Similarity Search Algorithms over Top-k Rankings and Class-Constrained Objects

Thesis approved by
the Department of Computer Science
Technische Universität Kaiserslautern
for the award of the Doctoral Degree

Doctor of Engineering (Dr.-Ing.)

to

Evica Milchevski

Date of defense:
August 15, 2019

Dean:
Prof. Dr.-Ing. Stefan Deßloch

Reviewers:
Prof. Dr.-Ing. Sebastian Michel (TU Kaiserslautern)
Prof. Dr.-Ing. Avishek Anand (Leibniz University, Hannover)
Prof. Nikolaus Augsten, PhD (University of Salzburg)

PhD committee chair:
Prof. Dr. Katharina Anna Zweig

D 386

To my parents...

Abstract

In this thesis, we consider the problem of processing similarity queries over a dataset of top-k rankings and class constrained objects. Top-k rankings are the most natural and widely used techniques to compress a large amount of information into a concise form. Spearman's Footrule distance is used to compute the similarity between rankings, considering how well rankings agree on the positions (ranks) of ranked items. This setup allows the application of metric distance-based pruning strategies, and, alternatively, enables the use of traditional inverted indices for retrieving rankings that overlap in items. Although both techniques can be individually applied, we hypothesize that blending these two would lead to better performance. First, we formulate theoretical bounds over the rankings, based on Spearman's Footrule distance, which are essential for adapting existing, inverted index based techniques to the setting of top-k rankings. Further, we propose a hybrid indexing strategy, designed for efficiently processing similarity range queries, which incorporates inverted indices and metric space indices, such as M- or BK-trees, resulting in a structure that resembles both indexing methods with tunable emphasis on one or the other. Moreover, optimizations to the inverted index component are presented, for early termination and minimizing bookkeeping. As vast amounts of data are being generated on a daily bases, we further present a distributed, highly tunable, approach, implemented in Apache Spark, for efficiently processing similarity join queries over top-k rankings. To combine distance-based filtering with inverted indices, the algorithm works in several phases. The partial results are joined for the computation of the final result set. As the last contribution of the thesis, we consider processing k-nearest-neighbor (k-NN) queries over class-constrained objects, with the additional requirement that the result objects are of a specific type. We introduce the MISP index, which first indexes the objects by their (combination of) class belonging, followed by a similarity search sub index for each subset of objects. The number of such subsets can combinatorially explode, thus, we provide a cost model that analyzes the performance of the MISP index structure under different configurations, with the aim of finding the most efficient one for the dataset being searched.

Zusammenfassung

In dieser Arbeit betrachten wir das Problem, Ähnlichkeitsanfragen über einem Datensatz von Top-k-Ranglisten oder Objekten mit klassenspezifischen Selektionskriterien zu berechnen. Top-k-Ranglisten sind eine natürliche und geläufige Technik, um große Mengen an Informationen in einer prägnanten Form darzustellen. Spearman's Footrule Distanz wird benutzt, um die Ähnlichkeit zwischen Ranglisten zu berechnen. Sie berücksichtigt dabei, wie stark die Listen im Bezug auf die Positionen (Ränge) der aufgeführten Elemente übereinstimmen. Dieser Problemstellung erlaubt die Anwendung von Strategien basierend auf metrischen Distanzen, um Objekte, die nicht im Ergebnis enthalten sein werden, frühzeitig zu eliminieren. Alternativ dazu können auch traditionelle invertierte Indexe über Ranglisten benutzt werden, um solche Ranglisten zu finden, die in mindestens einem Eintrag mit der Anfragerangliste überlappen. Obwohl beide Techniken eigenständige Ansätze darstellen, liegt die Vermutung nahe, dass eine Kombination der beiden zu einer verbesserten Effizienz führen könnte. Zunächst präsentieren wir ein theoretisches Modell, um bestehende mengenbasierte Ansätze über invertierten Indexen auch für Top-k-Ranlisten anwendbar zu machen. Weiterhin stellen wir eine hybride Indizierungsstrategie vor, die zur effizienten Verarbeitung ähnlichkeitsbasierter Bereichsanfragen entworfen ist. Diese Strategie umfasst sowohl invertierte Indexe als auch Indexe für metrische Räume, wie beispielsweise M- oder BK-Bäume. Sie resultiert in einer Struktur, die beide Indizierungsmethoden abbildet und einen zwischen ersterer und letzterer justierbaren Schwerpunkt bietet. Außerdem stellen wir Optimierungen für die Anfrageverarbeitung über der Invertierten-Index-Komponente vor, die frühzeitige Terminierung und Minimierung von Buchführung ermöglicht. Enorme Mengen von Daten werden täglich generiert, weswegen wir einen verteilten und hochgradig anpassbaren Ansatz vorstellen, der mittels Apache Spark implementiert wurde, um Ähnlichkeits-Verbundanfragen über Top-k-Ranglisten effizient beantworten zu können. Distanz-basierte Filter werden in einem mehrphasigen Algorithmus mit invertierten Indizes kombiniert, wobei partiellen Ergebnisse für die Berechnung der finalen Ergebnismenge kombiniert werden. Als letzten Beitrag der Arbeit betrachten wir die Verarbeitung von k-Nächste-Nachbar-Anfragen über Objekten mit klassenspezifischen Selektionskriterien, wobei vorgegeben wird, dass Ergebnisobjekte einem spezifischen Typ angehören. Wir präsentieren den MISP-Index, der Objekte zunächst anhand der Klassenzugehörigkeit (oder einer Kombination von Klassen), darauf folgend mittels Ähnlichkeitssuche auf einem

Subindex für jede Teilmenge von Objekten indiziert. Die Anzahl solcher Teilmengen kann kombinatorisch explodieren, weswegen wir ein Kostenmodell vorstellen, das die Performanz der MISP Indexstruktur unter verschiedenen Konfigurationen analysiert, sodass die effizienteste für den zu durchsuchenden Datensatz gefunden wird.

Contents

Chapter 1

Introduction

Current data processing trends are increasingly moving away from standard database techniques, and are leaning toward unstructured, or semi-structured data, where exact-match queries cannot be applied and different information retrieval methods are needed. The flourishing of research fields like machine learning, natural language processing or data exploration, and the prevalence of data handling frameworks like Apache Hadoop [HDP], Apache Spark [SPK], MongoDB [MDB] and HBase [HBS] further reinforce this shift toward non-traditional data management methods. Similarity searching is one way of answering the information need when faced with such schema-free data. For instance, consider the case of image search engines, like Google Image Search, that offer query by example search—as input the user provides an image, and as a result the system returns images similar to the input. Similarity searching has been an active research field for decades now, and some of its focuses include similarity search index structures and algorithms in metric space [Sam06, CNBM01], sets and strings [MAB16], or developing scalable, distributed algorithms [FAB+18].

In this thesis, we address the task of similarity search over top-k rankings and class-constrained objects, both being of great value for a large number of diverse applications, e.g., business analysis, data cleansing, or for searching in the geospatial domain. Top-k rankings allow users and algorithms to effectively and efficiently inspect the best performing items within a certain category. Considering the information deluge we are facing today, rankings are a ubiquitous information representation. They are used in databases for business intelligence and other forms of insight-seeking analyses, in politics and other social aspects of our lives for mutual comparison, or expressing dominance. Often, they are also crowd-sourced through mining user polls on the Web, in portals such as IMDB or Netflix (for movie ratings), or specifically created by users in form of favorite lists on personal websites. Similarity search over top-k rankings can give access to valuable analytical insights. For instance, consider the rankings in Figure 1.1 representing favorite lists from IMDB users, where they rate their preferences for TV series. Based on their similarity, we could suggest to users

series that they would like to watch. Considering the lists of Alice and Bob as similar, we could propose the TV series *Disenchantment* to Bob and *The Americans* to Alice. Another application would be to detect communities within the users database, based on the similarities of the lists.

Class-constrained objects are as well omnipresent, as all objects can have a set of classes or attributes associated to them. For instance, movies can be classified by different genres, the year when they were produced or their director, geospatial data can be classified by their type, for instance, sightseeing, recreational, restaurant, administrative or also more fine-grained, such as cuisine of a restaurant. To contemplate the need for similarity search over class-constrained objects, consider Bob, who is planning a trip to Brazil. His first destination is São Paulo and he wants to see where to go from there. He likes large cities, so he wants to find the nearest cities to São Paulo with over million inhabitants. In this thesis, we specifically aim at answering such questions, where the user is interested in finding the nearest objects that also belong to a concrete, user-defined, set of classes.

	Alice	**Bob**	**John**
1.	Narcos	Narcos	The Big Bang Theory
2.	Stranger Things	Stranger Things	The Office
3.	Black Mirror	Sherlock	Narcos
4.	Disenchantment	Black Mirror	Modern Family
5.	Sherlock	The Americans	Planet Earth

Figure 1.1: Example top-k lists of favorite TV Series for IMDB users.

1.1 Problem Statement

In this thesis, we address three types of similarity search queries, namely, similarity range queries, similarity join, and a variation of the nearest neighbors query. The former two we solve over a dataset of top-k rankings, while the latter, over class-constrained objects. In all three cases the focus is on the query execution performance. In the following, we define each one more formally and discuss the challenges that come with them.

The first problem that we address in this thesis is to efficiently process similarity range queries over a dataset of top-k rankings $\mathcal{T}$, defined as:

Definition 1 (Similarity Range Search) *Given a dataset $\mathcal{T}$ of top-k rankings, a distance function d, a user-defined query ranking q, and distance threshold θ, the task is to efficiently find all rankings in $\mathcal{T}$ with distance below or equal to θ, i.e., $\{\tau_i | \tau_i \in \mathcal{T} \wedge d(\tau_i, q) \leq \theta\}$.*

Furthermore, we also address the similarity joins problem for top-k rankings,

where we specifically focus on devising an efficient distributed solution, defined as:

Definition 2 (Similarity Joins) *Given two dataset $\mathcal{S}$ and $\mathcal{T}$ of top-k rankings, a distance function d, and distance threshold θ, the task is to efficiently find all pairs $(\tau_i \in \mathcal{S}, \tau_j \in \mathcal{T})$ with distance below or equal to θ, i.e., $\{(\tau_i \in \mathcal{S}, \tau_j \in \mathcal{T}) \wedge d(\tau_i, \tau_j) \leq \theta\}$.*

Without loss of generality, in this thesis, we discuss the self join problem, i.e., to find all pairs $\{(\tau_i, \tau_j) | \tau_i, \tau_j \in \mathcal{T}, i \neq j \wedge d(\tau_i, \tau_j) \leq \theta\}$.

For the similarity range queries, we assume that the dataset $\mathcal{T}$ remains the same, while the query ranking q and the threshold θ change at query time. This entails indexing the rankings in $\mathcal{T}$. Thus, the challenge here is developing an index structure that allows efficient answering of this type of queries over top-k rankings. Since the threshold value changes at query time, no assumption should be made about its value when designing the index structure. While the index construction time and the memory consumption should be taken into consideration, the focus here is on minimizing the query answering time. On the other hand, for similarity join queries we assume that the threshold θ is given upfront, and thus, could possibly be used for improving the performance. However, since the similarity join result set includes all pairs of rankings in $\mathcal{T}$ with distance smaller than θ, this entails processing all rankings in $\mathcal{T}$. In fact, the naive solution has $O(n^2)$ complexity, where n is the size of the dataset, as all records need to be compared to each other. The challenge here is to determine ranking-specific filtering techniques which would reduce the number of comparisons that need to be made. In addition, since we focus on developing a distributed solution, we need to handle the data distribution, which ideally should be equal among all nodes, and the communication or data shuffling between nodes, which should be kept as low as possible.

In this thesis, as a distance measure for comparing the rankings we use Spearman's Footrule. Fagin et al. [FKS03] 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. This immediately suggests employing metric data structures, like BK-trees [BK73] or M-trees [CPZ97], for indexing and similarity search, however, as we will show later in our thesis, a better performance can be achieved when these are combined with an inverted index.

Furthermore, in this thesis we address the problem of a special type of k nearest neighbor query, named class-constrained k-NN (CCk-NN).

Definition 3 (Class-Constrained k Nearest Neighbor) *Given a dataset of objects $O = \{o_1 \dots o_n\}$, where each object has a set of classes $C_{o_r} \subseteq C$ associated to it, from one global domain of classes C, a distance function d, a user-specified query q, and a user-specified query classes $C_q \subseteq C$. The task is to find k objects $O_R \subseteq O$, such that $\forall o_r \in O_R : C_{o_r} \supseteq C_q$, and there is* no *object $o_s \in O$ and $o_r \in O_R$ such that $d(q, o_s) < d(q, o_r)$ and $C_{o_s} \supseteq C_q$.*

A naive solution to this problem would be for each object in the dataset O to check if it satisfies the query classes C_q and if so, compare it to q. However, a better performance can be reached when the objects are indexed using an inverted index. Then, the dataset objects that satisfy the classes C_q, can be determined upfront. Further sequential scan over these classes is needed to find the nearest objects to q. Alternatively, similarly as with the rankings, the dataset objects can be indexed using a spatial index structure like the R-tree [Gut84] or the M-tree [CPZ97]. To process the CCk-NN query, the index structure's k-NN retrieval mechanism can be used. By executing a k'-NN query, where $k' > k$ and check which of the results satisfy the classes in C_q. If the results found are less than k the process should be repeated with larger k'. Ideally, the CCk-NN search process can be implemented in an incremental fashion and stops right after k objects with classes C_q are found. In this thesis, similarly as before, we make the hypothesis that a combination of an inverted index with a spatial index structure would be a more suitable solution for such type of queries. This opens a new set of challenges, like for instance, how these two should be combined in order to achieve the best query performance, while keeping the index construction time and memory consumption low.

1.2 Contributions

With this work, we make the following contributions to the area of similarity search:

- We devise theoretical bounds for top-k rankings, based on the Spearman's Footrule distance between them, which find its application in several different problems where top-k rankings are used. Especially important is their use for the adaptation of existing set-based similarity search and join algorithms for handling top-k rankings.
- We present a novel index structure, coined Coarse index, and algorithms for processing similarity range queries over a dataset of top-k rankings. The presented index structure combines two well known indexing techniques, typically used independently from each other in different problem settings, to bring the best out of both worlds.
- We present a theoretical cost model that based on the dataset distribution can compute the performance sweet spot of our proposed Coarse index. Furthermore, trough experiments, by using two real-world datasets, we show that the cost model correctly predicts the parameters of the Coarse index, that would lead to its best performance.
- As a complement to the proposed approach for similarity range queries over top-k rankings, we propose an efficient solution to the problem of similarity join for top-k rankings. The proposed solution is implemented in Apache Spark [SPK], a highly established platform for large scale data

analytics, and thus is able to handle large amounts of data. Furthermore, we adapt existing set-based distributed MapReduce approaches to top-k rankings and implement them using Apache Spark.

- To complete our contributions, we also address the third type of similarity queries, the k nearest neighbors (k-NN). Concretely, we analyze the problem of k nearest neighbors where as input we are provided with a dataset of class-constrained objects, and the user is interested in the nearest objects to the query that also satisfy some set of classes, coined class-constrained k-NN. We propose MISP, an index structure for efficiently handling such type of similarity queries, and a cost model, that estimates the performance and memory consumption of the proposed index structure based on the properties of the input data.

1.3 Publications

The work presented in this thesis has been published in several peer-reviewed papers in conferences and workshops.

In [MAM15], we have introduced the problem of similarity search over top-k rankings and presented our Coarse index, described in Chapter 5, a hybrid index structure, which partitions the data according to their pairwise distance. The span of the partitions affects the performance of the Coarse index, and thus, we also conceive a cost model, that, depending on the data distribution, computes the partitioning sweet spot. In this paper, we also define some theoretical bounds over top-k rankings, which allow us to apply a well known set-based filtering technique to our problem setting.

- Evica Milchevski, Avishek Anand, and Sebastian Michel. The Sweet Spot between Inverted Indices and Metric-Space Indexing for Top-K-List Similarity Search. 18th International Conference on Extending Database Technology (EDBT), 2015.

In [MNM18], we have addressed the problem of Class-Constrained k nearest neighbors (CCk-NN). In this paper we have focused on devising a cost model that estimates the performance of an index structure for answering CCk-NN queries.

- Evica Milchevski, Fabian Neffgen, and Sebastian Michel. Processing Class-Constraint K-NN Queries with MISP. 21st International Workshop on the Web and Databases (WebDB), 2018.

The theoretical bounds, presented in Chapter 4 and its applications have been published in different publications. In [MM16] we have presented our work on maintaining a set of crowd-sourced top-k rankings, based only on the distance, i.e., similarity, between the rankings.

- Evica Milchevski, and Sebastian Michel. Quantifying Likelihood of Change through Update Propagation across Top-k Rankings. 19th International Conference on Extending Database Technology (EDBT), 2016.

In [PMM16] we have extended the PALEO system [PM16], initially designed to reverse engineer an SQL query that, given a database and a sample top-k input list, would return the same input list, to be able to also find SQL queries that would result in lists *similar* to the input top-k list. In this paper, some of the bounds presented in Chapter 4 have been applied in order to improve the efficiency of the system.

- Kiril Panev, Evica Milchevski, and Sebastian Michel. Computing similar entity rankings via reverse engineering of top-k database queries. Workshop on Keyword Search and Data Exploration on Structured Data (KEYS), 2016.

In [PMMP16] a complete prototype implementation of PALEO has been demonstrated.

- Kiril Panev, Sebastian Michel, Evica Milchevski, and Koninika Pal. Exploring Databases via Reverse Engineering Ranking Queries with PALEO. 42nd International Conference on Very Large Databases (VLDB), 2016.

