# Fast Filtering of Search Results Sorted by Attribute

FRANCO MARIA NARDINI, ISTI-CNR, Pisa, Italy
ROBERTO TRANI, ISTI-CNR, Pisa, Italy
ROSSANO VENTURINI, University of Pisa, Italy

Modern search services often provide multiple options to rank the search results, e.g., sort "by relevance", "by price" or "by discount" in e-commerce. While the traditional rank by relevance effectively places the relevant results in the top positions of the results list, the rank by attribute could place many marginally relevant results in the head of the results list leading to poor user experience. In the past, this issue has been addressed by investigating the relevance-aware filtering problem, which asks to select the subset of results maximizing the relevance of the attribute-sorted list. Recently, an exact algorithm has been proposed to solve this problem optimally. However, the high computational cost of the algorithm makes it impractical for the Web search scenario, which is characterized by huge lists of results and strict time constraints. For this reason, the problem is often solved using efficient yet inaccurate heuristic algorithms. In this paper, we first prove the performance bounds of the existing heuristics. We then propose two efficient and effective algorithms to solve the relevance-aware filtering problem. First, we propose OPT-Filtering, a novel exact algorithm that is faster than the existing state-of-the-art optimal algorithm. Second, we propose an approximate and even more efficient algorithm, $\epsilon$-Filtering, which, given an allowed approximation error $\epsilon$, finds a (1-$\epsilon$)–optimal filtering, i.e., the relevance of its solution is at least (1-$\epsilon$) times the optimum. We conduct a comprehensive evaluation of the two proposed algorithms against state-of-the-art competitors on two real-world public datasets. Experimental results show that OPT-Filtering achieves a significant speedup of up to two orders of magnitude with respect to the existing optimal solution, while $\epsilon$-Filtering further improves this result by trading effectiveness for efficiency. In particular, experiments show that $\epsilon$-Filtering can achieve quasi-optimal solutions while being faster than all state-of-the-art competitors in most of the tested configurations.

CCS Concepts: • **Information systems** → **Retrieval tasks and goals**.

Additional Key Words and Phrases: Relevance-aware Filtering, Filtering Algorithms, Approximation Algorithms, Efficiency-Effectiveness trade-offs

**ACM Reference Format:**
Franco Maria Nardini, Roberto Trani, and Rossano Venturini. 2020. Fast Filtering of Search Results Sorted by Attribute. *ACM Transactions on Information Systems* 0, 0, Article 0 ( 2020), 24 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Search engine results are traditionally ranked by relevance to maximize the user experience. A classic example is Web search: a user formulates her information needs through a textual query and the Web search engine answers with the list of search results that maximizes the relevance with respect to the user query [2]. To provide more flexibility to the users, many modern search engines also provide additional options to rank the search results. For instance, social networks

Authors' addresses: Franco Maria Nardini, ISTI-CNR, Pisa, Italy, francomaria.nardini@isti.cnr.it; Roberto Trani, ISTI-CNR, Pisa, Italy, roberto.trani@isti.cnr.it; Rossano Venturini, University of Pisa, Italy, rossano.venturini@unipi.it.

1046-8188/2020/0-ART0 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

provide a “sort by time” of the stories posted by friends [13], while e-commerce platforms allow to “sort by price” the items matching a specific user query [8]. However, this flexibility comes at a cost. The top positions of the list sorted by attribute may contain results that are marginally relevant with respect to the information need of the user [21, 22], leading thus to a poor user experience. A simple example based on a real e-commerce motivates this observation. *Amazon.com* returns more than 100 000 results for the query “iPhone”. When results are sorted by relevance (default), the first page of results consists of several relevant iPhone models and related accessories. However, when the results are sorted by increasing price, the first page of results is a list of low-price phone covers and gadgets. The latter results list leads to a poor user experience as the user now needs to examine tens of results before she finds something relevant.

The problem above can be addressed as a filtering task aiming to keep at most $k$ results of the list to maximize the relevance of the list while preserving the sort by attribute. In particular, given an attribute-sorted list of $n$ search results having relevance scores $R = \langle r_1, \ldots, r_n \rangle$, a positive integer $k$ and a search quality metric $Q$, the FILTERING@$k$ problem asks to find the sub-list of results of size at most $k$ that maximizes $Q$, i.e., a list of strictly increasing indices $I$ such that $I = \text{argmax}\, Q(\langle r_{I_1}, \ldots, r_{I_{|I|}} \rangle)$ and $|I| \leq k$.

Solving the FILTERING@$k$ problem is hard when employing complex quality metrics such as the Discounted Cumulative Gain (DCG) [9]. Indeed, to find an optimal solution, it is not sufficient neither to select the $k$ most relevant results of $R$ nor to consider only the subsequences of exactly $k$ results. To support the two considerations above, we provide the following example. Given an attribute-sorted list of four results, whose relevance labels are $\langle 2, 2, 4, 1 \rangle$, the best DCG@3 is obtained by filtering out the first two results of the list, although each of them is more relevant than the last one. Moreover, the example shows that all sub-sequences of three results achieve a lower DCG@3 than the optimal solution consisting of only two results, i.e., $\langle 4, 1 \rangle$.

The FILTERING@$k$ problem was recently introduced by Spirin *et al.* [21], who proposed an optimal algorithm, based on Dynamic Programming [6], that runs in $\Theta(nk)$ time. The algorithm, called $\text{OPT}_{\text{DYN}}$, has however two important drawbacks. First, its time complexity makes it impractical for search engines that need to handle several thousands of results per query within small time budgets, e.g., 100ms for 99th-percentile per-query response time in Web search [1, 10, 12]. Second, it cannot be applied in a distributed setting [3] as it takes global decisions based on the complete list of results. For these reasons, the filtering problem is often addressed with two heuristics based on thresholding: Cutoff and Top$k$. Cutoff selects the results whose relevance is greater than a given threshold, while Top$k$ selects the $k$ most relevant results of the list. Spirin *et al.* further combine Top$k$ and Cutoff with $\text{OPT}_{\text{DYN}}$ to trade effectiveness for efficiency and obtain two new heuristics, i.e., Top$k$-OPT and Cutoff-OPT. The FILTERING@$k$ problem is thus solved by using either exact but slow algorithms or fast but inaccurate heuristics.

In this paper, we provide an analysis of the weaknesses of the above approaches, and we propose two novel efficient yet effective solutions to the FILTERING@$k$ problem exploiting different combinatorial properties of discounted cumulative gain based metrics, e.g., DCG [9] and RBP [16]. In particular, we extend a previous result by Nardini *et al.* [17] with the following contributions:

- we theoretically analyze the Cutoff-OPT and Top$k$-OPT heuristics in the filtering scenario [21]. We show that Cutoff-OPT does not provide any performance guarantee while Top$k$-OPT finds a 0.5−optimal filtering in $\Theta(n \log k + k^2)$ time.
- we present OPT-Filtering, a novel efficient exact algorithm that finds an optimal filtering in $\Theta\left(n \log k + \min(n, 2^k)\; k\right)$ time. The new algorithm has lower theoretically time complexity than the state-of-the art exact algorithm, i.e., $\text{OPT}_{\text{DYN}}$, and is much faster in practice.

- we introduce $\epsilon$-Filtering, a novel efficient approximate algorithm that trades-off between efficiency and effectiveness through a parameter $\epsilon$ that controls the approximation error. $\epsilon$-Filtering finds a (1-$\epsilon$)–optimal filtering in $\Theta\big(n + k^2 \log_{1\text{-}\epsilon}(\epsilon/k)\big)$ time.
- we provide a comprehensive experimental evaluation of OPT-Filtering and $\epsilon$-Filtering against state-of-the-art competitors on two real-world public datasets. Results show that our two algorithms achieve a speedup of up to two orders of magnitude w.r.t. the state-of-the-art competitor introduced by Spirin *et al.*, i.e., $\text{OPT}_{\text{DYN}}$. The experimental results also show that $\epsilon$-Filtering is faster than the two heuristics, i.e., Top$k$-OPT and Cutoff-OPT, while guaranteeing very small approximation errors.

The rest of the paper is organized as follows. Section 2 examines the related work, while Section 3 presents a theoretical analysis of the Cutoff-OPT and Top$k$-OPT heuristics. Sections 4 and 5 present our OPT-Filtering and $\epsilon$-Filtering algorithms. Section 6 provides a comprehensive assessment of all filtering algorithms on two real-world datasets. Lastly, Section 7 concludes the work.

## 2 RELATED WORK

The relevance-aware filtering problem can be solved with optimal but slow algorithms based on dynamic programming paradigm or fast but inaccurate heuristics. The goal of this section is to present known solutions for this problem.

**Optimal filtering**. Spirin *et al.* [21] design an optimal algorithm, $\text{OPT}_{\text{DYN}}$, to solve the relevance-aware filtering problem. This algorithm computes an optimal solution in $\Theta(nk)$ time employing a dynamic programming paradigm [6]. Given a results list $R$ and a target number $k$, $\text{OPT}_{\text{DYN}}$ iterates over the prefixes of $R$ and incrementally fills a memoization matrix $M_{n\times k}$ to obtain optimal solutions of any length $j \leq k$. Each entry $M[i, j]$ of the table stores the best relevance score achievable on the prefix of $R$ of length $i$ by selecting exactly $j$ elements. Clearly, the score of the empty prefix of $R$ is 0, hence $M[0, \cdot] = 0$, and the score of selecting no items is 0, hence $M[\cdot, 0] = 0$. At each step, $\text{OPT}_{\text{DYN}}$ could either *i*) appends the $i$-th element of $R$, $r_i$, to the optimal subsequence of length $j-1$ of the prefix $i-1$, or *ii*) takes the optimal subsequence of length $j$ of the prefix $i-1$. The algorithm chooses the alternative giving the largest score. Specifically, it fills the table using the following recursive definition

$$M[i, j] = \max \begin{cases} M[i-1, j] \\ M[i-1, j-1] + \texttt{score\_item}(r_i, j) \end{cases}$$

where $\texttt{score\_item}(r_i, j)$ is the gain of relevance score of the element $r_i$ when placed in position $j$, e.g., $(2^{r_i} - 1)/\log_2(j+1)$ for the DCG metric [9]. Clearly, the highest score in the last row of $M$ corresponds to the optimal solution. This solution can be reconstructed by tracing backs the choices made to get that score. The algorithm runs in $\Theta\,(nk)$ time, assuming that the score of an item can be computed in constant time.

**Heuristic filtering**. Cutoff and Top$k$ are two heuristics that can be employed to solve the relevance-aware filtering problem. Given a relevance threshold, the Cutoff heuristic selects the items whose relevance is at most the threshold. Instead, Top$k$ selects the $k$ most relevant items of the list. Spirin *et al.* experimentally showed that these two heuristics are more effective when they are used as a pre-filter for $\text{OPT}_{\text{DYN}}$, i.e., their results are further filtered by $\text{OPT}_{\text{DYN}}$ [21]. We call Cutoff-OPT and Top$k$-OPT the resulting approaches. They also present a comprehensive evaluation on Learning-to-Rank datasets [14, 20] that shows a better performance of the optimal solution with respect to Cutoff-OPT and Top$k$-OPT. However, they do not provide any theoretical analysis of these two heuristics, which are, instead, analysed in this paper. We first show that Cutoff-OPT does not provide any performance guarantee. We then prove that the effectiveness achieved by Top$k$-OPT may be up to two times worse than the optimum. Therefore, their application may lead to significant degradation of the relevance of the filtered list.

