A General Top-k Algorithm for Web Data Sources

Mehdi Badr, Dan Vodislav

A general top-k algorithm for web data sources*

Mehdi Badr and Dan Vodislav

ETIS, CNRS, University of Cergy-Pontoise, France
firstname.lastname@u-cergy.fr

Abstract. Several algorithms for top-k query processing over web data sources have been proposed, where sources return relevance scores for some query predicate, aggregated through a composition function. They assume specific conditions for the type of source access (sorted and/or random) and for the access cost, and propose various heuristics for choosing the next source to probe, while generally trying to refine the score of the most promising candidate. We present *BreadthRefine* (BR), a generic top-k algorithm, working for any combination of source access types and any cost settings. It proposes a new heuristic strategy, based on refining all the current top-k candidates, not only the best one. We present a rich panel of experiments comparing BR with state-of-the art algorithms and show that BR adapts to the specific settings of these algorithms, with lower cost.

Keywords: top-k queries, web data, multi-criteria information retrieval, ranking

1 Introduction

Data retrieval applications accorded an increasing importance these last years to ranked queries, compared to traditional boolean queries. This is, to a large extent, the consequence of the great development of web applications integrating huge volumes of heterogeneous and multimedia data: text, images, video, maps, etc. On one hand, this introduced the need for fuzzy, approximate answers, ranked by relevance, when querying such large and heterogeneous amount of data, on the other hand the new web data types came with intrinsic ranked predicates: text and image similarity, location proximity, user preferences, etc.

In this context, we address the problem of processing top-k queries over a set of web sources. A query is expressed through a set of ranked predicates over a common set of data objects. Each predicate is independently evaluated by some web source and returns a relevance score for any input object. A monotone aggregation function combines partial scores from each predicate into the final object score relative to the query. The query returns objects having the k best global scores.

Consider the example of a tourist in Paris, looking for buildings constructed around 1750, near to his current location and similar to the one he just photographed. Objects are here the buildings in Paris and the query consists of three ranked predicates: p_1: proximity of the construction date, p_2: spatial proximity to the current location, and p_3: similarity with a given picture.

In SQL-like syntax, e.g. the one proposed by RankSQL [10], this query could be expressed such as in Figure 1, if we consider $k = 5$ and a simple aggregation function based on the sum of the individual scores.

The sources that evaluate the predicates could be, for instance:

* The original publication is available at www.springerlink.com

```
select * from Building b
order by proximity(b.year, 1750) +
          closeness(b.address, here()) +
          similarity(b.image, myImage)
limit 5 ;
```

S_1 (S-source)	S_2 (SR-source)	S_3 (R-source)
(o_2, 0.4)	(o_3, 0.9)	(o_1, 0.9)
(o_1, 0.3)	(o_1, 0.2)	(o_2, 0.7)
(o_4, 0.25)	(o_4, 0.15)	(o_3, 0.8)
(o_3, 0.2)	(o_2, 0.1)	(o_4, 0.6)

Fig. 1. Example query and sources

- S_1: a database server storing historical information about buildings (for p_1);
- S_2: a geographic/map service (for p_2);
- S_3: an image database indexing fronts of buildings in Paris (for p_3).

Note that the ranking predicates are *dependent on the query*, for instance, even if S_2 always evaluates spatial proximity between an object and a reference point, this point depends on the query.

To execute such a query, one must access the web sources to get partial scores for objects. The access to the web sources during query processing has the following main properties:

- Access is limited to operations allowed by the web source interface and there is no control on the inside mechanisms. Generally, a web source may allow two kinds of access: *sorted*, where each access returns the next score/object in decreasing order, and/or *random*, where each access returns the score of a given object. We call *S-sources* sources with sorted access only, *R-sources* those with random access only, and *SR-sources* those with both accesses.
- Accessing web sources is expensive, the cost of accessing sources dominates the cost of the other algorithm operations.

The naive approach would be to get all the partial scores for all the objects, then to compute their global scores, to order them by descending score and get the first k results. In practice, this method is very expensive and many efficient algorithms have been proposed for various cases of access types and cost settings.

Some algorithms, such as NRA [4] and StreamCombine [6] consider only S-sources, while algorithms such as TA [4], CA [4] and QuickCombine [5] consider only SR-sources. A third category considers specific heterogeneous access configurations, e.g. one S-source and several R-sources for Upper [2] and MPro [3], or several SR-sources and several R-sources for some extensions of TA and Upper. The NC framework [14] is the only approach that addresses any combination of access types. All these algorithms are further detailed in the related work section.

Cost settings considered by these algorithms fall generally into two categories: (i) no difference between the cost of sorted and random accesses (NRA, StreamCombine, TA, QuickCombine), and (ii) random more expensive than sorted access (CA, Upper, MPro). However, other cost settings are possible, e.g. sorted more expensive than random access. Consider for instance the case of a source evaluating image similarity by using an index on disk for sorted access, but keeps image descriptors in memory; random access requires here no disk access and is faster than sorted access. Again, NC is the only attempt to adapt to any cost settings.

The general idea behind all these top-k algorithms is to maintain a list of candidate objects and the interval $[L, U]$ of possible global scores for each of them. At the beginning, the interval is obtained by aggregating the minimum / maximum source scores. Monotonicity of the aggregation function ensures that further source accesses will refine these intervals, by decreasing the upper bound U and increasing the lower bound L. The algorithm stops when the score of the best k candidates cannot be exceeded by the other objects.