1.4 Outline of the Thesis

This thesis is organized as follows. In Chapter 2 we present the background concepts, algorithms and frameworks needed for understanding the work presented in this thesis. Chapter 3 discusses related work in the filed of similarity search. Here, we specifically describe work for similarity search in metric space and similarity join and search for sets and strings. In Chapter 4 we present theoretical bounds for top-k rankings based on Spearman's Footrule distance between them. In Chapter 5 and 6 we present our approaches for similarity search over top-k rankings. Specifically, in Chapter 5 our work on similarity range queries, where a novel index structure is presented, and in Chapter 6 we present distributed approaches for efficiently solving the problem of similarity joins over top-k lists. In Chapter 7 we present our work on the special type of k nearest neighbors queries, the class-constrained k nearest neighbors. Finally, in Chapter 8 we summarize our work and present directions for future work.

Chapter 2

Background and Preliminaries

In this chapter, we present background knowledge necessary for understanding the work described in this thesis. We start by introducing the concepts of similarity search where we explain the different types of queries, distance functions, and index structures applied in our methods. The fundamentals of metric space and the indexing techniques for metric space are also explained. We continue by discussing rankings, top-k rankings and the distance functions that can be used for comparing them. Finally, the core concepts behind platforms for distributed data processing, specifically MapReduce and Apache Spark, are explained.

2.1 Similarity Search

2.1.1 Metric Space

In this thesis, we mainly deal with metric distance functions, or simply *metrics*. The following definition defines metric space M [O'S06]:

Definition 4 *Suppose M is a set and d is a real function defined on the Cartesian product $M \times M$. Then d is called a metric on M if, and only if, for each $a, b, c \in M$:*

- **Positive property:** $d(a, b) \geq 0$ *with equality if, and only if,* $a = b$.
- **Symmetric property:** $d(x, y) = d(y, x)$
- **Triangle inequality:** $d(x, z) \leq d(x, y) + d(y, z)$

M is called a metric space, and d is a metric distance, or simply a metric. One of the most well-known metric distances is the absolute-value function on $\mathbb{R}$. In general, if the distance between two objects a and b is smaller, the objects

are more similar or closer to each other. In some domains, it can be assumed that d never exceeds some value r, i.e., for all $a, b \in M$, $d(a, b) \leq r$. Then, the metric space is called *bounded metric space*. In this thesis, we consider such bounded metric spaces.

2.1.2 Distance Functions

There are many different distance functions that belong to the metric space. They can be divided into *discrete* and *continuous* functions. Discrete functions have a limited set of possible values, whereas continuous functions map onto infinitely many possible values.

Minkowski Distances

The most well-known family of continuous functions are the *Minkowski* distances, called L_p metrics, $L_p : \mathbb{R}^n \times \mathbb{R}^n \mapsto \mathbb{R}_+$:

$$L_p[(x_1, \ldots, x_n), (y_1, \ldots, y_n)] = \sqrt[p]{\sum_{i=1}^{n} |x_i - y_i|^p} \tag{2.1}$$

for $p \in \mathbb{R}$ and $p \geq 1$.

For $p = 1$, the metric is called *Manhattan*, or the *taxicab* distance, since it resembles the way for measuring the distance from one place to another using the parallel network of roads in the city of New York.

For $p = 2$, the Minkowski distance is called the *Euclidean* distance and it is the most widely used distance function in vector space.

Hamming Distance

The *Hamming* distance is used to quantify the distance (similarity) of two strings a and b of equal length n. Initially described in the paper of R. W. Hamming [Ham50], it measures the number of positions at which the two strings differ. For comparing strings with different lengths, the shorter string is padded with null characters to reach the length n, to make the Hamming distance computation possible. The Hamming distance is a metric for a fixed length of words n. Figure 2.1 shows three example strings where the characters which differ are marked with red. The Hamming distance between the strings s_1=house and s_2=hoouse is 4, $h(s_1, s_2) = 4$, since only the first 2 characters are matching. Similarly, we can compute $h(s_1, s_3) = 3$ and $h(s_2, s_3) = 4$.

Edit Distance

The *edit* distance, also called Levenshtein distance, is similar to the Hamming distance. It is calculated as the minimum number of edit operations that are

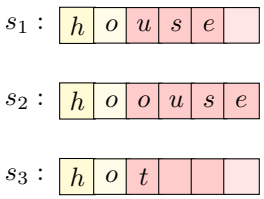


Figure 2.1: Example strings where the difference in characters is marked with red. The Edit and the Hamming distance are the same between (s_1, s_3) and (s_2, s_3), with values 3 and 4, respectively. For the pair (s_1, s_2), $h(s_1, s_2) = 4$ while $e(s_1, s_2) = 1$.

needed to transform one string into the other. The edit operations considered include: inserting a character into a string, deleting a character from the string, or, replacing one character with another. Considering the example strings in Figure 2.1, the Edit distance for strings s_1 and s_2 is 1, $e(s_1, s_2) = 1$, since inserting the character o at position 3 in s_1 would make the two strings identical. The edit distance for pairs (s_1, s_3) and (s_2, s_3) is the same as their Hamming distance, i.e., $e(s_1, s_3) = 3$ and $e(s_2, s_3) = 4$.

There are several variations of the edit distance which include adding either weights to the operations considered, or to the characters being replaced. These variations are useful for spelling corrections of strings.

Jaccard Distance

The Jaccard distance is the complement of the Jaccard coefficient [Jac12], computed as:

$$d_J(s_1, s_2) = 1 - J(s_1, s_2)$$

where s_1 and s_2 are sets. The Jaccard coefficient is the standard measure used for measuring the similarity between two sets s_1 and s_2. It is defined as:

$$J(s_1, s_2) = \frac{|s_1 \cap s_2|}{|s_1 \cup s_2|}$$

The maximum value of the Jaccard coefficient is 1, meaning that the two sets have no overlap at all, and its minimum value is 0, meaning that the two sets are identical. For empty sets, the Jaccard coefficient is 1.

2.1.3 Similarity Queries

A similarity query is a type of query where at input we are given a query object q and, usually, some type of distance or similarity constraint. As output, all objects that satisfy the constraint need to be returned, and usually, these are objects that are close to, or similar to, the query object q [ZADB06].

Range Query

This type of queries is probably the most used and well known type of similarity queries, denoted as $\mathcal{R}(q, r)$, where r is the query radius. Given some dataset D, a query q and a distance (similarity) radius r, the goal of the range query is to retrieve all objects, $o \in D$ with distance (similarity) within r:

$$\mathcal{R}(q, r) = \{o \in D, d(o, q) \leq r\}$$

Note that the query object q does not need to be a member of the dataset D. If the search radius r is zero, then the query is called a **point query**. In this case, only the elements that are duplicates to the query object q should be returned as results

Nearest Neighbor Query

In cases where the search range is difficult to specify, for instance, when we do not have any a priori knowledge of the data, it would be more appropriate for searching for those objects that are closest to the query object q. These type of queries, where instead of a search range, we are searching for the nearest object to q with respect to some distance function d, are called the *nearest neighbor queries*. This concept can be generalized to searching for the k nearest neighbors to q, called k *nearest neighbor queries*, or k-NN. Formally, the result should be:

$$\text{k-NN}(q) = \{R \subseteq D, |R| = k \wedge \forall o \in R, y \in D - R : d(q, o) \leq d(q, y)\}$$

If the size of the dataset to be searched is smaller than k, then the whole dataset is returned as a result. When there are more than k objects with the same distance to the query object, ties are broken arbitrarily.

Reverse Nearest Neighbor Query

In certain application scenarios, like for instance in decision support systems, a different type of k-NN query can be useful, where instead of searching for the nearest object to q, we want to retrieve all objects that have q as their nearest neighbor. This is called *reverse nearest neighbor queries*, and the general case is called reverse k nearest neighbor queries (k-RNN). In the general case we want to retrieve all objects that have q in their k-NN result set. Formally:

$$\text{k-RNN}(q) = \{R \subseteq D, \forall o \in R : q \in k - NN(o) \wedge \forall o \in D - R : q \notin \text{k-NN}(o)\}$$

Note that for this type of queries, also objects that have a large distance to the query object can be in the result set.

Similarity Join

The last type of similarity query that we are going to discuss in this thesis, is the *similarity join*, or in this thesis also referred to as all-pairs similarity search, which usually finds its application in the fields of data cleaning or data integration. The similarity join between two datasets X and Y retrieves all pairs of objects ($x \in X, y \in Y$) whose distance (similarity) is smaller (larger) than a given distance (similarity) threshold, θ. More specifically:

$$SimJoin(X, Y, \theta) = \{(x, y) \in X \times Y : d(x, y) \leq \theta\}$$

We also define a *similarity self join* where we have only one dataset X and we want to find all pairs of objects $(x_i \in X, x_j \in X,), i \neq j$, such that their distance, $d(x_i, x_j)$, is smaller or equal to θ.

2.1.4 Data Partitioning

The main task in this thesis is searching for similar objects. This is usually done by first indexing the data using some kind of index structure, and then, given a query object and a distance (similarity) threshold, searching this index. In metric space, the index structures are constructed such that no assumption is being made about the data being indexed. The only assumption made is that the distance between the objects in the dataset can be computed, and used while indexing, and searching the data. These indexing methods divide the data space into subsets, based on the distance between the objects. When searching, certain partitions which share some piece of information (e.g., spatial closeness), can be ignored, again, depending only on the distance between the data in the partition and the query object.

There are two basic partitioning schemes that are applied by the metric index structures, *ball partitioning* and *hyperplane partitioning*, first defined by Uhlmann [Uhl91]. The here described concepts are used by several index structures referenced in this thesis.

Ball Partitioning

One possibility is to split the data objects o based on the distance d to one pivot element p. One partition is the data that is inside a ball around the pivot element, and another partition is formed by the data outside the ball [Uhl91]. The following formula describes this formally:

$$bp_1(o, p, r) = \begin{cases} 0 & d(o, p) \leq r \\ 1 & d(o, p) > r \end{cases} \tag{2.2}$$

This can be extended to use more than one radius. Basically, the distance cuts the space in $n + 1$ parts. For example, for $n = 2$:

$$bp_2(o, p, r_1, r_2) = \begin{cases} 0 & d(o,p) \leq r_1 \\ 1 & d(o,p) > r_1 \wedge d(o,p) \leq r_2 \\ 2 & d(o,p) > r_2 \end{cases} \tag{2.3}$$

An example of ball partitioning is given in Figure 2.2. In this example 2 dimensional data is used, which is divided in 2 (left) and 3 (right) regions around a chosen pivot.

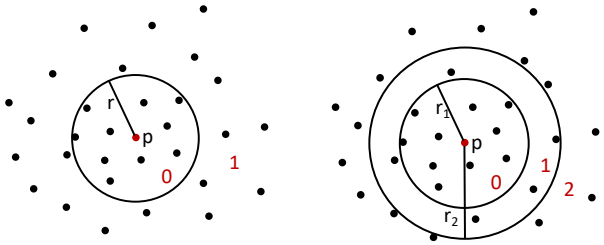


Figure 2.2: Example of ball partitioning where the space is divided in two (left) and three (right) partitions.

Hyperplane Partitioning

The *hyperplane partitioning* choses at least two pivot elements and divides the data into two, or more partitions by the relative distance of the objects to the pivots. A data object o belongs to the partition represented by the nearest pivot p_n. For instance, when we have two pivots, then:

$$hp_2(o, p_1, p_2) = \begin{cases} 0 & d(o,p_1) \leq d(o,p_2) \\ 1 & d(o,p_1) > d(o,p_2) \end{cases} \tag{2.4}$$

This strategy can be extended to use more pivots. Figure 2.3 illustrates the division of the data into two partitions.

2.1.5 BK-tree

The BK-tree is one of the earliest distance based index structures that works with discrete metrics. Proposed by Burkhard and Keller [BK73], the BK-tree resembles an n-ary search tree, where the data is partitioned according to the ball-partitioning principle.

Index Creation

The subtrees in the BK-tree group items according to their distance to the parent node. Thus, each node in the BK-tree can have at most m children, where m

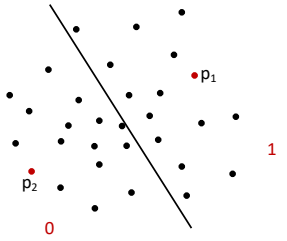


Figure 2.3: Example of hyperplane partitioning where the space is divided in two partitions

is the number of values that the distance function can take. For creating the tree, first a random object is chosen and is set as the root of the tree. Then, the distance from each element of the tree to the root node is computed, and according to the distance, the object is placed in a corresponding branch. This is repeated recursively, until there are no more than b children in one subset. b is called bucket capacity

Thus, the BK-tree, at each level of the tree, divides the data objects into m subsets, based on their distance to a randomly chosen pivot element.

Querying

A range search is done by traversing the tree from the root to the leaves, and the triangle inequality is used for deciding which subtrees have to be visited.

Given a query object q and a threshold distance θ, we start by computing the distance from the q to the root object r, $d(q, r)$. Then, all the branches with distance i, such that $d(q, r) - \theta \leq i \leq d(q, r) + \theta$ need to be searched. This is done recursively until a leave node is reached, where all the elements are compared to the query object. All compared objects, o, where the distance $d(q, o) \leq \theta$ are returned as result to the user. The triangle inequality ensures that that there are no false negatives.

2.1.6 iDistance

iDistance [JOT+05] is a distance based indexing strategy, specifically designed for high dimensional data that allows incremental searching. It utilizes the idea of mapping the objects into linear space, so that indexing these values can be achieved with other well known strategies. The authors propose using a B^+-tree, but other structures can be used as well.

iDistance uses the hyperplane partitioning method. Initially, i reference points (*pivots*) O_i are chosen. They do not have to be among the data points, but must be in the same data space. The pivots are used as an anchor for data points d, where each d gets assigned to a pivot element and the partition that it represents. The authors propose and discuss several partitioning methods.

Index Creation

The creation of the index is done in three main steps. First, according to the partitioning scheme, the pivots are chosen. Then, for each object d, the distances to all pivots O_i is calculated and the closest pivot is chosen as a reference point. In the last step, for each object d, an index key is calculated based on the distance to its reference point, $dist(d, O_i)$:

$$key_d = i \cdot c + dist(d, O_i)$$

where c is a constant and i is the index of the partition to which the object d belongs. These keys are indexed using a B^+-tree. c is used to achieve mapping of the objects that belong to one partitions into regions, and it should be set to sufficiently large number, preferably greater or equal to the maximum distance d^+ of the data space. Note that two points around a pivot element can be equidistant to the pivot, and thus can have the same index key, but this does not affect the correctness of the index, as long as the chosen structure can handle multiple entries per value.

Querying

k-NN queries q, with k as parameter, are done by performing a range search with radius r and incrementally expanding r until k results are found. Range queries are executed by analyzing the distances of the query to each pivot O_i. Under certain circumstances, partly or all data points associated with this pivot can be pruned. Based on the triangle inequality, it follows that:

$$dist(O_i, q) - dist(p, q) \leq dist(O_i, p) \leq dist(O_i, q) + dist(p, q)$$

and thus all points with $dist(p, q) \leq r$ must fulfill the following equation [JOT+05]:

$$dist(O_i, q) - r \leq dist(O_i, p) \leq dist(O_i, q) + r$$

For easier pruning, $dist\_max_i$ is remembered, which is the distance between O_i and the point furthest away from it. If $dist(O_i, q) - r \leq dist\_max_i$ holds, then the partition represented by O_i has to be searched, otherwise it can be pruned.

The actual search is done by scanning the one dimensional index structure for every qualifying pivot O_i. To be more precise, the bounds of $i \cdot c$ and $(i+1) \cdot c$ are considered, because everything outside this bounds belongs to another pivot. For every found data object, the real distance is calculated, because in the one dimensional index there can be false positives due to the overlap in the keys. When the values are stored in an index structure, which supports interval queries on the values, then searching can be done efficiently.

$$
\begin{array}{lll}
\text{e} & \longrightarrow < R_1, R_3 > & \\
\text{a} & \longrightarrow < R_1, R_2, R_3 > & R_1 = \{a, b, c, d, e\} \\
\text{b} & \longrightarrow < R_1 > & R_2 = \{a, c, d\} \\
\text{d} & \longrightarrow < R_1, R_2, R_3 > & R_3 = \{a, d, e\} \\
\text{c} & \longrightarrow < R_1, R_2 > &
\end{array}
$$

Figure 2.4: Example inverted index for three relations R_1, R_2, and R_3.

2.1.7 Inverted Index

The *inverted index*, or sometimes also called inverted file, is one of the first major concepts introduced in the field of information retrieval [MRS10]. Its application since then has expanded greatly, including answering similarity search queries for set-valued types of data.

An inverted index consists of two components—a *dictionary* $\mathcal{D}$ of *objects* and the corresponding *posting lists* (aka. index list) that record for each object information about its occurrences in the document or relation.

Figure 2.4 shows an example inverted index for three relations, R_1, R_2 and R_3, with dictionary $\mathcal{D} = \{a, b, c, d, e\}$. For instance, if we look at the posting list for object b we can see that this object is only a member of R_1.

The initial and main purpose of the inverted index was efficient answering of keyword queries in search engines. However, inverted indices can be used for many other tasks, like for instance, in our specific case, for similarity search of top-k rankings. In this case, we can either use the standard inverted index, where rank information would be lost, or, to support arbitrary ranking queries, an inverted index which contains all items from the rankings in its dictionary and record rank information in its index lists, like the inverted index depicted in Figure 2.5. Thus, the entry $(\tau_{12} : 1)$ found in the index list for item b conveys that the object occurs at rank 1 in ranking τ_{12}.