**Related problems**. Carmel *et al.* explore a similar problem in mail search, where the standard sort by time of the search results may negatively affect the user experience [4, 5]. To this end, the authors investigate a mixed approach promoting the most relevant results on top of time-ranked results. Experimental results on Yahoo! Mail show that supplementing time-sorted results with relevant results leads to better performance than the traditional time-sorted view. This proposal by Carmel *et al.* is only partially related to ours. While they investigate how to couple relevance ranking with time-based ranking to enable a two-dimensional view in mail search, we propose to directly address the relevance-aware filtering of result lists by removing irrelevant results from an attribute-ordered list of results. Trotman *et al.* present a new information retrieval task called high accuracy recall task and suggest the metrics to measure the quality of the results in this setting [22]. The task is to identify as many relevant documents, and as few non-relevant documents as possible, such that the precision remains high regardless of the order. The high accuracy recall task is only partially related to FILTERING@$k$. Indeed, it is agnostic with respect to the ordering of the results, while our work aims to maximize a target list-wise metric that takes into account the order of presentation of the results. Recently, Vorobev *et al.* investigate the FILTERING@$k$ problem from a machine learning point of view [23]. In particular, the authors propose new machine learning algorithms to learn to select a set of items maximizing the quality of the attribute-ordered list, which are based on the direct optimization of the resulting list quality. The work by Vorobev *et al.* is orthogonal to our one. While they investigate new machine learning algorithms that estimate the scores used to select the top-$k$ results, taking into account the attribute order, we propose two algorithmic solutions that filter the list of results starting from the estimated relevance scores.

## 3 HEURISTIC FILTERING

The goal of this section is to analyze the two aforementioned heuristics Cutoff-OPT and Top$k$-OPT. We show that Cutoff-OPT may be arbitrarily worse than any optimal algorithm OPT, while Top$k$-OPT is 0.5−optimal, i.e., it may find a solution whose relevance score is 0.5 times the score of the optimal solution. We present the results by using DCG [9] as search quality metric $Q$.

### 3.1 Cutoff-OPT

Cutoff-OPT applies $\text{OPT}_{\text{DYN}}$ to the elements of the list whose relevance is greater than a given threshold. Cutoff-OPT does not provide any performance guarantees. Indeed, its performance strictly depends on the fixed threshold: the higher the threshold, the more efficient and less accurate the filtering as an increasing number of elements is pruned from the list. It is thus trivial to find worst-case lists for Cutoff-OPT, where either *i*) the algorithm is not able to filter any element, or *ii*) it filters all of them. In the former case, Cutoff-OPT does not improve the time complexity of $\text{OPT}_{\text{DYN}}$, i.e., it runs in $\Theta(nk)$ time, while its solution is arbitrarily worse than the optimal one in the latter case.

### 3.2 Top$k$-OPT

Top$k$-OPT provides stronger guarantees than Cutoff-OPT. First, Top$k$-OPT runs in $\Theta\big(n \log k + k^2\big)$ time in the worst case. Indeed, it selects the top-$k$ elements in $\Theta\big(n \log k\big)$ time and filters them in $\Theta\big(k^2\big)$ time by using $\text{OPT}_{\text{DYN}}$. Then, as far as the quality of its solution is concerned, we prove that *i*) there exist an infinite family of lists for which the solution provided by Top$k$-OPT is roughly two times worse than the optimum, *ii*) this is the worst case, i.e., the approximation factor is at most 0.5.

The first point is proven by using the following family of lists. We have a list $R_k$ for each possible value of the parameter $k$ of the FILTERING@$k$ problem. The list $R_k = \langle m', \dots, m', 1, m, \dots, m\rangle$ of length $2k-1$ is made of two subsequences of equal elements of length $k-1$ separated by 1. The value of $m'$ is $(1-d(k))/\sum_{i=1}^{k-1} d(i)$, where $d(i)$ is the discount factor of the DCG metric, i.e., $1/\log_2(i+1)$. Instead, $m$ is smaller than $m'$ but infinitely close to it. For any value of the parameter $k$, Top$k$-OPT selects the first $k$ elements of $R_k$, while any optimal algorithm OPT takes its last $k$ elements. The scores achieved by the two strategies are 1 and $\left(2 - d(k) + (1 - 2d(k) + d(k)^2)/\sum_{i=1}^{k-1} d(i)\right)$, respectively. For increasing values of $k$, the score of OPT tends to be 2 times better than the score of Top$k$-OPT, i.e., Top$k$-OPT is 0.5–optimal. For example, for $k = 20$, OPT is roughly 1.7 times better than Top$k$-OPT.

As far as the second point above is concerned, we need to prove that the worst approximation factor of Top$k$-OPT is at most 0.5 as stated by the following theorem.

THEOREM 3.1. *Top$k$-OPT is* 0.5*–optimal.*

PROOF. Let $M$ be the largest relevance in $R$. We start by showing that every optimal solution must contain an element of relevance $M$. Let $O$ be an optimal solution that does not contain $M$. Let $o_p$ be the element of $O$ closest to the element having relevance $M$ in $R$. The solution $O'$ obtained from $O$ by substituting $o_p$ with $M$ is still a sub-list of $R$ and has score

$$\mathrm{DCG}(O') = \mathrm{DCG}(O) + \left(2^M - 2^{o_p}\right) d(p),$$

which is greater than the relevance of $O$ because $M > o_p$, thus contradicting the hypothesis that $O$ is optimal.

Let $O$ and $O'$ be the solutions of OPT and Top$k$-OPT, respectively. As a consequence of the previous fact, both $O$ and $O'$ contain an element having relevance $M$. Moreover, their relevance score cannot be smaller than the score of the list containing only $M$. As we are interested in the worst-case scenario, we focus on the case where $O$ and $O'$ have the largest difference in the score. $O$ and $O'$ differ the most when they share the least number of results. In particular, it holds when all elements of $O$ but $M$ are not in $O'$, i.e., $O$ contains no one of the Top$k$ elements but $M$.

Let $m$ be the second largest value of $O$ and $m'$ be the smallest value of $O'$. Since the discount $d(\cdot)$ is a monotonically decreasing function, the solution $O$ having the maximum score is the one with all the elements in decreasing order. In particular, $O$ achieves the maximum score when all elements but $M$ are equal to $m$, i.e., $O = \langle M, m, ..., m\rangle$. For the same reason, the solution $O'$ having the minimum score is the one with all elements but $M$ equal to $m'$ and placed on the left of $M$, i.e., $O' = \langle m', ..., m', M\rangle$. For instance, this is the case of the list $R = \langle m', ..., m', M, m, ..., m\rangle$ whose two sublists of identical elements contain $k$-1 elements each.

Note that $O$ obtains the maximum score when $m$ is the maximum value not selected by Top$k$, thus when $m$ is smaller than but infinitely close to $m'$. Without loss of generality, we can assume $m = m'$ to simplify the discussion. Therefore, the scores of $O$ and $O'$, expressed as a function of $m$ and $k$, are respectively

$$Q(O) = g(M)\, d(1) + \sum_{p=2}^{k} g(m)\, d(p)\,, \text{ and}$$

$$Q(O') = \max \begin{cases} g(M)\, d(1) \\ \sum_{p=1}^{k-1} g(m)\, d(p) + g(M)\, d(k) \end{cases}$$

where $g(r)$ is the gain factor of the DCG metric, i.e., $2^r - 1$.

By studying the derivative of the approximation function with respect to $m$, i.e., $\frac{\partial}{\partial m}\left(Q(O')/Q(O)\right)$, we can find that the worst approximation error is reached when the two terms of the max function are equal, so when $Q(O') = g(M)\, d(1)$. Hence, in the worst case, $m$ is such that

$$g(m) = \frac{g(M)\,(d(1) - d(k))}{\sum_{p=1}^{k-1} d(p)}.$$

Finally, by substituting the value above into the score of $O$ and by replacing $d(1)$ with its real value, i.e., 1, we have

$$Q(O) = g(M)\left(2 - d(k) + \frac{1 - 2d(k) + d(k)^2}{\sum_{p=1}^{k-1} d(p)}\right).$$

Therefore, the score of $O$ tends to $2g(M)$ for $k$ tending to infinity and, since the score of $O'$ is exactly $g(M)$, in the worst-case scenario Top$k$-OPT is 0.5−optimal. □

Finally, it is worth highlighting that we could not improve the above weak guarantees by increasing the number of elements selected by the Top$k$ pre-filtering in Top$k$-OPT. Indeed, we can easily change each list $R_k$ to match the same worst-case scenario and, thus, proving the following corollary of Theorem 3.1.

Corollary 3.2. *The solution obtained by applying OPT to the top $k'$ elements of $R$, with any $k' \geq k$, is* 0.5*−optimal.*

Proof. Let $R$ be the adversarial list used in the proof of Theorem 3.1 extended on the left with many identical elements, i.e., $R = \langle m', ..., m', M, m, ..., m \rangle$. The first run of identical elements of $R$ contains $k' - 1$ values, while the second run contains $k - 1$ elements. The Top$k'$ pre-filtering selects the first $k'$ elements of $R$, then the OPT filtering selects the singleton element $M$, whose score is $g(M)$ and is optimal if we consider only these elements. Instead, the optimal filtering of the entire list $R$ is composed of the last $k$ elements, whose score tends to $2 \cdot g(M)$ for $k$ that tends to infinity, as shown in the proof of Theorem 3.1. □

In summary, Top$k$-OPT reduces the time complexity of the filtering from $\Theta(nk)$ to $\Theta(n \log k + k^2)$ at the cost of a 0.5 approximation error in the worst case.

## 4 EXACT FILTERING

The efficiency issues of the exact filtering OPT$_{\text{DYN}}$ motivate us to study a more efficient exact solution to the Filtering@$k$ problem. In this section, we propose OPT-Filtering: a new exact filtering algorithm running in $\Theta\left(n \log k + \min(n, 2^k)k\right)$ time. It is very competitive with respect to OPT$_{\text{DYN}}$ when $k$ is smaller than the logarithm of $n$, e.g., when $k \leq 20$. In all other cases, it is never slower than OPT$_{\text{DYN}}$, namely, the two bounds coincide. Indeed, the asymptotic complexity of OPT$_{\text{DYN}}$ is $\Theta\,(nk)$ and both $(n \log k)$ and $\left(\min(n, 2^k)k\right)$ are lower or equal than $nk$ for all values of $n$ and $k$. OPT-Filtering is composed of two lossless pruning steps, i.e., left-pruning and right-pruning, followed by the exact yet slow OPT$_{\text{DYN}}$ filtering. The two pruning steps are designed to discard much of the elements and do not affect the optimality of the solution, thus guaranteeing the exactness of the solution. We start by examining the two pruning strategies separately, and then we analyze the OPT-Filtering algorithm.

### 4.1 Left-pruning

This lossless pruning prunes out any result whose preceding subsequence contains a decreasing skyline of at least $k$ elements with greater or equal relevance. A *decreasing skyline* of a list is a subsequence of the elements such that each result of the skyline is more or equally relevant than each other appearing after it in the list. As a consequence, each element of the skyline “dominates” all others appearing after it in the list. The idea behind this pruning is quite intuitive. Let $a$ be an element discarded by the left-pruning. Then, $a$ is preceded by a decreasing subsequence of at least $k$ elements more or equally relevant than itself. Thus, we can select one of them in place of $a$ while preserving the effectiveness of the solution. The example below shows this intuition.

Example 4.1. *Let $R$ be the list $\langle 1, \dots, 1, 0.9\rangle$ of 21 elements and $k = 20$. The element $a = 0.9$ is the only element discarded by the left pruning. Every solution that includes the element $a$ can be improved by replacing it with a non selected element on its left. Indeed, the solution of $k$ elements $\langle 1, \dots, 1, 0.9\rangle$ has DCG of about 6.99, while the solution $\langle 1, \dots, 1\rangle$ has DCG of about 7.*

Let *left-height* of an element $a$ be the number of elements having a greater or equal relevance than $a$ and constituting a decreasing skyline of the sequence preceding $a$. Given a solution $O$, we say that an element having left-height $h$ is a *left-defect* of $O$ if it occurs in the first $h$ positions of $O$. The following lemma proves the intuition above.