We illustrate the behavior of such an algorithm through the example in Figure 1. Suppose S_1 is a S-source, S_2 a SR-source, S_3 a R-source; scores are presented in descending order for S/SR sources and by object id for R-sources. Individual predicate scores belong to the $[0, 1]$ interval, so the initial global score interval is $[0, 3]$ for all objects. Note that we consider *algorithms without wild guesses*, i.e. objects are not known in advance, they are 'discovered' by sorted accesses. We note *candidates* the set of candidates and U_{unseen} the maximum score of objects not yet discovered. Initially, $candidates = \emptyset$ and $U_{unseen} = 3$.

- A sorted access to S_1 retrieves (o_2, 0.4), so with one partial score known for o_2 its global score interval becomes $[0.4, 2.4]$, i.e. $candidates$=$\{(o_2, [0.4, 2.4])\}$. Also U_{unseen} becomes 2.4 because further scores in S_1 cannot exceed 0.4.
- A sorted access to S_2 retrieves (o_3, 0.9). This adds a new candidate (o_3), lowers U_{unseen} to 2.3 (because further S_2 scores cannot exceed 0.9), but also lowers the maximum global score of o_2 to 2.3, because the maximum score of S_2 is now 0.9. $candidates$=$\{(o_2, [0.4, 2.3]), (o_3, [0.9, 2.3])\}$.
- A random access to S_2 for o_2 retrieves (o_2, 0.1). This changes only the global score interval of o_2. $candidates$=$\{(o_2, [0.5, 1.5]), (o_3, [0.9, 2.3])\}$.
- A random access to S_3 for o_3 retrieves (o_3, 0.8) and changes the global score interval of o_3. $candidates$=$\{(o_2, [0.5, 1.5]), (o_3, [1.7, 2.1])\}$.
- A sorted access to S_2 retrieves (o_1, 0.2). This adds a new candidate (o_1), lowers U_{unseen} to 1.6, but does not lower the maximal global score of the other candidates because o_2 and o_3 already know their S_2 scores. $candidates$=$\{(o_2, [0.5, 1.5]), (o_3, [1.7, 2.1]), (o_1, [0.2, 1.6])\}$.
- The minimum global score of o_3 exceeds both U_{unseen} (maximum global score of unseen objects) and the maximum global score of all the other candidates, we can conclude that o_3 is the best (top-1) object.

Execution may continue depending on the value of k, but notice that *top-k objects could be returned without knowing their exact scores and order*. Most of the existing algorithms return exact scores for top-k and are iterative, i.e. return first top-1, then top-2, etc. However, in many applications the exact global score is not needed (e.g. image retrieval), but only the top-k objects with approximate ordering.

The difference between the top-k algorithms mentioned above consists in the heuristics proposed for the choice of the next source to probe and of the candidate to refine through random accesses. Note that all the algorithms that must select a candidate to further refine focus on the current "best" candidate.

Our goal is to propose a new generic top-k algorithm that works for any combination of source access types and cost settings, and returns the set of top-k objects, possibly without complete scoring information. This algorithm uses a new heuristic approach, that refines on all the top-k candidates, instead of favoring only the best current candidate.

The contributions of this paper may be resumed as follows:

- We present *BreadthRefine* (BR), a new, generic top-k algorithm, able to adapt to any combination of source access types and to any cost settings. To the best of our knowledge, excepting NC [14], this is the first generic top-k algorithm for web sources. Unlike NC, that mixes heuristics and sampling optimization, our algorithm is only based on a simple heuristics.
- We propose a new heuristic approach, that do not focus only on the best current candidate, but considers all the top-k candidates. Our experiments show that this heuristics produces better results than the usual approach.
- We report a rich experimental evaluation comparing BR to existing algorithms for various source types and cost settings, and show that BR is less expensive.

The rest of the paper is organized as follows: the next section presents related work, then Section 3 describes the BR algorithm, Section 4 reports the experiments comparing BR to existing algorithms, then we end with conclusions and future work.

2 Related work

Top-k query processing techniques have been largely studied in the last decade at different levels: query model, access types, implementation structures, integration in database engines [10][7][9], etc. The survey [8] presents a rich overview of these various approaches. In this context, we address the problem of selection queries (no joins), for web sources with any configuration of access types and any cost settings. We do not compare with algorithms having additional information about objects in sources, e.g. the rank in BPA [1]. Join queries are addressed e.g. by the J* [13] or the Rank-Join [7] algorithms.

Algorithms for top-k selection queries proposed so far focus on specific access types and cost settings for the sources. They fit with the general method illustrated in the example of Section 1, i.e. maintaining a list of candidates with global score interval and accessing sources following their own heuristics.

Among the algorithms that consider only S-sources, the best known is *NRA* (No Random Access) [4]. NRA consults sources following a simple round-robin strategy, with no specific order. Such as for BR, final results may have incomplete scoring information. Efficient NRA implementations such as LARA [11] that reduce the overhead of candidate updates are not relevant in our context, since we ignore this overhead compared to the access time to web sources. *StreamCombine* [6] is a variant of NRA that selects at each step the next source to probe following the benefit that the source may provide. Benefit considers three factors: (i) the importance of the source in the global score (e.g. the coefficient in the aggregation function, for a weighted sum), (ii) the decrease of the source score (bigger decreases favor faster algorithm termination), and (iii) the number of candidates in the current top-k not yet seen in the source and that will consequently lower their upper bound. We adopt in BR a similar notion of benefit for sorted sources.

Algorithms that consider only SR-sources adopt a different approach: the global score of a candidate discovered through a sorted access is completely evaluated through random probes of the other sources. The consequence is that candidates have exact scores, not intervals, so there is no need to maintain a list of candidates. The termination test simply compares the score of the k-th candidate with the threshold U_{unseen}.

