The Sweet Spot between Inverted Indices and Metric-Space Indexing for Top-K–List Similarity Search*

Evica Milchevski
University of Kaiserslautern
Kaiserslautern, Germany
milchevski@cs.uni-kl.de

Avishek Anand
L3S Research Center and
University of Hannover
Hannover, Germany
anand@l3s.de

Sebastian Michel
University of Kaiserslautern
Kaiserslautern, Germany
smichel@cs.uni-kl.de

ABSTRACT

We consider the problem of processing similarity queries over a set of top-k rankings where the query ranking and the similarity threshold are provided at query time. Spearman's Footrule distance is used to compute the similarity between rankings, considering how well rankings agree on the positions (ranks) of ranked items (i.e., the L1 distance). This setup allows the application of metric index structures such as M- or BK-trees and, alternatively, enables the use of traditional inverted indices for retrieving rankings that overlap (in items) with the query. Although both techniques are reasonable, they come with individual drawbacks for our specific problem. In this paper, we propose a hybrid indexing strategy, which blends inverted indices and metric space indexing, resulting in a structure that resembles both indexing methods with tunable emphasis on one or the other. To find the sweet spot, we propose an assumption-lean but highly accurate (empirically validated) cost model through theoretical analysis. We further present optimizations to the inverted index component, for early termination and minimizing bookkeeping. The performance of the proposed algorithms, hybrid variants, and competitors is studied in a comprehensive evaluation using real-world benchmark data consisting of Web-search–result rankings and entity rankings based on Wikipedia.

1. INTRODUCTION

One common way to counter the information deluge is the formation of concise rankings that allow users and algorithms to effectively and efficiently inspect the best performing items within a certain category. Ranking schemes are used to impose an order between items—such as Google's PageRank or more traditional OLAP-style aggregation and ranking functions used in databases for business intelligence and other forms of insight-seeking analyses. Besides tangible facts and objective ranking schemes, rankings are often also crowd-sourced through mining user polls on the Web, in portals such as IMDB (for movie ratings) or rankopedia.com, or specifically created by users in form of favorite lists on personal websites or used in dating portals for matchmaking. A core characteristic of such rankings is that they are rather tiny in length, compared to the global domain of items that could be ranked—consider the top-10 movies of all time compared to the total number of produced movies or the top-10 Web search results for Britney Spears compared to more than 100 million documents about her in Google's index. Access to rankings can serve ad-hoc information demands or give access to deeper analytical insights. Consider for instance the task of query suggestion in web search engines that is based on finding historic queries by their result lists with respect to the currently issued query, or dating portals that let users create favorite lists that are used to search for similarly minded mates.

As a generic access substrate for such services, we consider querying sets of top-k rankings by means of distance functions. That is, retrieving all rankings that have a distance to the query less than or equal to a user-provided threshold. We specifically focus on Spearman's Footrule that is the L1 distance metric between two rankings. Fagin et al. [18] show that there is a metric Spearman's Footrule adaptation for top-k rankings, whose ranked items do not necessarily match or overlap at all. Dealing with metrics immediately suggests employing metric data structures like M-trees [14] for indexing and similarity search. On the other hand, similar rankings, for reasonable query thresholds, should in fact overlap in some (or all) of the items they rank. Searching overlapping sets for ad-hoc queries [22, 30] or joins [25] is a well studied research topic. Inverted indices or signature trees are used to indexing tuples based on their set-valued attributes [22]. Such indices are very efficient to answer contained-in, equal-to, or overlaps-with queries, but do not exploit the distances between the indexed rankings as metric index structures do. In this work, we study a hybrid index structure that smoothly blends an inverted index with metric space indexing. With an assumption-lean but highly accurate theoretic cost model, we further show that the estimated sweet spot reaches runtime performances almost identical to the manually tuned one.

1.1 Problem Statement and Setup

As **input** we are provided with a set $\mathcal{T}$ of rankings τ_i (Table 1). Each ranking has a domain D_{τ_i} of items it contains. We consider fixed-length rankings of size k, i.e., $|D_{\tau_i}| = k$, but investigate the impact of various choices of k on the

*This work has been partially supported by the Excellence Cluster on Multimodal Computing and Interaction (MMCI) and the German Research Foundation (DFG) under grant MI 1794/1-1.

$\mathcal{T}$	
ranking id	**ranking content**
τ_1	$[2, 5, 4, 3]$
τ_2	$[1, 4, 5, 9]$
τ_3	$[0, 8, 5, 7]$

Table 1: Sample set $\mathcal{T}$ of rankings (items are represented by their ids).

query performance. The considered rankings do not contain any duplicate items.

Rankings are represented as arrays or lists of items, where the left-most position denotes the top ranked item. Without loss of generality, in the remainder of the paper, we assume that items are represented by their ids. The rank of an item i in a ranking τ is given as $\tau(i)$.

A distance function d quantifies the distance between two rankings—the larger the distance the less similar the rankings are. Therefore, for a given query ranking q, distance function d, and distance threshold θ, we want to find all rankings in $\mathcal{T}$ with distance below or equal to θ, that is,

$$\{\tau_i | \tau_i \in \mathcal{T} \wedge d(\tau_i, q) \leq \theta\}$$

In this work, we focus on the computation of Spearman's Footrule distance, but the proposed coarse index can be applied to any metric distance function. A more detailed introduction to rankings specifically top-k rankings, metric distance functions, and how to work with items i that are not in a ranking τ is described in Section 3.

The objective of this work is to study in-memory indexing and query processing techniques, with the overall aim to decrease the average query response time. We consider adhoc similarity queries over rankings, where the query ranking and query similarity threshold are specified at query time.

1.2 Contributions and Outline

In this work, we make the following contributions:

- we present a coarse index and a cost model that allows automated tuning of the coarsening threshold for optimal performance
- we derive distance bounds for early stopping / pruning inside position-augmented inverted indices—concepts that are largely orthogonal to each other and can be combined
- we show the results of a carefully conducted experimental evaluation involving a suite of algorithms and hybrids under realistic workloads derived from realworld rankings

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 presents background information on rankings and discusses distance functions for rankings. Section 4 introduces a coarse, hybrid index that indexes partitions of rankings. Section 5 describes a cost model that allows picking the sweet spot between inverted-index-access time and result-validation time. Section 6 shows how to compute distance bounds and to enable effective pruning of entire index lists at runtime. Section 7 presents the experimental evaluation, while Section 8 concludes the paper.

2. RELATED WORK

There is an ample work on computing relatedness between ranked lists of items, such as to mine correlations or anti-correlations between lists ranked by different attributes; like age and weight. Arguably, the two most prominent similarity measures are Kendall's tau and Spearman's Footrule. Fagin et al. [18] study comparing top-k lists, that is, lists capturing a subset of a global set of items, rendering the lists incomplete in nature. In the scenarios motivating our work, like similarity search of favorite/preference rankings, the lists are naturally incomplete, capturing, e.g., only the top-10 movies of all times. In this work, we focus on the computation of Spearman's Footrule distance, for which Fagin et al. [18] show that it retains its metric properties also for incomplete rankings under certain assumptions (cf., Section 3).

We primarily distinguish two indexing paradigms for handling ranked lists. First, considering the similarity metric among them and applying indexing techniques for metric spaces. Second, treating ranked lists as plain sets and indexing them using methods like inverted indices.

Helmer and Moerkotte [22] present a study on indexing set-valued attributes as they appear for instance in object-oriented databases. Retrieval is done based on the query's items; the result is a set of candidate rankings, for which the distance function can be computed. For metric spaces, data-agnostic structures for indexing objects are known, like the M-tree by Ciaccia et al. [14, 37]. For discrete metrics, the tree structure proposed by Burkhard and Keller [10] resembles an n-ary search tree, called BK-tree, where subtrees group items according to their (discrete) distance to the parent node. Similarly, Ganti et al. [21] present single-pass algorithms for clustering data in metric distance space using a R*-tree–style [7] structure for mapping objects to (evolving) clusters. The vantage-point tree [32, 36] partitions the space by choosing vantage points (pivots) that segment the space into two areas, similar to the k-d tree [8]. Chávez and Navarro [12] describe an algorithm to create non-overlapping partitions of data in a metric space based on pivots and fixed-diameter or fixed-size partitions; several ways to choose pivots are studied. We consider indexing clusters of rankings to shrink the size of the inverted index, by considering partitions of rankings within a pre-determined distance threshold (Section 4)—effectively trading-off cluster retrieval time and final result validation cost. The partitioning can be done in any of the above ways; we choose the BK-tree [10]. The book by Hanan Samet [26] gives a comprehensive overview of indexing techniques for metric spaces. The recent work by Wang et al. [34] propose MapReduce [16] algorithms for all-pairs similarity search in metric spaces. Previously, Jacox and Samet [24] proposed sequential algorithms for the similarity join problem in metric spaces.