Lemma 4.2. *There exists an optimal filtering that does not contain left-defects.*

Proof. By contradiction, let us assume that all existing optimal solutions contain at least one left-defect. Let $\hat{O}$ be any optimal solution having the lowest number of defects and whose leftmost defect is the closest to the first position of the solution. The contradiction arises by showing that starting from $\hat{O}$ we can build a new solution which is either more relevant, with fewer defects, or with the left defect closer to the first position of the solution.

Let $\hat{l}$ be the leftmost defect of $\hat{O}$, $h$ its left-height and $\hat{p}$ be its position in $\hat{O}$. By definition, $\hat{l}$ is "dominated" on the left – within $R$ – by a decreasing skyline of at least $h$ elements having relevance greater than or equal to $\hat{l}$ and having a smaller left-height. At least one of them is not in $\hat{O}$ because there are less than $h$ elements before $\hat{l}$ in $\hat{O}$. Let $l$ be the leftmost of them, and $p$ be the position within $\hat{O}$ where $l$ should be placed if selected. We differentiate between the following two cases.

*i*) All elements of $\hat{O}$ between $p$ and $\hat{p}$ are greater than or equal to $\hat{l}$. Let $O$ be the solution obtained from $\hat{O}$ by removing $\hat{l}$ and by inserting $l$. By definition, $l$ is greater than or equal to all elements between $p$ and $\hat{p}$. Thus, the solution $O$ has a score greater than or equal to the one of $\hat{O}$. Moreover, the elements shifted by one position on the right do not introduce new defects because they are moving away from the first position of the solution. If $l$ introduces a defect, $O$ has the same number of defects of $\hat{O}$ and has the leftmost defect closer to the first position than $\hat{O}$. Otherwise, if $l$ does not introduce a defect, $O$ has a lower number of defects than $\hat{O}$. Therefore, the solution $O$ contradicts the hypothesis.

*ii*) At least one element of $\hat{O}$ between $p$ and $\hat{p}$ is strictly smaller than $\hat{l}$. Let $\hat{l}'$ be the leftmost element with this property. Let $O$ be the solution obtained from $\hat{O}$ by removing $\hat{l}'$ and by inserting $l$. The solution $O$ has a score strictly greater than $\hat{O}$, thus contradicting the optimality of $\hat{O}$. □

Let *left-$k$-maximal* be any element whose left-height is smaller than $k$. The following lemma proves the correctness of the left-pruning, and it directly follows from Lemma 4.2 by showing that any non-left-$k$-maximal element causes a left-defect into the solution. Indeed, a non-left-$k$-maximal element has a left-height greater or equal than $k$, and any viable solution contains at most $k$ results.

Lemma 4.3. *There exists an optimal filtering composed of only left-$k$-maximal elements of $R$.*

The left-pruning can be performed in $\Theta(n)$ time by scanning the list left-to-right and by employing a stack to keep the decreasing subsequence of elements preceding the current element. At each step of the scanning, *i*) all elements smaller than the current element are popped out from the stack, and *ii*) the current element is pushed on top of the stack if the size of the stack is smaller than $k$. The current element is not left-$k$-maximal when it is not pushed into the stack, and its left-height is the position where it is placed into the stack. The stack contains the elements in decreasing order; thus whenever we compare the current element with the top of the stack we either pop an element (first operation), or push a new one (second operation). Since the number of elements is $n$, the number of push and pop operations is $\Theta(n)$.

The left-pruning, however, does not guarantee an effective reduction of the elements surviving the pruning, which in turn must be processed by $\text{OPT}_{\text{DYN}}$. There are lists where it does not discard any element. For example, when the elements of $R$ are all distinct and sorted in increasing order, all of them are left-$k$-maximal.

## 4.2 Right-pruning

This lossless pruning prunes any element followed by at least $k$ elements having a greater or equal relevance. The intuition behind this pruning is the following. Let $a$ be an element pruned out by the right-pruning; there are at least $k$ elements on its right that could be selected by any optimal algorithm OPT in place of $a$, without worsening the solution. The example below shows this intuition.

Example 4.4. *Let $R$ be the list $\langle 0.9, 1, \ldots, 1\rangle$ of 21 elements and $k = 20$. The element $a = 0.9$ is the only element discarded by the right-pruning. Every solution that includes the element $a$ can be improved by replacing it with a non selected element on its right. Indeed, the solution of $k$ elements $\langle 0.9, 1, \ldots, 1\rangle$ has DCG of about 6.9, while the solution $\langle 1, \ldots, 1\rangle$ has DCG of about 7.*

Let *right-height* of an element $a$ be the number of elements after $a$ having a greater or equal relevance. Given a solution $O$, we say that an element having right-height $h$ is a *right-defect* of $O$ if it occurs in the last $h$ positions of $O$. The following lemma proves the intuition above.

Lemma 4.5. *There exists an optimal filtering that does not contain right-defects.*

Proof. By contradiction, let us assume that all existing optimal solutions contain at least one right-defect. Let $\hat{O}$ be any optimal solution having the lowest number of defects and whose rightmost defect is the closest to the last position of the solution. The contradiction arises by showing that starting from $\hat{O}$ we can build a new solution which is either more relevant, with fewer defects, or with the rightmost defect closer to the last position of the solution.

Let $\hat{r}$ be the rightmost defect of $\hat{O}$, $h$ its right-height and $\hat{p}$ be its position in $\hat{O}$. By definition, there are at least $h$ elements having relevance greater than or equal to $\hat{r}$ on its right – within $R$ – and having a smaller right-height. At least one of them is not in $\hat{O}$ because there are less than $h$ elements after $\hat{r}$ in $\hat{O}$. Let $r$ be the leftmost of them, and $p$ be the position within $\hat{O}$ where $r$ should be placed if selected. We differentiate between the following two cases.

*i*) All elements of $\hat{O}$ between $\hat{p}$ and $p$ are greater than or equal to $\hat{r}$. Let $O$ be the solution obtained from $\hat{O}$ by removing $\hat{r}$ and by inserting $r$. The solution $O$ has a score greater than or equal to the one of $\hat{O}$. Moreover, the elements shifted of one position on the left do not introduce new defects because they are moving away from the last position of the solution. If $r$ introduces a defect, $O$ has the same number of defects of $\hat{O}$ and has the rightmost defect closer to the last position than $\hat{O}$. Otherwise, if $r$ does not introduce a defect, $O$ has a lower number of defects than $\hat{O}$. Therefore, the solution $O$ contradicts the hypothesis.

*ii*) At least one element of $\hat{O}$ between $\hat{p}$ and $p$ is strictly smaller than $\hat{r}$. Let $\hat{r}'$ be the rightmost element with this property. Let $O$ be the solution obtained from $\hat{O}$ by removing $\hat{r}'$ and by inserting $r$. The solution $O$ has a score strictly greater than $\hat{O}$, thus contradicting the optimality of $\hat{O}$. □

Let *right-$k$-maximal* be any element whose right-height is smaller than $k$. The following lemma proves the correctness of the right-pruning, and it directly follows from Lemma 4.5 by showing that any non-right-$k$-maximal element causes a right-defect into the solution. Indeed, a non-right-$k$-maximal element has a right-height greater or equal than $k$, and any viable solution contains at most $k$ results.

Lemma 4.6. *There exists an optimal filtering composed of only right-$k$-maximal elements of $R$.*

The right-pruning can be computed in $\Theta(n \log k)$ time by employing a priority queue to compute all right-$k$-maximal elements scanning the list right-to-left. At each step of the scan, the priority queue contains the top-$k$ elements seen so far. When the current element enters into the queue, it is right-$k$-maximal. If we are interested in computing also the right-heights of the right-$k$-maximal elements, we can use an AVL tree [6] of size $k$ to maintain the top-$k$ elements and compute the number of elements greater or equal of the current right-$k$-maximal element. Since AVL trees implement this lookup operation in $\Theta(\log k)$, the time complexity is still $\Theta(n \log k)$.

The right-pruning alone, however, does not guarantee an effective pruning of the original list. Indeed, there are lists where it does not reduce the number of elements to process with $\text{OPT}_{\text{DYN}}$. For example, when the elements of $R$ are all distinct and sorted in decreasing order, all of them are right-$k$-maximal. We thus discuss how to address the lack of guarantees of the left and right prunings by combining them to form an exact pruning strategy with strong guarantees.

## 4.3 Exact-Pruning

Left-pruning and right-pruning do not provide guarantees on the number of results surviving the pruning. For this reason, we show that we can effectively combine the two ideas into an efficient lossless pruning, called Exact-Pruning, guaranteeing that the number of non-pruned results is limited. Let *$k$-maximal* be any element whose left-height plus right-height is smaller than $k$. Exact-Pruning prunes out any non-$k$-maximal result from the list, as described in Algorithm 1. The following lemma proves the correctness of the proposed lossless pruning, and it directly follows from Lemma 4.2 and 4.5 by showing that any non-$k$-maximal element causes a left-defect or a right-defect into the solution. Indeed, a non-$k$-maximal element has left-height plus right-height greater or equal than $k$, and any viable solution contains at most $k$ results; thus it can occur neither in the head nor in the tail of any solution.

Lemma 4.7. *There exists an optimal filtering composed of only $k$-maximal elements of $R$.*

We have now to prove that the number of results not pruned by this pruning is limited, as stated by the following property.

Property 4.8. *$R$ contains at most $\left(2^k - 1\right)$ $k$-maximal elements.*

Proof. We prove by induction that for any $k > 0$, a list of at least $2^k$ elements contains at least one non-$k$-maximal result.

**Base case,** $k = 1$. Any combination of 2 elements contains an element violating either the left or the right maximality. Let $r_i$ and $r_j$ be the relevances of the first and second element. If $r_i \geq r_j$, than the $j$-th result is not left-$k$-maximal. Contrarily, if $r_i < r_j$, than the $i$-th result is not right-$k$-maximal. If there are more than two elements, it is trivial to show the claim.

**Inductive case,** $k > 1$. Let $p$ be the position of the most relevant result $r_{\max}$ into the list $R$. In case of ties, let us select the rightmost result with this property. Since $r_{\max}$ is the most relevant result of the list, it contributes to increase the right-height of all elements on its left by one. For the same reason, it also contributes to increase the left-height of all elements on its right by one. Since an element is $k$-maximal if its left-height plus its right-height is smaller than $k$, we can consider only the elements on the left or on the right of $p$ to find a non-($k$-1)-maximal result.

**Algorithm 1** Exact Pruning

```
Input: A list $R$ of $n$ relevances and a size threshold $k$
Output: The list of the $k$-maximal elements of $R$
 1: $left\_maximals$ = List()
 2: $stack$ = Stack()
 3: for ($i$ = 1; $i \leq n$; $i$ += 1) do                                ▷ Left-Pruning
 4:     while ($stack$.size() > 0 and $R[i] > stack$.top()) do
 5:         $stack.pop()$
 6:     end while
 7:     if ($stack.size() < k$) then
 8:         $left\_height = stack$.size()
 9:         $stack$.push($R[i]$)
10:         $left\_maximals$.append($\langle i, left\_height \rangle$)
11:     end if
12: end for
13: $result$ = List()
14: $tree$ = AVLTree()
15: for ($j = left\_maximals$.size(); $j > 0$; $j$ -= 1) do              ▷ Right-Pruning
16:     $\langle i, left\_height \rangle = left\_maximals[j]$
17:     $right\_height = tree$.num_greater_equal($R[i]$)
18:     if ($left\_height + right\_height < k$) then
19:         if ($tree$.size() < $k$) then
20:             $tree$.insert($R[i]$)
21:         else if ($R[i] > tree$.min()) then
22:             $tree$.replace_min($R[i]$)
23:         end if
24:         $result$.append($i$)
25:     end if
26: end for
27: return $result$.reverse()
```

Let $R_l$ and $R_r$ be the lists composed only by the elements on the left and right of $p$, respectively. One of the two lists contains at least $2^{k-1}$ results as $R$ contains at least $2^k$ elements. Thus, we can inductively show that it contains at least one non $(k$-1)-maximal result. □

The property above states that the number of $k$-maximal results is bounded by a quantity that is independent of the length of $R$ and its elements. To identify the $k$-maximal results, the algorithm prunes all non-left-$k$-maximal results, from left to right, and stores the left-height of each non-pruned result. Then, it computes all right-heights from right to left and prunes all non-$k$-maximal elements. The order of the two pruning operations matters as the pruning of a non-left-$k$-maximal result could affect the right-height of a less relevant result on its left, while the pruning of a non-right-$k$-maximal result does not affect the left-height of any element on the right. As already described for the left and right pruning, the left-heights and right-heights can be computed in $\Theta(n)$ and $\Theta(n \log k)$ time, respectively. Therefore, the time complexity of this pruning is $\Theta(n \log k)$.