TA (Threshold Algorithm) [4], the best known SR-source algorithm, consults sources in a way similar to NRA, following a round-robin strategy. *QuickCombine* [5] is a variant of TA that uses the same idea as StreamCombine to select the next sorted source to probe. The benefit considers only the two first factors presented above for StreamCombine, the last one being not relevant. *CA* (Combined Algorithm) [4] is a variant of TA that considers random accesses being h times more expensive than sorted ones. It combines NRA with TA to reduce the number of random accesses by performing h sorted accesses in each source before a complete evaluation of the best candidate by random probes. We adopt in BR a similar idea for taking into account the possible difference between random and sorted costs. Many other extensions of TA have been proposed, such as *TAz* [2], which considers an additional set of R-sources. TAz acts as TA for the sorted accesses, but includes the R-sources in the random probe phase.

Algorithms with sorted access and controlled random probes, such as *Upper* [2] and *MPro* [3], typically consider one S-source and several R-sources. The random cost exceeds the sorted

cost, but is different from one R-source to another. Both Upper and MPro consider complete scoring of the final top-k result.

Upper maintains candidates following an *estimated* score. At each step, it considers candidate o with the highest upper score U: if $U < U_{unseen}$ a sorted access is performed in order to reduce U_{unseen}, otherwise a random probe for o is done. A benefit is computed for each R-source and the best source is selected for the random probe. Two cases are considered. If o belongs to the current top-k, the benefit is the ratio between the expected decrease δ of U and the access cost. Otherwise, o has more chances to be out of the top-k and one computes the decrease Δ of U necessary to prove that o is not a top-k object. The source benefit in this case is the ratio between $min(\delta, \Delta)$ and the access cost. An extension of Upper for several SR-sources and several R-sources is presented in [12] - in this case sorted access is performed in NRA manner over the SR-sources, while random access considers all the SR/R-sources not yet probed.

Like Upper, *MPro* considers at each step the candidate with the highest upper score and performs a random probe for it or a sorted access if U_{unseen} is higher. But unlike Upper, MPro fixes for all the candidates *the same order* (schedule) for probing the R-sources. The best schedule may be determined by various methods, such as sampling optimization, proposed by the authors.

The only algorithm that aims at genericity is *NC* (Necessary Choices) [14], an extension of MPro. Like BR, NC is generic and adapts to any source access type and cost settings. NC identifies *necessary choices* (i.e. accesses that are necessary at a given execution state to obtain the final result) as belonging to accesses for the current top-k upper bound candidates. NC proposes an algorithm framework that performs only necessary accesses and defines an algorithm called SR/G in this framework that computes for each S/SR-source a *limit score*. SR/G gives priority to sorted accesses in sources that did not reach the limit score. More precisely, the algorithm considers at each step the candidate with the highest upper score and chooses a sorted access for it, if possible (i.e. in a sorted source that did not return the object, nor reach its limit). If not possible, a random probe is selected following a fixed schedule, like for MPro. Limit scores are determined by sampling optimization and simulation.

Compared to NC, BR addresses a slightly different problem, by computing top-k objects possibly without complete scoring. Therefore necessary choices in the NC context are not relevant for BR. We do not compare in this paper BR with NC because, source sampling being not always possible, we only focus on fully heuristic-based top-k algorithms. Even if NC authors claim that sampling may replaced by estimation of the source limits, our tests indicate that NC is very sensible to the quality of this estimation. One should first define good limit estimation heuristics for NC, which is out of the scope of this paper.

3 The *BreadthRefine* algorithm

Unlike top-k algorithms presented in Section 2, the *BreadthRefine* (BR) algorithm covers any configuration of source access types and cost settings and proposes a new heuristic approach: refining the scores of all the top-k candidates, instead of focusing on the best one. We first present the BR data and query model, then the general BR algorithmic framework and several algorithm variants in this framework.

Data and query model We consider a set of data objects $\mathcal{O} = \{o_1, ..., o_n\}$, a top-k multi-criteria query q over these objects, and a set of web sources $\mathcal{S} = \{S_1, ..., S_m\}$ able to evaluate the query criteria.

Definition 1. ***Query:*** A top-k multi-criteria query q *is defined by (i) a number of objects* k *to return, (ii) a set of ranked predicates (criteria)* $P_q = \{p_1, ..., p_m\}$*, depending on the query, and (iii) a monotone score aggregation function* $\mathcal{F}$*.*

Predicates $p_j : \mathcal{O} \rightarrow [min_j, max_j]$ *return for any object a score in a given interval, while function* $\mathcal{F} : \mathbb{R}^m \rightarrow \mathbb{R}$ *aggregates predicate scores into a global object score. Each predicate* p_j *is independently evaluated by source* S_j*.*

Definition 2. ***Source:*** A source S_j *is characterized by (i) its access type (S, R or SR), and (ii) a cost per access, noted* $\mathrm{C_s(S_j)}$ *for sorted and* $\mathrm{C_r(S_j)}$ *for random access. The minimum and maximum scores in* S_j *are noted* min_j *and* max_j*, as mentioned in Definition 1.*

The set of sources $\mathcal{S}$ *can be partitioned following the access type in three disjoint subsets, possibly empty:* $\mathcal{S} = \mathcal{S}_S \cup \mathcal{S}_R \cup \mathcal{S}_{SR}$*.*

A sorted source (of type S or SR) provides an access function

getNext *:* $\mathcal{S}_S \cup \mathcal{S}_{SR} \rightarrow \mathcal{O} \times \mathbb{R} \cup \{\mathrm{nil}\}$

returning the next couple (o, s), where o is the object with the next score in decreasing order and s its score (if exists), or the special value nil *otherwise.*

A random source (of type R or SR) provides an access function

getScore*:*$(\mathcal{S}_R \cup \mathcal{S}_{SR}) \times \mathcal{O} \rightarrow \mathbb{R}$

returning the score of the given object in the source.