Augmenting the inverted index with rank information allows computing the Footrule distance on the fly. For score-ordered index lists used in top-k query processing, there is a large variety of work. Most prominently, the family of threshold algorithms [18] and variants like the work by Bast et al. [4] that is emphasizing on disk-I/O optimal access. For k nearest neighbor (KNN) or similarity queries, the per-dimension information of the indexed objects is not presorted by "score" as this depends on the query that is not known a priori. Work on KNN search in databases [9] transforms the KNN problem into a range query over the involved dimensions, that can be answered using standard database

τ	A ranking
$\tau(i)$	The rank of item i in ranking τ
$F(\tau_i, \tau_j)$	Footrule distance between τ_i and τ_j
$d(\tau_i, \tau_j)$	Distance between τ_i and τ_j
d_{max}	maximum distance between two rankings
$\mathcal{T}$	Set of rankings to be indexed
k	Size of rankings
$\mathcal{D}_\tau$	Items contained in ranking τ
$\mathcal{D}$	Global domain of items
q	Query ranking
θ	Similarity threshold, set at query time
θ_C	Maximum pairwise similarity within a partition
$\mathcal{P}_i$	A partition of rankings. Partitions are pairwise disjoint.

Table 2: Overview of notation used in this paper

indices that support range queries, like B+ trees [6]. Work on similarity join in databases [2, 27, 28] focuses on defining and implementing the similarity join as relational operators. Mamoulis [25] addresses processing joins between sets (relations) of tuples with set-valued attributes. Terrovitis et al. [29] considers containment queries over sets with skewed data distributions. The work in [30] proposes combination of trees and inverted files to answer superset, subset and equality queries over set-valued attributes. Recently, the most prominent technique for answering set similarity joins are the prefix-filtering based methods [35, 5, 11]. The main idea behind this method is to reduce the size of the inverted index. This is done by imposing a total ordering of the elements in the universe $\mathcal{U}$ (or what we refer to as the dictionary $\mathcal{D}$), sorting the elements in the records, and then, based on the threshold value, indexing only a prefix, and not the complete records. Similarly, we propose a technique for dropping some of the elements in the query (Section 6), however, our technique does not require the threshold value to be known during index construction. Additionally, we do not require a global ordering to be imposed on the items in the rankings, i.e., the rankings keep their original structure. Wang et al. [33] propose the AdaptJoin algorithm that improves on previous prefix filtering work by using variable length prefix scheme and a cost model that selects the most efficient prefix length for each object. They further propose the AdaptSearch algorithm for processing ad-hoc queries using the same adaptive framework. As rankings can also be seen as plain sets, AdaptSearch can be applied for computing relatedness between rankings as well.

When processing top-k, KNN, or similarity queries, the ultimate goal is to identify the final result objects as soon as possible, without exhaustive evaluation of scores/distances. In the NRA algorithm by Fagin et al. [19] over score-sorted index lists, this is achieved by maintaining score bounds for each seen object. We come back to this in Section 6.

3. BACKGROUND ON RANKINGS AND DISTANCE FUNCTIONS

Pairwise similar rankings can be retrieved by means of distance functions, like Kendall's Tau or Spearman's Footrule distance, over all pairs or selectively for a given query ranking. We first discuss metrics over complete rankings over a single domain and then we discuss results on computing distances for top-k lists (incomplete rankings).

Complete rankings are considered to be permutations over a fixed domain $\mathcal{D}$. We follow the notation by Fagin et al. [18] and references within. A permutation σ is a bijection from the domain $\mathcal{D} = \mathcal{D}_\sigma$ onto the set $[n] = \{1, \dots, n\}$. For a permutation σ, the value $\sigma(i)$ is interpreted as the rank of element i. An element i is said to be ahead of an element j in σ if $\sigma(i) < \sigma(j)$. For two permutations σ_1, σ_2 over the same domain, the Kendall's tau $K(\sigma_1, \sigma_2)$ and Spearman's Footrule $F(\sigma_1, \sigma_2)$ measures are two prominent ways to compute the distance between σ_1 and σ_2. Both measures are distance metrics, that is, they have symmetry property, i.e., $d(x, y) = d(y, x)$, are regular, i.e., $d(x, y) = 0$ iff $x = y$, and suffice the triangle inequality $d(x, z) \leq d(x, y) + d(y, z)$, for all x, y, z in the domain. Spearman's Footrule metric is the L_1 distance between two permutations, i.e., $F(\sigma_1, \sigma_2) = \sum_i |\sigma_1(i) - \sigma_2(i)|$ and in this work we specifically focus on this metric, but the proposed coarse index can be applied to any metric distance function. We refer the reader to Table 2 for an overview of the notation used in this paper.

We consider incomplete rankings, called top-k lists in [18]. Formally, a top-k list τ is a bijection from D_τ onto $[k]$. The key point is that individual top-k lists, say τ_1 and τ_2 do not necessarily share the same domain, i.e., $D_{\tau_1} \neq D_{\tau_2}$. Fagin et al. [18] discuss how the above two measures can be computed over top-k lists.

There exists a Spearman's Footrule adaptation that is also a metric for top-k lists by considering an artificial rank l for items not contained in a ranking, i.e., $\tau(i) = l$ if $i \notin D_\tau$. Consider the rankings $\tau_1 = [2, 5, 6, 4, 1], \tau_2 = [1, 4, 5]$, and $\tau_3 = [0, 8, 4, 5, 7]$. For a rank $l = 6$ for not-contained items, we obtain $F(\tau_1, \tau_2) = 15$, $F(\tau_2, \tau_3) = 17$, and $F(\tau_1, \tau_3) = 22$.

In this work, we assume that $\tau(i)$ takes values from 0 to $k - 1$ (instead of 1 to k), and we fix the value of l to k as suggested in [18]. It is clear that this does not affect our algorithms. We further consider only rankings of same size k, thus the largest possible value of the Footrule distance is $k \times (k + 1)$ and occurs if two disjoint rankings are compared. The smallest distance is 0, for the compared rankings are identical. In the rest of the paper, for ease of presentation, we use normalized values for the Footrule distance and θ, ranging from 0 to 1, i.e., $d_{max} = 1$.

4. FRAMEWORK

Rankings can be considered as plain sets and accordingly indexed in traditional inverted indices [22] that keep for each item a list of rankings in which the item appears. At query time such a structure allows efficiently finding those rankings that have one or more items in common with the query ranking. A compact example is given below:

$$\text{item a} \longrightarrow < \tau_1, \tau_5, \tau_7 > \qquad \textit{inverted index}$$

$$\text{item b} \longrightarrow < \tau_4, \tau_9, \tau_{12}, \tau_{19} >$$

The key point of using inverted indices is their ability to efficiently reduce the global amount of all rankings to potential candidates by eliminating the rankings with maximum distance d_{max} to the query. This is done in the first query processing phase, namely the **filtering phase**. In this phase, for a given query ranking q and a user defined threshold θ, the inverted index is queried for each item in $\mathcal{D}_q$. The obtained index lists are merged to identify all rankings that have at least one overlapping item with the query ranking q. These are considered candidates.

For each of them, the distance function $d(q, \tau)$ is evaluated to identify the true results, i.e., the rankings where $d(q, \tau) \leq \theta$. This is done in the **validation phase**. We refer to this as the Filter and Validate (F&V) algorithm. Naturally, we assume that the query threshold θ is strictly smaller than the maximum possible distance d_{max}.

Although the inverted index is good for finding rankings (sets) that intersect with the query, the F&V comes with two drawbacks:

(i) It naively indexes all rankings and, hence, is of massive size, despite the fact that often rankings are (near) duplicates

(ii) The validate phase evaluates the distance function on each ranking separately, although known metric index structures suggest pre-computing distances among (similar) rankings for faster identification of true results

While directly using metric index structures, like M-Trees [14] or BK-Trees [10], appears promising at first glance, they are not ideal for boiling down the space to intersecting rankings. In fact, we show in our experiments that using metric data structures is an order of magnitude slower than using pure inverted indexes.