Querying through Filter and Validate

In the standard inverted index, a separate full access for each of the found rankings is required; as there is no positional information given. We call this method *Filter and Validate* (short, F&V). This is the most basic inverted-index–based method, consisting of the following two phases:

Filter Phase: To process a ranking query q with similarity threshold θ, the inverted index is queried for each item in the the query. The obtained index lists are merged to identify the distinct rankings that overlap with the query ranking (in at least one item). These are considered candidates.

Validation Phase: For each identified candidate ranking θ, the distance $d(q, \tau)$ to the query is computed. And all θ with $d(q, \tau) \leq \theta$ are returned as results.

inverted index with ranks

item 2 $\longrightarrow$ $<(\tau : 0),(\sigma : 0)>$
item 1 $\longrightarrow$ $<(\tau : 1)>$

Figure 2.5: Example position-augmented posting lists for the top-k rankings shown in Figure 2.6

.

Doing so, the plain inverted index is sufficient, as no rank information is required. This Filter and Validate approach is also independent of the actually employed distance function, which does not even have to be a metric. Again, we obviously assume $\theta < 1$. Otherwise, rankings τ_i that are not overlapping with q at all, would be missed in the results, as these cannot be identified in the filter phase using an inverted index.

In the inverted index augmented with rank information, this additional lookup is not required. In the latter, the index lists can be sorted by ranking id for efficient multi-way merge operations; or be ordered by rank to be able to derive early stopping and pruning thresholds. These more efficient processing techniques are discussed later in the thesis.

Processing Keyword Queries

Our inverted index processing techniques follow some well-known methods for processing keyword queries from the information retrieval field.

The task of keyword queries is, given a small set of keywords, to return the most relevant documents, according to some scoring function, from a large collection of documents. To achieve this inverted indices are used. There are many different approaches for realizing this, and various implementation details, which we are not going to describe here. Instead, we are going to briefly outline two of the most common approaches for processing the index lists in the case of keyword queries:

Term-at-a-Time: In a Term-at-a-Time (TaaT) query processing, the terms of the keyword query are processed one at a time. This means, that for each term in the query, we retrieve its posting list, and read it completely, before starting to process the posting list for the next term. In order to compute the score for each document that contains the terms, an accumulator is stored and updated while processing the lists.

Document-at-a-Time: In a Document-at-a-Time (DaaT) query processing, the posting lists of the terms in the keyword query are processed concurrently. In this type of query processing, the posting lists are ordered by the document id, and with this, the score of one document can be completely evaluated at

once. The posting lists are traversed in parallel and aligned on a matching document.

2.2 Top-k Rankings

Complete rankings are considered to be permutations over a fixed domain $\mathcal{D}$. We follow the notation by Fagin et al. [FKS03] and references within. A permutation σ is a bijection from the domain $\mathcal{D} = \mathcal{D}_\sigma$ onto the set $[n] = \{1, \ldots, n\}$. 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 instance, in Figure 2.6, item 2 is ahead of item 5 in τ, thus $\tau(2) < \tau(5)$ holds.

τ

rank $\tau(i)$	item i
0	2
1	1
2	5
3	3

σ

rank $\sigma(i)$	item i
0	2
1	5
2	7

Figure 2.6: Two top-k lists containing items with domains $\mathcal{D}_\tau = \{1, 2, 3, 5\}$ and $\mathcal{D}_\sigma = \{2, 5, 7\}$.

We consider incomplete rankings, called top-k lists in [FKS03]. 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., $D_{\tau_1} \neq D_{\tau_2}$. Intuitively, a top-$k$ ranking is an ordered list of distinct items, which represents an incomplete ranking of the total amount of known items. Figure 2.6 shows two top-k rankings τ and σ of different length with two items in common, called overlapping items.

2.2.1 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 introduce metrics over complete rankings over a single domain and then we discuss results on computing distances for top-k lists (incomplete rankings).

For two permutations σ_1 and 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.

The Kendall's Tau between two permutations σ_1, σ_2 is computed as the sum over the concordant and discordant pairs in one ranking. To put that more formally, let $\mathcal{P} = \{(i, j) | i \neq j \text{ and } i, j \in D\}$ be the set of unordered pairs in the domain of the rankings. Then, Kendall's Tau for two permutations σ_1, σ_2 is computed such that, for each pair (i, j) in $\mathcal{P}$ we add one to the distance if i and j

σ_1

rank $\sigma_1(i)$	item i
0	2
1	1
2	3

σ_2

rank $\sigma_2(i)$	item i
0	2
1	3
2	1

Figure 2.7: Two permutations σ_1 and σ_2 containing items from the domain $\mathcal{D} = \{1, 2, 3\}$, $P = (1, 2), (1, 3), (2, 3)$. $K(\sigma_1, \sigma_2) = 1$, $F(\sigma_1, \sigma_2 = 2)$

are in the opposite order in σ_1 and σ_2. For instance, consider the permutations σ_1 and σ_2 given in Figure 2.7. Their Kendall's Tau distance is $K(\sigma_1, \sigma_2) = 1$ since only the pair $(1, 3)$ is in an opposite order, i.e., in σ_1 item 1 is ranked higher then item 3 while in σ_2, item 3 is the better ranked one. The other pairs in $\mathcal{P}$ are in the same order in both permutations.

Spearman's Footrule metric is the L_1 distance between two permutations, i.e., $F(\sigma_1, \sigma_2) = \sum_i |\sigma_1(i) - \sigma_2(i)|$. Thus, the Footrule distance for the permutations σ_1 and σ_2 given in Figure 2.7 is $F(\sigma_1, \sigma_2) = 0 + 1 + 1 = 2$.

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.

Fagin et al. [FKS03] discuss how the above two measures can be computed over top-k lists. For both Kendall's Tau and for Spearman's Footrule distance, they propose several adaptations, depending on how the non-overlapping items are handled. In this thesis, we specifically focus on using Spearman's Footrule adaptation that is also a metric for top-k lists, and thus in the following we only describe this measure. For a solution of the similarity search problem for top-k rankings, where Kendall's Tau is used as a distance function, we refer the reader to [Pal18].

Fagin et al. [FKS03] define the *Footrule distance with location parameter* l, such that for the items that belong to only one ranking σ_1 an artificial rank l, $l > k$ is assumed, i.e., $\sigma_1(i) = l$ if $i \notin D_{\sigma_1}$. Considering the rankings in Figure 2.6, we can calculate the distance of τ_1 and τ_2 by setting $l = 4$:

$$\begin{aligned} F(\tau_1, \tau_2) &= |\tau_1(2) - \tau_2(2)| + |\tau_1(1) - \tau_2(1)| + |\tau_1(5) - \tau_2(5)| + |\tau_1(3) - \tau_2(3)| \\ &\quad + |\tau_1(7) - \tau_2(7)| \\ &= |0 - 0| + |1 - 4| + |2 - 1| + |3 - 4| + |4 - 2| \\ &= 7 \end{aligned}$$

One can see another very intuitive behavior of this distance measure. When comparing an item i, with $i \in \tilde{D}_{\tau_1}$ and $i \notin \tilde{D}_{\tau_2}$, there is a smaller penalty when i is ranked worse in τ_1 than if it is further ahead in the ranking. Fagin et al. prove that this measure is a metric, for all values of the parameter l.

In this thesis, we assume that $\tau(i)$ takes values from 0 to $k - 1$ (instead of 1 to k), It is clear that this does not affect our algorithms. When it comes to the calculation of the Footrule distance between two rankings τ_1 and τ_2,

$$\begin{aligned} \textbf{Map}: &\quad (k_1, v_1) &&\mapsto list(k_2, v_2) \\ \textbf{Reduce}: &\quad (k_2, list(v_2)) &&\mapsto list(v_3) \end{aligned}$$

Figure 2.8: Map and Reduce function. Note that k_i are keys while v_i are values.

we fix the value of l to k as suggested in [FKS03]. 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 thesis, for ease of presentation, unless otherwise specified, we use normalized values for the Footrule distance and θ, ranging from 0 to 1, i.e., $d_{max} = 1$.

2.3 Platforms for Distributed Data Processing

2.3.1 MapReduce

In 2004, Jeffrey Dean and Sanjay Ghemawat presented MapReduce [DG04], a framework for analyzing and processing big data. This approach is based on data being organized as key-value pairs, which are then processed by a map and reduce concept, adapted from functional programming. The strength of this framework lays in the concurrency of scalable data processing in a shared-nothing cluster and the high fault tolerance on the worker side as well as on the master side.

Figure 2.8 depicts the basic syntax of the user-defined Map and Reduce functions. First, workers retrieve the key-value pairs and execute the Map function. Each worker processes one chunk of the input data file, called a split. For each record, the Map function emits a user defined set of intermediate key-value pairs (k_2, v_2). The next step, called Shuffle phase, redistributes the pairs with identical keys k_2 to the same workers, called Reducers. Then those nodes apply the predefined Reduce function on the retrieved values v_2, usually performing some kind of merging operation, and output a list of resulting values v_3. Finally, these are collected and returned. This process can be seen in Figure 2.9

In order to reduce network traffic and usage of memory space a Combine function was introduced, which is executed before the Shuffle phase and basically does the same task as the Reducer function, at each mapper node.

Some example problems that can easily be implemented and executed using MapReduce are counting the frequencies of the words in a collection of documents, building an inverted index, a distributed grep, etc. A popular open-source implementation of MapReduce is Apache Hadoop [HDP].

Alongside criticism about the incompatibility with other DBMS tools and

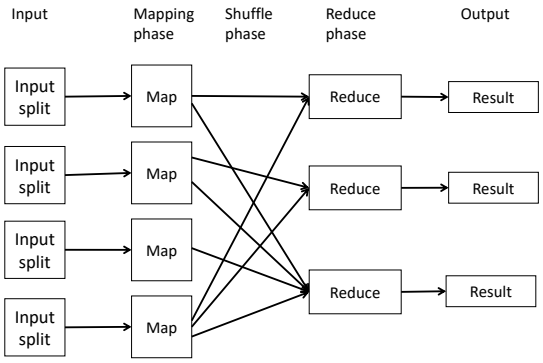


Figure 2.9: MapReduce data processing flow.

being a low-level language not able to describe larger programs efficiently, the main flaw of the framework is the restrictive programming paradigm based on acyclic data flows, which can result in an unnecessary slow computation of complex algorithms.

2.3.2 Apache Spark

Apache Spark [SPK] is a general purpose, open-source platform that enables easy and fast development and execution of distributed application. It is considered as an evolution of MapReduce since it provides the same capabilities with an improved performance. Additionally, several other functionalities are provided and many libraries are built on top of its core. Parallelization of applications is easier when using Spark due to the notions of RDD, *transformations* and *actions* used in the platform. Spark performs computations on Java Virtual Machines (JVMs).

An important characteristic of Apache Spark is its ability to execute iterative processes, using mostly the main memory in order to reduce disk I/O, thus reducing the overall execution time of the application. This is the main difference between Spark and MapReduce, thus the better performance of the former, as shown by Shi et al. [SQM+15]. The reason for such difference is that when using MapReduce it is necessary to have a map and a reduce phase, and the reduce phase necessarily needs to store its results. If a job can not be done in a single map and reduce phase, a second map and reduce is necessary, and thus, the data written by the first phase needs to be read from disk again. Apache Spark does not have this drawback, and keeps intermediate results in memory, whenever possible.

On its own, Spark is not a data storage solution. However, it supports different data storage systems, as for example HDFS [SKRC10], Apache Cassandra [CSD], HBase [HBS], or the local file-system as well, when running in local mode. Furthermore, Spark is often used together with a cluster manager, which coordinates the distribution of Spark applications. Supported cluster

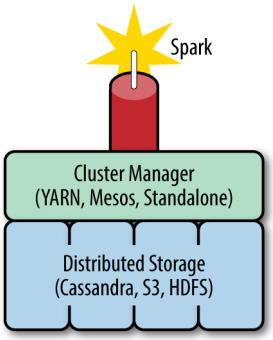


Figure 2.10: Distributed data processing ecosystem, including Spark (Image source: [KW17]).

managers include *Hadoop YARN* or *Mesos*, where the management is done by these external platforms, creating containers where the Spark application will run using the desired amount of resources, or the so called *Standalone Cluster Manager* for more simple setups where Spark takes care of the entire management. Figure 2.10 shows a diagram of the data processing ecosystem, including Spark [KW17].

In addition to its core functionalities, the Spark supporting infrastructure includes several APIs, like GraphX for handling graphs, MLib for machine learning tasks, Spark Streaming to handle streaming data and Spark SQL for working with structured data. In this thesis we rely only on the core components of Spark, since it provides all necessary operations for implementing our algorithms.

Resilient Distributed Dataset (RDD)

Spark represents datasets as RDDs, which are immutable, distributed collections of elements stored across the nodes of a cluster, i.e., the executors [KW17]. Once created, RDDs are partitioned among the available nodes of a cluster. This way each node, or executor, handles a subset of the data. RDDs are evaluated lazily, meaning that, instead of directly computing each RDD transformation, the computation is performed only at the end, when the final RDD data needs to be computed. Since they are immutable, and fault-tolerant, whenever an operation is performed on an RDD, a new RDD is created based on the previous one and both can be accessed through their pointers.

Furthermore, RDDs can contain any type of Java, Python, or Scala objects, as well as user defined classes, thus, providing flexibility on the possible data types to be processed. Spark works preferably with the system's main memory, i.e., stores RDDs in the main memory of the executors, allowing faster repeated computations. However, in the case of large intermediary result, it is also possible to store data on disk.

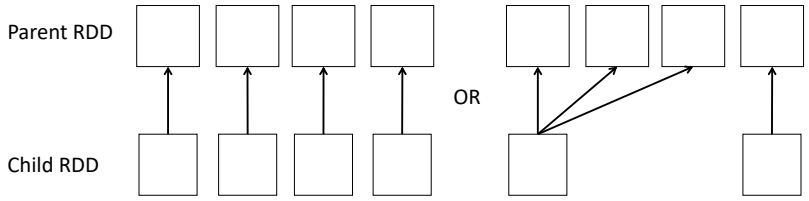
Transformations, Actions, and Lazy Evaluation

Spark supports two types of operations, which defer in the way they are evaluated, and the result that they return [KW17]:

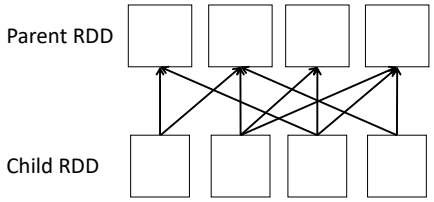
Transformations: Transformations are operations executed on RDDs, which are lazy evaluated and always return a new RDD as a result. We refer to these RDDs as parent and child RDD of the transformation. Based on their dependencies, there are two types of transformations, transformation with wide dependencies, and transformations with narrow dependencies. The type of the transformation has a great effect on its evaluation. Transformation with narrow dependencies are those where there are only simple, finite dependencies between the partitions of the child RDD and the parent RDD, and thus, can be determined at design time. On the other hand, this is not the case with the transformations with wide dependencies, where data usually needs to be partitioned in a specific way, and thus, usually require shuffling of the data across partitions. These transformations are usually more expensive, and thus need to be used carefully. Figure 2.11 shows the difference in the dependency graph between the partitions of parent and child RDDs for the two types of transformations. Several transformations are available and the most common ones with narrow dependencies are *map* and *filter*, and transformations with wide dependencies include *sort*, *join*, *groupByKey*, etc..

Actions: Actions, on the other hand, do not transform the RDDs, but rather, they bring the data to the driver, or write it to disk. Common actions are *count* and *saveAsTextFile*: the first counts the number of elements in the collection and the second stores the RDD to persistent storage. Note that all Spark programs must have at least one action, since actions are the ones that force the evaluation of the program.

Lazy evaluation: Another key characteristic of Spark is lazy evaluation. Spark does not begin computing the partitions of an RDD, until an action is called. To achieve this, Spark first creates a *directed acyclic graph* (DAG) of execution stages, each consisting of possibly multiple transformations and actions. Actions are leafs in the execution DAG, since the edges in a DAG are based on the dependencies between the partitions of the RDDs. Each execution DAG with one action and all the transformations needed for computing that action, forms a *job* in Spark. Jobs are further divided into *stages* by wide transformations that require shuffling of the data. This means that one stage ends, and another stage begins whenever there is a need for a network communication between worker nodes. In addition, the partitioning of the RDDs can also play a role in the forming of the stage boundaries. Depending on the type of the transformations, the engine can optimize the execution, allowing Spark to avoid unnecessary data reads and writes, reducing I/O. For instance, Spark can combine narrow transformations, like $m$ap and $f$ilter, performed on the same RDD,



(a) Narrow transformations



(b) Wide transformations

Figure 2.11: Dependency graph between the partitions of parent and child RDDs for the two types of transformations (Image source: [KW17]).

to avoid multiple passes over the data. Stages further consist of *tasks*, which are the smallest unit of computation in Sparks's execution hierarchy. Each task represents one local computation.

Modes, Runtime, and Tuning

In Spark, two deployment modes are provided: *local* and *cluster* [KKWZ15]. The first aims at local machines, mainly for development purposes, where one can develop and test an application without the need of a cluster. The second mode is for production applications, running in a cluster with up to several thousand nodes. There are two main interfaces for interacting with the platform, namely *spark-shell* and *spark-submit*, both capable of running in either deployment mode. On one hand, spark-shell offers an interactive way to run commands and check for its outcomes, which is very handy during development since one can easily check the behavior of the application and can take advantage of running it either in a local machine or in a cluster itself, which is desirable when big amounts of data are being processed. On the other hand, spark-submit allows submitting an entire task to Spark's environment and run it. This, for example, can be done providing a Java or Scala *.jar* file and setting the main class and the method to be called.