We note score*(o,* S_j*) the score retrieved for object o in source* S_j*.*

We note crtmax_j *the largest score that source* S_j *could further return. For pure random sources* $S_j \in \mathcal{S}_R$*,* $crtmax_j = max_j$ *(constant). For sorted sources* $S_j \in \mathcal{S}_S \cup \mathcal{S}_{SR}$*,* crtmax_j *is the score returned by the last sorted access to* S_j[1] *(initially* $crtmax_j = max_j$*).*

Definition 3. ***Candidates:*** A candidate *in the algorithm is an object that has been already returned by a sorted access. For each candidate c, the algorithm maintains* $L(c)$ *(*$U(c)$*), i.e. the lower (upper) bound of the global score for c.* $L(c)$ *(*$U(c)$*) is computed by aggregating the scores of c in the sources where it has been already consulted, and the minimum (maximum) score in the other sources.*

$L(c) = \mathcal{F}(l_1, ..., l_m)$*,* l_j =score*(c,* S_j*) if c consulted in* S_j*, else* $l_j = min_j$

$U(c) = \mathcal{F}(u_1, ..., u_m)$*,* u_j =score*(c,* S_j*) if c consulted in* S_j*, else* $u_j = crtmax_j$*.*

We note L_k *(*U_k*) the k-th value in decreasing order for* $L(c)$ *(*$U(c)$*) among the candidates - for less than k candidates, the value is* nil*.*

A candidate is called viable *if it still has chances to belong to the final top-k result. The viability condition for c is* $U(c) \geq L_k$*. It is simple to prove using monotonicity that once a candidate becomes non-viable, it will remain non-viable and could be removed.*

We note U_{unseen} *the upper bound of the global score for any object that is not yet a candidate. Initially,* $U_{unseen} = \mathcal{F}(max_1, ..., max_m)$

The BR algorithm framework The basic idea of the BR algorithm is to maintain the top-k candidates as a whole instead of looking only at the best candidate. Also, BR is able to handle any kind of source access; the choice of the access type is the central issue at each step.

Figure 2 presents the general BR framework, from which various algorithm variants may be instantiated. BR maintains a set of candidates, initially empty, and the maximum score of unseen objects, U_{unseen}.

[1] We consider that an object retrieved through a sorted access in a SR-source is not further accessed by a random access in the same source.

Framework $BR(q, \mathcal{S})$
  candidates $\leftarrow \emptyset$
  $U_{unseen} \leftarrow \mathcal{F}(max_1, ..., max_m)$
  repeat
    //choice between sorted or random access
    if $|candidates| < k$ **or** $U_k < U_{unseen}$ **or** ***CostCondition()*** **then**
      $S_j \leftarrow$ ***BestSortedSource(*** $\mathcal{S}$ ***)*** *//choice of a sorted source*
      $(o, s) \leftarrow getNext(S_j)$ *//sorted access to the selected source*
      Update *candidates* and U_{unseen}
    else *//random access*
      $c \leftarrow$ ***ChooseCandidate(*** **candidates,** k ***)*** *//choice of a top-k candidate*
      $S_j \leftarrow$ ***BestRandomSource(*** $\mathcal{S}$**,** c ***)*** *//choice of a random source*
      $s \leftarrow getScore(S_j, c)$ *//random access to the selected source*
      Update *candidates*
    endif
  until $|candidates| = k$ and $L_k \geq U_{unseen}$
  return *candidates*

Fig. 2. The BR algorithm framework

At each step, BR first chooses the type of access to be performed. Sorted access is preferred in the following cases:

- A group of top-k candidates does not exist yet, i.e. if $|candidates| < k$.
- An unseen object could belong to the current upper bound top-k group, i.e. $U_k < U_{unseen}$. A sorted access will decrease U_{unseen} and eliminate unseen objects from the top-k group.
- If the cost-related condition *CostCondition* is true. This condition enables BR to adapt to various cost settings, e.g. to force sorted accesses when random probes are expensive.

If a sorted access is decided, the *BestSortedSource* function chooses the best sorted source, then performs the sorted access[2]. Consequently, the candidate set and U_{unseen} are updated; the update of the candidate set consists in several actions:

- Add the object to the candidate set, if it is seen for the first time.
- Update the upper and lower bounds for all the candidates. In fact, besides the retrieved object, the update only affects the upper bound of candidates that have not been retrieved yet in the source.
- Remove non-viable candidates.

In the case of a random access, the *ChooseCandidate* function chooses a candidate in the current top-k group. Then a random source is selected with *BestRandomSource*, among those not yet probed for the candidate. The source is probed and the candidate set is updated.

The algorithm ends when the candidate set is reduced to k objects (after removing non-viable candidates) and when the unseen objects cannot change anymore the final result ($L_k \geq U_{unseen}$). The result is the set of k candidates, with possibly incomplete scores.

BR algorithm variants By instantiating *CostCondition*, *BestSortedSource*, *ChooseCandidate* and *BestRandomSource* functions, one may obtain various BR algorithms. We present three variants: *BR-Cost*, *BR-Basic* and *BR-First*.