To harness the pruning power of inverted indices but at the same time not to ignore the metric property of the Footrule distance, we present a hybrid approach that blends both performance sweet spots by representing near duplicate rankings by one representative ranking, which is then put into an inverted index. That way, depending on how aggressive this coarsening is, the inverted index drastically shrinks in size, hence, lower response time, and the validation step is benefiting from the fact that near duplicate rankings are represented by a metric index structure.

Below, we describe more formally how such an index organisation is realized and how queries are processed on top of it. We present a highly accurate cost model that allows trading-off the coarsening threshold to find the optimal trade-off between the inverted index cost and the cost to validate rankings in the metric index structure.

4.1 Index Creation

The aim is to group together rankings that are similar to each other—with a quantifiable bound on the maximum distance. That is, partitions $\mathcal{P}_i$ of similar rankings are created, and each represented by one $\tau_m \in \mathcal{P}_i$, the so called medoid of the partition. It is guaranteed that $\forall \tau_i \in \mathcal{P} : d(\tau_m, \tau) \leq \theta_C$. The distance bound θ_C is called the partitioning threshold. We write $\tau_m \prec \tau$ to denote that ranking τ is represented by ranking (medoid) τ_m.

To find partitions of rankings, we employ a BK-tree [10], an index structure for discrete metrics, such as the Footrule distance. Figure 1 depicts the general shape of such a BK-tree. Ignoring for a moment the different colors and black, solid circles: each node represents an object (here, ranking) and maintains pointers to subtrees whose root has a specific, discrete distance. We create such a BK-tree for the given rankings. Then, in order to create partitions of similar rankings, the tree is traversed and, for each node, the children with distance above θ_C are considered in different partitions. The procedure continues recursively on these children. The children within distance $\leq \theta_C$ are forming a partition with their root node, which acts as the medoid. In Figure 1, each partition is illustrated by its root (representative ranking) shown as a black, solid circle, and the green subtrees below it (those with distance 1 or 2). A partition is not represented as a plain set (or list) of rankings, but by the corresponding subtree of the BK-tree. The immediate benefit is that these subtrees (that are full-fledged BK-tree themselves) are used to process the original query (with threshold θ) on the clusters, without the need to perform an exhaustive evaluation of the partition's rankings. Alternatively, any algorithm that creates (disjoint) partitions of objects within a fixed distance bound can be used, such as the approach by Chávez and Navarro [12], which randomly picks medoids, assigns objects to medoids, and continues this procedure until no object is left unassigned. We use this simple model to reason about the trade-offs of our algorithm below.

Irrespective of the way to find medoids and their partitions, medoids are rankings, too, and can be indexed using inverted indices. In Section 6 we further propose techniques for more efficient retrieval of the rankings indexed with an inverted index.

4.2 Query Processing

LEMMA 1. *For given query threshold θ and partitioning threshold θ_C, at query time, for query ranking q, all medoids τ_m with distance $d(\tau_m, q) \leq \theta + \theta_C$ need to be retrieved in order not to miss a potential result ranking.*

Lemma 1 ensures that rankings $\{\tau_i | \tau_m \prec \tau_i \wedge d(\tau_i, q) \leq \theta \wedge d(\tau_m, q) > \theta\}$ will not be omitted from the result set. In other words, Lemma 1 avoids missing result rankings with distance $\leq \theta$, which are represented by a medoid with distance $> \theta$. On the other hand, since the medoids are indexed using an inverted index, we assume that $\theta + \theta_C < 1$. This is needed because medoids τ_m that are not overlapping with q at all, cannot be retrieved from the inverted index.

For each of the found medoids τ_m (i.e., $d(\tau_m, q) \leq \theta + \theta_C$), the rankings $\mathcal{R} := \{\tau | \tau_m \prec \tau\}$ are potential result rankings. For each such candidate ranking $\tau_i \in \mathcal{R}$ it needs to be checked if in fact $d(q, \tau_i) \leq \theta$. The rankings $\tau_i \in \mathcal{R}$ with $d(q, \tau_i) > \theta$ are so called *false positives* and according to Lemma 1 there are *no false negatives.* As for each affected medoid τ_m, the rankings in $\mathcal{R}$ are represented in form of a BK-tree (or any other metric index structure), it is the task of this tree to identify the true result rankings (i.e., eliminating the false positives).

Algorithm 1, depicts the querying using the relaxed query threshold, and the subsequent retrieval of result rankings. In this algorithm, as well as in the actual implementation, the partitions, represented by the medoids, are arranged as BK-trees, created at partitioning time.

It is clear that the partitioning threshold θ_C affects the cost for querying the metric index structure: The larger the partitions are (i.e., the larger θ_C is) the larger is the tree to be queried. On the other hand, then, there are less medoids to be indexed in the inverted index. This apparent tradeoff is theoretically investigated in the following section to find the design sweet spot between the naive inverted index and the case of indexing the entire set of rankings in one metric index structure.

```
method: **processCoarse**
input: QueryProcessor over Medoids qp, double $\theta$, $\theta_C$,
       Map:Int→ BK-Tree map
output: list of query results rlist
rTemp ← qp.execute($\theta$+$\theta_C$) ▷ query with relaxed threshold
for each id ∈ rTemp
    tree ← map[id]
    rList.addAll(tree.execute($\theta$))
return rList
```

Algorithm 1: Query processing using the coarse index.

5. PARAMETER TUNING

Setting the clustering threshold θ_C allows tuning the performance of the coarse index. For a clustering threshold $\theta_C = 0$, only duplicate rankings are grouped together, whereas for $\theta_C = 1$ there is only one large group that consists of all rankings. That means, for larger θ_C the inverted index becomes smaller, with more work to be done at validation time inside the retrieved clusters. For smaller θ_C the inverted index is larger, but clusters are smaller, hence, less work to be done in the validation phase. There are, hence, two separate costs: **filtering cost**—the cost for querying the inverted index, and, **validation cost**—the cost for validating the partitions represented by the medoids returned as results by the inverted index, in order to get the final query answers.

We try to make as few assumptions as possible and for now we assume we know only the distribution of pairwise distances. That is, for a random variable X that represents the distance between two rankings, we know the cumulative distribution function $P[X \leq x]$, hence, we know how many rankings of a population of n rankings are expected to be within a distance radius r of any ranking, i.e., $n \times P[X \leq \theta_C]$. We assume that medoids are also just rankings (by design) and are accordingly distributed. According to the clustering method described by Chávez and Navarro [12], we randomly select medoids, one after the other. After each selected medoid, all rankings that are not yet assigned to any medoid before and that are within distance θ_C to the current medoid are assigned to it. The process ends as soon as no ranking is left unassigned:

The radius r of the created partitions around the medoids is modeled as $P[X \leq \theta_C]$. We are interested in the number of medoids that need to be created to capture all rankings in the database. This resembles the *coupon collector problem* [20]. The solution to this problem describes how many coupons a collector needs to buy, in expectation, to capture all distinct coupons available. The first acquired coupon is unique with probability 1. The second pick is not seen before with probability $(c-1)/c$; c denoting the total number of distinct coupons. The third pick with probability $(c-2)/c$, and so on. In the case of medoids and their partitions, we specifically consider the variant of the coupon collector problem with package size larger or equal to one, i.e., batches of coupons are acquired together. Within each such package, there are no duplicate coupons. Figure 2 depicts the generic sampling of the ranking space, where fixed-diameter circles are forming the partitions around the medoid at the center. The deviation from the standard coupon collectors problem is that for picking medoids, in each round of picks, the medoid itself has not been selected before. Thus, the number

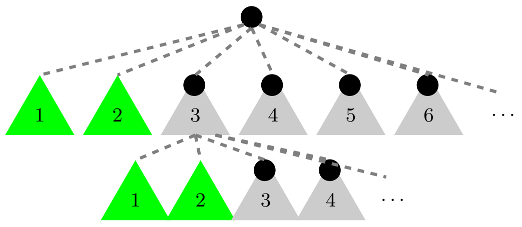

Figure 1: Creating partitions based on the BK-tree. The green (distance 1 and 2) subtrees are indexed by their parent node (medoid, as black dot). Distance 0 is not shown here.

of "coupons" that need to be acquired to get the i^{th} distinct coupon, given package size $p = P[X \leq \theta_C] \times n$, and a total of c distinct coupons, which in our case is the number of distinct rankings n is then:

$$h(n,i,p) = \begin{cases} 1, & \text{if } i \bmod p = 0 \\ \frac{n-(i \bmod p)}{n-i}, & \text{otherwise} \end{cases} \tag{1}$$

And overall, the number of medoids (packages) is given as

$$M(n,\theta_C) = p^{-1}\sum_{i=0}^{n-1} h(n,i,p) \tag{2}$$

This gives us the expected number of medoids indexed by the inverted index. Next, we first reason about the cost for validating the partitions, and then we discuss the filtering cost, i.e., the cost for querying the inverted index.