Spark has one *driver* and more *executors*. *ExecutorBackends* represent a bridge between the driver and an executor, i.e., there is one back-end per executor. As there is one single driver running in the environment, several back-ends are connected to it updating the status of the executors as the task progresses and send heartbeats to check if the executors are still alive and executing. Figure

2.12 shows such architecture, including also the placement of resource manager when it is used.

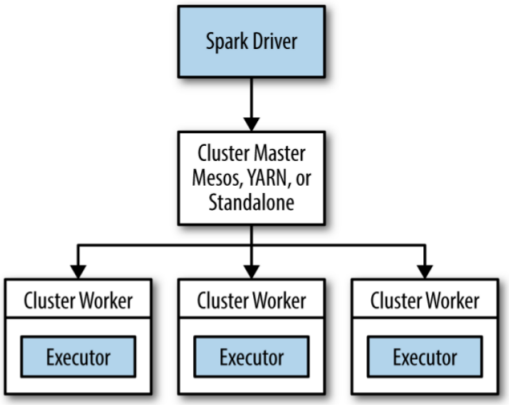


Figure 2.12: Spark Runtime Architecture (Image source: [KKWZ15])

Finally, tuning Spark to make full use of available resources is also required to achieve the best possible performance using the underlying hardware. One can tune the total number of executors, the memory allocated for each executor and the memory allocated for the driver in each node of the cluster. This can be done either by passing the configuration as Spark's arguments when running an application or using the interactive shell utility, or by setting and configuring files inside Spark's installation. General recommendation is, in order to increase the parallelism, to allocate to each executor more than one core; the exact number depending on the configuration of the cluster.

Chapter 3

Related Work

This chapter gives an overview of state-of-the-art approaches for (all-pairs) similarity search. We primarily distinguish two indexing paradigms for handling ranked lists. First, considering the distance function used, i.e., Spearman's Footrule distance, for which Fagin et al. [FKS03] show that it retains its metric properties also for incomplete rankings under certain assumptions, and applying indexing techniques for metric spaces. Second, treating ranked lists as plain sets and indexing them using methods like inverted indices. Thus, in Section 3.1.1 we describe works that can handle different types of similarity search queries for set-valued attributes. In Section 3.1.2 we describe indexing techniques in metric space. Section 3.1.3 focuses on indexing techniques that can handle incremental k-NN queries. Section 3.2.1 presents work on in memory all-pairs similarity search approaches and Section 3.2.2 presents distributed approaches.

3.1 Similarity Search

3.1.1 Indexing Techniques for Set-Valued Atributes

Helmer and Moerkotte [HM03] 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 further validation should be computed. The focus in their work is on set containment queries, i.e., given a query q the task is to find all sets $s \in S$ such that $q\theta s$, where $\theta \in \{\subseteq, \equiv, \supseteq\}$. They compare four indexing methods, three signature-based retrieval methods, and an inverted index. The signature based methods work such that sets are represented using binary signatures. Using a coding function, each set is mapped into a signature of length b, where k bits are set. During search time, the signature of the query is checked against the signature of the object. If both signatures are equal, then the pair is a possible result and needs to be verified. Otherwise, we can be sure that the two sets are not equal. The authors propose and compare three indexing schemes for the set

signatures, sequential, hierarchical and partitioned. In their experimental study they showed that inverted index seems to perform better than signature-based index structures.

Mamoulis [Mam03] addresses processing joins between relations of tuples with set-valued attributes. They propose new algorithms for processing set containment joins over set-valued attributes by using inverted indices, and show that inverted indices perform better than signature based indices.

Terrovitis et al. [TPVS06] propose a new indexing structure, the HTI index, that is a combination of inverted index and trie-trees for indexing set valued attributes with the aim of efficient evaluation of containment queries. The proposed index performs especially well for skewed datasets. To adapt the trie for indexing sets, they first impose a canonical ordering on the sets, i.e., the sets are sorted according to the items frequency in a descending order. Then, they insert the items of the sets to the trie, such that, for items that share a common prefix, only the additional items are added. For each trie node, the set to which this node belongs is stored. The HTI index combines the trie with an inverted index, such that, the items that appear most frequently in the set, are indexed using a trie, and the less frequent items, are indexed using an inverted index.

Terrovitis et al. [TBV+11] further address the problem of answering set containment queries for skewed datasets. For this purpose they present an ordered inverted file (OIF) which is again a combination of two index structures, an inverted index, where the index lists are ordered using a specific ordering, and a B-tree index, which is used for fast access of the needed positions in the inverted lists.

All of these indices also support k-NN queries over set-valied attributes. Other work on k-NN search in databases [BCG02] transforms the k-NN problem into a range query over the involved dimensions, that can be answered using standard database indices that support range queries, like B+ trees [BM72].

Note, however, that these works assume that data resides on disk, and thus, concentrate on reducing the number of page accessed on the disk, while in our work we assume that the data can completely fit in the main memory.

3.1.2 Metric Space Indexing Techniques

The book by Hanan Samet [Sam06] or the article by Chávez et al. [CNBM01] give a comprehensive overview of indexing techniques for metric spaces.

When the data can be represented using vectors, then we are dealing with a special case of the metric space, called vector space. In vector space the coordinate information is used when designing and searching the index structures. The most popular index structures for vector space, also called spatial access methods, are the R-tree [Gut84], the kd-tree [Ben75] and the quad-tree [Sam84].

The R-tree [Gut84] hierarchically groups nearby objects in a minimum bounding rectangle (MBR). It is a height-balanced tree, that contains internal non-leaf nodes and leaf nodes. Each node corresponds to a page on the disk and

each node has a predefined minimum and maximum number of entries. The non-leaf internal nodes contain pointers to the children nodes and they store the smallest bounding rectangle, that contains the rectangles of its children. Each leaf node also contains a minimum bounding rectangle, which is the bounding box of the data objects which are indexed. Additionally, the leaf node contains pointers to the data objects stored in the corresponding node. When searching the tree, the MBR of the nodes are checked against the MBR of the query object, and if there is an overlap, the corresponding subtree is searched. The quad-tree [Sam84] is a tree structure where each internal node has at most four children. The idea is that the data is divided in a grid like manner. The data are placed in the "right" quadrant by comparing its coordinates to the ones of the parent node. The kd-tree [Ben75] is an improvement over the quad-tree. It is a binary tree where the data at each level is divided into two partitions based on the value of one coordinate.

Furthermore, for metric spaces, many data-agnostic structures for indexing objects are known. In addition to the BK-tree, described in Chapter 2, another index structure, designed to work with a discrete metric distance functions, is proposed by Baeza-Yates et al. [BYCMW94]. They propose Fixed Query Trees (FQT), which gets its name because the keys on one level of the tree are the same, and thus, the comparisons on one level do not depend on those previously made. As keys are chosen random elements from the objects being indexed, and the leave nodes contain at most b objects. The tree is build recursively until there is a leave node with less than b elements. When searching the tree, the distance from the query to the key node is computed, and again, based on the triangle inequality, only parts of the tree are searched recursively. Further improvements over the FQT are also presented, like the Fixed Height FQT [BYCMW94] and the Fixed Queries Arrays [CMN01], both further reducing the number of distance function computations done while searching the tree.

Similarly, there are also many metric index structures that adapt to continuous distance functions. The Vantage Point (VP) tree [Yia93] is a binary tree, where at each level a random element is chosen as a root, i.e., a pivot. Then, the distance of every element in the dataset to the root element is calculated and the median is computed. Then, every element with distance smaller than the median is placed in the left subtree, and the rest, in the right subtree. For searching the tree the splitting median value is used, and the distance of the query to the pivot element, to decide which subtree needs to be searched. This is similar to the k-d tree [Ben75]. There are many variations of the VP tree [BÖ97, Yia93, Chi94] all trying to reduce the distance function computations done while searching.

Another option to partition the data is by the generalized hyperplane partitioning strategy. In this partitioning strategy instead of choosing only one point to partition the data, two objects are chosen, and data is partitioned depending on its closeness to these objects. The generalized hyperplane tree (gh-tree) [Uhl91] is a tree that applies this partitioning strategy. Two random

elements are used to split the space. Points closer to the first element are placed in the left subtree and those closer to the second are placed in the right subtree. This is applied recursively, resulting in a binary tree. While searching the tree, the distance from the query to the reference objects is computed, and based on its distance, and the triangle inequality, it can be calculated if the subtree needs to be searched or not. The Bisector tree [KM83] is again a binary tree which is very similar to the gh-tree. The only difference is that the radius of each partition is kept and used while searching the tree, to decide which subtree needs to be searched. This, depending on the data, can sometime lead to tighter bounds around the partitions. There are many improvements over the Bisector tree, like the work by Noltemeier et al. [NVZ92] where the covering radius of the subtrees is decreased when moving down the tree.

One of the most well known metric index structures is the M-tree by Ciaccia et al. [CPZ97, ZSAR98], which behaves similarly to the R-tree [Gut84]. R-trees can only store vector space data, or spatial data. M-trees generalize this principle for metric space. They are balanced trees that aim at reducing the I/O cost of the index, in addition to reducing the distance function computations. M-trees are also capable of dynamic inserts and deletions and several strategies for splitting of nodes have been proposed.

All of the aforementioned approaches have the problem that as the (intristic) dimensionality of the data increases, they start to perform poorly. There are some proposed approaches addressing this specific issue. Chávez and Navarro [CN05] 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. Similarly, in our proposed approach for similarity search of top-k rankings (Chapter 5) 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—effectively trading-off cluster retrieval time and final result validation cost. The partitioning can be done in any of the above ways; however, we choose the BK-tree [BK73].

3.1.3 K-NN Queries under Categorical Constraints

All of the aforementioned works can handle standard k-NN queries, however, they are not designed for handling class-constraint k-NN (CCk-NN) queries, which we tackle in our work presented in Chapter 7, and thus, would perform worse for these types of queries. In this section we present approaches that are more suitable for handling this special type of k-NN queries.

Hjaltason and Samet [HS99] propose an algorithm that can compute the $k+n$ nearest neighbors, if needed, without accessing the already searched data again, i.e., the processing is incremental. This way CCk-NN queries can be efficiently answered, since if retrieving the first k nearest neighbors results in having items that do not satisfy the constraints, the index needs to be further searched for additional nearest neighbors that do satisfy the imposed constrains.

To allow the computation of the $k+n$ nearest neighbors, they employ a modification of an R-tree [Gut84]. The standard strategy to find nearest neighbors within an R-tree is to make a depth-first search on the tree to find the rectangle, which holds the query point at the leaf node. Because neighbors do not need to reside in the same node as the query point, the tree is tracked back higher and the larger bounding box is searched, until all k items are found. To support the idea of getting always the next nearest neighbor, they implement a "best-first" approach, backed up by a priority queue. This queue contains bounding boxes and objects in such a way, that they are sorted by their distance to the query point, and thus, allows for finding the next nearest object. It is also ensured, that bounding boxes have higher priority than all their contained objects. When now the first element of the queue is picked, all objects closer to the query objects have already been examined, thus, the next nearest neighbor or a bounding box would be found. In the case when a bounding box is found, its sub bounding boxes or contained objects are inserted into the queue with respect to their distance to q. The threshold of the population can vary, thus, the query has to be able to dynamically expand the search area. On the other hand, when a point is hit, it means that the next nearest neighbor to q is identified. It is guaranteed, that the point is in the queue and is the next nearest, since its bounding box triggered the inclusion of the point and the boxes distance to q is always smaller. This procedure is repeated until the queue is emptied.

Another indexing strategy that allows incremental searching is the iDistance, presented by Jagadish et al. [JOT+05], already explained in Chapter 2.

A similar problem to the CCk-NN search is the problem of location-based web search. The main task of this problem is to determine documents that are in terms of content and location relevant to the query. Many approaches that combine information retrieval techniques with nearest neighbors search have been proposed. These are summarized in [JP08] and [CCJW13]. Zhou et al. [ZXW+05] investigate and compare the performance of three indexing strategies: indexing the data using both an inverted index and an R*-tree, indexing the first using an inverted index, and then, indexing the objects in each posting list using an R*-tree, and indexing the data first using an R*-tree, then an inverted index. They show that the hybrid indices perform better than double indexing the data. Cong et al. [CJW09] propose a new indexing structure named IR-tree. The IR-tree is in fact an R-tree whose nodes are augmented with inverted files which can provide estimation of the document's score, in a certain subtree, while searching the tree. This allows searching the tree in an R-tree style. Although our approach for handling CCk-NN search also combines inverted indices with a tree-based indexing structure, we address the more general problem of handling any type of data where classes are present. Therefore, the challenges presented are different, and thus, the proposed solution as well.

3.2 Similarity Joins

3.2.1 In-memory All-Pairs Similarity Search

There is an ample work on computing the all pair similarity join for sets or strings. Mann et al. [MAB16] summarize and compare the in-memory based approaches.

Previous approaches are mainly based on a filter and verification framework, which uses inverted indices as the initial filter for pairs that do not have any items in common. First, in the filter phase, a set of candidate pairs is generated. This set is kept small by applying additional filters that filter out the dissimilar pairs. In the next verification phase, the candidates are verified by computing their true similarity score in order to find the true results.

Recently, the most prominent techniques for answering set similarity joins are the prefix-filtering based methods [XWLY08, BMS07, CGK06]. The main idea behind this method, initially proposed by Chaudhuri et al. [CGK06] and Sarawagi and Kirpal [SK04] is to reduce the size of the inverted index, and thus, the number of candidate pairs. It works by first imposing a total ordering of the elements in the universe $\mathcal{U}$, sorting all records in the dataset in the same canonical order, and then, indexing only a prefix of the records with an inverted index. The size of the prefix depends on the threshold, the length of the records, and the distance, i.e., similarity measure used and its value. The prefix-filtering principle guarantees that all similar records for the given similarity value will have at least one common token in their prefixes. The initial papers propose that the records should be sorted by the ascending frequency of the elements in the sets, which makes the index lists significantly smaller, and thus, less candidate pairs will be generated. The framework guarantees that no true candidate will be missed, however, the generated candidates still need to be verified if they qualify given the similarity threshold.

There are many works [BMS07, XWLY08, RLW+13, WLF12, WQL+17] that propose improvements over the initial prefix-filtering algorithm.

Bayardo et al. [BMS07] focus on algorithms for finding the similar pairs given a large collection of sparse vectors. For comparing the vectors, they use the Cosine similarity measure. They propose the AllPairs algorithm where in addition to filtering based on the prefix of the vectors, they further filter candidates by sorting the vectors in the dataset by decreasing order of their maximum weight. Then, by keeping an upper bound of the score, they are able to index fewer features of the vector, while still ensuring the correctness of the algorithm. In addition to this, they introduce filtering of the candidates by their size, and by computing an upper bound between the candidates and filtering them, if the upper bound cannot reach the threshold.

Xiao et al. [XWLY08] further improve the AllPairs algorithm. They propose the PPJoin+ algorithm, one of the best known prefix-filtering algorithms. PPJoin+ algorithm outperforms previous AllPairs algorithm by introducing suf-

fix filtering, filtering based on the tokens positions, and by indexing fewer tokens from the records. The positional filtering works by computing an upper bound of the overlap between the unseen parts of the two records, based on the position of the token under consideration, and the fact that all the records are sorted in a canonical order. Then, if the sum of the overlap computed so far and the estimated upper bound for the unseen part is smaller than the threshold, the candidate pair is filtered out. The suffix filtering works by probing the records for random tokens in the yet unseen, suffix of the record. Then, using this probing token, the suffix is partitioned into a left and right partition. By using an equivalent Hamming distance constraint to the overlap constraint, and the sizes of these partitions, they are able to further eliminate candidate pairs.

Wang et al. [WLF12] propose the AdaptJoin algorithm. They claim that if the length of the prefix of the records is varied, a better performance can be achieved, because lengthier prefixes have more pruning power, and shorter prefixes require longer verification time. Thus, they present an adaptive prefix scheme where by using a cost model, they estimate for each object, which length of the prefix would lead to the best performance. The adaptive index they build supports storing different prefix lengths for the objects. Furthermore, they also propose an adaptive framework for processing similarity search queries, AdaptSearch. For creating the adaptive prefix indexing scheme, the threshold needs to be known upfront, which is not the case for similarity search queries. Thus, they propose grouping the sets by their size and creating an adaptive indexing framework for the maximum threshold that can be applied for each group of rankings.

Recently Wang et al. [WQL+17] motivated by the conclusions presented in [MAB16] proposed an approach that improves upon existing prefix-filtering approaches by introducing index level and answer-level skipping. The index level skipping reduces the unnecessary checks done by position and length based filters, by using length-sorted skipping blocks in the posting lists augmented with the positions of the elements in the sets. The answer-level skipping is based on the idea that the answer sets of similar sets should be also similar, thus, the already computed answer set of one set is used for computing the answer set of another, similar, set.

Since top-k rankings can be seen as sets as well, all of the presented approaches can, with modifications, be applied to our problem setting as well. Note however, that some of the filtering techniques presented could not be applied to our setting, or if applied, would lead to no or very little benefit. Therefore, in Chapter 5 and Chapter 6 we present solutions better suitable for our problem setting.