[2] We consider that *BestSortedSource* does not return a source with no more objects.

BR-Cost is the reference BR algorithm that we propose. It refines the set of top-k candidates in a breadth-first manner and adapts to various cost settings. *BR-Cost* uses weighted sum aggregation function and employs source selection methods similar to existing algorithms.

- *ChooseCandidate* selects the least refined candidate (with the least random probes) from the current upper bound top-k group.
- *CostCondition* aims at reducing the number of random accesses (if more expensive than sorted ones), in a way similar to the CA algorithm. More precisely, if r is the average ratio between the cost of random and sorted accesses, then once a random probe is processed, the next one is possible only after at least r sorted accesses.
- *BestSortedSource* selects the sorted source S_j with respect to a benefit similar to the one proposed by StreamCombine [6], i.e. $B_j = coef_j N_j \delta_j / C_s(S_j)$, where $coef_j$ is the weight of S_j in the aggregation function, N_j the number of top-k candidates not yet seen in S_j, δ_j the expected decrease of the score in S_j and $C_s(S_j)$ the access cost. Here N_j measures the number of top-k objects that will be concerned by the decrease of the upper bound, and $coef_j$ and δ_j the amount of this decrease.
- *BestRandomSource* selects the random source with respect to a benefit $B_j = coef_j \times (crtmax_j - min_j)/C_r(S_j)$. Here $coef_j$ and $crtmax_j - min_j$ measure the variation of the candidate's upper/lower bound when the real score will replace the upper/lower source score.

BR-Basic is a variant of *BR-Cost* which does adapt to various cost settings, i.e. *CostCondition* systematically returns *false*. Comparing BR-Basic with BR-Cost will reveal the importance of cost adaptation.

BR-First is a variant of *BR-Basic* that uses the classical approach for choosing a candidate to refine, i.e. always select the best one - the highest upper bound in this case. Note that this is still a particular case of the general BR heuristics, the best candidate belonging to the top-k. Comparing *BR-First* with *BR-Basic* will measure the benefit of the new heuristics.

4 Experiments

In this section we report the experimental evaluation of the BR algorithms over synthetic data. We evaluate genericity and adaptivity by comparing BR with state of the art algorithms in their specific access type and cost configurations. We also measure the importance of the BR heuristics and of cost adaptivity by comparing the BR algorithms variants.

Data sets We generate synthetic score lists for each source, each list representing the predicate scores for a given query, as values in the [0,1] interval. Sources are independent and have similar data distribution. The sorted access cost C_s is the same for any source, idem for the random cost C_r. We consider three variants of data distribution:

- *Uniform*: values are uniformly distributed.
- *Gaussian*: values are generated from three overlapping Gaussian bells.
- *Zipfian*: values are generated from a Zipf function with 1000 distinct values and Zipfian parameter $z = 1$.

Parameters and default settings All the experiments measure *the execution cost*, which is the cost of all the source accesses performed during execution. More precisely, if Ns_j (Nr_j) is the number of sorted (random) accesses to source S_j, then the cost of the algorithm is:

$$cost = \sum_{S_j \in \mathcal{S}_S \cup \mathcal{S}_{SR}} Ns_j C_s(S_j) + \sum_{S_j \in \mathcal{S}_R \cup \mathcal{S}_{SR}} Nr_j C_r(S_j) \quad (1)$$

Each result is the average of 8 measures over different randomly generated data sources. We consider various parameters in the experiments:

- The number of objects: 10000 (default), 20000, 40000, 60000, 80000, 100000.
- The number k of returned objects: 20, 40, 50 (default), 60, 80, 100.
- The number of S-sources N_s, of R-sources N_r and of SR-sources N_{sr} (default value for each one: 3).
- The cost setting: $C_r(=5) > C_s(=1)$ (default), $C_r = C_s(=1)$, $C_r(=1) < C_s(=5)$
- Data distribution for all the sources: uniform (default), gaussian, zipfian.

Algorithms tested Experiments are grouped by source access type in three categories, in each case the set of algorithms adapted to this setting is considered:

- No R-sources: NRA and MPro.
- No S-sources: Upper, TAz and MPro.
- All the source types exist: MPro.