Cost for Validating Partitions

The number of medoids retrieved is following again the given distribution of pairwise distances. Since we query the inverted index with threshold $\theta + \theta_C$ we obtain

$$E[\textit{retrieved medoids}] = P[X \leq \theta + \theta_C] \times M \tag{3}$$

where M, for brevity, denotes $M(n,\theta_C)$.

Assuming that the retrieved medoids have the same size on average, i.e., n/M for a total number of rankings n, we have

$$E[\textit{candidate rankings}] = P[X \leq \theta + \theta_C] \times n \tag{4}$$

candidate rankings retrieved that need to be checked against the distance to the query ranking. This is also very intuitive.

For the case of brute-force evaluation of such candidate rankings this is multiplied with the cost of computing the distance measure. The cost of representing the partitions by full-fledged BK-tree is expected to be lower, but it introduces a complexity to the model. Our goal is to provide an easy to compute, and yet accurate model. For a more complex reasoning about the cost of querying the BK-tree we refer the reader to [3].

Cost for Retrieving Partitions

When querying the inverted index with a threshold $\theta + \theta_C$ to find the resulting medoids, the overall cost is based on the average index list length and the final medoids to be checked against the threshold. We should first estimate the average size of an index list in an inverted index.

We assume that the popularity of items in the rankings follows Zipf's law with parameter s. Sorting all items by their popularity (frequency of appearance in the rankings),

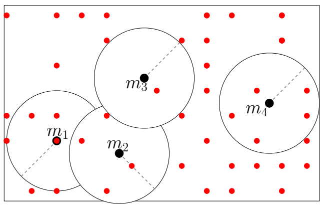

Figure 2: Four medoids with fixed-diameter partitions.

the law states that the frequency of the item at rank i is given by $f(i; s, v) = \frac{1}{i^s H_{v,s}}$, where $H_{v,s}$ is the generalized harmonic number and v is the total number of items. The size of the index list for an item is equal to the number of rankings that contain the item, i.e., $n \times f(i; s, v)$ for the i^{th} most popular item; where n is the number of indexed rankings. Consider a random variable Y representing sizes y_i of index lists for items i. We are interested in $E[Y] = \sum_i y_i \, P[y_i]$ and assume that the chance of item i, that is the i^{th} most popular item, to be selected as a query item is following the same Zipf distribution, $f(k; s, v)$. That means, the items appearing frequently in the data are also used often in the queries. The average size of an index list is then given as $E[Y] = \sum_i n \times f(i; s, v)^2$. This is a generic result for inverted indices, which in our cost model is applied on an inverted index over M medoids (not n rankings) that together have v' distinct items; v' is derived thereafter, so the expected length of an inverted list for the inverted index is:

$$E[\textit{index list length}] = \sum_i M \times f(i; s, v')^2 \tag{5}$$

For each query, k such index lists need to be accessed. This is one part of the cost caused by the retrieval of the medoids. For these $k \times E[\textit{index list length}]$ medoids, we have to compute the distance function, assuming that there are no duplicate medoids retrieved.

The expectation of distinct items v' within the medoids is derived as follows. The probability that an item, out of a global domain of v items, is not selected into a single ranking of size k is $(\frac{v-1}{v})^k$, but we do know that a ranking does not contain duplicate items, hence, $P[\neg \textit{selected}] = \frac{v-1}{v} \times \frac{v-2}{v-1} \cdots \frac{v-k}{v-k+1} = \Pi_{i=0}^{k} \frac{v-i}{v-i+1} = 1 - (\frac{k}{v})$. The probability that an item, out of a global domain of v items, is not selected into a single ranking of size k, knowing that the items in the ranking are unique, is $P[\neg \textit{selected}] = 1 - (\frac{k}{v})$. The probability *not* to be selected in *any* of the M medoid rankings is then $(1 - \frac{k}{v})^M$. And thus

$$E[v'] = v \times \left(1 - \left(1 - \frac{k}{v}\right)^M\right) \tag{6}$$

To compute the overall cost, the above estimates are combined as shown in Table 3. To bring both parts of the overall cost to a comparable unit, we precompute the cost (runtime) of a single Footrule computation $Cost_{Footrule}(k)$ (for various k) and the cost (runtime) to merge k lists of a certain size, $Cost_{merge}(k, size)$.

Figure 3 shows the model for vary θ_C for the two datasets used in the experimental evaluation (we refer the reader to Section 7 for a description of the datasets). We empirically estimated the skewness parameter s from samples of the

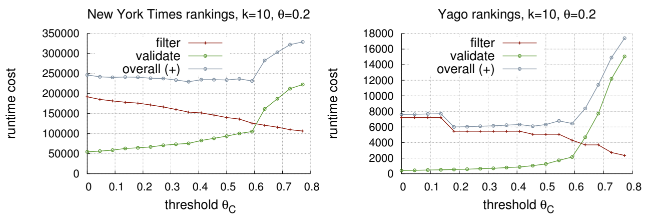

Figure 3: The behavior of the theoretically derived performance for varying θ_C.

datasets—$s = 0.87$ for the New York Times dataset (left plot) and $s = 0.53$ for the Yago dataset (right plot)—and fitted it in the above estimate of the expected index list length.

Find medoids for query:	
Inv. Index Cost:	$Cost_{merge}(k, \sum_i f(i; s, v')^2 \times M)$
	$+$
Validation Cost:	$k \left(\sum_i f(i; s, v')^2 \times M\right) \times Cost_{Footrule}(k)$
Validation of retrieved rankings:	
Validation Cost:	$n \times P[X \leq \theta + \theta_C] \times Cost_{Footrule}(k)$

Table 3: Model of query performance (∼runtime) of the coarse index.

6. INV. INDEX ACCESS & OPTIMIZATIONS

Medoids are rankings as well and thus they can be indexed using inverted indices. In this section, two optimizations over inverted indices are presented.

First, a minimum-overlap criterion is derived; it indicates how many of the k index lists can be dropped from consideration, guaranteeing that no true result ranking can possibly be missed.

For the second optimization, for a query ranking of size k, the k corresponding index lists are accessed one after the other, and the contained information in each list in the form of $(\tau_i, \tau(i))$ are continuously aggregated for each (seen) ranking. For each ranking observed during accessing the index lists, upper and lower bounds for the true distance are derived, to allow accepting or rejecting final result rankings early.

6.1 Pruning by Query-Ranking Overlap

Consider a ranking τ with $\mathcal{D}_\tau \cap \mathcal{D}_q = \emptyset$, i.e., the items in τ are not at all overlapping with the query's items. It is easy to see that the Footrule distance is $F(\tau, q) = k \times (k+1) = L(k)$, considering rankings of length k. $L(k)$ is used to denote this lowest possible distance[1]. In the case of zero overlap, $L(k)$ is also the exact distance. In general, considering an overlap of size ω between $\mathcal{D}_q$ and $\mathcal{D}_\tau$, the smallest possible Footrule distance $L(k, \omega)$ in that case is given when the ω overlapping items are perfectly matched and positioned in the top of both lists, hence, $L(k, \omega) = L(k - \omega)$. For a given query threshold θ, rankings with an overlap of ω items can be safely ignored if $L(k, \omega) > \theta$. In practice, this means that some index lists can be entirely omitted from being accessed.

[1]We use the naming lower and upper bounds for distances instead of best and worst distances, for clarity.

It is immediately clear how to turn this insight into enhancements of algorithms that work with an inverted index: Solving $L(k, \omega) = \theta$ for ω tells that rankings τ with $F(\tau, q) \leq \theta$ must not have an overlap smaller than $\omega = \lfloor 0.5(1 + 2k - \sqrt{1 + 4\theta}) \rfloor$. From this it immediately follows that $k - \omega + 1$ index lists are sufficient to retrieve all the candidate lists since a ranking missing from these lists must have an overlap smaller than ω.

If we further take into consideration the position of the ω overlapping items, i.e., that they are positioned at the top of both lists, then we can ensure correctness by retrieving $k-\omega$ lists if at least one of the retrieved lists is of an item positioned in the top ω places. In this case, we can miss rankings that have an overlap of ω with the query, but we will never miss rankings that have overlap of ω where these ω items are positioned at the top ω places. This leads immediately to the following lemma.