Jacox and Samet [JS08] proposed in memory algorithms for the similarity join problem in metric spaces, later used as basis for the distributed approaches. The algorithm they propose partitions the data recursively, using either the ball partitioning method, or the hyperplane partitioning method, until the partitions are small enough so that objects in it can be compared using a nested loop

join. To keep the correctness of the algorithm, also some objects from different partitions need to be compared. Specifically, the objects within distance θ, where θ is the threshold used, of the radius of the partitions need to be also compared, in order not to miss any true result.

Li et al. [LDWF11] present PassJoin, an algorithm specifically designed to solve the problem of similarity join for strings, where the distance measure used is the edit distance. They partition the strings into $\theta + 1$ non-overlapping segments, where θ is the distance threshold. Based on the pigeon hole principle, it is guaranteed that if two strings s and r have edit distance higher than θ then s must contain at least one substring, which matches a segment of r. The non-overlapping segments are indexed using an inverted index. To compute the join, they traverse the dataset D in an ordered manner, starting with the shortest strings, and for each string $s \in D$ they check if any subset of the string matches some of the indexed segments. These are candidate results and need to be verified. They further develop techniques for choosing which substrings of a string need to be taken for querying the inverted index, and for efficient evaluation of the candidate pairs.

3.2.2 MapReduce-based All-pairs Similarity Search

To handle larger datasets, many distributed solutions for all-pairs similarity search have also been proposed. Recently, Fier et al. [FAB+18] summarized and compared the MapReduce-based all-pairs similarity search solutions.

Vernica et al. [VCL10] present a distributed solution, referred to as VJ, based on the well known prefix-filtering method. The algorithm they propose is a three stages approach. The first stage is needed so that the records could be sorted into the same canonical order. To do this, first, they sort the tokens in the records according to the frequency of the elements in the sets. Then, in the next stage, the sorted tokens are loaded in the memory of the mappers and while processing the records, they are first sorted in ascending frequency, and then the element as key, and the whole set as value is emitted, but only those elements that belong to the prefix. Then on each machine, for all the rankings that share at least one element, in the reduce phase the PPJoin+ algorithm [XWLY08], is used to find the similar rankings. In the final stage, duplicate pairs must be removed, since the same pair can be generated at several machines and the complete records are retrieved.

The *V-SMART* algorithm adopts a different idea from prefix filtering, of distributively computing the ingredients of the similarity measure which are later joined to compute the final results. Their approach is initially presented for the purpose of self-similarity joins of multisets, but with no, or little modifications can be applied to sets as well. In their approach they, first, analyze the similarity measures and conclude that the similarity measures used for comparing sets, e.g., Jaccard, Dice, Cosine, can be computed using partial results. The partial results for these measures can be computed by scanning the elements in only one

dataset, also called as unilateral functions, combined with scanning the elements of the intersection of the two multisets (called conjunctive functions), which can be more efficiently done by using an inverted index. For instance, the Jaccard similarity for two sets S_1 and S_2 is computed as: $J(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$, where $S_1 \cup S_2 = |S_1| + |S_2| - |S1 \cap S_2|$. The size of each set can be computed by scanning the elements in each set, and the size of the intersection, by scanning the elements in the intersection of the two sets. Thus, they propose *V-SMART* algorithm that works in two stages, a joining stage and a similarity stage. In the first stage, the joining stage, by scanning the elements in each set, the partial result is computed and joined to all the elements in the sets. In the similarity stage, the algorithm as input takes the output from the first stage, builds an inverted index, and then, while traversing the posting lists for each element s_i, emits pairs of sets together with the information needed to compute the intersection between the elements. Then, the similarity between each pair of sets is computed and only the qualifying onces are written to disk.

Deng et al. present MassJoin [DLH+14]. This approach is based on PassJoin [LDWF11], a main memory method for string similarity joins. Deng et al. implement the signature based algorithm in Map Reduce using two stages. The first stage they call the filter stage, where they filter the sets that do not have a matching signature. The mapper reads the sets from both datasets and generates the signatures accordingly. For each signature, it emits the signature as key, and the set id. All the candidate pairs that share the same signature will end up at the same reducer. As output the reducer for each set from the first dataset generates a list of candidate sets from the other datasets and outputs it as $< sid, list(rid) >$. Note that all the set pair that do not end up at the same reducer are not candidates automatically since they do not share any signature. In the next stage, called verification stage, the candidate pairs generated from the filtering stage are verified using two MapReduce jobs. To improve the performance filtering techniques are proposed that reduce the number of messages send between different stages.

Rong et al. present FS-Join [RLS+17]. They claim that their algorithm outperforms the competitors because it addresses some of the issues that previous approaches had, i.e., it does not generate duplicate results and achieves better load balancing. The dataset is both vertically and horizontally partitioned, by dividing each set into segments, and then, partitioning the data according to the segments. In addition, they propose length filters for further performance improvement.

Since Spearman's Footrule distance is a metric, metric space approaches can also be applied for finding the similar rankings. Wang et al [WMP13] present an algorithm for solving the all-pairs similarity search problem where the distance (similarity) function is a metric. They first, based on the triangle inequality, define a partitioning scheme. They state that, for a dataset D, there is an all pair similarity search (APSS) division if the set can be divided into N inner sets $\cup_{i=1}^{N} I_i = D$ and N outer sets O_i, $O_i \subseteq D$, such that $\cup_{i=1}^{N} SimSet(I_i, I_i \cup O_i) =$

$SimSet(D, D)$. The goal is to find an APPS division of the dataset D into a worksets W_i, where each W_i is a pair of inner set and an outer set $< I_i, O_i >$. In order to create an APPS division of the dataset they define a partitioning of the input dataset D into N disjoint partitions P_i, $P_i \cup P_j = \emptyset, \cup_{i=1}^{N} P_i = D$, created by randomly choosing N centroids p_i and assigning each point $p \in D$ to the partition represented by the closest centroid. As inner set of a workset W_i all the data in one partition are assigned, $I_i = P_i$ and to the outer sets O_i belong all the points $p_j \in D$ such that $p_j | d(p_j, c_i) \leq (r_i + \theta) \wedge p_j \notin P_i$, where c_i is the centroid of P_i, $r_i = max_{p_i \in P_i}, d(c_i, p_i)$ is the radius of P_i and θ is the distance threshold. The algorithm for computing the APPS division of a dataset D consists of two main stages. In the first stage, N centroids are sampled and the partition statistics, such as the radii of the partitions, are computed. In the second stage, the all pair similarity join is computed. The basic approach that the authors propose results in many duplicate pairs being computed and in uneven partitioning. Therefore, the authors propose optimizations based on reducing the number of duplicate pairs and algorithms for repartitioning, as well as compression techniques.

Similarly to the work in [WMP13], Sarma et al. [SHC14] propose a method for all-pairs similarity search in metric spaces. Their method, as they claim, works very well for very small distance thresholds. In fact, the experiments they perform using only threshold up to 0.1. The algorithm they propose works in three stages. In the first phase, similarly to [WMP13], they compute the statistic for the dataset for creating the partitions. In the second stage, they decide into which partition centroid each point should be mapped to. The novelty in this work is that they apply here some filtering techniques, both distance specific and not, which lead to having tighter partitions, and thus, fewer comparisons. In the last stage they compute the similar pairs for all partitions. They also propose a load balancing method using a $2D$ hashing technique.

Interestingly, Fier et al. [FAB+18] came to the conclusion that the approach proposed by Vernica et al. [VCL10] outperforms the other approaches in most scenarios. Therefore, in this thesis we compare our approach, presented in Chapter 6, to the one presented in [VCL10].

Chapter 4

Theoretical Bounds for Top-k Rankings

4.1 Introduction

In this chapter, we present theoretical bounds for top-k rankings, some of which found its use in the similarity search algorithms presented in the next chapters. The described bounds are especially important for adapting existing set-based similarity techniques to the setting of top-k rankings, like for instance, the prefix-filtering framework, however, they can be applied to other problem settings concerned with top-k rankings. For instance, consider the task of maintaining a set of crowdsourced entity rankings where the similarity between the rankings is used to reason about the degree of change in a set of rankings due to an update in one ranking. Since the distance between two rankings resembles not only structural but semantic similarity as well, it is reasonable to assume that once a ranking changes, it is more likely that similar rankings change, rather than dissimilar ones. If one ranking changes, this means that items that are present in this ranking changed, respectively their features. Such changes might or might not propagate to other rankings with a distance λ to the affected ranking. The bounds presented here can be used to reason about the likelihood of such a propagation. Another problem setting where these bounds are applied is in the case of exploring databases by using top-k rankings [PMM16, PMMP16]. Specifically, a user submits a top-k ranking to the system which then aims at returning a set of queries that do return a list similar to the input, when executed on the given database instance. In this scenario, the bounds are used to make the searching for the similar ranked lists more efficient.

The work presented here has been published in several of our works, depending on its application. The bounds applied to the problem of similarity search for rankings have been published at EDBT 2015 [MAM15], the ones considered in the problem for maintaining crowdsourced entity rankings have been

published at EDBT 2016 [MM16], while at KEYS 2016 [PMM16] and PVLDB 2016 [PMMP16] the bounds and its applications related to the problem of exploration of databases through top-k rankings have been published.

4.2 Bounds on Overlap

In this thesis, we often assume that only the distance between two top-k rankings is known, and, based only on this, we want to apply some pruning techniques, or make further assumption over the data. Therefore, in this section, we reason about the overlap bounds that two top-k rankings, τ_i and τ_j, with Footrule distance θ, i.e., $d(\tau_i, \tau_j) = \theta$ of each other, can have.

Lemma 4.2.1 (Minimum Overlapping Criterion) *Let τ_i and τ_j be two top-k rankings, both with length k, such that the Footrule distance between them is θ, i.e., $d(\tau_i, \tau_j) = \theta$. The* minimum *overlap ω_{min} of these rankings is given by*

$$\omega_{min} = \lfloor 0.5 \cdot (1 + 2 \cdot k - \sqrt{1 + 4 \cdot \theta}) \rfloor .$$

Proof Assume two top-k rankings τ_i and τ_j have an overlap of ω. The smallest Footrule distance between them is achieved when the overlapping items are placed at the top ω ranks of both rankings and they are perfectly aligned. Because the w best ranked items have a partial distance of 0, we can treat this Footrule distance as the distance between two disjoint rankings of length $k - \omega$. This distance can be calculated by

$$d(\tau_i, \tau_j) = (k - \omega) \cdot (k - \omega + 1). \tag{4.1}$$

Solving

$$(k - \omega) \cdot (k - \omega + 1) = \theta$$

for ω derives the smallest overlap possible between τ_i and τ_j, since rankings with an overlap smaller than ω or when the ω overlapping items are not on the top positions, would lead to a strictly higher distance.
Solving the equation leads to

$$\begin{aligned} & (k - \omega) \cdot (k - \omega + 1) && = \theta \\ \Leftrightarrow \quad & w^2 - 2 \cdot \omega \cdot k - w + k^2 + k - \theta && = 0 \\ \Leftrightarrow \quad & w^2 + \omega \cdot (-2 \cdot k - 1) + (k^2 + k - \theta) && = 0 \end{aligned}$$

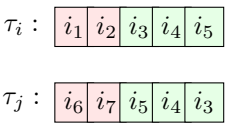


Figure 4.1: Example rankings with $k = 5$ and $\omega = 3$ with maximum Footrule distance $U(k, \omega) = 22$.

which resolves in

$$\begin{aligned}\omega_{1,2} &= \frac{2 \cdot k + 1}{2} \pm \sqrt{\frac{(-2 \cdot k - 1)^2}{4} - (k^2 + k - \theta)} \\ &= \frac{2 \cdot k + 1}{2} \pm \sqrt{\frac{4 \cdot \theta + 1}{4}} \\ &= 0.5 \cdot (2 \cdot k + 1 \pm \sqrt{4 \cdot \theta + 1}).\end{aligned}$$

Because of

$$k < 0.5 \cdot (2 \cdot k + 1 + \sqrt{4 \cdot \theta + 1}) = \omega_1, \tag{4.2}$$

meaning that the ranking length is smaller than ω, only the solution

$$\omega = \lfloor 0.5 \cdot (1 + 2 \cdot k - \sqrt{1 + 4 \cdot \theta}) \rfloor \tag{4.3}$$

is reasonable.

Note that, we require the floor function since $\omega_{min} \in \mathbb{N}$ and using the ceil function could lead to a Footrule distance greater than θ. □

Lemma 4.2.2 (Maximum Overlapping Criterion) *Let τ_i and τ_j be two top-k rankings, both with length k, such that the Footrule distance between them is θ, i.e., $d(\tau_i, \tau_j) = \theta$. The* maximum *overlap ω_{max} of these rankings is given by*

$$\omega_{max} = min(k, \lfloor (-1 + \sqrt{1 - 2 \cdot \lambda + 2 \cdot k + 2 \cdot k^2}) \rfloor).$$

Proof We again follow the same reasoning. Assume two top-k rankings τ_i and τ_j have an overlap of ω. The *maximum* Footrule distance, $U(k, \omega)$, that these top-k rankings can have, occurs when the items that they do *not* have in common are positioned in the top $(k - \omega)$ places and the common items are positioned in the last ω items in both rankings, but in reverse order.

To give an example, consider two top-k rankings τ_i and τ_j with $k = 5$ and $\omega = 3$, shown in Figure 4.1. Note that the actual items do not matter, what matters is whether the items are overlapping or not, and their positions.

Therefore:

$$U(k, \omega) = F_{max}(\omega) + 2 \cdot \sum_{i=\omega+1}^{k} i \tag{4.4}$$

where, $F_{max}(\omega)$ is the maximum Footrule distance that two *complete* rankings with size ω can have. According to Fagin et al. [FKS03], $F_{max}(\omega)$ can be computed as:

$$F_{max}(\omega) = \begin{cases} \frac{\omega^2}{2} & \text{if } \omega \text{ is even} \\ \frac{(\omega+1)\cdot(w-1)}{2} & \text{if } \omega \text{ is odd} \end{cases} \tag{4.5}$$

We use $F_{max}(\omega)$ to compute the distance between the items $i \in \mathcal{D}_{\tau_1} \cap \mathcal{D}_{\tau_2}$ in the two rankings. Since these items are always the same, positioned in the last ω positions, we can also consider them as permutations stemming from one domain $\mathcal{D} = \mathcal{D}_{\tau_1} \cap \mathcal{D}_{\tau_2}$. We use the summation $\sum_{i=\omega+1}^{k} i$ to factor in the remaining items in τ_1, $i \in \mathcal{D}\tau_1 \setminus (\mathcal{D}_{\tau_1} \cap \mathcal{D}_{\tau_2})$, and for τ_2 analogously.

Solving $U(k, \omega)$ by ω gives us the formula for computing the maximum overlap, ω_{max}, of two rankings, when the Footrule distance θ between them is known, and their size is k. □

4.2.1 Different Size Rankings

In this thesis, the presented algorithms work only on rankings of same size k. This is because when evaluating the performance of different algorithms, we want to focus mainly on the core performance of the algorithm. In this section we discuss how the aforementioned bounds can be extended to rankings of different length, and how an additional length filtering technique can be derived, based on the Footrule distance between the top-k rankings.

When we are considering the problem of comparing rankings of different size, for the Footrule distance with location parameter l, we follow the principle used for comparing top-k rankings of same size and set l using the largest ranking in the dataset, i.e., $l = max(|\tau_i|)$. Then, for two top-k rankings τ_i and τ_j, where $|\tau_i| = l_i$, $|\tau_j| = l_j$, $l_i < l_j$, the best Footrule distance that can be achieved is:

$$L(l_i, l_j) = \sum_{i=1}^{l_j - l_i} l_j - i \tag{4.6}$$

This is the case when $D_{\tau_i} \subset D_{\tau_j}$ and all the l_i overlapping items are positioned at the top ranks in both rankings, and perfectly aligned. If we assume that the rankings have ω overlapping entities where $\omega < l_i$ then the best distance that can be achieved is:

$$L(l_i, l, \omega) = L(l_i, l) + L(l_i, \omega) \tag{4.7}$$

and the worst distance that can be achieved is:

$$U(l_i, l, \omega) = L(l_i, l) + U(l_i, \omega) \tag{4.8}$$

Where $L(l_i, \omega)$ and $U(l_i, \omega)$ are the aforementioned lowest Footrule distance (Equation 4.1) and highest Footrule distance (Equation 4.4) that can be achieved

by two rankings of same length l_i and overlap of ω, respectively. By using the same logic as for rankings of same size, by setting $L(l_i, l, \omega) = \theta$ and solving this equitation by ω, we can find the minimum number of overlapping entities that we need to have between the two rankings to have a Footrule distance $d(\tau_i, \tau_j) \leq \theta$. Furthermore, $L(l_i, L)$ can be used for deriving a length filter based on the Footrule distance between τ_i and τ_j, $d(\tau_i, \tau_j) = \theta$. Solving $L(l_i, L) = \theta$ by l_i gives us the minimum length that a ranking τ_i needs to have in order $d(\tau_i, \tau_j) \leq \theta$, $|\tau_j| = l - 1$, while solving it by l gives us the maximum length that a ranking τ_j needs to have in order $d(\tau_i, \tau_j) \leq \theta$.

4.3 Prefix Size for Top-k Rankings

One of the main tasks that we are concerned with in this thesis is similarity search for top-k rankings. As discussed in Chapter 3, recent state-of-the-art similarity search techniques rely on the prefix filtering technique, where, given a distance threshold θ, we need to index only a prefix of the record of size p in order to find all similar records.

In this section we derive this prefix size for top-k rankings, where the Footrule distance is used as a metric for comparing the top-k rankings. We propose two different techniques for computing this prefix, the later providing slightly tighter prefix sizes than the first, however, the former allows more freedom in choosing the items in the prefix.