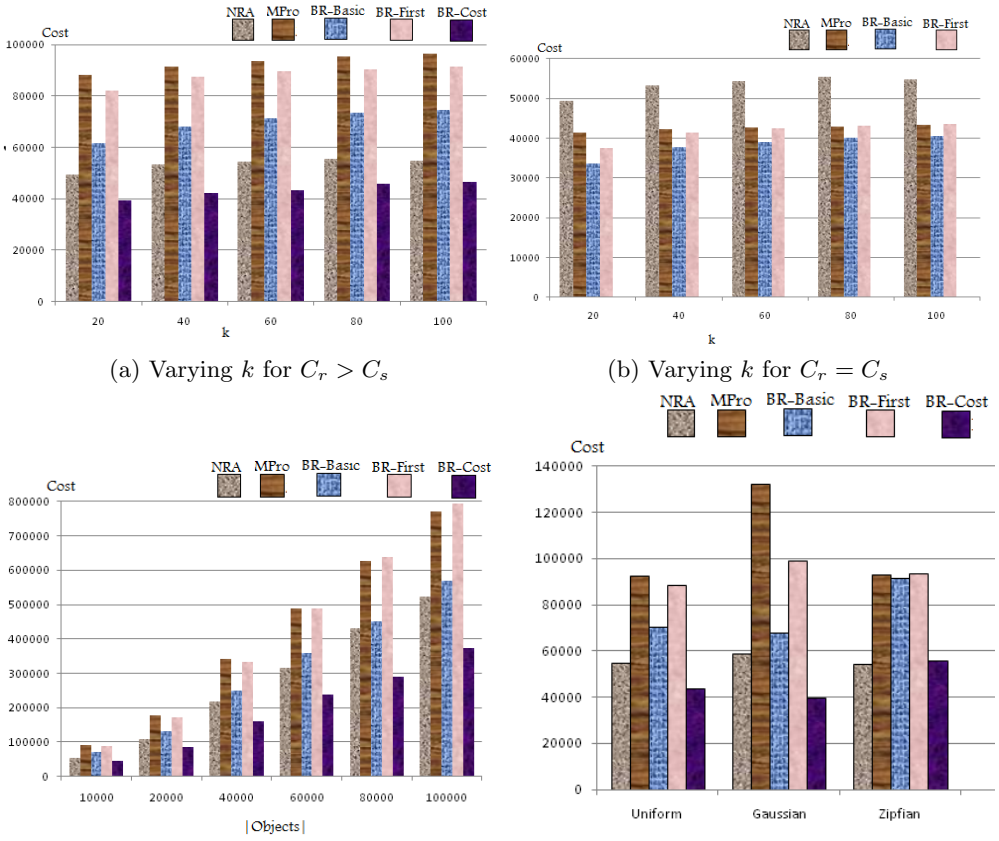
We adapted the MPro algorithm to cover all the cases as follows: sorted sources are combined with NRA to behave as a single S-source. SR-sources may be considered by MPro either as sorted or random, depending on the context: sorted when there are no S-sources, random when there are no R-sources. In the case of all the source types, we considered both settings (SR as sorted and SR as random) and reported the average cost. Also for random probes we considered a fully heuristic variant for MPro, in which scheduling uses the same order based on benefit as BR.

SR-sources + S-sources We compare BR-Cost, BR-Basic and BR-First with NRA and MPro. We first vary k in two cases: $C_r > C_s$ (default setting) and $C_r = C_s$. The case $C_r < C_s$ is not favorable to NRA and is considered later for MPro.

Figure 3(a) presents results for $C_r > C_s$. Cost increases with k for all the algorithms, but BR-Cost is significantly less expensive than MPro and even than NRA, which does no random access. The importance of considering cost settings in BR is obvious when comparing BR-Cost with BR-Basic. Also the benefit of the BR heuristic is confirmed by BR-First which behaves worse than BR-Basic. Figure 3(b) considers the case $C_r = C_s$, where BR-Cost is the same as BR-Basic. BR-Basic/Cost is still the best, but the difference with BR-First and MPro decreases.

Next, we vary the number of objects for the default cost settings. Figure 3(c) shows that the cost increases almost proportionally for all the algorithms, so the above conclusions are unchanged.

Figure 3(d) illustrates the variation of data distribution. For uniform and gaussian distributions, BR-Cost is the best algorithm and BR-Basic outperforms BR-First. Zipfian distribution, with many identical values, is a special case, but BR-Cost still have good results. BR-Cost and NRA have almost the same cost, while differences between MPro and BR variants become insignificant.