LEMMA 2. *For given query threshold θ and ranking size k, $k - \omega$ index lists are sufficient to retrieve all the candidate rankings τ with $F(\tau, q) \leq \theta$, where $\omega = \lfloor 0.5(1 + 2k - \sqrt{1 + 4\theta}) \rfloor$.*

This is a generic result, independent of the actual choice of the index lists that can be dropped. Still, the expected impact of the candidate pruning is larger if the largest lists are dropped. In fact, experiments will show that specifically for the query-log–based benchmark, drastic performance gains can be enjoyed, literally for free. For the remaining lists and rankings within, the exact distance still needs to be determined, as there are obviously so called *false positives* with distance larger than the query threshold. But the above lemma guarantees that there are *no false negatives*, i.e., no ranking τ with $F(\tau, q) \leq \theta$ is missed.

Algorithms that make use of this dropping of entire index lists carry the suffix **+Drop** in the title.

6.2 Partial Information

Instead of having only ranking ids stored in the inverted index, such that an additional lookup is required to get the actual ranking content, we can augment the inverted index to make it hold the rank information as well, such that the true distance can be directly computed.

inverted index w/ ranks

item a $\longrightarrow$ $<(\tau_1 : 3),(\tau_5 : 1),(\tau_7 : 4)>$

item b $\longrightarrow$ $<(\tau_4 : 2),(\tau_9 : 11),(\tau_{12} : 1),(\tau_{19} : 2)>$

In a **List-at-a-Time** fashion, the individual index lists determined by the query are accessed one after the other. Similarly to the NRA algorithm by Fagin et al. [19], for a ranking τ that has been seen only in a subset of the index lists, we can compute bounds for its final distance. This is done by keeping track of the common elements seen between the query q and ranking τ. The lower and upper bounds are computed by reasoning about the yet unseen elements: A lower bound distance $L(\tau, q)$ is given by assuming the best configuration of the unseen elements, that is, the remaining elements are common to both q and τ, and are additionally present in the same ranks in both rankings. Thus, their partial contribution to the Footrule distance is zero.

The upper bound distance $U(\tau, q)$ is obtained when none of the (yet) unseen elements in τ will be present in the query q. The partial distance contribution of such an item i, at rank $\tau(i)$ in τ is $|k - \tau(i)|$, and overall we have

$\tau_0 = [1, 2, 3, 4, 5]$	$\tau_5 = [4, 5, 1, 2, 3]$
$\tau_1 = [1, 2, 9, 8, 3]$	$\tau_6 = [1, 6, 2, 3, 7]$
$\tau_2 = [9, 8, 1, 2, 4]$	$\tau_7 = [7, 1, 6, 5, 2]$
$\tau_3 = [7, 1, 9, 4, 5]$	$\tau_8 = [2, 5, 9, 8, 1]$
$\tau_4 = [6, 1, 5, 2, 3]$	$\tau_9 = [6, 3, 2, 1, 4]$

Table 4: Sample set $\mathcal{T}$ of rankings

$$U(\tau, q) = L(\tau, q) + \sum_{i\,unseen} |k - \tau(i)|$$

The bounds allow pruning of candidates: If $L(\tau, q) > \theta$ we know that τ is not a result ranking, since $L(\tau, q)$ is monotonically non-decreasing. Similarly, if $U(\tau, q) \leq \theta$, we report τ as the result, as $U(\tau, q)$ is monotonically non-increasing. For small values of θ, many candidates can be evicted early on in the execution phase. For larger values of θ, candidate results can be reported early—reducing bookkeeping costs.

Consider for instance the set $\mathcal{T}$ of the rankings presented in Table 4 and a query $q = [7, 6, 3, 9, 5]$. The index list for item 7 is:

item 7 $\longrightarrow$ $<(\tau_3 : 0),(\tau_6 : 4),(\tau_7 : 0)>$

We can compute the bounds for the seen rankings, τ_3, τ_6, and τ_7. For all these rankings, we know the seen element is item 7 and we have 4 unseen elements, since $k = 5$. Thus, $L(\tau_3, q) = L(\tau_7, q) = 0$ and $L(\tau_6, q) = 4$, as $\tau_3(7) - q(7) = \tau_7(7) - q(7) = 0$, and $\tau_6(7) - q(7) = 4$ and for the unseen items we assume they are on the same position in all rankings. $U(\tau_3, q) = U(\tau_7, q) = 20$ and $U(\tau_6, q) = 24$, as we assume that all of the unseen elements are not present in τ_3, τ_6, and τ_7.

These distance bounds are used in the following online aggregation algorithm that encounters partial information. Algorithms that make use of this pruning for partial information carry the suffix **+Prune** in their title.

6.3 Blocked Access on Index Lists

When index lists are ordered according to the rank values, since the ranks are integers, there might be a sequence of index lists whose ranks are the same. We refer to this sequence of index lists as a block of index lists. Formally, we let the block $\mathcal{B}_{i@j}$ to denote the set of rankings in which item i appears at position j. We additionally have a secondary index, one for each index list, which stores the offsets of the individual blocks.

The advantage with such an index list organization strategy is that processing the entire index list can be avoided in many cases. We describe this in detail. It is obvious that result candidates which have a partial distance greater than θ can be pruned out. In such an index organization approach, we avoid processing blocks which would produce candidates with a partial distance greater than θ. Given a query $q = [q_1, \ldots, q_k]$ with a threshold θ, all result candidates obtained while traversing the block $\mathcal{B}_{i@j}$ have a partial distance of at least $|j - i|$. Thus, we modify the List-at-a-Time algorithm so that blocks, $\mathcal{B}_{i@j}$, where $|j - i| > \theta$ are omitted, avoiding processing the bulk of the index list.

Consider for instance the inverted index in Figure 4, constructed according to the rankings in Table 4. For the query $q = [3, 2, 1]$ and $\theta = 1$, blocks $\mathcal{B}_{3,1}$ need to be accessed for item 3, $\mathcal{B}_{2,1}$, $\mathcal{B}_{2,1}$ and $\mathcal{B}_{2,3}$ for item 2. Finally, blocks $\mathcal{B}_{1,2}$, $\mathcal{B}_{1,3}$ and $\mathcal{B}_{1,4}$ for item 1. In the process 17 out of 28 index item 1$\rightarrow \boxed{(\tau_0:0), (\tau_1:0), (\tau_6:0)}, \boxed{(\tau_3:1), (\tau_4:1), (\tau_7:1), (\tau_{10}:1)}, \boxed{(\tau_2:2), (\tau_5:2)}, \boxed{(\tau_9:3)}, \boxed{(\tau_8:4)}$

item 2$\rightarrow \boxed{(\tau_8:0)}, \boxed{(\tau_0:1), (\tau_1:1)}, \boxed{(\tau_6:2), (\tau_9:2)}, \boxed{(\tau_2:3), (\tau_4:3), (\tau_5:3), (\tau_{10}:3)}, \boxed{(\tau_7:4)}$

item 3$\rightarrow \boxed{(\tau_9:1)}, \boxed{(\tau_0:2)}, \boxed{(\tau_6:3)}, \boxed{(\tau_1:4), (\tau_4:4), (\tau_5:4)}$

item 4$\rightarrow \boxed{(\tau_5:0)}, \boxed{(\tau_{10}:2)}, \boxed{(\tau_0:3), (\tau_3:3)}, \boxed{(\tau_2:4), (\tau_9:4)}$

...

Figure 4: Inverted Index for rankings in Table 4 with highlighted blocks of same-rank entries

lists are processed which accounts for less than 50% index lists being accessed.

7. EXPERIMENTS

We implemented the described algorithms in Java 1.7 and report on the setup and results of an experimental study. The experiments are conducted on a quad-core Intel Xeon W3520 @ 2.67GHz machine (256KiB, 1MiB, 8MiB for L1, L2, L3 cache, respectively) with 24GB DDR3 1066 MHz (0.9 ns) main memory.

Datasets

Yago Entity Rankings: We have mined top-k entity rankings out of the Yago knowledge base, as described in [23]. The facts, in form of subject/predicate/object triples, are used to define constraints, for which the qualifying entities are ranked according to certain criteria. For instance, we generate rankings by focusing on type building and predicate located in New York, ranked by height. This dataset, in total, has 25,000 rankings.