## 4.4 OPT-Filtering: an exact filtering algorithm

The efficient exact filtering algorithm we are proposing, OPT-Filtering, is then the combination of Exact-Pruning and $\text{OPT}_{\text{DYN}}$. The following theorem shows that OPT-Filtering finds an optimal solution with a time complexity that is never higher than the one of $\text{OPT}_{\text{DYN}}$, which is $\Theta(nk)$.

THEOREM 4.9. *OPT-Filtering finds an optimal filtering in $\Theta\big(n \log k + \min(n, 2^k)k\big)$ time.*

PROOF. OPT-Filtering prunes the list in two steps then applies $\text{OPT}_{\text{DYN}}$ to the non-pruned results. It first computes all left-heights scanning the list left-to-right in $O(n)$ time. Then, OPT-Filtering computes all right-heights scanning the list right-to-left in $O(n \log k)$ by using an AVL tree, and prunes all non-$k$-maximal results. As shown in Property 4.8, at most $2^k - 1$ of the $n$ results are $k$-maximal, thus, it applies $\text{OPT}_{\text{DYN}}$ to these results in $\Theta(\min(n, 2^k)k)$ time. The optimality of the solution follows from Lemma 4.7. □

OPT-Filtering solves the FILTERING@$k$ problem by first applying Exact-Pruning and then $\text{OPT}_{\text{DYN}}$. It always finds the optimal solution, and it is also faster than the state-of-the-art optimal filtering when $k$ is smaller or equal than the logarithm of $n$, e.g., for one million results, when $k \leq 20$. In addition, to further enhance the efficiency of OPT-Filtering, we exploit the fact that there exists an optimal solution not containing left and right defects to improve the dynamic programming solution proposed by $\text{OPT}_{\text{DYN}}$. Within the memoization matrix used by the dynamic programming algorithm, each element $i$ participate to the $k$ possible positions that $i$ can occupy in the final solution. Interested readers can refer Section 2 for the details of the algorithm. Since in our setting we know the left and right heights of all $k$-maximal results, we can avoid the computation of the portions of the memoization matrix referring to the first left-height and the last right-height positions of each element. Indeed, if the element were placed in these positions by $\text{OPT}_{\text{DYN}}$, then it would cause either a left-defect or right-defect into the solution. This practical optimization does not improve the theoretical time complexity of OPT-Filtering, but it contributes to make our algorithm even more efficient.

## 5 APPROXIMATE FILTERING

The weak approximation guarantees provided by the existing heuristics motivate us to investigate a more efficient approximate solution to the FILTERING@$k$ problem. In this section, we propose $\epsilon$-Filtering: an approximate filtering algorithm that trades efficiency for effectiveness. The investigation of an approximate algorithm is also motivated by the fact that the relevances of the results are approximated using machine learning models in practice. Therefore, a small approximation on the quality of the filtering is negligible in the presence of inputs affected by an intrinsic error.

In detail, $\epsilon$-Filtering finds a (1-$\epsilon$)-approximation in $\Theta\big(n+k^2 \log_{(1\text{-}\epsilon)}(\epsilon/k)\big)$ time, for any $0 < \epsilon < 1$. $\epsilon$-Filtering is composed of three steps, followed by $\text{OPT}_{\text{DYN}}$: right-pruning, discretization, and thresholding. We already discussed the first step in the previous section. We now examine the remaining two steps.

### 5.1 Discretization

This step aims at decreasing the number of elements selected by the right pruning in the worst case. This is done by creating a new list $R_\epsilon$ from $R$ such that *i*) $R_\epsilon$ has a smaller number of distinct elements than $R$, and *ii*) the relevance of the optimal filtering of $R_\epsilon$ is guaranteed to be at least $(1 - \epsilon)$ times the relevance of the optimal filtering of $R$. This way, the use of the previous right pruning is more effective in terms of removed elements still almost preserving the optimality of the solution.

The idea is to discretize the relevance of the elements of $R$ to decrease the number of distinct elements. This discretization step trades-off between the approximation error of the solution and the obtained number of distinct elements. Let $\epsilon$ be the desired approximation error of $\epsilon$-Filtering, with $0 < \epsilon < 1$. Let $r_{\min}$ and $r_{\max}$ be the minimum and maximum relevance of the elements in $R$, respectively. The idea of the discretization step is to partition the range $[r_{\min}, r_{\max}]$ into $m$ intervals of elements. The elements belonging to the same interval are approximated in $R_\epsilon$ with the same relevance score, which equals the smallest relevance of the interval. The tricky part of this strategy is to decide how to partition the range $[r_{\min}, r_{\max}]$ to guarantee a $\epsilon$ approximation error and the minimum number of intervals $m$. Let $g(r)$ be the gain function used by the DCG metric, i.e., $g(r) = 2^r - 1$. We partition the range into maximal intervals such that the ratio between the gain $g()$ of the minimum and maximum relevance of each interval is at least (1-$\epsilon$). We call $\epsilon-$intervals the intervals in this partition. The following example shows how the discretization step works on the worst-case list shown for the right pruning.

Example 5.1. *Let $R$ be the list $\langle 3, 2.99, 2.98, \ldots, 0.01 \rangle$ of 300 elements and $\epsilon = 0.5$ the desired approximation error. Let $g(r)$ be the gain function used by the DCG metric, i.e., $g(r) = 2^r - 1$. The $\epsilon-$intervals of $R$ are the following ten intervals: $[0.01, 0.019)$, $[0.019, 0.039)$, $\ldots$, $[1.45, 2.17)$, $[2.17, 3.0]$. Within each interval, the ratio between the gain of the maximum and the gain of the minimum is (1-$\epsilon$). For example, for the last interval, we have $g(2.17)/g(3) \approx 0.5$. The list $R_\epsilon$ is as follows $\langle 2.17, \ldots, 2.17, 1.45, \ldots, 1.45, \ldots, 0.01 \rangle$.*

The following lemma extends Lemma 4.6 to $R_\epsilon$ and shows that the right-$k$-maximal elements of $R_\epsilon$ can be exploited to find a (1-$\epsilon$)-optimal filtering of $R$.

Lemma 5.2. *There exists a (1-$\epsilon$)-optimal filtering of $R$ made of only right-$k$-maximal elements of $R_\epsilon$.*

Proof. To prove the claim, we need to show that the optimal filtering of $R_\epsilon$ is (1-$\epsilon$)-optimal for $R$. Let $O$ be an optimal filtering of $R$. Let $\hat{O}_\epsilon$ be the filtering built by selecting from $R_\epsilon$ the same elements composing $O$. The gain of each element of $\hat{O}_\epsilon$ is at least (1-$\epsilon$) times the gain of its counterpart in $O$. Therefore, the relevance score of $\hat{O}_\epsilon$ is at least (1-$\epsilon$) times the relevance score of $O$. In particular, every optimal solution $O_\epsilon$ of $R_\epsilon$ has a score greater than or equal to the relevance score of $\hat{O}_\epsilon$, therefore $Q(O_\epsilon) \geq$ (1-$\epsilon$) $Q(O)$. The proof follows by applying Lemma 4.6 to $R_\epsilon$ and by showing that there exists a solution having score $Q(O_\epsilon)$ composed of only right-$k$-maximal elements of $R_\epsilon$. □

The discretization of $R$ into $R_\epsilon$ reduces the number of distinct elements from $n$ to $m$. The following property provides an upper bound for $m$.

Property 5.3. *The number of $\epsilon-$intervals of $R$ is: $m \leq \left\lceil \log_{(1-\epsilon)} \left( g(r_{min}) / g(r_{max}) \right) \right\rceil$.*

Proof. Each $\epsilon-$interval is such that the gain of the minimum element of the interval is (1-$\epsilon$)-times the gain of its maximum. Therefore, by starting from the most relevant element of $R$, $r_{\max}$, the number of intervals $m$ is the minimum value satisfying the following inequality

$$g(r_{\max}) \, (1\text{-}\epsilon)^m \leq g(r_{\min}) \, .$$

□

The discretization step introduces an approximation error $\epsilon$ to reduce the number of distinct elements in the list $R$ to $m$. In this way, the combination with the right-pruning guarantees that at most $mk$ elements are selected. However, the upper bound on $m$ depends on both the minimum and the maximum relevance in $R$, namely, $r_{\min}$ and $r_{\max}$. If the distance between these two values is large, then $m$ is large as well. In particular, it is possible to design an adversarial list $R$ where the discretization is ineffective, and the right pruning is forced to select all the elements. This is done by fixing a value of $r_{\min}$ and by choosing $r_{\max}$ to be large enough so that $m$ equals $n$. Then, $R$ is a decreasing list having an element for each $\epsilon-$interval. In this way, $R_\epsilon$ coincides with $R$ and the right-pruning selects all its $m = n$ distinct elements.

## 5.2 Thresholding

This step works by removing those elements whose relevance is below a given threshold. In this way, we increase the value of $r_{\min}$ in the resulting list, referred to as $R^-$, and, thus, we reduce $m$, i.e., the number of possible $\epsilon$−intervals. The threshold should be chosen so that *i*) the relevance of the optimal filtering of $R^-$ is at least $(1-\epsilon)$ times the relevance of the optimal filtering of $R$, and *ii*) the value of $m$ is guaranteed to be always much smaller than $n$. In this way, we can remove several elements with a small degradation of the solution, as shown in the following example.

Example 5.4. *Let $R$ be the list $\langle 5, 0.1, \ldots, 0.1\rangle$ of 10 elements and $k = 10$. The optimal filtering selects the full list $R$, whose DCG is about 31.25. If we remove all elements having relevance 0.1 then the list $R^-$ is $\langle 5\rangle$. The optimal filtering of $R^-$ has a DCG score of 31, namely, it is a 0.99-approximation.*

The following Lemma 5.5 states which threshold $t$ meets the desired approximation error $\epsilon$.