(a) Varying k for $C_r > C_s$

(b) Varying k for $C_r = C_s$

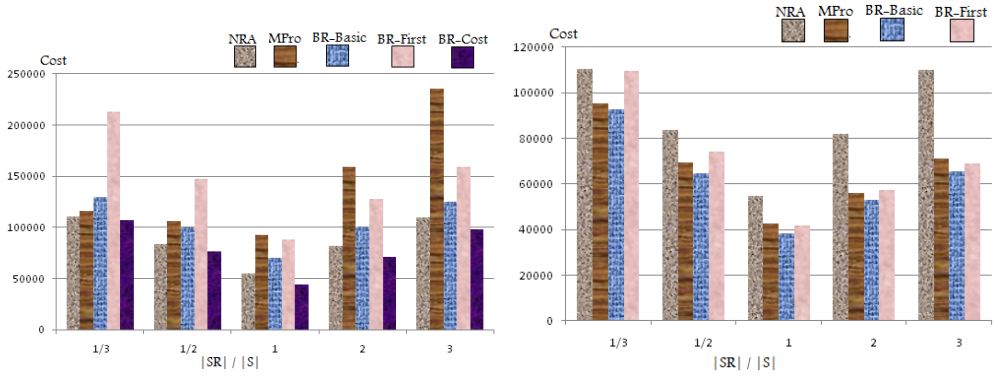
(c) Varying the number of objects

(d) Varying the distribution

Fig. 3. SR + S-sources: varying k, the number of objects and the distribution

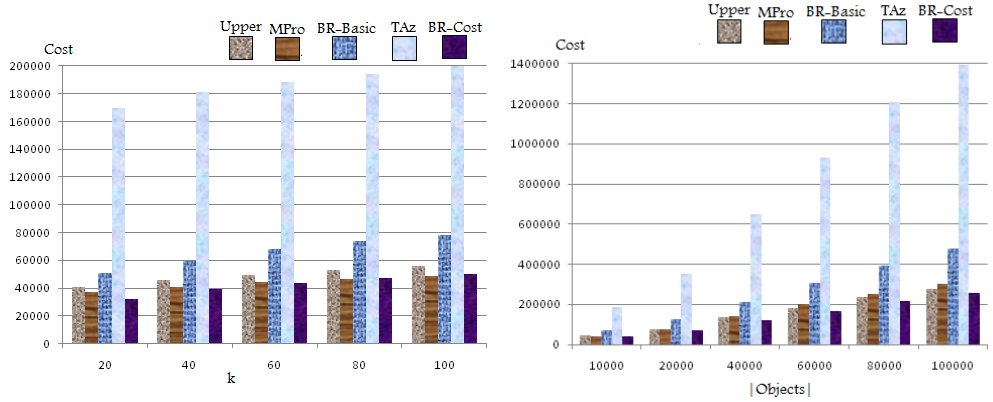
Finally, we study the impact of varying the ratio between the number of SR-sources and of S-sources. Figure 4 reports the results for the default setting (a) and for $C_r = C_s$ (b). Interestingly, in both cases algorithms have best results when |SR| = |S|. In the default setting, in all cases, BR-Cost has better cost, but it substantially outperforms the other algorithms when |SR| > |S|. Also BR-Basic is always significantly better than BR-First, the difference increases when |SR| < |S|. When $C_r = C_s$, BR-Basic/Cost, BR-First and MPro keep the same relative performances, while NRA behave worse when |SR| > |S|.

SR-Sources + R-Sources We compare BR with Upper, TAz and MPro. For space reasons, we only illustrate BR-Cost and BR-Basic, knowing that measures indicate that BR-First remains worse than BR-Basic. Results are presented in Figure 5. TAz is systematically outperformed by all the other algorithms and BR-Basic by BR-First. BR-Cost has globally the best results, even if Upper and MPro are close. Upper is slightly worse than MPro in the default setting, but scales better than MPro when the number of objects grows. Also Upper degrades when |SR| > |R|.

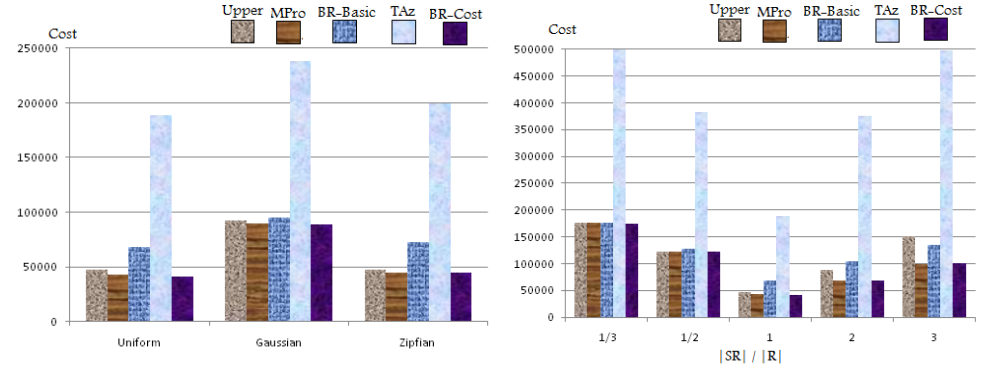


(a) $C_r > C_s$ (b) $C_r = C_s$

Fig. 4. SR + S-sources: varying |SR|/|S|



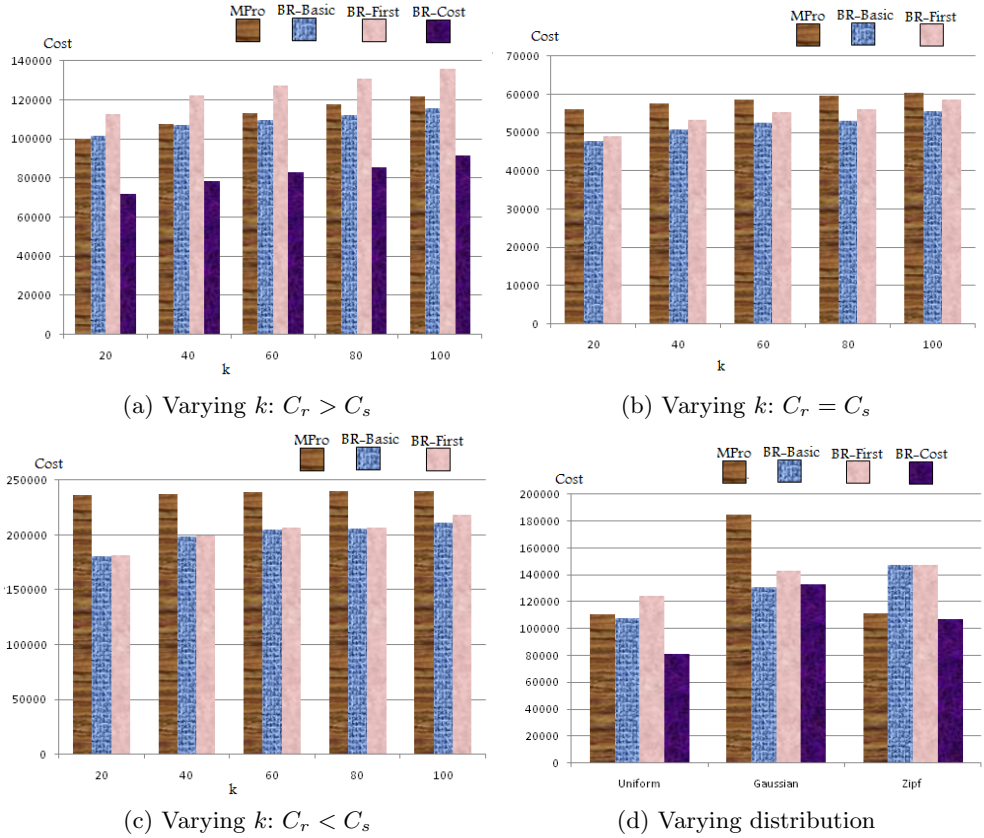
(a) Varying k (b) Varying the number of objects



(c) Varying distribution (d) Varying |SR|/|R|

Fig. 5. SR + R-sources

SR-Sources + S-Sources + R-Sources We compare BR-Cost, BR-Basic and BR-First with MPro, by first varying k for all the cost settings. Figure 6(a) reports the default case $C_r > C_s$. BR-Cost outperforms MPro, while BR-Basic remains better than BR-First and similar to MPro. When $C_r = C_s$ (subfigure b), BR-Basic/Cost is better than both MPro and BR-First and the difference augments when $C_r < C_s$ (subfigure c). This seems to indicate that MPro makes more sorted accesses than BR-Basic (the difference increases when C_s grows) and probably less random accesses.