NYT: We executed 1 million keyword queries, randomly selected out of a published query log of a large US Internet provider, against the New York Times archive [31] using a unigram language model with Dirichlet smoothing as a scoring model. Each query together with the resulting documents represents one ranking.

The two datasets are naturally very different: while the Yago dataset features real world entities that each occur in few rankings, the NYT dataset has many popular documents that appear in many query-result rankings.

Algorithms under Investigation

- the baseline approaches Filter and Validate (**F&V**) and Merge of Id-Sorted Lists (**ListMerge**) both described below
- filter and validate technique combined with the optimization based on dropping entire index lists (**F&V+ Drop**)
- blocked access with pruning (**Blocked+Prune**)
- blocked access with pruning based on both overlap and pruning (**Blocked+Prune+Drop**)
- query processing on the coarse index using the F&V technique (**Coarse**)
- query processing on the coarse index using the F&V+ Drop technique (**Coarse+Drop**)
- a competitor **AdaptSearch**, and **Minimal F&V** algorithm, both described below

Next to the actual algorithms, we implemented a minimal Filter and Validate algorithm (**Minimal F&V**) that has for each query materialized a single index list in an inverted index that contains exactly the true query-result rankings. For each of these, the Footrule distance is computed. The cost for the single index lookup and the Footrule computations serves as a lower bound for the performances of the discussed algorithms.

We also implemented **AdaptSearch** [33] as the most recent and competitive work on ad-hoc set similarity search in main memory. We implemented AdaptSearch by following the C++ implementation of the AdaptJoin algorithm available online[2]. We computed the size of the prefix of the query using the overlap threshold ω derived in Section 6. In the validation phase, AdaptSearch computes the Footrule distance for each of the candidate rankings.

The implementation of the M-tree is obtained from [15]. We implemented the BK-tree ourselves, according to the original work in [10]. The inverted index implementations make use of the Trove library[3].

Merge of Id-Sorted Lists with Aggregation: If the information within each index list is sorted by ranking id, and further contains rank information, the problem of computing the actual distances of the rankings to the query ranking can be achieved using a classical merge "join" of id-sorted lists. This is very efficient, in particular as the index lists do not contain any duplicates. Cursors are opened to each of the lists, and the distances of each ranking is finalized on the fly. There is no bookkeeping required as, at any time, only one ranking is under investigation (the one with the lowest id, if sorted in increasing order). Rankings do either qualify the query threshold or not. It is clear that this algorithm is threshold-agnostic, that is, its performance is not influenced by the query threshold θ; the index lists have to be read entirely.

We mainly focus on rankings of size 10 since in a previous study [1] we observed that at ranker.com most common are rankings of size 10.

Performance Measures

- Wallclock time: For all algorithms we measure the wallclock time needed for processing 1000 queries.
- Distance function calls: For the filter&validate algorithms, specifically F&V, F&V+Drop, Blocked+Prune+ Drop, Coarse, and Coarse+Drop, we measure the number of distance function computations performed.

For the coarse index processing techniques, we also investigate the performance of the individual phases.

7.1 Query Processing Performance

Inverted Index vs. Metric Index Structures

We first compare the two main concepts of processing similarity queries over top-k rankings: First, the use of met-

[2]https://github.com/sunlight07/similarityjoin

[3]http://trove.starlight-systems.com/

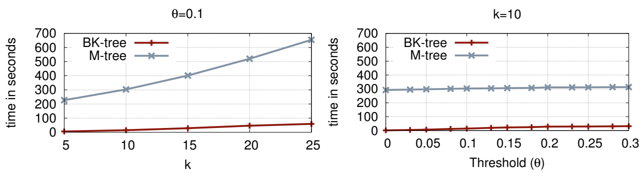

Figure 5: Performance of the M-tree vs. BK-tree (NYT)

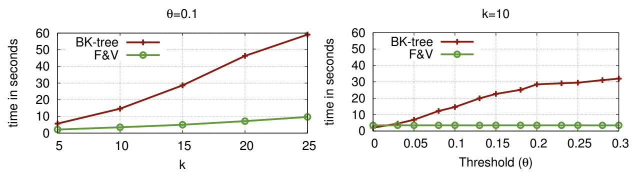

Figure 6: Performance of the BK-tree vs. the performance of inverted index (NYT)

ric index structures is compared, here, represented by the BK-tree and the M-tree [37] (Figure 5). Second, the use of inverted indices is compared to the BK-tree (Figure 6).

Figure 5 reports the query performance of the BK-tree compared to the M-tree and Figure 6 on the query performance of the BK-tree index structure versus the plain query processing using the inverted index with subsequent validation, i.e., filter and validate, F&V. We see that the inverted index performs orders of magnitudes better than the M-tree. Although the M-tree is a balanced index structure it still performs worse than the BK-tree. Chávez et al. [13] show that balanced index structures perform worse than unbalanced ones in high dimensions—we calculated the intrinsic dimensionality of both datasets to be around 13 (cf. [13] for the definition of intrinsic dimensionality). Despite the better performance of the BK-tree, the inverted index still outperforms it. Hence, only techniques using the inverted index paradigm are further studied.

Coarse Index Performance Based on θ_C

Next, we studied the performance of the coarse index for different θ_C values. We focus on the performance of the coarse index combined with the F&V technique as this combination resembles the model presented in Section 4 most. In Figure 7, the filtering and validation times are shown when varying θ_C and fixed $k = 10$, for both datasets. We see that the curves resemble the ones plotted for the cost model in Figure 3. Both dataset show a similar behavior of the execution time. The filtering time is reducing as we increase the value of θ_C, since the number of indexed medoids reduces. The validation time, on the other hand, is rising, since the size of the partitions is increasing proportionally with θ_C. Most importantly, we see that we can find a specific value of θ_C for which the coarse index performs optimally and this value depends on the value of $\theta + \theta_C$ as modeled in Section 4.

Cost Model Correctness

The performance of the coarse index if the trade-off value of θ_c as computed by the model is chosen, is shown in the plots in Figure 7 as a small rectangle. The vertical line denotes the difference between the performance of the coarse index in case of the two trade-off θ_c values—the modeled optimal one and the real optimal one. We observe that except for $\theta = 0.1$, for the NYT dataset, the difference in performance

	$\theta = 0.1$	$\theta = 0.2$	$\theta = 0.3$
NYT	29.47	10.23	4.75
Yago	3.28	0.41	2.38

Table 5: Difference in ms between the minimal performance of the coarse index, and the performance for the theoretically computed best value of θ_c ($k = 10$)

is smaller than 11ms (Table 5). For $\theta = 0.1$ the difference is 29.47ms. For the Yago dataset, the difference in performance is less than 4ms for any value of θ.

As we are considering the task of processing ad-hoc queries, even choosing the optimal value of θ_C for some previously defined maximum value of θ would result in a performance close to the optimal one, as the performance of the coarse index remains stable in this region. The major increase in the performance happens for very small values of θ_C or larger than the optimal θ_C. We show this in the experiments comparing different algorithms, where we set $\theta_C = 0.5$—the optimal value for $\theta = 0.3$.

We also measure the performance of the coarse index combined with the F&V+Drop technique as this should result in even bigger performance gains. For this technique, we measured the optimal value for θ_C to be 0.06, since for smaller values of $\theta + \theta_C$ we can drop more index lists.

Comparison of Different Algorithms

Next, we study the performance of different query processing methods performed over the two datasets; for rankings of size 10 and 20 and θ ranging from 0 to 0.3. First, in Figure 8 we compare the performance of the coarse index with the remaining techniques, for the NYT dataset. For a better visibility, we group the algorithms in the plots in two groups. The first (left) group contains the Coarse and Coarse+Drop techniques, the two baseline approaches F&V and ListMerge, and the competitors AdaptSearch and Minimal F&V. The second (right) group contains the remaining hybrid techniques.

We see that for all threshold values the coarse index, with and without dropping index list, significantly outperforms the AdaptSearch algorithm. In fact, the Coarse+Drop index outperforms the competitor by at least factor of 34. The coarse index outperforms the Minimal F&V technique by a factor of up to 7, since the number of Footrule distance function calls reduces significantly as shown in Figure 10. Dropping entire lists from the query even further boosts the performance of the coarse index, and results in up to 24 times better performance than the Minimal F&V. The baseline approaches, although threshold agnostic, perform worse than the rest of the algorithms. Increasing the values of θ degrades the performance of all the processing techniques except for the baseline F&V and ListMerge techniques, as they are threshold agnostic. In fact, because of its simple and efficient implementation, the ListMerge even outperforms the AdaptSearch algorithm for $\theta \geq 0.1$ for rankings with $k = 10$. For $k = 20$, since we increase the number of lists that need to be merged, the performance of the ListMerge is worse and thus the AdaptSearch outperforms it for all values of θ.