Lemma 5.5. *Let $R^-_\epsilon$ be the list obtained by removing the elements of $R$ below the threshold $t = g^{-1}\left(\epsilon\, g(r_{max})\,/k\right)$ and then by discretizing the remaining elements using the $\epsilon$−intervals. The optimal filtering of $R^-_\epsilon$ is a (1-$\epsilon$)-approximation.*

Proof. Let $O$ and $\hat{O}$ be the optimal filterings of $R$ and $R^-_\epsilon$, respectively. Since the ratio $\epsilon/k$ is a value strictly smaller than 1, the threshold $t$ is strictly smaller than $r_{\max}$. Let us consider the worst-case list whose results, a part of the maximum, are infinitely close to the threshold $t$, but smaller than it, and are all placed on the right of $r_{\max}$. The solution $\hat{O}$ is thus formed by the singleton $\langle r_{\max}\rangle$ while the solution $O$ is the list $\left\langle r_{\max}, \tilde{t}, \ldots, \tilde{t}\right\rangle$. Let $g(r)$ be the gain function used by the DCG metric, i.e., $g(r) = 2^r - 1$, and $d(r)$ be its discount function, i.e., $d(p) = \log_2(p+1)$. The approximation factor achieved by using $\hat{O}$ is

$$(1-\epsilon) = \frac{Q(\hat{O})}{Q(O)} = \frac{g(r_{\max})\, d(1)}{g(r_{\max})\, d(1) + g(\tilde{t})\sum_{i=2}^{k} d(i)},$$

which can be rewritten as

$$g(\tilde{t}) = \frac{\epsilon g(r_{\max})\, d(1)}{(1\text{-}\epsilon)\sum_{i=2}^{k} d(i)}.$$

The gain function $g()$ is a strictly increasing function, hence it admits an inverse function. Moreover, since the discount function $d()$ is always smaller or equal than 1, and the factor (1-$\epsilon$) is strictly smaller than 1, thus the denominator is strictly smaller than $k$. Therefore, the proof follows by using these rough approximations.

□

## 5.3 Approximate-Pruning

The right-pruning, the discretization and the thresholding steps can be combined together to effectively reduce the number of distinct values of the list $R$. Indeed, the threshold $t$ limits the number of $\epsilon$−intervals and, thus, the number of distinct values involved in the computation of the right-$k$-maximal elements. The resulting algorithm, called $\epsilon$-Pruning, is described in Algorithm 2. In detail, we can easily derive the following property by replacing $r_{\min}$ in Property 5.3 with the threshold $t$ of Lemma 5.5.

Property 5.6. *The number of $\epsilon$−intervals of $R^-_\epsilon$ is: $m \leq \left\lceil \log_{(1\text{-}\epsilon)}(\epsilon/k)\right\rceil$.*

The property above states that the maximum number of $\epsilon$−intervals of $R^-_\epsilon$ can be upper bounded by a quantity that is independent of the length of $R$ and its elements. As each right-$k$-maximal element $r$ is followed by at most $k$ elements having a relevance greater than or equal to $r$, the

**Algorithm 2** Approximate Pruning

**Input**: A list $R$ of $n$ relevances, a size threshold $k$ and a target approximation error $\epsilon$
**Output**: The list of the right-$k$-maximal elements of $R^-_\epsilon$

```
1:  $result$ = List()
2:  $heap$ = MinHeap()
3:  $cutoff$ = get_epsilon_cutoff($k$, $\epsilon$, max($R$))
4:  $i = n$
5:  for (; $heap$.size() $< k$ and $i > 0$; $i$ -= 1) do
6:      if ($R[i] > cutoff$) then
7:          $heap$.push($R[i]$)
8:          $result$.append($i$)
9:      end if
10: end for
11: $intervals$ = get_eps_intervals($\epsilon$, max($R$), $heap$.min())
12: for ($cur$ = 1; $i > 0$; $i$ -= 1) do
13:     if ($R[i] > intervals[cur]$) then
14:         $heap$.replace_min($R[i]$)
15:         $result$.append($i$)
16:         while ($intervals[cur] <$ $heap$.min()) $cur$ += 1
17:     end if
18: end for
19: return $result$.reverse()
```

number of right-$k$-maximal elements of $R^-_\epsilon$ can be upper bounded by a quantity depending on $\epsilon$ and $k$ only. Indeed, there are at most $km \leq k\lceil \log_{(1\text{-}\epsilon)}(\epsilon/k)\rceil$ right-$k$-maximal elements in $R^-_\epsilon$.

$\epsilon$-Pruning runs in $\Theta\,(n + km\log k)$ time as the maximum number of right-$k$-maximal elements of $R^-_\epsilon$ is $km$. The first factor is due to the cost of a linear scan of the list $R$, while the second factor is due to the cost of $\Theta(km)$ updates of a priority queue of size $k$. Therefore, $\epsilon$-Pruning is an efficient and effective way for reducing the number of elements to be processed by $\text{OPT}_{\text{DYN}}$. Notice that $\epsilon$-Filtering could employ the exact-pruning in place of the right-pruning step, but it would lead to the same theoretical efficiency and effectiveness guarantees. However, the exact-pruning relies on an AVL tree to compute the right-heights of the elements, which is substantially slower than the heap data structure used by the right-pruning. Therefore, $\epsilon$-Filtering benefits of fast and simple data structures and avoids the direct computation of left and right heights to fasten the filtering.

### 5.4 $\epsilon$-Filtering: an approximate filtering algorithm

The proposed algorithm, $\epsilon$-Filtering, applies $\epsilon$-Pruning and $\text{OPT}_{\text{DYN}}$ in sequence. Theorem 5.7 shows that it finds a (1-$\epsilon$)-optimal solution and trades-off effectiveness and efficiency through the approximation error $\epsilon$.

Theorem 5.7. *$\epsilon$-Filtering finds a (1-$\epsilon$)-optimal filtering in $\Theta\big(n + k^2\log_{(1\text{-}\epsilon)}(\epsilon/k)\big)$ time.*

Proof. $\epsilon$-Filtering is composed of two phases: $\epsilon$-Pruning and $\text{OPT}_{\text{DYN}}$. $\epsilon$-Pruning finds the right-$k$-maximal elements of $R^-_\epsilon$, which, according to Lemma 4.6, contain the optimal solution of the new approximate list. $\text{OPT}_{\text{DYN}}$ finds the optimal solution of the right-$k$-maximal elements of $R^-_\epsilon$, which, according to Lemma 5.5, is (1-$\epsilon$)-optimal.

As far as the time complexity is concerned, we need to prove that $\epsilon$-Filtering runs in $\Theta\big(n + k^2\log_{(1\text{-}\epsilon)}(\epsilon/k)\big)$ time. $\epsilon$-Pruning runs in $\Theta\,(n + km\log k)$ time, while $\text{OPT}_{\text{DYN}}$ runs in $\Theta\,(k^2m)$ time when applied to the $\Theta(km)$ elements selected by $\epsilon$-Pruning. Therefore, the claim follows from Property 5.6. □

Table 1. Main statistics of GoogleLocalRec and AmazonRel.

| Dataset | num queries | average num results | average relevance | max relevance |
|---|---|---|---|---|
| GoogleLocalRec | 10,000 | 16,000 | 0.76 | 2.97 |
| AmazonRel | 250 | 99,430 | 1.79 | 3.98 |

$\epsilon$-Filtering solves the relevance-aware filtering problem by combining three steps, i.e., right-pruning, discretization, and thresholding, with $\text{OPT}_{\text{DYN}}$. It provides strong guarantees on the effectiveness of the filtered list, and its performance is driven by the approximation error $\epsilon$, which trades efficiency for effectiveness.

## 5.5 Distributed setting

Many search services exploit distributed query processing architectures [3, 19] to deliver content to the users. Typically, a distributed environment consists of a set of indexes each one queried by a search engine. On top of them, a meta-search engine distributes the query to all search engines. Each search engine queries its index in parallel and produces a ranked list of top-$k$ results that is sent back to the meta-search engine. Finally, the meta-search engine aggregates the results and send them to the user.

$\text{OPT}_{\text{DYN}}$ cannot be applied in a distributed query processing architecture since it computes the optimal solution by taking into account all the results available. One possibility to overcome this issue is to deploy $\text{OPT}_{\text{DYN}}$ in the meta-search engine. However, sending all the results matching a query from a search engine to the meta-search engine is unfeasible. This would lead to high network communication overhead between machines. Moreover, by applying $\text{OPT}_{\text{DYN}}$ on partial lists of results, i.e., on a search engine level, the final solution is not guaranteed to be optimal.

$\epsilon$-Filtering can be applied in a distributed query processing architecture as it preserves the approximation guarantees. The application of $\epsilon$-Filtering to a distributed scenario requires that *i*) each search engine applies $\epsilon$-Pruning to the local results and sends the right-$k$-maximal elements to the meta-search engine, then *ii*) the meta-search engine merges the received lists by preserving the per-attribute sorting and applies $\epsilon$-Filtering to the merged list. By exploiting Property 5.6, we can prove that the number of elements transferred by each search engine is at most $k\lceil \log_{(1\text{-}\epsilon)}(\epsilon/k)\rceil$, thus confirming the feasibility of the approach.

## 6 EXPERIMENTS

In this section, we present a comprehensive experimental assessment of OPT-Filtering, $\epsilon$-Filtering and state-of-the-art competitors. First, we describe the datasets along with the experimental settings used to assess the different strategies. Then, we compare the proposed OPT-Filtering and $\epsilon$-Filtering to state-of-the-art competitors. Lastly, we study the impact of the parameter $\epsilon$ on the performance achieved by $\epsilon$-Filtering. We perform the analysis by varying the approximation threshold $\epsilon$, the number $n$ of the results to filter, the number $k$ of results to return, and the quality metric $Q$.

## 6.1 Experimental settings

**Datasets**. We evaluate the performance of all filtering algorithms on two public datasets, namely GoogleLocalRec and AmazonRel, that we built and released to facilitate the assessment of filtering algorithms. The two datasets represent two different real-world use cases, i.e., a recommendation scenario and a search scenario, and exploit two ad-hoc state-of-the-art solutions to estimate the relevance. Moreover, both datasets contain many results per query which allow us to perform a comprehensive assessment of the various filtering methods. We report in Table 1 the main statistics of the two datasets.

The GoogleLocalRec dataset employs Google Local data and a state-of-the-art recommender system, TransFM$_{\text{CONTENT}}$, that recently achieved the best performance in the task of sequential recommendation [18], i.e., the task of predicting the next action of a user starting from her historical interactions. The dataset consists of an extensive collection of geographically localized businesses and temporally labeled reviews by users [18]. In particular, we focused on the city of New York, which is the city with the highest number of businesses and active users that wrote reviews, and we employ the TransFM recommendation model to produce, given an active user, a list of relevant businesses to recommend to her. Therefore, we followed the methodology described by Pasricha and McAuley [18] to prepare the dataset and train the recommendation model using the reviews of the businesses within a radius of 50Km from the New York centre. We then randomly selected 10,000 active users and for each of them: *i*) we sorted the businesses by distance from the last known location of the user, and *ii*) we estimated their relevance for the user employing the TransFM recommendation model. The GoogleLocalRec dataset that we built is thus composed of 10,000 users, also referred to as queries, each one supplemented with a list of 16,000 recommendations.

The AmazonRel dataset employs Amazon data and a winning solution[1] of the Crowdflower Search Results Relevance[2] Kaggle competition to estimate the relevance of the Amazon products with respect to the user queries. The Crowdflower Search Results Relevance competition aims to transform the most effective machine learning solutions into open-source products to help small e-commerce sites to measure the relevance of the search results they provide. Since the test set distributed as part of the Kaggle competition contains only a few results associated to each textual query, we extended the list of results of each query with the Amazon products having at least one of the query terms in the title or in the description. To this end, we employed the extensive list of Amazon products of the public dataset introduced by McAuley *et al.* [15], whose data come from the Amazon e-commerce website and span from May 1996 to July 2014. In particular, for each test query: *i*) we sorted the results by price, and *ii*) we estimated their relevance with respect to the query employing the aforementioned relevance model, which relies on ensembles [7] of several machine learning models to estimate the relevance of each search result. The AmazonRel dataset that we built is thus composed of 250 queries, characterized by at least 500 results each, with about 100,000 results per query, on average.