(a) Varying k: $C_r > C_s$

(b) Varying k: $C_r = C_s$

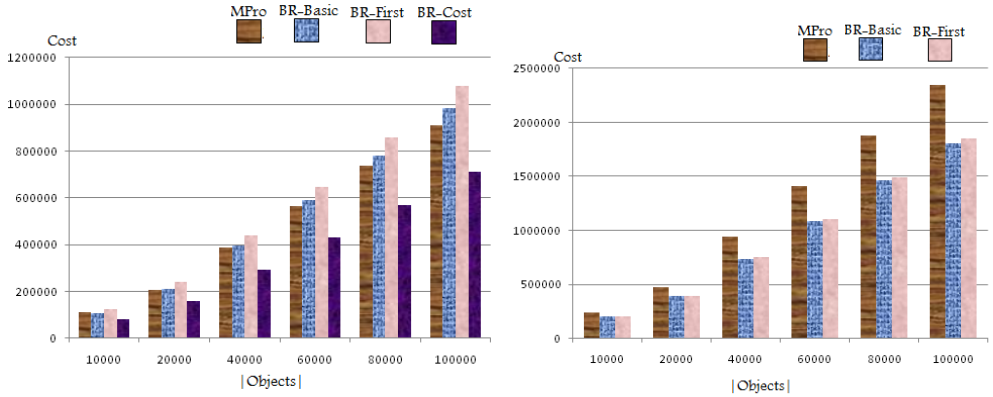
(c) Varying k: $C_r < C_s$

(d) Varying distribution

Fig. 6. SR + S + R-sources: varying k and the distribution

Figure 6 (d) illustrates the impact of data distribution: BR-Cost remains globally the best, while BR-First is the worse BR variant. There is little difference between BR variants for gaussian distribution, while zipfian favors MPro compared to BR-Basic and BR-First.

Figure 7 reports the impact of the number of objects for $C_r > C_s$ (a) and $C_r < C_s$ (b). In both cases, the cost augments almost proportionally for the BR variants and BR-Cost is always better than the other algorithms. We remark that MPro scales better than BR-Basic and BR-First when $C_r > C_s$ and significantly worse when $C_r < C_s$, indicating that MPro makes slightly less random accesses, but much more sorted accesses than BR-Basic.



(a) $C_r > C_s$ (b) $C_r < C_s$

Fig. 7. SR + S + R-sources: varying the number of objects

5 Conclusion

In this paper we proposed *BreadthRefine* (BR), a generic top-k algorithm, able to adapt to any combination of source access types and to any cost settings. It adopts a new heuristic approach, by refining the scores of all the top-k candidates instead of focusing on the best one. Experiments on synthetic data clearly indicate that BR successfully adapts to various settings, with globally better execution cost than algorithms designed for that specific case. Comparison with the classical approach of favoring the best candidate shows that BR's breadth-first heuristics produces better results.

Future work will focus on the advantages of breadth-first heuristics in approximating the top-k final results.

References

1. R. Akbarinia, E. Pacitti, and P. Valduriez. Best position algorithms for top-k queries. In *VLDB*, pages 495–506, 2007.
2. N. Bruno, L. Gravano, and A. Marian. Evaluating top-k queries over web-accessible databases. In *ICDE*, pages 369–, 2002.
3. K. C.-C. Chang and S. won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD Conference*, pages 346–357, 2002.
4. R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.
5. U. Güntzer, W.-T. Balke, and W. Kießling. Optimizing multi-feature queries for image databases. In *VLDB*, pages 419–428, 2000.
6. U. Güntzer, W.-T. Balke, and W. Kießling. Towards efficient multi-feature queries in heterogeneous environments. In *ITCC*, pages 622–628, 2001.
7. I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. *VLDB J.*, 13(3):207–221, 2004.
8. I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.
9. C. Li, K. C.-C. Chang, and I. F. Ilyas. Supporting ad-hoc ranking aggregates. In *SIGMOD Conference*, pages 61–72, 2006.

10. C. Li, K. C.-C. Chang, I. F. Ilyas, and S. Song. Ranksql: Query algebra and optimization for relational top-k queries. In *SIGMOD Conference*, pages 131–142, 2005.
11. N. Mamoulis, K. H. Cheng, M. L. Yiu, and D. W. Cheung. Efficient aggregation of ranked inputs. In *ICDE*, page 72, 2006.
12. A. Marian, N. Bruno, and L. Gravano. Evaluating top-k queries over web-accessible databases. *ACM Trans. Database Syst.*, 29(2):319–362, 2004.
13. A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. In *VLDB*, pages 281–290, 2001.
14. S. won Hwang and K. C.-C. Chang. Optimizing top-k queries for middleware access: A unified cost-based approach. *ACM Trans. Database Syst.*, 32(1):5, 2007.