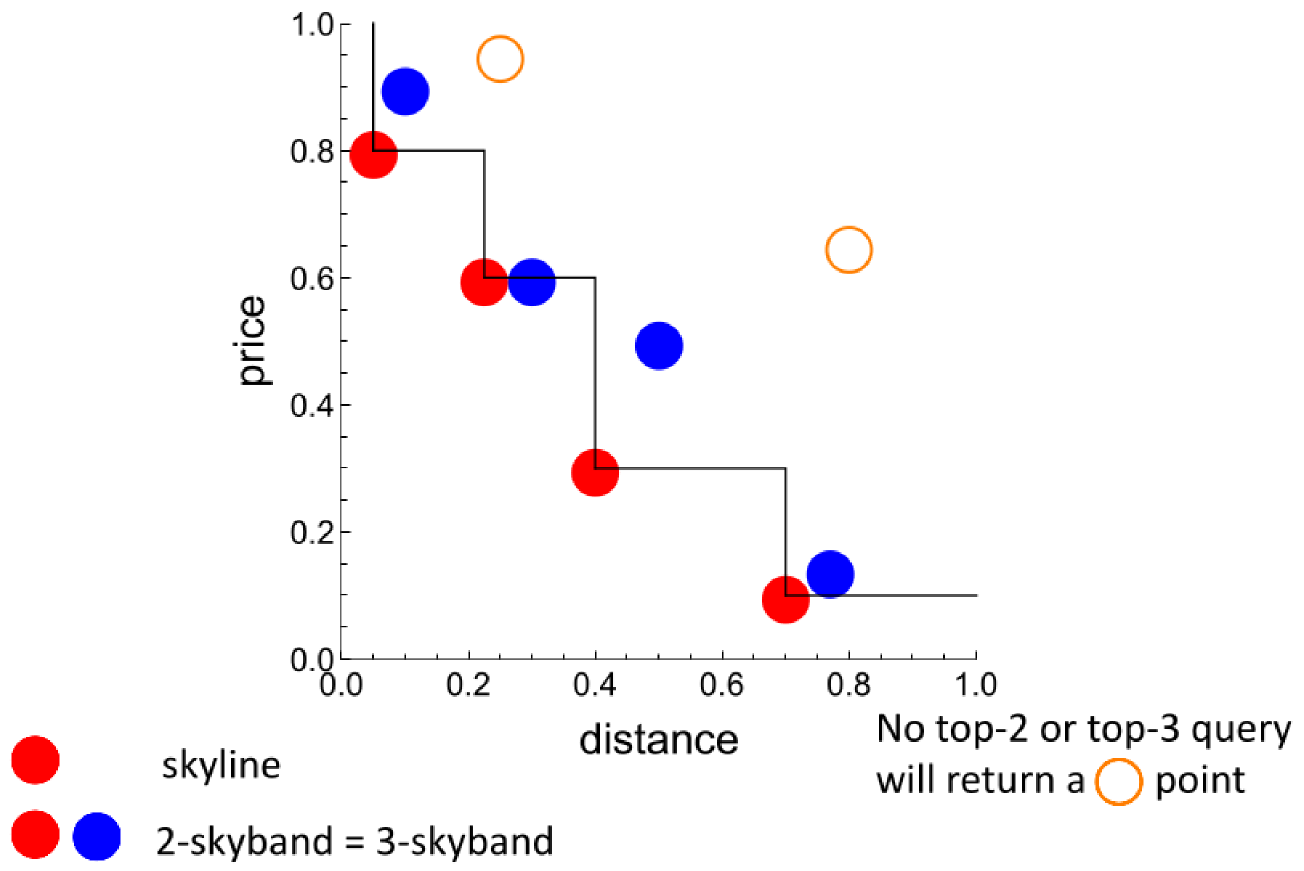
Skylines – main aspects

Pros:

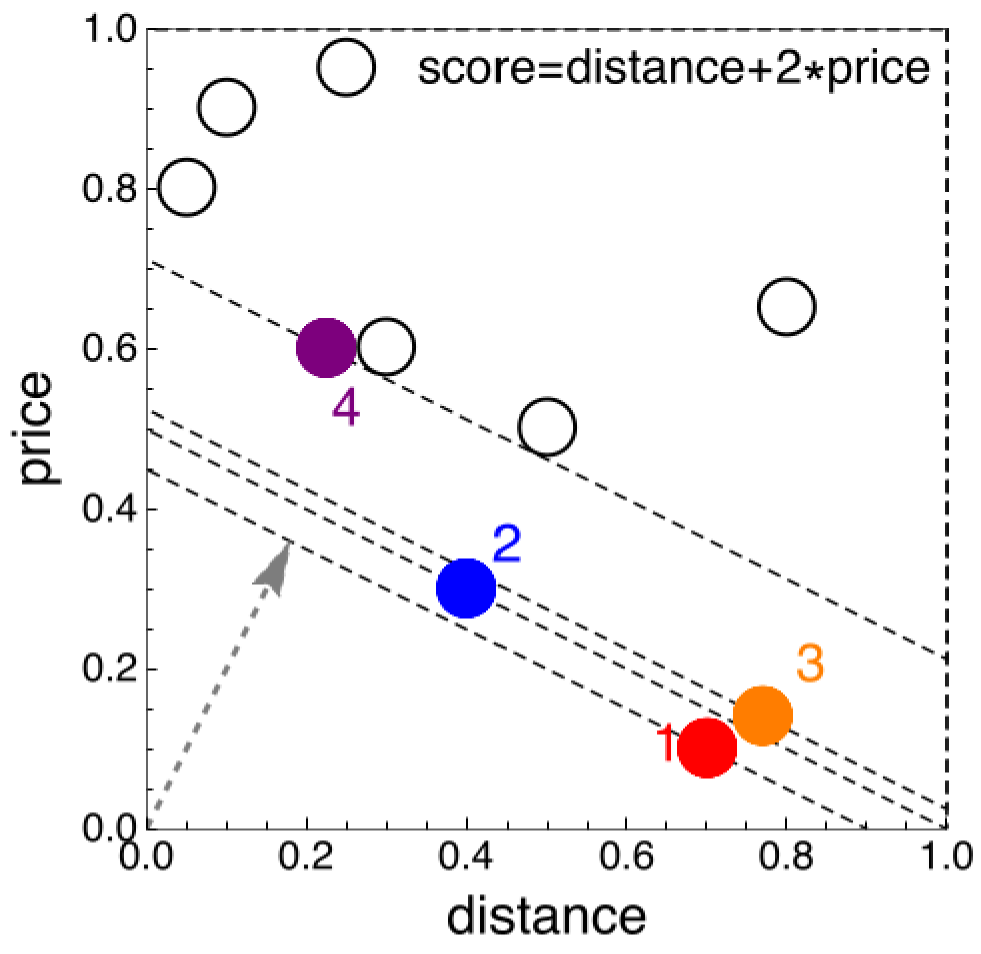
- Effective in identifying potentially interesting objects if nothing is known about the preferences of a user
- Very simple to use (no parameters needed!)
  Cons:
- Too many objects returned for large, anti-correlated datasets
- Computation is essentially quadratic in the size of the dataset (and thus not so efficient)
- Agnostic wrt. known user preferences (e.g., price is more important than distance)
  Extension: k-skyband = set of tuples dominated by less than k tuples.
- Skyline = 1-skyband
- Every top-k result set is contained in the k-skyband
  Another drawback is that a skyline query can return too much data(worse for dataset with uncorrelated data). If no point dominate the others skyline is useless as it returns nothing.

Comparing different approaches

Example: skyline/k-skyband query



Example: ranking query



	Ranking Queries	Skyline Queries
Simplicity	NO	YES
Overall view of interesting results	NO	YES
Control of cardinality	YES	NO
Trade-off among attributes	YES	NO

Extending Skylines and Top-k Queries

Main Objectives

- Output size control
- Preferences and flexibility (no magic numbers)
- Ability to retrieve all interesting data points
- Stability of results wrt. preferences/weights
- Balance of results (wrt. preferences/weights)
- Scale invariance: stretching axes has no effect
- Stability wrt “junk”: adding/removing dominated tuples has no effect

Selected Approaches

Early approaches (aim: output size control)

- Counting dominated tuples
- Representative Skylines
- Regret-minimizing Sets
  Newer approaches
- Flexible Skylines
- ORD and ORU
- Directional Queries

Top-k dominating query

Add a rank to the skyline tuples, this rank represent the tuples that are dominates.
Problems:

- Scale-invariant
- Not stable wrt the presence or not of interesting points
- Too many ties in the ranking(not good when talking about ranking)
- Non-skyline tuples may beat skyline tuples
- No preferences can be specified

Overall Dominance

Select a subset of the skyline that maximizes the number of tuples dominated by at least one tuple in the the subset
Problems:

- Not Stable
- Scale-Invariant
- No preferences can be specified

Distance-based representative skyline

Select a set R of k skyline tuples that minimizes the distance between any skyline tuples not in R to the closest tuple in R(represent the skyline as much as possible)
Problems:

- Stable
- Not scale-invariant: result can change if we change the scale
- NP-hard
- No preferences can be specified

Regret-Minimizing Sets

Problem of finding a subset of the skyline to satisfy the minimization of the regret:
for a scoring function f and a set D, let $Top_f(D) = max_{x \in D} f(x)$ (top score via f in D). The regret of S⊂D is $(Top_f(D) - Top_f(S))/Top_f(D)$. Find a set S of size k minimizing its maximum regret for any linear scoring function.
Problems:

- may be used to add cardinality control to skylines
- stable and scale-invariant: not affected by stretching the x-axis
- no preferences: no way to accomodate preferences
- only linear scoring functions(heavy restriction)

Skyline Revisited

We replace the set of the monotonic scoring functions with a subset of it. We revisit the notion of dominance:

$$t \prec_F s \text{ iff } (1)\; \forall f \in F.\, f(t) \leq f(s) \; (2)\; \exists f \in F.\, f(t) < f(s)$$

Condition (2) can be omitted if the function can distinguish between any two tuple: for every two tuple they are distinguishable if they don't have the same value for at least one scoring function
Skyline is in a Non-Dominated F-skyline if it is not dominated by one tuple in the skyline.