**Metrics**. We employ two search quality metrics $Q$ to assess the performance of all filtering algorithms under different evaluation criteria:

- DCG $(\langle r_1, ..., r_n\rangle) = \sum_{p=1}^{n} \frac{2^{r_p}-1}{\log_2(p+1)}$ combines an exponential gain function for relevance, $g(r) = 2^r - 1$, with a smooth discount factor for position, $d(p) = 1/\log_2(p+1)$. It strongly promotes relevant results.
- DCG-LZ $(\langle r_1, ..., r_n\rangle) = \sum_{p=1}^{n} \frac{r_p}{p}$, combines a linear gain function for relevance, $g(r) = r$, with a Zipfian discount factor for position [11], $d(p) = 1/p$. It highly penalizes the results that are not in the top positions of the ranked list.

The two aforementioned metrics are widely used within the IR community [9, 11, 24]. Due to their differences in $g(\cdot)$ and $d(\cdot)$, they provide a very different evaluation: DCG takes into account the relevance more than the rank, while DCG-LZ takes into account the rank more than the relevance.

[1] https://github.com/geffy/kaggle-crowdflower
[2] https://www.kaggle.com/c/crowdflower-search-relevance

Table 2. Average filtering time in milliseconds, speedup (above) and worst approximation error (below) achieved by all filtering algorithms on the GoogleLocalRec dataset by setting $n$ to 16, 000 and by varying $k$.

| $k =$ | 20 | 50 | 100 | 200 |
|---|---|---|---|---|
| OPT$_{\text{DYN}}$ | 0.117 | 0.163 | 0.279 | 0.488 |
| Cutoff-OPT | 0.033 (4×) | 0.039 (4×) | 0.055 (5×) | 0.081 (6×) |
| (*no-proven-approx*) | 0.20 | 0.29 | 0.34 | 0.38 |
| Top$k$-OPT | 0.017 (7×) | 0.022 (7×) | **0.034 (8×)** | **0.056 (9×)** |
| (0.5-*approx*) | 0.15 | 0.14 | 0.13 | 0.11 |
| OPT-Filtering | 0.026 (4×) | 0.076 (2×) | 0.156 (2×) | 0.330 (2×) |
| $\epsilon$-Filtering | 0.018 (7×) | 0.029 (6×) | 0.053 (5×) | 0.105 (5×) |
| ($\epsilon$=0.001) | 0 | 0 | 0 | 0 |
| $\epsilon$-Filtering | 0.013 (9×) | 0.023 (7×) | 0.046 (6×) | 0.095 (5×) |
| ($\epsilon$=0.01) | 0 | 0 | 0 | 0 |
| $\epsilon$-Filtering | **0.010 (11×)** | **0.018 (9×)** | **0.034 (8×)** | 0.075 (6×) |
| ($\epsilon$=0.1) | 0.08 | 0.07 | 0.06 | 0.04 |

**Filtering algorithms**. We perform a comparison of the performance of OPT-Filtering and $\epsilon$-Filtering against three state-of-the-art filtering algorithms: OPT$_{\text{DYN}}$, Top$k$-OPT and Cutoff-OPT. The three competitors, introduced by Spirin *et al.* [21], have been thoroughly discussed in Sections 2 and 3. In our experiments, we employ a query-based relevance threshold for Cutoff-OPT, which is defined as $(r_{\max} + r_{\min})/2$, where $r_{\max}$ ($r_{\min}$) is the maximum (minimum) relevance of the list.

**Testing details**. We implemented all algorithms in C++17 using GCC 9.2.1 with the highest optimization settings for the compilation. We performed all experiments on a machine with eight Intel Core i9-9900K cores clocked at 3.60GHz and 64GB of RAM. All timings refer to the average execution time (in milliseconds) of ten independent runs. To ease the reproducibility of the experiments, we release[3] the two datasets and our implementation of OPT-Filtering, $\epsilon$-Filtering, OPT$_{\text{DYN}}$, Top$k$-OPT, and Cutoff-OPT.

## 6.2 Experimental results

We now provide a comprehensive assessment of the performance of all filtering algorithms performing two sets of experiments using the DCG-LZ metric: one by varying the cut $k$, and one by varying the length $n$ of the lists of results. The two sets of experiments provide a twofold view of the impact of the problem parameters $k$ and $n$ on the filtering performance of the various algorithms. For this evaluation, we employ $\epsilon$-Filtering varying the approximation threshold $\epsilon \in \{0.1, 0.01, 0.001\}$.

### Assessment by varying $k$

We assess the filtering performance of all algorithms on the two datasets by varying the cut $k \in \{20, 50, 100, 200\}$ and setting $n = 16,000$, i.e., the maximum length of the results lists available in the GoogleLocalRec dataset. To do so, we employ the first 16, 000 results sorted by attribute of each query and discard those queries with less than 16, 000 results. We summarize in Tables 2 and 3 the results of the experiments on the two datasets, i.e., GoogleLocalRec and AmazonRel. In detail,

[3]https://github.com/hpclab/fast-approximate-filtering

Table 3. Average filtering time in milliseconds, speedup (above) and worst approximation error (below) achieved by all filtering algorithms on the AmazonRel dataset by setting $n$ to 16,000 and by varying $k$.

| $k$ = | 20 | 50 | 100 | 200 |
|---|---|---|---|---|
| OPT$_{\text{DYN}}$ | 0.112 | 0.164 | 0.290 | 0.537 |
| Cutoff-OPT | 0.041 (3×) | 0.049 (3×) | 0.069 (4×) | 0.107 (5×) |
| (*no-proven-approx*) | 0.23 | 0.33 | 0.39 | 0.43 |
| Top$k$-OPT | 0.015 (7×) | 0.019 (9×) | 0.026 (11×) | **0.045 (12×)** |
| (0.5-*approx*) | 0.14 | 0.14 | 0.13 | 0.13 |
| OPT-Filtering | 0.021 (5×) | 0.054 (3×) | 0.116 (3×) | 0.254 (2×) |
| $\epsilon$-Filtering | 0.012 (9×) | 0.019 (9×) | 0.035 (8×) | 0.074 (7×) |
| ($\epsilon$=0.001) | 0 | 0 | 0 | 0 |
| $\epsilon$-Filtering | 0.010 (11×) | 0.017 (10×) | 0.031 (9×) | 0.067 (8×) |
| ($\epsilon$=0.01) | 0 | 0 | 0 | 0 |
| $\epsilon$-Filtering | **0.007 (17×)** | **0.011 (14×)** | **0.021 (14×)** | **0.045 (12×)** |
| ($\epsilon$=0.1) | 0.06 | 0.05 | 0.05 | 0.04 |

we report for each algorithm and cut $k$: *i*) the average filtering time, *ii*) the speedup achieved over OPT$_{\text{DYN}}$, and *iii*) the worst approximation error achieved on all queries.

On both datasets, the filtering time of OPT$_{\text{DYN}}$ ranges from about 0.1ms ($k = 20$) to about 0.5ms ($k = 200$). Cutoff-OPT and Top$k$-OPT are faster than OPT$_{\text{DYN}}$, as expected. Indeed, Cutoff-OPT speeds-up from 3× to 6× over OPT$_{\text{DYN}}$, while Top$k$-OPT achieves even better speedups, from 7× to 12× over OPT$_{\text{DYN}}$, and it is the faster filtering when $k = 200$. It is worth reminding that we proved in Section 3 that Cutoff-OPT does not offer approximation guarantees while Top$k$-OPT is 0.5−optimal, i.e., it may find a solution whose relevance score is 0.5 times the score of the optimal solution. For this reason, we assess the worst approximation error achieved by the two heuristics in practice on the two datasets. We notice that Cutoff-OPT achieves an approximation error from about 0.20 ($k = 20$) to about 0.40 ($k = 200$), while Top$k$-OPT achieves lower errors, from about 0.15 ($k = 20$) to about 0.12 ($k = 200$). Even if Top$k$-OPT shows better performance than Cutoff-OPT, the experimental results show that, also in practice, the two heuristics are far from being optimal.

The proposed OPT-Filtering algorithm optimally solves the filtering problem. The average time spent by OPT-Filtering to filter the lists of results ranges from about 0.02ms ($k = 20$) to about 0.3ms ($k = 200$). It shows a valuable speedup from 2× to 5× with respect to the exact competitor OPT$_{\text{DYN}}$ on both datasets. In particular, it achieves the best speedups on the two datasets when employing small values of $k$. Therefore, the proposed exact algorithm provides a new optimal filtering algorithm which is more efficient than the state-of-the-art optimal algorithm OPT$_{\text{DYN}}$.

The proposed $\epsilon$-Filtering algorithm, driven by the parameter $\epsilon$, finds a (1-$\epsilon$) approximate filtering solution. The approximation threshold $\epsilon$ also drives the complexity of the algorithm, which takes more time to compute a more accurate solution when requiring a lower approximation error. As expected, $\epsilon$-Filtering achieves the best speedups with respect to OPT$_{\text{DYN}}$ when employing $\epsilon = 0.1$, i.e., when admitting up to 10% of error. In this setting, the speedups achieved by $\epsilon$-Filtering with respect to OPT$_{\text{DYN}}$ range from 6× to 17×, while the worst approximation error ranges from 0.08 to 0.04. $\epsilon$-Filtering achieves noteworthy speedups, from 5× to 9×, even when employing $\epsilon = 0.001$, i.e., when we admit up to 1‰ of error. In particular, when $\epsilon \in \{0.01, 0.001\}$, the proposed $\epsilon$-Filtering algorithm always found the optimal solution (zero error) on all queries of the two datasets, for all

Table 4. Average filtering time in milliseconds, speedup (above) and worst approximation error (below) achieved by all filtering algorithms on the AmazonRel dataset by setting $k$ to 100 and by varying $n$.

| $n =$ | 50,000 | 100,000 | 200,000 | 500,000 |
|---|---|---|---|---|
| OPT$_{\text{DYN}}$ | 1.263 | 8.562 | 17.480 | 43.924 |
| Cutoff-OPT | 0.204 (6×) | 0.441 (19×) | 1.490 (12×) | 1.098 (40×) |
| (*no-proven-approx*) | 0.20 | 0.04 | 0.02 | 0.00 |
| Top$k$-OPT | 0.057 (22×) | 0.100 (86×) | 0.183 (96×) | 0.429 (102×) |
| (0.5-*approx*) | 0.11 | 0.10 | 0.11 | 0.05 |
| OPT-Filtering | 0.165 (8×) | 0.219 (39×) | 0.310 (56×) | 0.570 (77×) |
| $\epsilon$-Filtering | 0.056 (22×) | 0.082 (105×) | 0.126 (139×) | 0.255 (172×) |
| ($\epsilon$=0.001) | 0 | 0 | 0 | 0 |
| $\epsilon$-Filtering | 0.051 (25×) | 0.076 (113×) | 0.119 (146×) | 0.249 (177×) |
| ($\epsilon$=0.01) | 0 | 0 | 0 | 0 |
| $\epsilon$-Filtering | **0.035 (36×)** | **0.050 (171×)** | **0.084 (207×)** | **0.180 (244×)** |
| ($\epsilon$=0.1) | 0.05 | 0.05 | 0.05 | 0.03 |

values $k$, while achieving speedups ranging from 5× to 11× over OPT$_{\text{DYN}}$. The result highlights that $\epsilon$-Filtering achieves better speedups and lower approximation errors than Top$k$-OPT and Cutoff-OPT. Moreover, $\epsilon$-Filtering is also able to compute solutions that are very close to, or optimal, with a significant reduction of the filtering time with respect to the state-of-the-art optimal algorithm.

## Assessment by varying $n$

We assess the filtering performance of all algorithms on the AmazonRel dataset, which contains up to 600,000 results per query, by varying the length of the list $n \in \{50,000, 100,000, 200,000, 500,000\}$ and setting $k = 100$. To this end, for each value of $n$, we employ the first $n$ results sorted by attribute of each query and discard those queries with less than $n$ results. We summarize in Table 4 the results of this experiment on the AmazonRel dataset. As done for the previous analysis, we report for each algorithm and length $n$: *i*) the average filtering time, *ii*) the speedup achieved over OPT$_{\text{DYN}}$, and *iii*) the worst approximation error achieved on all queries.