For rankings of size 10, all hybrid techniques outperform AdaptSearch, but not the coarse index. The Blocked+Prune algorithm dynamically computes the best score for the yet unseen blocks to decide when to terminate further schedul-

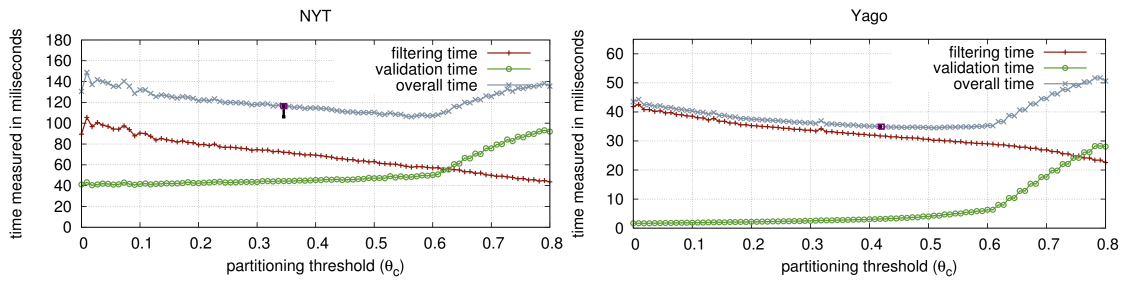

Figure 7: Trend of the filtering and validation time of the coarse index for $k = 10$, $\theta = 0.2$ and varying θ_C. The small rectangle depicts the performance of the coarse index if θ_C was chosen by the model and the vertical line the difference in performance.

ing of blocks. In cases where the best blocks will not result in similar rankings, Blocked+Prune terminates early. Thus, when searching for exact matches, the Blocked+Prune technique performs especially well, outperforming AdaptSearch by a factor of 1.2. Same as for the coarse index, dropping lists further improves the performance of the Blocked+Prune technique. Increasing the values of θ degrades the performance of all the processing techniques. The Blocked+Prune+Drop technique performs worse than the F&V+Drop, because sorting the lists adds some overhead to the processing while the pruning is not so effective. The F&V+Drop technique is performing very well, in fact we measured its performance to be very close to the Minimal F&V, especially for small values of θ. Although they are both based on the same concept, F&V+Drop performs better than AdaptSearch, first because it drops one index list more than AdaptSearch, and second, because we are processing relatively short rankings, thus the simple algorithms perform well.

Most of the query processing techniques display the same behavior in the experiments performed on the Yago dataset (Figure 9). What is different here is that none of the processing techniques perform as good as the Minimal F&V, which shows a runtime close to zero. This is due to the fact that the items in the Yago dataset are more equally distributed. In this dataset we have small clusters of similar rankings. However, the clusters seem to be different among them, allowing more rankings to be pruned early on. Moreover, for the Yago dataset the Blocked+Prune technique performs very poorly. We believe this is because the overhead of sorting the index list is too big for the small index size. In fact, we measured that for 35% of the queries sorting of the index lists accounts for a third of the execution time, when $k = 10$. The percentage increases as we increase k. The simple baseline ListMerge technique surprisingly outperforms the coarse index and the AdaptSearch algorithm. We believe that this happens because of the small data size and the size of the rankings. Still, ListMerge does not perform better than the Coarse+Drop technique, except for $\theta = 0.3$ and $k = 10$. For this dataset, the AdaptSearch algorithm shows better performance, performing better than the coarse index in most of the cases. However, the Coarse+Drop technique and some of the hybrid techniques still outperform the competitor, AdaptSearch.

Distance Function Computations

The difference in performance between the Coarse, Coarse+Drop, F&V+Drop and Blocked+Prune+Drop algorithms can be explained by looking at the number of distance functions calls, shown in Figure 10. We see that for the Yago dataset the final result set is very small, practically almost 1, and the number of distance function computations performed by all the algorithms is significantly larger than the final result set. On the other hand, for the NYT data set—where we have a skewed distribution of the items—the number of false positives is very small, resulting in a very good performance of the F&V+Drop and Blocked+Prune+Drop processing techniques. Combining these with the coarse index even further reduces the number of distance function computations, i.e., the number of distance function computations is smaller than the final result set. This is because for the exact matching rankings in one partition, the Footrule distance is not computed again during query processing time.

	size in MB		**construction time in sec.**	
	NYT	Yago	NYT	Yago
Plain Inverted Index	480	24	3.37	0.03
Augmented Inverted Index	661	38	5.72	0.11
Delta Inverted Index	417	35	3.63	0.086
BK-tree	276	11	1206.75	12.11
M-tree	265	11	35.00	0.47
Coarse Index	367	26	1392.35	19.57

Table 6: Size and construction time of indices for $k = 10$

7.2 Index Size and Construction Time

In Table 6 the size and the index construction time is shown for both datasets for $k = 10$. Delta Inverted Index is the index used in the AdaptSearch algorithm. For the coarse index, we set $\theta_C = 0.5$. We see that all the indices are smaller than 1GB. All indices store the complete rankings, thus their sizes do not differ significantly. The rank-augmented inverted index requires the most storage as it keeps both the complete rankings, and the position augmented index lists to support different processing techniques.

The construction time of the coarse index is the most expensive one, as we need to build a BK-tree, partition it and add the medoids to the inverted index. The construction of the BK-tree is expensive as the tree is unbalanced and in worst case, we need $O(n^2)$ distance computations. The M-tree index construction time is lower than the BK-tree. Both construction times are worse than the one of the inverted index; creating the inverted index does not imply making any distance computations. However, the construction time of the plain inverted index is cheaper than the augmented one, as we do not consider the position of the rankings.

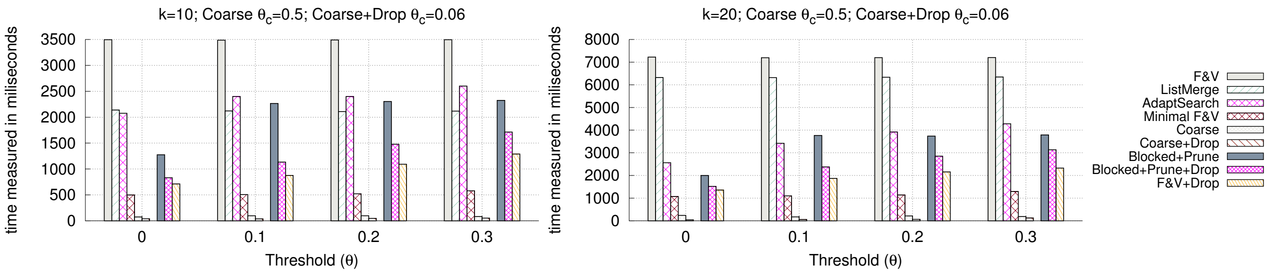

Figure 8: Comparing query processing over coarse index with baseline and competitor approaches (left block) and with other hybrid methods over inverted index (right block) (NYT).

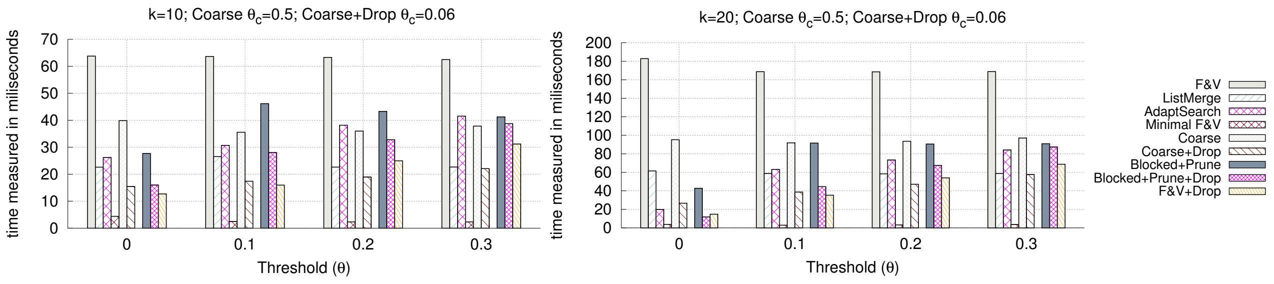

Figure 9: Comparing query processing over coarse index with baseline and competitor approaches (left block) and with other hybrid methods over inverted index (right block) (Yago).