F-dominance Regions

We can extend to the notion of F-dominance to the dominance region t as we use the fact that it is all the set of points F-Dominates by t.

$$DR(t, F) = \{s | t \prec_F s\}$$

Flexible skylines – basic properties

Relationship with skylines:

- PO(r; M) = ND(r; M) = Sky(r)
- PO(r; F) ⊆ ND(r; F) ⊆ Sky(r)
  Relationship with top-k queries:
- PO(r; { f }) ≈ ND(r; { f }) ≈ top-1 query wrt. F(beware of ties)
  Monotonicity: if F1 ⊆ F2 then
- ND(r; F1) ⊆ ND(r; F2)
- PO(r; F1) ⊆ PO(r; F2)
  There can be ties and ND returns all the tying tuple, PO return nothing

xtensions of Flexible Skylines

Idea: leverage the k-skyband to target the potential top k (instead of just top 1) tuples
$ND_k(r;F)$ = tuples F-dominated by less than k tuples in r
PO_k(r;F) = tuples in r that are top-k for at least one function in F
Both NDk and POk (essentially) coincide with

- Top-k query if F is a single scoring function
- k-Skyband if F = M (all monotone functions)
  The idea is using the same definition of before but introduce the k parameter as you need to be inside the top-k results.
  The idea is that now we can accomodate the use of preference in skyline and we are flexible in respect to the weights.
  Pros:
- User preferences (via constraints on weights)
  - More robust with respect to magic numbers
  - E.g., interval ci - ε ≤ wi ≤ ci + ε instead of wi = ci
- Reduced output size
- Computed efficiently for Lp norms with linear constraints on weights
  Cons:
- Cardinality of output not directly controllable
  - Even less so for extensions based on k-skybands
- Still slow for loose constraints

Adding cardinality control

Extension of flexible skyline of level k that tries to extend control on the output size. The idea is considering linear scoring function and the starting point we use $ND_k(ORD)$ or $PO_k(ORU)$.
The idea is to use a zero distance from our weight and obtain results until we obtain a number of result equal to the dimension of our data set.
Aim: Output-Size Specified (OSS) operators
• Idea:

- Collect user preferences (weight vector $w =< w_1, \ldots, w_d >$)
- Apply either ND_k (called ORD) or PO_k (ORU)
- Limit their output size to a user-defined number m
  - Use a set F of linear scoring functions whose weight vector w' is at a distance at most ϱ from w so that the output size is exactly m
  - We get more tuples as ϱ grows
    - If ϱ=0 then m=k and we get the top-k results with weights w
    - If ϱ unbounded, then m=|k-Skyband| with ORD
      - (possibly less with ORU)

Features of ORD and ORU

Pros:

- OSS operators
  - output size may be easier than constraints on weights
- Personalized with user preferences (weights)
- Flexible (weights used loosely)
  Cons:
- Too many size parameters (k and m)
  - When k=m, it’s a standard linear top-k query (ρ = 0)
- Restricted to linear functions
  - The most common choice, but…

Beyond Linear Queries

The limits of linear top-k queries is that they can give interesting results but difficult to retrieve. An indicator of difficulty can be best rank(scale-invariant and not stable).
Convex and concave tuples: Linear scoring functions (L1) and convex hulls: brank(t)=1 ⇔ t is the top-1 by some linear function ⇔ t ∈ convex hull (of dataset + frontier points)
We can use the concavity degree as an indicator of difficulty. A tuple t is convex if brank(t)=1, concave otherwise. NB: ND(r;L1)=Sky(r), but, generally, PO(r;L1)⊂Sky(r). Amount of non-linearity needed to rank t as top-1. cdeg(t) = smallest p to rank tuple t as top-1 with f ∈ Lp. Here, cdeg(t2)=2(t2 is top with a function in L2). Both scale-invariant and stable
We can use as an indicator of interest the exclusive volume. evol(t) = volume of the region dominated only by t. evol(t) expresses the potential of dominating other tuples that we would miss if we excluded t. Only relevant for skyline tuples. Both scale-invariant and stable
Another indicator of interest os the grid resistance. gres(t) = smallest value of g^{-1} for which t is no longer in the skyline when snapping values to a grid with g equal-size intervals along each axis. g^{-1} is the grid step. Only relevant for skyline tuples. Here, gres(t2) ≤ 1/4. Both scale-invariant and stable

Features of Directional Queries

Pros:

- Increased balance of results
- Increased overall exclusive volume
- Increased overall grid resistance
- Can retrieve all difficult tuples
- Retains main advantages of top-k queries
  Cons:
- What is the right mix (β) of linear + balance?

- Monotonicity of the scoring function may be lost