The time required by OPT$_{\text{DYN}}$ to filter the lists significantly degrades when dealing with longer lists: from about 1ms for lists of 50,000 items to about 44ms for lists of 500,000 items. The huge variation of the filtering time required by OPT$_{\text{DYN}}$ confirms that it is not always a feasible choice. Cutoff-OPT and Top$k$-OPT are faster than OPT$_{\text{DYN}}$, and the gap increases with longer lists. Indeed, Cutoff-OPT speeds-up from 6× to 40× over OPT$_{\text{DYN}}$, while Top$k$-OPT achieves even better speedups, from 22× to 102× over OPT$_{\text{DYN}}$. We also observe that the worst approximation errors achieved by Cutoff-OPT and Top$k$-OPT decrease when dealing with long lists. Cutoff-OPT achieves approximation errors from about 0.20 ($n$ = 50,000) to exactly 0 ($n$ = 500,000), while Top$k$-OPT achieves lower errors, from about 0.11 ($n$ = 50,000) to about 0.05 ($n$ = 500,000).

The proposed OPT-Filtering exact algorithm achieves relevant speedups ranging from 8× ($n$ = 50,000) to 77× ($n$ = 500,000) over OPT$_{\text{DYN}}$. It is thus much faster than the exact competitor OPT$_{\text{DYN}}$ and the gap increases when increasing the number of items to filter. Moreover, it is even faster than the Cutoff-OPT heuristic filtering that does not offer approximation guarantees.

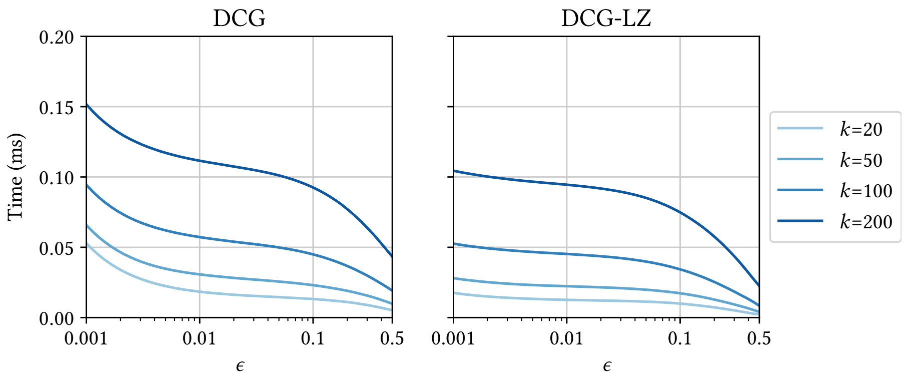

Fig. 1. Average filtering time of $\epsilon$-Filtering by varying $\epsilon$ and $k$.

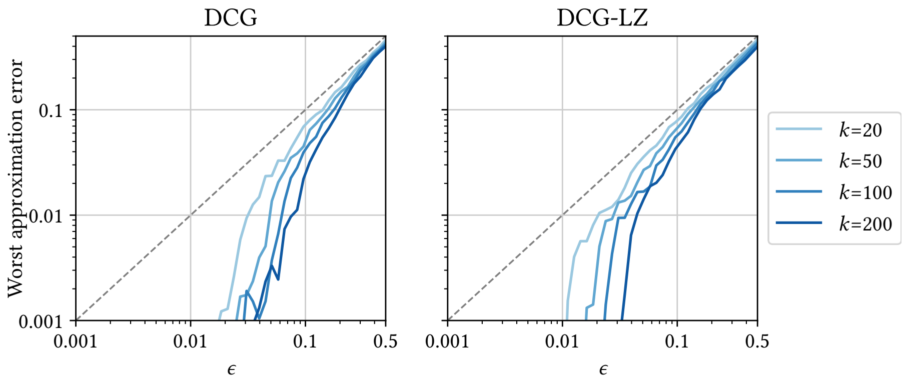

Fig. 2. Worst approximation error of $\epsilon$-Filtering by varying $\epsilon$ and $k$.

The proposed $\epsilon$-Filtering algorithm outperforms all competitors in terms of filtering time and approximation error. When employing $\epsilon = 0.1$, i.e., when admitting up to 10% of error, $\epsilon$-Filtering speeds-up the filtering from 36× to 244× with respect to $\text{OPT}_{\text{DYN}}$, committing an approximation error of about 0.05%. It also achieves important speedups from 22× to 172× over $\text{OPT}_{\text{DYN}}$ when admitting up to 1‰ of error, i.e., $\epsilon = 0.001$, outperforming all heuristic algorithms with much better approximation guarantees. In particular, when employing $\epsilon \in \{0.01, 0.001\}$, $\epsilon$-Filtering always found the optimal solution on all queries, for all values $n$. We also observe that the time spent by Top$k$-OPT grows faster than the one of $\epsilon$-Filtering when increasing $n$. This result confirms the theoretical analysis we provide in Section 5, where we show that the time complexity of Top$k$-OPT, i.e., $\Theta\left(n \log k + k^2\right)$, has a factor $\log k$ more than the one of $\epsilon$-Filtering, which is $\Theta\big(n + k^2 \log_{(1\text{-}\epsilon)} \frac{\epsilon}{k}\big)$.

$\epsilon$-Filtering is the best performing method for all values of $n$ and $\epsilon$. When employing $\epsilon = 0.1$, $\epsilon$-Filtering is remarkably faster than the two heuristics achieving a speedup of up to 244× on lists of 500,000 items. Moreover, its performance does not degrade significantly when requiring better solutions. Indeed, when decreasing $\epsilon$ to lower values, i.e., 0.01 and 0.001, $\epsilon$-Filtering always found the optimal solution on all tested queries achieving a speedup of up to 177× over $\text{OPT}_{\text{DYN}}$.

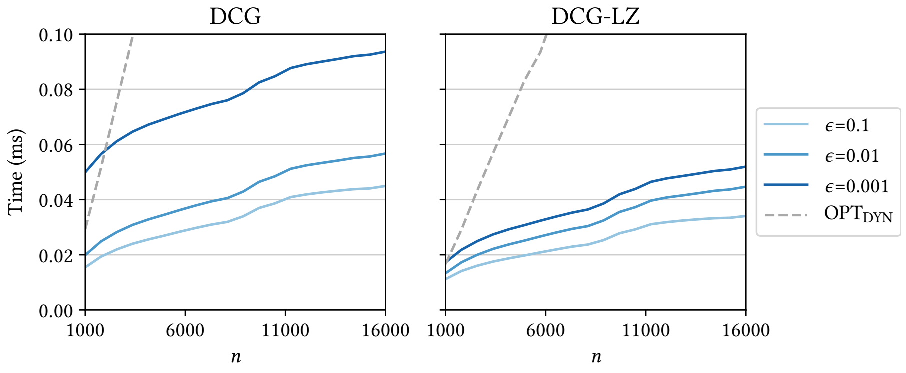

Fig. 3. Average filtering time of $\epsilon$-Filtering and $\text{OPT}_{\text{DYN}}$ by varying $n$ and $\epsilon$.

## 6.3 Assessment of the efficiency-effectiveness trade-offs achieved by $\epsilon$-Filtering

We now analyze the performance of $\epsilon$-Filtering varying the trade-off parameter $\epsilon$, which drives the efficiency of the algorithm through the granularity of the approximation. We present this analysis employing the search quality metrics DCG and DCG-LZ, and assessing the behaviour of $\epsilon$-Filtering when varying the problem parameters $k$ and $n$. We report the results achieved on the 10,000 test queries of the GoogleLocalRec dataset as we observed similar findings on the AmazonRel dataset.

### Assessment by varying $k$

We show in Figure 1 the average filtering time in milliseconds (y-axis) required by $\epsilon$-Filtering when varying $\epsilon$ (x-axis) from 0.001 to 0.5. We perform this assessment for different values of the cut $k$, i.e., $k \in \{20, 50, 100, 200\}$, that are interesting for real-world applications. The time required by $\epsilon$-Filtering decreases when increasing the approximation threshold $\epsilon$, i.e., when requiring less accurate solutions. The reported results are always below 0.15 milliseconds for all the tested combinations of $\epsilon$ and $k$, which is a negligible amount of time for a query processing pipeline. In detail, $\epsilon$-Filtering takes from 0.05ms ($\epsilon = 0.5$) to 0.15ms ($\epsilon = 0.001$) when filtering the list to $k = 200$ results. We observe that the time required by the algorithm smoothly increases when requiring approximation thresholds $\epsilon$ between 0.005 and 0.1, for all values of $k$ tested. $\epsilon$-Filtering is thus a valuable filtering algorithm for online applications characterized by tight time constraints, even requiring very small approximation errors. We also observe that the parameter $k$ affects efficiency more than the approximation threshold $\epsilon$. As we showed in Theorem 5.7, the time complexity of $\epsilon$-Filtering is quadratic with respect to the parameter $k$. Figure 1 clearly shows this result: when $k$ moves from 100 to 200, the difference in time is greater than the difference in time achieved by varying $\epsilon$ of one order of magnitude.

To measure the approximation error achieved in practice by $\epsilon$-Filtering, we assess the worst (maximum) approximation error achieved on all test queries. In Figure 2, we report the worst approximation error (y-axis) achieved when varying $\epsilon$ (x-axis) from 0.001 to 0.5. The dashed diagonal line represents the theoretical approximation error, i.e., those points whose approximation error is equal to the approximation threshold $\epsilon$. We notice that the approximation error achieved in practice rapidly decreases when decreasing the approximation threshold $\epsilon$. In particular, it is always far from the theoretical approximation error and, when requiring less than one per cent of error, i.e., $\epsilon = 0.01$, the algorithm finds an optimal solution for all tested queries and for all values of $k$. Therefore, in practice, $\epsilon$-Filtering achieves very good approximations of the optimal solution.

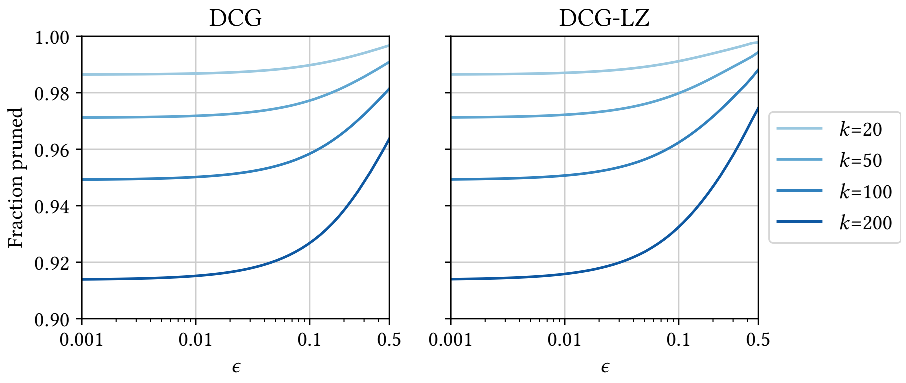

Fig. 4. Average fraction of pruned elements of $\epsilon$-Filtering by varying $\epsilon$ and $k$.