Considering Lemma 4.2.1, we define the following corollary:

Corollary 4.3.1 *For top-k rankings of size k and a minimum overlapping criterion ω_{min}, the prefix size, p, can be computed as:*

$$p = k - \omega_{min}$$

.

This corollary immediately follows from Lemma 4.2.1. It is clear that we can set the prefix size to $p = k - \omega_{min} + 1$, since we know that the minimum overlap between the top-k rankings is ω_{min}, and thus, they must have at least one overlapping item. We further limit the size of the prefix to $p = k - \omega_{min}$ by considering the way that the minimum overlap was computed. Since, as already discussed, the items of both rankings need to be ranked at the top w places, we can define $p = k - \omega_{min}$, if we include at least one top ω_{min} ranked item in the prefix.

The above definition of the prefix of a top-k ranking mainly relies on the overlap between the items, and it does not impose restrictions on which items should be included in the prefix. Next, we define an ordered prefix of top-k rankings, where we specify the number of best p ranked items that need to be included in the prefix of a ranking in order to have at least one item overlapping.

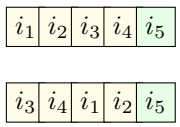


Figure 4.2: Example rankings with $k = 5$ and $p = 2$ with maximum Footrule distance $d(\tau_i, \tau_j) = 8$.

Lemma 4.3.2 (Ordered Prefix) *For a given distance threshold θ and a ranking length k, when the distance function used is Spearman's Footrule, the size of the* ordered prefix p_o *of the top-k rankings is given by*

$$p_o = \lfloor \frac{\sqrt{\theta}}{\sqrt{2}} \rfloor + 1$$

Proof The lowest Footrule distance that two top-k rankings τ_i and τ_j can have, when none of the first p items of each ranking are not overlapping, $L(p, k)$, is when the items are overlapping in the rankings, but they are positioned in the next p places in the other ranking. This is so, because the partial Footrule distance of an item we get either by the difference in its positions, when they are overlapping, or as $k - \tau_i(i)$ when the item is non overlapping. As an example, consider the rankings τ_i and τ_j where $p = 2$, shown in Figure 4.2.

For the items i positioned in the first p places in ranking, where $p < \frac{k}{2}$, the partial distance of the items being overlapping and placed at the next p places is always lower than if an item is non overlapping. $L(p, k)$ can be computed as $\frac{(p*2)^2}{2}$.

Solving $L(p, k) = \theta$ gives us the first $p = \lfloor \frac{\sqrt{\theta}}{\sqrt{2}} \rfloor$ items that can be non-overlapping in case of θ. Taking one more item guaranties that we will not miss any candidate pair. Note that this only holds when $\theta \leq \frac{k^2}{2}$. In the case when $\theta > \frac{k^2}{2}$ computing the formula for the ordered prefix size is more complicated and is out of the scope of this thesis. Using values of $\theta \leq \frac{k^2}{2}$ is more than enough for our problem setting, as it is common practice to use values of $\theta \leq 0.4$[1]. □

4.4 Bounds on Item Positioning

To increase the pruning power of our searching techniques, we further compute bounds on the positioning of the items in two top-k rankings τ_i and τ_j, when the Footrule distance between them is known. These bounds allow us to perform positional pruning while searching for similar rankings. In order to derive these bounds on the position of the items, we leverage the properties of the Footrule distance. We first define the concept of **displacement of an item i**, η_i as:

[1] $\theta = \frac{k^2}{2}$ is around $0,45$ when normalized, depending on the value of k.

Definition 5 Displacement of an Item *For two top-k rankings τ_i and τ_j, $\tau_i \neq \tau_j$, we define a displacement of an item i, $i \in \mathcal{D}_{\tau_i} \cap \mathcal{D}_{\tau_j}$, denoted with η_i, as the difference of the position of the item in the two rankings, i.e., $\eta_i = |\tau_i(i) - \tau_j(i)|$. In the case when $i \in \mathcal{D}_{\tau_i} \setminus \mathcal{D}_{\tau_j}$, $\eta_i = k - \tau_i(i)$ or vice versa.*

The Footrule distance of two lists τ_i and τ_j is in fact a sum over the displacements of the items in $\mathcal{D}_{\tau_i} \cup \mathcal{D}_{\tau_j}$. Next, with the following lemma we define the maximum value for the displacement of any item i in any two list τ_i and τ_j, where their Footrule distance is $d(\tau_i, \tau_j) = \lambda$:

Lemma 4.4.1 (Maximum Displacement) *The maximum displacement of an item i for two top-k lists τ_i and τ_j of size k with Footrule distance $d(\tau_i, \tau_j) = \lambda$ is*

$$max(\eta_i) = min\{\lfloor \frac{\lambda \cdot k \cdot (k+1)}{2} \rfloor, k\}$$

Proof Restricting the maximum displacement $max(\eta_i)$ to k is clear. Considering the fact that the rankings (lists) are of size k (have k items), by the definition of a displacement it follows that the $max(\eta_i)$ for two rankings τ_i and τ_j cannot be larger than k for any distance $d(\tau_i, \tau_j)$.

Now lets look at the case of showing that the maximum displacement $max(\eta_i)$ for any two rankings τ_i and τ_j with Footrule distance $d(\tau_i, \tau_j) = \lambda$, cannot be larger than $\frac{\lambda \cdot k \cdot (k+1)}{2}$. Top-k rankings are bijections from the set $\mathcal{D}_{\tau_i}$ to itself, where $\mathcal{D}_{\tau_i} \neq \mathcal{D}_{\tau_j}$ for two top-k lists τ_i and τ_j. The Footrule distance, in this case, is a sum not only over the displacement of the overlapping items, but over the displacement of the missing items as well. Since we are working with lists of same size, we have the same number m of missing items in τ_j and τ_i. Therefore, if we ignore the fact that the missing items are different in the two top-k lists, and just consider that we have m missing items in each list and $k - m$ common items, there is a bijection between the set of k items and the lists τ_i and τ_j. Now, let us assume that $max(\eta_i)$ is larger than $\frac{\lambda \cdot k \cdot (k+1)}{2}$. If $max(\eta_i) > \frac{\lambda \cdot k \cdot (k+1)}{2}$ then we have to have at least one more item j displaced in order to keep the bijection between the two sets. Since $F(\tau_i, \tau_j) = \sum_{i \in \mathcal{D}} \eta_i$ it follows that we have at least one item whose displacement is $\eta_j < \frac{\lambda \cdot k \cdot (k+1)}{2}$, or we have more than one displaced item, but the sum of their displacements is smaller than $\frac{\lambda \cdot k \cdot (k+1)}{2}$. However, this would break the bijection relation between the two permutations, which is not possible, proving that $max(\eta_i) \leq \frac{\lambda \cdot k \cdot (k+1)}{2}$.

Therefore, it holds that $max(\eta_i) = min\{\lfloor \lambda \cdot k \cdot \frac{(k+1)}{2} \rfloor, k\}$. □

Chapter 5

Similarity Range Queries over Top-k Rankings

5.1 Introduction

This chapter is based on our own publication at EDBT 2015 [MAM15] and presents a novel index structure for efficient processing of similarity range queries over top-k rankings. This specific problem has several notable applications. For instance, consider 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. As already mentioned before, for comparing the ranking we use Spearman's Footrule metric adaptation for top-k rankings [FKS03]. Dealing with metrics immediately suggests employing metric data structures like M-trees [CPZ97] 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 [HM03, TPVS06] or joins [Mam03] is a well studied research topic. Inverted indices or signature trees are used to indexing tuples based on their set-valued attributes [HM03]. Such indices are very efficient to answer contained-in, equal-to, or overlaps-with queries, but do not exploit the distances between the indexed objects, as metric index structures do. In this chapter, we present 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.

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

Table 5.1: Sample dataset $\mathcal{T}$ of rankings where items are represented by their ids.

5.1.1 Problem Statement and Setup

As **input** we are provided with a dataset $\mathcal{T}$ of rankings τ_i. 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 query performance. The considered rankings do not contain any duplicate items. Table 5.1 shows an example dataset $\mathcal{T}$ of three rankings.

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 this work, we assume that items are represented by their ids. The rank of an item i in a ranking τ is given as $\tau(i)$. In Table 5.2 a summary of the notations used throughout this chapter is shown.

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 thesis, we focus on the computation of Spearman's Footrule distance, but the proposed coarse index can be applied to any metric distance function, or any distance function that satisfies the triangle inequality.

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 ad-hoc similarity queries over rankings, where the query ranking and query similarity threshold are specified at query time. We make the natural assumption that the query threshold θ is strictly smaller than the maximum possible distance d_{max}.

5.1.2 Contributions and Outline

With this work we make the following contributions:

- To the best of our knowledge, this work is the first to consider the problem of similarity search of top-k rankings. We present a coarse index for efficient processing of similarity range queries.

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 5.2: Overview of notation used in this paper

- We present 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 describe how, unless apparent
- We show the results of a carefully conducted experimental evaluation involving a suite of algorithms and hybrids under realistic workloads derived from real-world rankings

The remainder of this chapter is organized as follows. Section 5.2 introduces a coarse, hybrid index that indexes partitions of rankings. Section 5.3 describes a cost model that allows picking the sweet spot between inverted-index-access time and result-validation time. Section 5.4 shows how to apply the bounds presented in Chapter 4 to our problem setting and to enable effective pruning of entire index lists at runtime. Section 5.5 presents the experimental evaluation, while Section 5.6 summarizes the work in this chapter.

5.2 Framework

As discussed before in Chapter 2, rankings can be considered as plain sets and accordingly indexed in traditional inverted indices [HM03] 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.

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. Although the inverted

index is good for finding rankings (sets) that intersect with the query, the F&V algorithm, described in Chapter 2 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 validation 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 [CPZ97] or BK-trees [BK73], appears promising at first glance, they are not ideal for reducing down the space to the intersecting rankings. In fact, we show in our experiments that using metric data structures is an order of magnitude slower than using pure inverted indices.

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 organization 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.

5.2.1 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 [BK73], an index structure for discrete metrics, such as the Footrule distance. Figure 5.1 depicts the general shape of 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

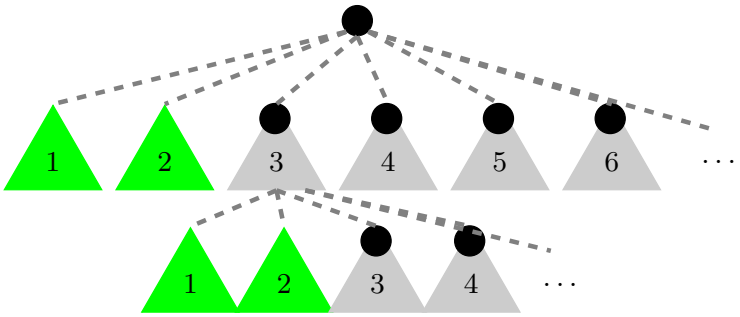


Figure 5.1: Creating partitions based on the BK-tree. The green (distance 1 and 2) subtrees are indexed by their parent node (medoid, as black dot). Distance 0 is not shown here.

partition with their root node, which acts as the medoid. In Figure 5.1, 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 [CN05], 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 of finding medoids and their partitions, medoids are rankings too and can be indexed using inverted indices. In Section 5.4 we further propose techniques for more efficient retrieval of the rankings indexed with an inverted index.

5.2.2 Query Processing

Lemma 5.2.1 *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.*

Proof The largest distance between any ranking τ and its representative medoid τ_m, $\tau_m \prec \tau$, is θ_C, i.e., $d(\tau_m, \tau) \leq \theta_C$. The triangle inequality states that for three rankings τ_m, τ and q it holds that $d(\tau_m, q) \leq d(q, \tau) + d(\tau_m, \tau)$. Since we want to retrieve only those rankings that have $d(q, \tau) \leq \theta$, it directly follows that $d(\tau_m, q) \leq \theta + \theta_C$. □

Lemma 5.2.1 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 5.2.1 avoids

method: **processCoarse**
input: QueryProcessor over Medoids qp, double θ, θ_C,
Map:Int→ BK-tree map
output: list of query results rlist
1 rTemp ← qp.execute(θ+θ_C) ▷ query with relaxed threshold
2 **for each** id ∈ rTemp
3 tree ← map[id]
4 rList.addAll(tree.execute(θ))
5 **return** rList

Algorithm 1: Query processing using the coarse index.

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 5.2.1 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.

5.3 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 [CN05], 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* [FGT92]. 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 5.2 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 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:

$$h(n,i,p) = \begin{cases} 1, & \text{if } i \bmod p = 0 \\ \frac{n-(i \bmod p)}{n-i}, & \text{otherwise} \end{cases} \tag{5.1}$$

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

$$M(n,\theta_C) = p^{-1} \sum_{i=0}^{n-1} h(n,i,p) \tag{5.2}$$

This gives us the expected number of medoids indexed by the inverted index.

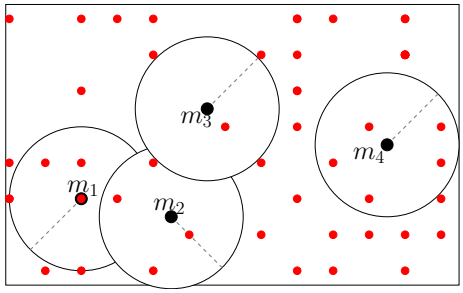


Figure 5.2: Four medoids with fixed-diameter partitions.

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

$$E[\textit{retrieved medoids}] = P[X \leq \theta + \theta_C] \times M \tag{5.3}$$

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

$$E[\textit{candidate rankings}] = P[X \leq \theta + \theta_C] \times n \tag{5.4}$$

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 [BYCMW94].

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), the law states that the frequency of the item at rank i is given

by:

$$f(i; s, v) = \frac{1}{i^s H_{v,s}} \tag{5.5}$$

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:

$$E[\textit{index list length}] = \sum_i M \times f(i; s, v')^2 \tag{5.6}$$

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:

$$P[\neg \textit{selected}] = \frac{v-1}{v} \times \frac{v-2}{v-1} \ldots \frac{v-k}{v-k+1} = \Pi_{i=0}^{k} \frac{v-i}{v-i+1} = 1 - (\frac{k}{v}) \tag{5.7}$$

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

$$E[v'] = v \times \left(1 - \left(1 - \frac{k}{v}\right)^M\right) \tag{5.8}$$

To compute the overall cost, the above estimates are combined as shown in Table 5.3. 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,

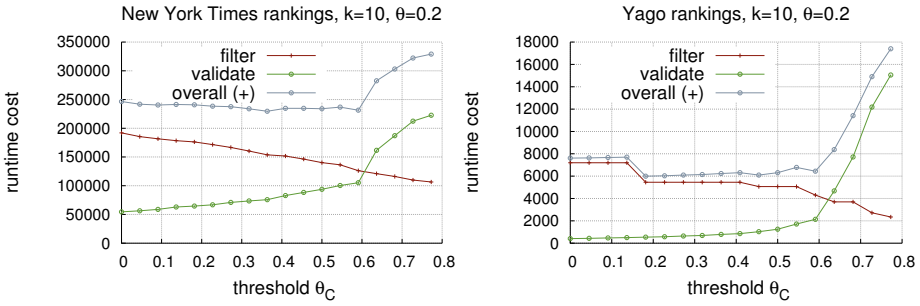


Figure 5.3: The behavior of the theoretically derived performance for varying θ_C.

$Cost_{merge}(k, size)$. If a subset of the inverted index lists are dropped according to Section 5.4.1, the above reasoning remains the same, except that the parameter k is adapted according to Corollary 4.3.1.

Figure 5.3 shows the model for vary θ_C for the two datasets used in the experimental evaluation (we refer the reader to Section 5.5 for a description of the datasets). We empirically estimated the skewness parameter s from samples of the 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 5.3: Model of query performance (∼runtime) of the coarse index.

5.4 Inverted 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, we describe how the minimum-overlap criterion and the prefix sizes, derived in Chapter 4 can be applied here; it is used for calculating 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.

5.4.1 Pruning by Query-Ranking Overlap

This inverted index optimization is based on the prefix-filtering framework described in Chapter 3. By using either Corollary 4.3.1 or Lemma 4.3.2, at query time, based on the threshold θ, we compute the (number of) posting lists, p, that need to be retrieved from the inverted index. Note that, since we do not know the threshold θ upfront, we have to index the complete rankings. Furthermore, in our problem setting, due to the way that the prefix size is computed, no additional sorting of the rankings is required.

Independent of the actual choice of the prefix used (either Corollary 4.3.1 or Lemma 4.3.2) $k-p$ index lists 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. Therefore, in this work we use the prefix computation based on Corollary 4.3.1. 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 we can guarantee 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.

5.4.2 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. This possible extension of the inverted index with rank information was already mentioned in Chapter 2. Here we describe the exact query processing algorithm using such rank-augmented inverted index.

inverted index with 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. [FLN03], 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

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

Table 5.4: Sample set $\mathcal{T}$ of rankings

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

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

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 5.4 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.