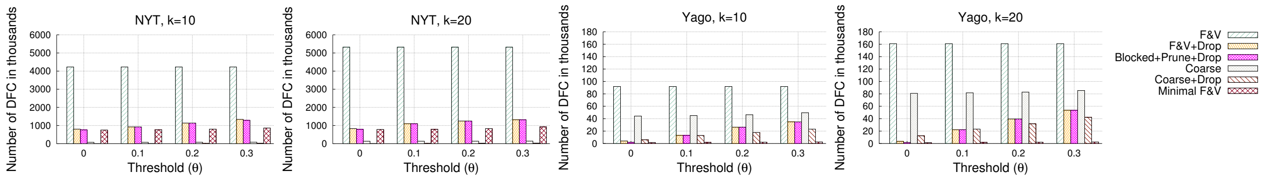

Figure 10: # of distance function calls (DFC) for different query processing methods (Coarse θ_c=0.5; Coarse+Drop $\theta_c = 0.06$)

It is difficult to compare the complexity of the construction time of the different index structures, since the complexity of the metric index structures is usually measured in distance function computation, as this is the most costly operation. On the other hand, in the case of the inverted index there are no distance functions performed during construction at all.

Lessons Learned

Combining the coarse index with the proposed optimizations on the inverted index always leads to performance improvements, independent of the distribution of the items in the dataset. The experiments demonstrate that the Coarse+Drop technique outperformed state-of-the-art algorithm for similarity search, AdaptSearch, for both datasets. The simple yet accurate model for picking the optimal trade-off point (cf., Section 4) leads close to the best performance of the coarse index. When the query threshold is not known, we can tune the coarse index for the maximum query threshold that we might have. In these cases, the coarse index shows to perform better for a skewed dataset. When having a dataset where the items are unevenly distributed, the F&V+Drop algorithm alone results in huge gains as we only process the smallest index lists. These, as the distribution of the items is skewed, can often contain only few false positives. On the contrary, when the dataset contains chunks of rankings similar to each other, i.e, we have more evenly distributed items, the effect of the early pruning of rankings is most expressed. Thus in these cases, using the Blocked+Prune+Drop algorithm, which combines the early pruning with dropping of entire index lists, leads to the biggest benefits, for small values of θ. Varying the size of the rankings does not have a great impact on the different algorithms. Only when having very small ranking sizes, for instance k=5, the simple baseline ListMerge shows to perform well.

8. CONCLUSION AND OUTLOOK

In this paper, we addressed indexing mechanisms and query processing techniques for ad-hoc similarity search inside sets of rankings. We specifically considered Spearman's Footrule distance for top-k rankings and investigated the trade-offs between metric index structures and inverted indices, known in the literature for indexing set-valued attributes. The presented coarse index synthesizes advantages of metric-space indexing and the ability of inverted indices to immediately dismiss non-overlapping rankings. To understand and automatically tune the necessary partitioning of the rankings, we developed an accurate theoretic cost model; and showed by experiments that it allows reaching performance close to the optimal trade-off point. Further, we presented an algorithm that avoids accessing blocks of an index list during query processing thereby improving performance. We derived upper and lower distance bounds for such an online processing and, further, studied the impact of dropping entire parts of the query depending on the tightness of the query threshold. The presented approaches are to a large extent orthogonal and, by a comprehensive performance evaluation using two real-world datasets, we showed that the individual benefits add up, showing better performance than the competitor, AdaptSearch.

As ongoing work we consider processing large batches of queries, instead of the single ad-hoc queries we addressed in this work. We believe that an approach similar to the coarse indexing can be fruitful here: the query batch can be partitioned into related medoid rankings to prune the search space of potential result rankings.

9. REFERENCES

[1] F. Alvanaki, E. Ilieva, S. Michel, and A. Stupar Interesting event detection through hall of fame rankings. *DBSocial*, 2013.

[2] N. Augsten, A. Miraglia, T. Neumann, and A. Kemper. On-the-fly token similarity joins in relational databases. *SIGMOD*, USA, 2014.

[3] R. A. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity Matching Using Fixed-Queries Trees. In *CPM*, 1994.

[4] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. IO-Top-k: Index-access Optimized Top-k Query Processing. *VLDB*, pages 475–486, 2006.

[5] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling Up All Pairs Similarity Search. *WWW*, pages 131–140, 2007.

[6] R. Bayer and E. M. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Inf.*, 1, 1972.

[7] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. *SIGMOD*, 1990.

[8] J. L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM*, 18(9), 1975.

[9] N. Bruno, S. Chaudhuri, and L. Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM Trans. Database Syst.*, 27(2), 2002.

[10] W. A. Burkhard and R. M. Keller. Some Approaches to Best-match File Searching. *Commun. ACM*, 16(4), 1973.

[11] S. Chaudhuri, V. Ganti, and R. Kaushik. A Primitive Operator for Similarity Joins in Data Cleaning. *ICDE*, 2006.

[12] E. Chávez and G. Navarro. A compact space decomposition for effective metric indexing. *Pattern Recognition Letters*, 26(9), 2005.

[13] E. Chávez, G. Navarro, R. A. Baeza-Yates, and J. L. Marroquín. Searching in metric spaces. *ACM Comput. Surv.*, 33(3), 2001.

[14] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. *VLDB*, 1997.

[15] E. R. D'Avila. M-Tree Implementation at GitHub. `https://github.com/erdavila/M-Tree`.

[16] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *OSDI*, 2004.

[17] C. Dwork, R. Kumar, M. Naor, and D. Sivakumar. Rank aggregation methods for the Web. *WWW*, 2001.

[18] R. Fagin, R. Kumar, and D. Sivakumar. Comparing Top k Lists. *SIAM J. Discrete Math.*, 17(1), 2003.

[19] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4), 2003.

[20] P. Flajolet, D. Gardy, and L. Thimonier. Birthday Paradox, Coupon Collectors, Caching Algorithms and Self-Organizing Search. *Discrete Applied Mathematics*, 39(3), 1992.

[21] V. Ganti, R. Ramakrishnan, J. Gehrke, A. L. Powell, and J. C. French. Clustering Large Datasets in Arbitrary Metric Spaces. *ICDE*, 1999.

[22] S. Helmer and G. Moerkotte. A performance study of four index structures for set-valued attributes of low cardinality. *VLDB J.*, 12(3), 2003.

[23] E. Ilieva, S. Michel, and A. Stupar. The essence of knowledge (bases) through entity rankings. *CIKM*, 2013.

[24] E. H. Jacox and H. Samet. Metric space similarity joins. *ACM Trans. Database Syst.*, 33(2), 2008.

[25] N. Mamoulis. Efficient Processing of Joins on Set-valued Attributes. *SIGMOD*, 2003.

[26] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[27] Y. N. Silva, W. G. Aref, and M. H. Ali. The similarity join database operator. *ICDE*, pages 892–903, 2010.

[28] Y. N. Silva, W. G. Aref, P. Larson, S. Pearson, and M. H. Ali. Similarity queries: their conceptual evaluation, transformations, and processing. *VLDB J.*, 22(3), 2013.

[29] M. Terrovitis, P. Bouros, P. Vassiliadis, T. K. Sellis, and N. Mamoulis. Efficient answering of set containment queries for skewed item distributions. *EDBT*, 2011.

[30] M. Terrovitis, S. Passas, P. Vassiliadis, and T. K. Sellis. A combination of trie-trees and inverted files for the indexing of set-valued attributes. *CIKM*, 2006.

[31] The New York Times Annotated Corpus. `http://corpus.nytimes.com`.

[32] J. K. Uhlmann. Satisfying General Proximity/Similarity Queries with Metric Trees. *Inf. Process. Lett.*, 40(4), 1991.

[33] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *SIGMOD Conference*, 2012.

[34] Y. Wang, A. Metwally, and S. Parthasarathy. Scalable all-pairs similarity search in metric spaces. *KDD*, 2013.

[35] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient Similarity Joins for Near Duplicate Detection. *WWW*, 2008.

[36] P. N. Yianilos. Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces. *SODA*, 1993.

[37] P. Zezula, P. Savino, G. Amato, and F. Rabitti. Approximate Similarity Retrieval with M-Trees. *VLDB J.*, 7(4), 1998.

[38] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.