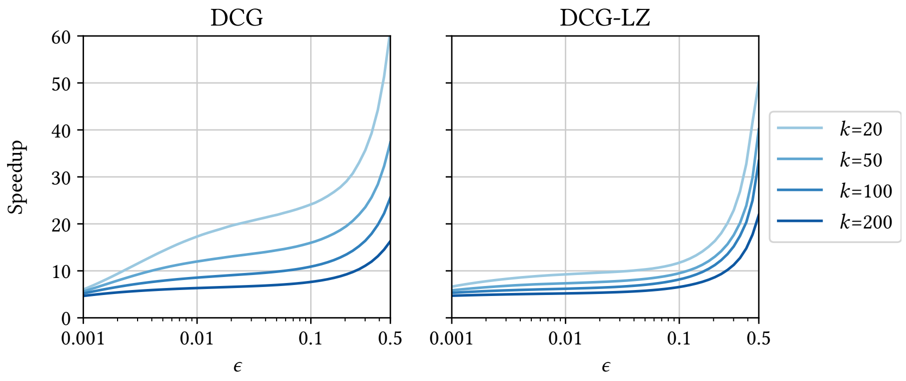

Fig. 5. Average speedup of $\epsilon$-Filtering by varying $\epsilon$ and $k$.

## Assessment by varying $n$

We show in Figure 3 the impact of the number $n$ of results to filter (x-axis) on the processing time required by $\epsilon$-Filtering and $\text{OPT}_{\text{DYN}}$ (y-axis). We perform this assessment setting the cut $k$ to 100 and varying $n$ between 1,000 and 16,000: for each value of $n$, we consider only the first $n$ results sorted by attribute of each test query. Moreover, we test different values of the approximation threshold $\epsilon$ for $\epsilon$-Filtering, i.e., $\epsilon \in \{0.1, 0.01, 0.001\}$. Figure 3 shows the average filtering time required by $\epsilon$-Filtering and $\text{OPT}_{\text{DYN}}$ to process the 10,000 test queries. The figure clearly shows that the time required by $\text{OPT}_{\text{DYN}}$ grows very rapidly when increasing $n$, while the time required by $\epsilon$-Filtering increases very slowly in the same range. In detail, the time required by $\text{OPT}_{\text{DYN}}$ increases by 16× when passing from 1,000 to 16,000 results, while the time required by $\epsilon$-Filtering only doubles in the same range. Experimental results highlight a valuable property of the proposed algorithm: given an approximation threshold $\epsilon$, the filtering time required by $\epsilon$-Filtering is almost constant when increasing $n$. $\epsilon$-Filtering is thus a valuable filtering algorithm whose performance does not degrade significantly when filtering the long list of results.

## Speedup

In section 5.4 we show that the approximation threshold $\epsilon$ guarantees the desired approximation error by controlling the granularity of the pruning of $\epsilon$-Filtering, which in turn affect the number of elements filtered by $\text{OPT}_{\text{DYN}}$. In Figure 4, we show the average fraction of results of the list that are pruned by $\epsilon$-Filtering (y-axis) by varying the approximation threshold $\epsilon$ (x-axis). The figure shows that the number of elements pruned by $\epsilon$-Filtering grows when decreasing the approximation threshold $\epsilon$ above 0.01. Specifically, the proposed algorithm filters out more than 95% of the elements, on average, when requiring a cut $k$ of up to 100. This massive reduction of the original list to less than 5% of the elements boosts the filtering efficiency as it fastly reduces the amount of work done by the slow $\text{OPT}_{\text{DYN}}$ algorithm.

Figure 5 reports the average speedup achieved by $\epsilon$-Filtering with respect to $\text{OPT}_{\text{DYN}}$ (y-axis) by varying the approximation threshold $\epsilon$ (x-axis). Results show a speedup of about one order of magnitude for all tested combinations of parameters. In particular, $\epsilon$-Filtering with $\epsilon = 0.01$ is from 9 to 18 times faster than $\text{OPT}_{\text{DYN}}$ when using the DCG metric, and from 6 to 9 times when using the DCG-LZ metric. This difference is mainly due to the fact that the DCG-LZ metric is computationally cheaper than the DCG metric. With $k = 20$ and $\epsilon = 0.05$, i.e., up to five per cent of error, $\epsilon$-Filtering is 14 and 9 times faster than $\text{OPT}_{\text{DYN}}$ using the DCG and DCG-LZ metrics, respectively, and the speedup further improves of about 30% when increasing $\epsilon$ to 0.1, i.e., up to ten per cent of error. Therefore, $\epsilon$-Filtering provides a wide range of efficiency-effectiveness trade-offs, showing speedups of at least 5× over $\text{OPT}_{\text{DYN}}$ even in the most adverse conditions.

## 7 CONCLUSIONS

In this article, we presented two novel algorithms for solving the relevance-aware filtering problem, i.e., OPT-Filtering and $\epsilon$-Filtering. OPT-Filtering is an exact filtering algorithm that efficiently finds an optimal solution by using combinatorics and dynamic programming, while $\epsilon$-Filtering is an approximate filtering algorithm that further improves the time complexity of OPT-Filtering by using approximation to trade efficiency for effectiveness. Given an approximation threshold $\epsilon$, $\epsilon$-Filtering finds a (1-$\epsilon$)–optimal filtering, i.e., the relevance of its solution is at least (1-$\epsilon$) times the optimum. We proposed a comprehensive evaluation of OPT-Filtering and $\epsilon$-Filtering against three state-of-the-art competitors, i.e., $\text{OPT}_{\text{DYN}}$, Top$k$-OPT, and Cutoff-OPT, on two real-world public datasets. Experiments show that OPT-Filtering speeds up the filtering from 2× to 77× over the exact filtering $\text{OPT}_{\text{DYN}}$, while $\epsilon$-Filtering speeds up the filtering up to 244× also outperforming the heuristics. Furthermore, when decreasing the approximation threshold $\epsilon$ to values smaller than 0.01, $\epsilon$-Filtering finds the optimal solution on all tested queries while achieving from 5× to 177× speedup over $\text{OPT}_{\text{DYN}}$.

## 8 ACKNOWLEDGEMENTS

This work is partially supported by the “Algorithms, Data Structures and Combinatorics for Machine Learning” (MIUR-PRIN 2017) and by the OK-INSAID (MIUR-PON 2018, grant agreement Nº ARS01_00917) projects.

## REFERENCES

[1] Ioannis Arapakis, Xiao Bai, and Berkant Barla Cambazoglu. 2014. Impact of response latency on user behavior in web search. In *Proceedings of the 37th International ACM Conference on Research and Development in Information Retrieval (SIGIR).* 103–112.

[2] Ricardo A. Baeza-Yates and Berthier A. Ribeiro-Neto. 2011. *Modern Information Retrieval - the concepts and technology behind search, Second edition.* Pearson Education Ltd., Harlow, England.

[3] Berkant Barla Cambazoglu and Ricardo A. Baeza-Yates. 2011. Scalability Challenges in Web Search Engines. In *Advanced Topics in Information Retrieval*. Vol. 33. Springer, 27–50.

[4] David Carmel, Guy Halawi, Liane Lewin-Eytan, Yoelle Maarek, and Ariel Raviv. 2015. Rank by Time or by Relevance?: Revisiting Email Search. In *Proceedings of the 24th ACM International Conference on Information and Knowledge Management, (CIKM)*. ACM, 283–292.

[5] David Carmel, Liane Lewin-Eytan, Alex Libov, Yoelle Maarek, and Ariel Raviv. 2017. Promoting Relevant Results in Time-Ranked Mail Search. In *Proceedings of the 26th International ACM Conference on World Wide Web (WWW)*. 1551–1559.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition*. MIT Press.

[7] Dirk Van den Poel. 2012. Book Review: Ensemble Methods: Foundations and Algorithms. *IEEE Intell. Informatics Bull.* 13, 1 (2012), 33–34.

[8] Esra Ilbahar and Selçuk Çebi. 2017. Classification of design parameters for E-commerce websites: A novel fuzzy Kano approach. *Telematics Informatics* 34, 8 (2017), 1814–1825.

[9] Kalervo Järvelin and Jaana Kekäläinen. 2002. Cumulated gain-based evaluation of IR techniques. *ACM Transactions on Information Systems (TOIS)* 20, 4 (2002), 422–446.

[10] Myeongjae Jeon, Saehoon Kim, Seung-won Hwang, Yuxiong He, Sameh Elnikety, Alan L. Cox, and Scott Rixner. 2014. Predictive parallelization: taming tail latencies in web search. In *Proceedings of the 37th International ACM Conference on Research and Development in Information Retrieval (SIGIR)*. 253–262.

[11] Evangelos Kanoulas and Javed A. Aslam. 2009. Empirical justification of the gain and discount function for nDCG. In *Proceedings of the 18th International ACM Conference on Information and Knowledge Management (CIKM)*. 611–620.

[12] Ron Kohavi, Alex Deng, Brian Frasca, Toby Walker, Ya Xu, and Nils Pohlmann. 2013. Online controlled experiments at large scale. In *Proceedings of the 19th International ACM Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 1168–1176.

[13] Jimmy Lin, Yulu Wang, Miles Efron, and Garrick Sherman. 2014. Overview of the TREC-2014 Microblog Track. In *Proceedings of The Twenty-Third Text REtrieval Conference, (TREC) (NIST Special Publication)*, Vol. 500-308. National Institute of Standards and Technology (NIST).

[14] Tie-Yan Liu. 2009. Learning to Rank for Information Retrieval. *Foundations and Trends in Information Retrieval* 3, 3 (2009), 225–331.

[15] Julian J. McAuley, Christopher Targett, Qinfeng Shi, and Anton van den Hengel. 2015. Image-Based Recommendations on Styles and Substitutes. In *Proceedings of the 38th International ACM Conference on Research and Development in Information Retrieval (SIGIR)*. 43–52.

[16] Alistair Moffat and Justin Zobel. 2008. Rank-biased precision for measurement of retrieval effectiveness. *ACM Trans. Inf. Syst.* 27, 1 (2008), 2:1–2:27. https://doi.org/10.1145/1416950.1416952

[17] Franco Maria Nardini, Roberto Trani, and Rossano Venturini. 2019. Fast Approximate Filtering of Search Results Sorted by Attribute. In *Proceedings of the 42nd International ACM Conference on Research and Development in Information Retrieval (SIGIR)*. 815–824.

[18] Rajiv Pasricha and Julian J. McAuley. 2018. Translation-based factorization machines for sequential recommendation. In *Proceedings of the 12th International ACM Conference on Recommender Systems (RecSys)*. 63–71.

[19] Mr Biraj Patel and Dr Dipti Shah. 2011. Meta search ranking strategies. *International Journal of Information and Computing Technology (ICT)* 976, 5999 (2011), 24–25.

[20] Tao Qin, Tie-Yan Liu, Jun Xu, and Hang Li. 2010. LETOR: A benchmark collection for research on learning to rank for information retrieval. *Information Retrieval* 13, 4 (2010), 346–374.

[21] Nikita V. Spirin, Mikhail P. Kuznetsov, Julia Kiseleva, Yaroslav V. Spirin, and Pavel A. Izhutov. 2015. Relevance-aware Filtering of Tuples Sorted by an Attribute Value via Direct Optimization of Search Quality Metrics. In *Proceedings of the 38th International ACM Conference on Research and Development in Information Retrieval (SIGIR)*. 979–982.

[22] Andrew Trotman, Surya Kallumadi, and Jon Degenhardt. 2018. High Accuracy Recall Task. In *The SIGIR 2018 Workshop On eCommerce co-located with the 41st International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR) (CEUR Workshop Proceedings)*, Vol. 2319. CEUR-WS.org.

[23] Aleksandr Vorobev, Aleksei Ustimenko, Gleb Gusev, and Pavel Serdyukov. 2019. Learning to select for a predefined ranking. In *Proceedings of the 36th International Conference on Machine Learning, (ICML) (Proceedings of Machine Learning Research)*, Vol. 97. PMLR, 6477–6486.

[24] Yining Wang, Liwei Wang, Yuanzhi Li, Di He, and Tie-Yan Liu. 2013. A Theoretical Analysis of NDCG Type Ranking Measures. In *Proceedings of the 26th Annual JMLR Conference on Learning Theory (COLT)*. 25–54.