5.4.3 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

item 1→	$(\tau_0 : 0), (\tau_1 : 0), (\tau_6 : 0)$	$(\tau_3 : 1), (\tau_4 : 1), (\tau_7 : 1), (\tau_{10} : 1)$	$(\tau_2 : 2), (\tau_5 : 2)$	$(\tau_9 : 3)$	$(\tau_8 : 4)$
item 2→	$(\tau_8 : 0)$	$(\tau_0 : 1), (\tau_1 : 1)$	$(\tau_6 : 2), (\tau_9 : 2)$	$(\tau_2 : 3), (\tau_4 : 3), (\tau_5 : 3), (\tau_{10} : 3)$	$(\tau_7 : 4)$
item 3→	$(\tau_9 : 1)$	$(\tau_0 : 2)$	$(\tau_6 : 3)$	$(\tau_1 : 4), (\tau_4 : 4), (\tau_5 : 4)$	
item 4→	$(\tau_5 : 0)$	$(\tau_{10} : 2)$	$(\tau_0 : 3), (\tau_3 : 3)$	$(\tau_2 : 4), (\tau_9 : 4)$	
item 5→	$(\tau_5 : 1), (\tau_8 : 1)$	$(\tau_4 : 2)$	$(\tau_7 : 3)$	$(\tau_0 : 4), (\tau_3 : 4), (\tau_{10} : 4)$	
item 6→	$(\tau_4 : 0), (\tau_9 : 0)$	$(\tau_6 : 1)$	$(\tau_7 : 2)$		
item 7→	$(\tau_3 : 0), (\tau_7 : 0)$	$(\tau_6 : 4)$			
item 8→	$(\tau_2 : 1)$	$(\tau_1 : 3)$	$(\tau_8 : 3)$		
item 9→	$(\tau_2 : 0), (\tau_{10} : 0)$	$(\tau_1 : 2), (\tau_3 : 2), (\tau_8 : 2)$			

Figure 5.4: Inverted Index for rankings in Table 5.4 with highlighted blocks of same-rank entries.

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 5.4, constructed according to the rankings in Table 5.4. 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 lists are processed which accounts for less than 50% index lists being accessed.

5.5 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 [IMS13]. 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 [NYT] 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** [WLF12] as the most recent and competitive work on ad-hoc set similarity search in main memory at that time. We implemented AdaptSearch by following the C++ implementation of the AdaptJoin algorithm available online[1]. We computed the size of the prefix of the query using the overlap threshold ω derived in Section 5.4. In the validation phase, AdaptSearch computes the Footrule distance for each of the candidate rankings.

The implementation of the M-tree is obtained from [MTI]. We implemented the BK-tree ourselves, according to the original work in [BK73]. The inverted index implementations make use of the Trove library[2].

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 [AIMS13] we observed that at ranker.com most common are rankings of size 10.

[1] https://github.com/sunlight07/similarityjoin
[2] http://trove.starlight-systems.com/

Performance Measures

- Wall-clock time: For all algorithms we measure the wall-clock 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.

5.5.1 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 metric index structures is compared, here, represented by the BK-tree and the M-tree [ZSAR98] (Figure 5.5). Second, the use of inverted indices is compared to the BK-tree (Figure 5.6).

Figure 5.5 reports the query performance of the BK-tree compared to the M-tree and Figure 5.6 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. [CNBM01] 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. [CNBM01] 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 is the most comparable to the model presented in Section 5.2. In Figure 5.7, 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 5.3. 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 5.2.

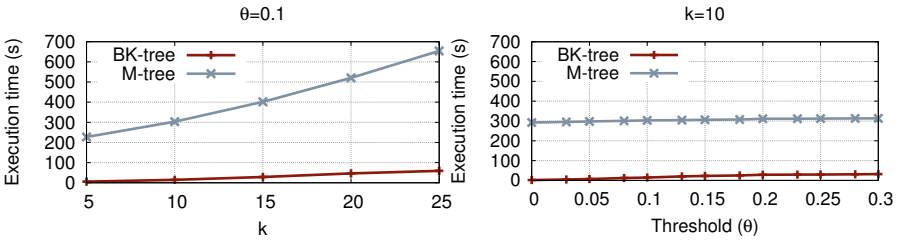


Figure 5.5: Performance of the M-tree vs. BK-tree for NYT dataset

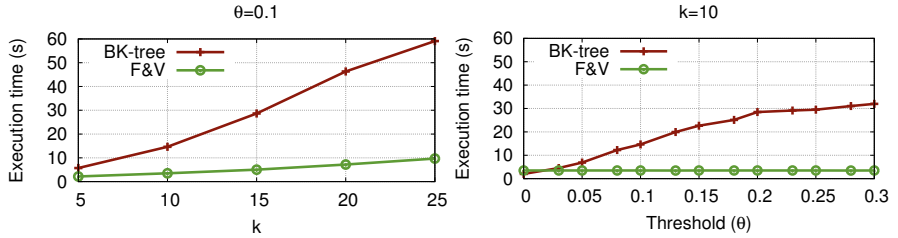
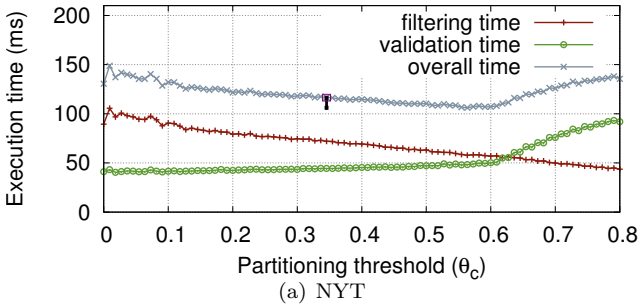
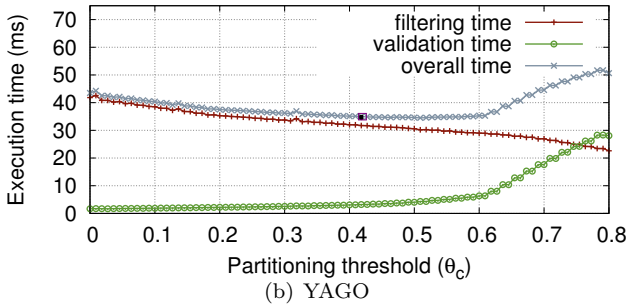


Figure 5.6: Performance of the BK-tree vs. the performance of inverted index for NYT dataset.



(a) NYT



(b) YAGO

Figure 5.7: 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.

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 5.7 as a small rectangle.

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

Table 5.5: Difference in ms between the minimal performance of the coarse index, and the performance for the theoretically computed best value of θ_c ($k = 10$)

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 is smaller than 11ms (Table 5.5). 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

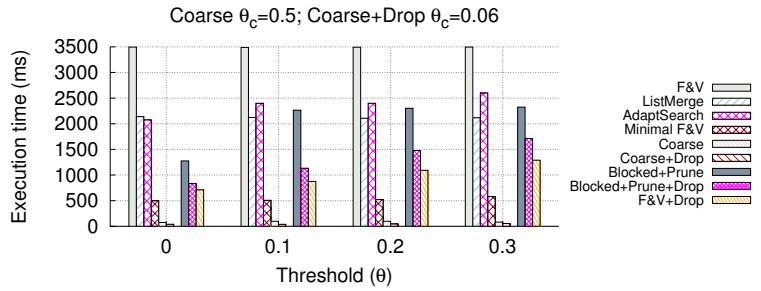
Next, we study the performance of different query processing methods performed over the two datasets; for rankings of size 10 and 20 and θ ranging from 0 to 0.3. First, in Figure 5.8 we compare the performance of the coarse index with the remaining techniques, for the NYT dataset. For a better visibility, 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 5.10. 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

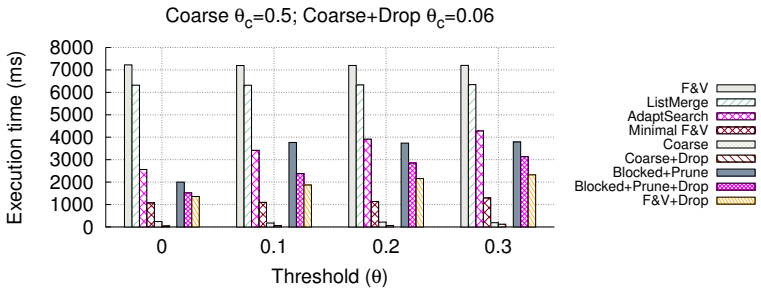
agnostic, perform worse than the rest of the algorithms. Increasing the values of θ degrades the performance of all the processing techniques except for the baseline F&V and ListMerge techniques, as they are threshold agnostic. In fact, because of its simple and efficient implementation, the ListMerge 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 scheduling 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 5.9). 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.



(a) $k = 10$



(b) $k = 20$

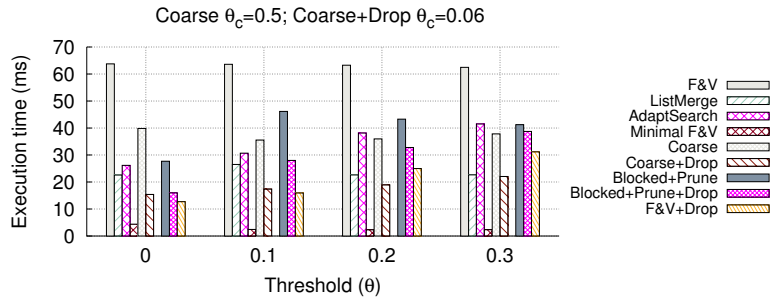
Figure 5.8: Comparing query processing over coarse index with baseline and competitor approaches (left block) and with other hybrid methods over inverted index (right block) for NYT dataset, for $k = 10$ and $k = 20$.

Distance Function Computations

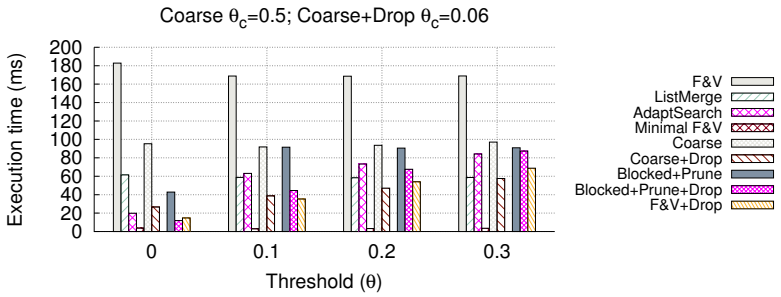
The difference in performance between the Coarse, Coarse+ Drop, F&V+Drop and Blocked+Prune+Drop algorithms can be explained by looking at the number of distance functions calls, shown in Figures 5.10 and 5.11 . We see that for the Yago dataset the final result set is very small, practically almost 1, 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.

5.5.2 Index Size and Construction Time

In Table 5.6 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



(a) $k = 10$



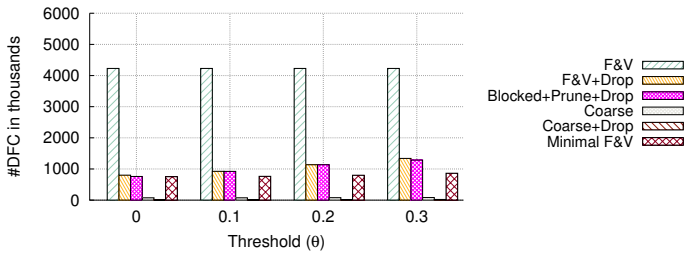
(b) $k = 20$

Figure 5.9: Comparing query processing over coarse index with baseline and competitor approaches (left block) and with other hybrid methods over inverted index (right block) for Yago dataset, for $k = 10$ and $k = 20$.

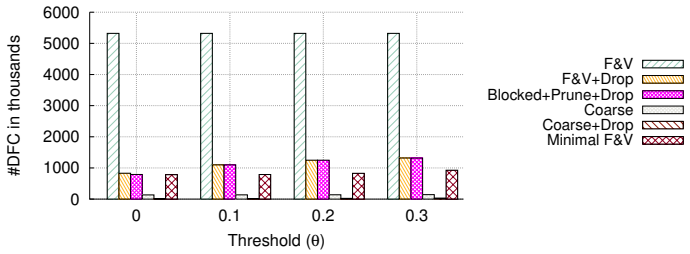
than 1GB. All indices store the complete rankings, thus their sizes do not differ significantly. The rank-augmented inverted index requires the most storage as it keeps both the complete rankings, and the position augmented index lists to support different processing techniques.

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.

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.

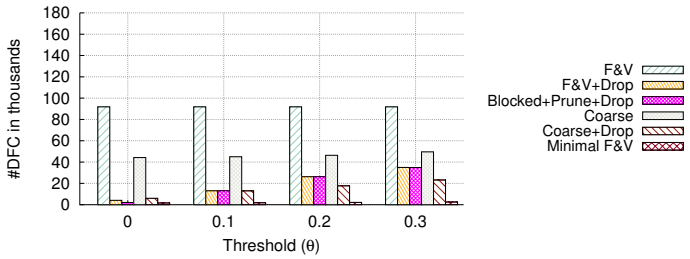


(a) $k = 10$

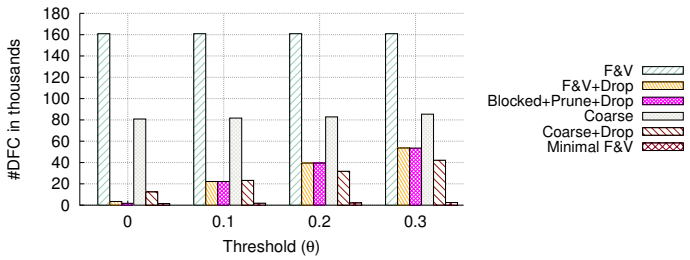


(b) $k = 20$

Figure 5.10: Number of distance function calls (DFC) for different query processing methods for NYT dataset (Coarse θ_c=0.5; Coarse+Drop $\theta_c = 0.06$)



(a) $k = 10$



(b) $k = 20$

Figure 5.11: Number of distance function calls (DFC) for different query processing methods for Yago dataset (Coarse θ_c=0.5; Coarse+Drop $\theta_c = 0.06$)

Lessons Learned

Combining the coarse index with the proposed optimizations on the inverted index always leads to performance improvements, independent of the distribution

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

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 5.2) 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.

5.6 Summary

In this chapter, 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.

Chapter 6

Distributed Similarity Joins over Top-k Rankings

6.1 Introduction

As a natural extension to the problem of processing similarity range queries over top-k rankings, in this chapter, we present algorithms for efficiently solving the problem of similarity joins over top-k rankings. The work presented in this chapter has not been published yet, however, is under submission. Similarity joins have been a popular research topic in the database community for more than a decade. Previous research in this topic is concerned with solving the problem of similarity join for sets [GIJ+01, CGK06, XWLY08, WLF12], strings [JLFL14] or the more general problem of finding the similar objects in metric space [JS08]. However, to the best of our knowledge, the problem of similarity joins over top-k rankings has not been addressed so far.

Since presently huge amounts of data are being generated and processed, in this chapter, we specifically focus on solving the problem of distributed similarity joins for top-k rankings. Recently, Fier et al. [FAB+18] summarized and compared existing distributed solutions on similarity joins for sets, strings and metric space. In their study, they showed that the existing distributed solutions in Map Reduce do not scale well, and propose that Apache Spark [SPK] is used as a platform for developing new alternative solutions. Thus, in this thesis, we focus on studying an efficient and scalable top-k rankings similarity joins using Apache Spark [SPK].

The similarity join algorithms for top-k rankings can be applied in some of the same contexts as the similarity range approach. For instance, in the case of query suggestion or expansion in search engines based on finding similar queries by comparing their result lists, or in a dating portal where we can use the preferences and affinities of users, presented in a form of top-k lists, for matchmaking, where we compare *all* the lists, in order to find people whose interests

match. For instance, consider Table 6.1 containing favorite movies of members of some dating portal. By comparing the lists we see that Alice and Chris have similar taste so the system should match them for a date. Furthermore, these algorithms can be applied in the case of recommender systems, where the similarity between the top sold (liked, favored) items for different clients can help in recommending products.

	Alice	**Bob**	**Chris**
1.	Pulp Fiction	The Schindler List	Indiana Jones
2.	E. T.	Lord of the Rings	Pulp Fiction
3.	Forrest Gump	Avengers	Forrest Gump
4.	Indiana Jones	Indiana Jones	E. T.
5.	Titanic	E. T	Titanic

Table 6.1: Example top-k lists of favorite movies for users of a dating portal

As in the prior chapter, Spearman's Footrule distance is used as a distance measure for comparing two top-k lists, and thus, the presented solution, similarly as in Chapter 5, combines an inverted index approach with metric, distance-based, filtering. Furthermore, the proposed approach is implemented in Apache Spark, and thus, is better tailored to the properties of this platform. In contrast to MapReduce, where each stage is composed of only a *map* and *reduce* function, and the data from each stage is written to disk, Apache Spark is more suitable for iterative processing of data and performs the computation in memory. Thus, we propose an iterative approach that computes the similarity join in several stages while storing the intermediate results in memory. By using the triangle inequality, the number of candidate pairs generated is reduced—very similar rankings are clustered together, and then, only the cluster representatives are joined, reducing the size of the data joined. Through a detailed experimental study, we show that our algorithm outperforms the competitor, for larger values of the distance threshold, θ.

6.1.1 Problem Statement and Setup

Similarly as in Chapter 5 as **input** we are provided with a set $\mathcal{T}$ of fixed-length rankings of size k, τ_i, where each ranking has a domain D_{τ_i} of items it contains. Furthermore, the ranked items in a ranking are represented as arrays or lists of items, where the left-most position denotes the top ranked item. In addition, each ranking has an id associated with it. We follow the same notation as in Chapter 5.

The problem that we want to solve in this chapter is: *Given a dataset of top-k rankings $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ and a distance threshold θ we want to efficiently find all pairs (τ_i, τ_j), $\tau_i, \tau_j \in \mathcal{T}$, $i \neq j$, where the distance d between τ_i and τ_j is smaller or equal to θ, i.e., $d(\tau_i, \tau_j) \leq \theta$.*

In this work, as in Chapter 5, for comparing the rankings Spearman's

Footrule distance is used. The main difference to our work on similarity range queries, presented in Chapter 5, is first, that the distance threshold is known beforehand, and second, that for this problem, we do not have a preprocessing step where the data is indexed before being queried, but instead the whole dataset is being processed at join time. Furthermore, our focus is on developing a distributed solution, which introduces additional challenges, like handling the data distribution.

6.1.2 Contributions and Outline

The contributions of our work can be summarized as follows.

- We adapt existing set-based similarity join algorithms to the problem of top-k rankings. We furthermore implement and adapt these algorithms to the Apache Spark framework.
- We introduce a new iterative, highly configurable, algorithm that combines metric space distance-based filtering with state-of-the-art set-based similarity join algorithms.
- We propose further optimization to the proposed algorithm by presenting a method for repartitioning large partitions.
- We implemented our methods and competitors in Apache Spark and through an extensive experimental study on two real-world datasets, we show that our methods consistently outperform state-of-the-art approaches for larger values of the threshold θ.

The rest of the chapter is structured as follows. In Section 6.2 we describe how set-based algorithms can be adapted for processing top-k rankings using Spearman's Footrule distance. The clustering algorithm and its components are introduced in Section 6.3. Section 6.4 proposes a Spark based repartitioning technique. We experimentally evaluate the presented approaches in Section 6.5. Finally, we summarize the chapter in Section 6.6.

6.2 Adaptation of Set-Based Algorithms to Top-k Rankings

6.2.1 Vernica Join

To find all pairs of similar top-k rankings for a given set $\mathcal{T}$ and a threshold θ we can use the Vernica Join (VJ) algorithm, described in Chapter 3. This algorithm is based on the prefix-filtering method, therefore, we can use the prefix sizes, defined in Chapter 4 in order to be able to apply it on top-k rankings.

The first step in the VJ algorithm is counting the frequency of the elements in the sets and ordering them by frequency. This step is not needed for top-k

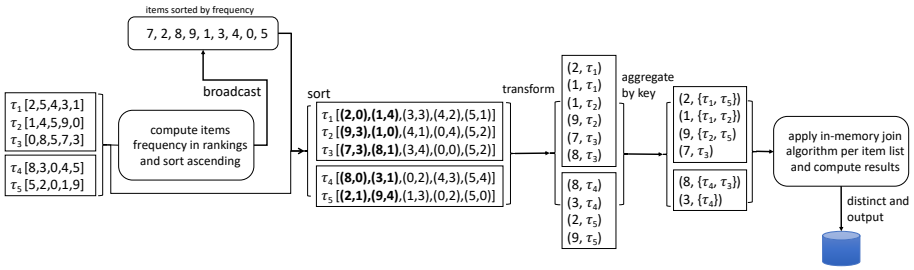


Figure 6.1: Example of computing the similarity join for top-k rankings using the Vernica Join algorithm in Spark.

rankings and it can be skipped. However, since most real-world datasets follow a skewed distribution, through experiments we concluded that reordering the rankings by the item's frequency leads to major performance gains, and thus, we keep this step for top-k rankings as well. This entails that only the prefix size based on the overlap between the rankings, defined in Corollary 4.3.1, can be applied. To perform the reordering, we first count the frequency of the items in the rankings. Then, in order to make this collection available to all the nodes, in Spark, we use a broadcast variable which is cached on each machine and then used to sort the items of all rankings $\tau \in \mathcal{T}$ by increasing order of their frequency. Note that, while we reorder the items in the rankings, we still need to keep track of their original rank for the computation of the Footrule distance. In the next step, we transform the rankings into (i_{id}, τ) pairs, where as key we have the item id, and as value we have the ranking. This we only do for the items that belong to the prefix of the ranking. Then, in the next step, in order to bring all rankings that share an item to the same partition, we aggregate the tuples (RDD) created in the previous step by key. In the next step, for the rankings that share an item, a main memory approach for finding the similar pairs is applied. For the rankings on each item list, we again sort the items by frequency, and index their prefixes using an inverted index. In addition, we apply a position filter, based on Lemma 4.4.1 in order to filter out more candidate pairs. In Chapter 4 we proved that two rankings τ_i and τ_j cannot have distance smaller than θ if at least one of the items in the rankings have a difference in their ranks larger than $\frac{k*(k+1)*\theta}{2}$, i.e., if there is at least one item $i \in \tau_i, \tau_j$, such that, $|\tau_i(i) - \tau_j(i)| > \frac{k*(k+1)*\theta}{2}$, we can be sure that $d(\tau_i - \tau_j) > \theta$. For the candidate pairs that pass the filters, we compute the Footrule distance. Before writing the final result, we remove the duplicate pairs. Figure 6.1 illustrates the algorithm through an example.

Improved Memory Usage

Previous similarity join distributed approaches were designed and implemented in MapReduce. As discussed in Chapter 2, Spark as a successor of MapReduce, has different characteristics than MapReduce, and thus existing approaches need

to be adapted to the computational properties of Spark in order to improve their performance. Large datasets in Spark are represented as RDDs, which are immutable, distributed collections of objects, stored in the memory of the executors. This means that for every transformation of an RDD, a new RDD is created. In addition to this, Spark runs in the JVM, and thus garbage collection can easily cause performance issues for Spark jobs. Thus, keeping objects in the memory of Spark's executors is not recommended, since this can lead to crashes or performance overhead for large datasets. Instead, working with iterators is more native to the Sparks computational model, since this allows the framework to spill some data to disk, when needed.

The VJ algorithm that shows the best performance, according to [VCL10], works such that rankings that share the same item are distributed to different partitions. Next, on each partition, an in memory join algorithm is executed, to compute the rankings with distance smaller than θ. This entails, first, storing a dictionary of the items, second, storing an inverted index for the rankings for this partition, and storing the partial result sets until the final computation is done. In addition to this, since Spark works with immutable objects, sorting the objects for performing the per partition in memory join, imposes creating new objects for each ranking. This means that VJ algorithm can lead to having both of the issues that we mentioned above, bad performance caused by the garbage collector overhead and memory overhead crashes, due to keeping data structures and objects in memory.

Instead, we claim that a nested loop modification to the VJ algorithm, is more native to Spark's processing style. Instead of indexing the rankings per partition, we propose using iterators to loop through the rankings in a nested loop fashion. For each ordered pair of rankings, (τ_i, τ_j), where $\tau_i < \tau_j$ that passes the position filter defined above, we compute the Footrule distance, and output those pairs where $d(\tau_i, \tau_j) \leq \theta$. This approach, as we will show in our experiments, performs better for large datasets, since allows Spark to spill the data to disk when needed.

6.3 Approach

Driven by the idea that similar rankings should have similar results sets, we propose an approach where we introduce a pre-processing step, where very similar rankings are grouped together, forming clusters. Then, only one representative ranking from the clusters, called centroid, is considered in the next similarity join phase. The idea is that in this way we would reduce the number of records being joined, and thus, reduce the execution time of the main joining phase, which is actually the most expensive part of the similarity join algorithms. Since Spark is suitable for iterative processing, adding an additional phase should not be a performance issue. Note that in contrast to the work presented in Chapter 5, the execution time of this step is included in the total execution time. Thus, this pre-processing phase should be very efficient, so that we do not end up

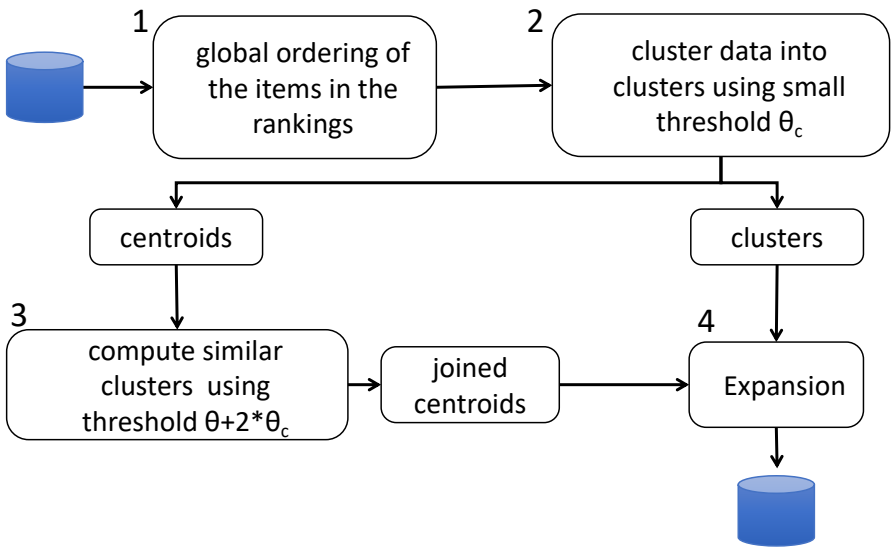


Figure 6.2: Overall architecture. The algorithm has four main phases: ordering, clustering, joining and expansion.

with having a higher overhead than benefit. Another key observation is that the triangle inequality can be used for forming and expanding the clusters after finding the similar centroids, to compute the final result set more efficiently.

Making use of the above observations, we propose our approach consisting of four main phases: Ordering, Clustering, Joining and Expansion, depicted in Figure 6.2.

Ordering: The first phase of our approach is ordering the items in the rankings by their occurrence, i.e., items that occur less frequently in the rankings, are moved to the top positions of the rankings. In our proposed algorithm, as later described, the VJ similarity join algorithm is applied twice, once for clustering the rankings, and once for finding the similar clusters. Instead of reordering the rankings twice, we choose to do this only once, using the original dataset $\mathcal{T}$. As our approach does not depend on the similarity join algorithm used for clustering or for joining the clusters, the re-ordering of the items in the rankings can be skipped if it is of no use to the joining algorithm applied later on. The reordering is done just for determining which items will be included into the prefix of the rankings, while the rankings still preserve their original item ordering for the computation of the distance. The first steps of Figure 6.1 show an example of the ordering phase.

Clustering: The second phase of our approach is forming clusters, such that similar rankings will belong to the same cluster, $\mathcal{C}_i$. First, a similarity join algorithm is executed to find the similar rankings that need to be grouped together. Then, clusters are formed such that the pairwise distance between each member of the cluster and its representative is at most θ_c. In this work, we will

refer to θ_c as the clustering threshold. Note that in contrast to other similarity join algorithms in metric space, where formed clusters have different radius, the radius of all clusters formed by our approach is bounded by the clustering threshold, θ_c. Similarly as in Chapter 5, we write $\tau_i \prec c_i$ to denote that ranking τ_i belongs to the cluster, $\mathcal{C}_i$, represented by ranking (centroid) c_i. Rankings in the dataset $\mathcal{T}$ that do not have any similar rankings with distance smaller than the clustering threshold, θ_c, form singleton clusters, i.e., one element clusters.

Joining: In this phase a similarity join algorithm is executed over the centroids with a threshold $\theta_o = \theta + 2 * \theta_c$. Using threshold θ_o instead of θ is necessary in order to insure the correctness of the algorithm. Note that any similarity join algorithm can be applied here, however, similarly to Fier et al. [FAB+18], our experiments showed that VJ is the most efficient one, and thus we use this algorithm.

Expansion: In the last step of the algorithm, the final result set is computed by joining the results from the joining phase with the formed clusters in the clustering phase. The members of the joined clusters from the joining phase are checked against each other if the distance between them is smaller than θ. Using the metric properties of the distance measure, we are able to directly filter out some candidates, and thus compute the final result list more efficiently.

Before we describe each phase more formally, how each phase is realized and how the final join result is computed, we first discuss the correctness of the proposed algorithm.

Lemma 6.3.1 *For given join threshold θ and clustering threshold θ_c, in the joining phase, all pairs of centroids c_i, c_j with distance $d(c_i, c_j) \leq \theta + 2 * \theta_c$ need to be retrieved in order not to miss a potential join result.*

Lemma 6.3.1 ensures that pairs $\{(\tau_i, \tau_j) | \tau_i \prec c_i, \tau_j \prec c_j \wedge d(\tau_i, \tau_j) \leq \theta \wedge d(c_i, c_j) > \theta\}$ will not be omitted from the result set.

In other words, Lemma 6.3.1 avoids missing result rankings with distance $\leq \theta$, which are represented by centroids which are with distance larger than θ from each other.

This follows from the fact that for all rankings $\{\tau_i | \tau_i \prec c_i \wedge d(\tau_i, c_i) \leq \theta_c\}$. It follows that for any pair of rankings $\{\tau_i, \tau_j | \tau_i \prec c_i, \tau_j \prec c_j\}$ the distance of the corresponding centroids $d(c_i, c_j)$ must be $\leq \theta + 2 * \theta_c$. Thus, using a threshold $\theta_o = \theta + 2 * \theta_c$ in the joining phase is enough to ensure that no true result will be missed.

6.3.1 Clustering

When forming the clusters, the following points need to be considered:

- To ensure correctness, the radius of all the clusters should be the same, i.e., for any ranking $\tau_i \in \mathcal{C}_i$, represented by a top-k ranking c_i, $d(\tau_i, c_i) \leq \theta_c$.

- The clustering method should be very efficient, otherwise the cost of the clustering would overweight its benefit.

- The performance of the expansion phase depends on the clusters formed.

We address each point individually while explaining our design choices for the clustering algorithm.

For forming the clusters, we could turn to existing methods [WMP13, SHC14], where, first, the centroids of the clusters are randomly chosen, and then, by computing the distance from the centroids to the other points in the dataset, the members of the clusters are found. However, considering that we aim at forming equal range clusters, where the points are very close to each other, this approach has two main drawbacks, which make it not suitable for our use case. First, due to the very small clustering threshold, and the random choice of the clusters, it could happen that for some, or in the worst case, for all of the chosen centroids, there are no other points in the dataset such that their distance to the centroids is smaller than the clustering threshold, θ_c. This leads to having singleton clusters which do not cause any performance benefit in the joining phase. Another drawback of this approach is that the number of clusters needs to be chosen upfront.

Figure 6.3 shows through an example the creation of the clusters. First, to find the rankings that are very similar to each other, instead of selecting the centroids first and comparing the distance for each point to the centroids, we execute a similarity join algorithm with the clustering threshold over the whole dataset $\mathcal{T}$. Here any similarity join algorithm can be applied, however, since prefix filtering approaches are especially efficient for very small thresholds, in our implementation we use the VJ algorithm. Note that the rankings have already been sorted, so we do not perform any additional sorting in this phase. The result of the VJ algorithm are all pairs of rankings (τ_i, τ_j) whose distance is smaller than the clustering threshold, i.e., $d(\tau_i, \tau_j) \leq \theta_c$. The clusters are formed such that, from the pairs, we take the first ranking as the cluster centroid, and the second one as their member. This does not only keep the clustering phase efficient, but also simplifies the expansion of the results in the last phase, since then the expansion can simply be performed by joining the result set from the joining and clustering phase. Furthermore, this way we can also efficiently apply filters based on the distance of the elements to their centroids, explained in Section 6.3.3. Clusters formed this way theoretically correspond to clusters formed by grouping the results by the first ranking, and taking the first ranking as the centroid. For instance, in Figure 6.3, the following clusters would be formed $\mathcal{C}_1 = \{\tau_1, \tau_2, \tau_5\}$, $\mathcal{C}_2 = \{\tau_3, \tau_4\}$ with centroids τ_1 and τ_3, respectively.

Since Spearman's Footrule distance is a metric, we know that for any two rankings $\tau_i, \tau_j \in C_i$ it holds that $d(\tau_i, \tau_j) \leq 2 \cdot \theta_c$, and thus, members of the same clusters can directly be written to disk as partial results, as long as $\theta_c \cdot 2 < \theta$.

By creating the clusters in this way, all of the afore-listed requirements are satisfied. The radius of all the clusters is the same, i.e., θ_c, and both forming

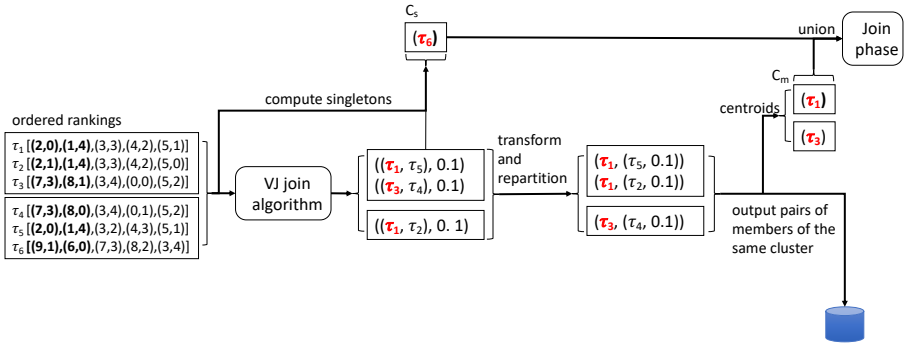


Figure 6.3: Example of how clusters are formed and centroids (marked with red) are chosen, where $\theta_c = 0.1$.

the clusters and expanding the result set is kept simple, and thus, very efficient. One minor drawback of this approach is that the formed clusters would be overlapping, however, resolving this would negatively impact the performance of both the clustering and the expansion phase.

As input to the next, joining phase, we union the set of centroids $\mathcal{C}_m$ that contains all centroids representatives of clusters with at least two members, i.e., $|C| \geq 2$ with the set of centroids $\mathcal{C}_s$ that represent the singleton clusters i.e., $|C| = 1$. The set $\mathcal{C}_s$ is derived from the original dataset, by finding those rankings $\tau_i \in \mathcal{T}$ where there is no other ranking $\tau_j \in \mathcal{T}$, such that, $d(\tau_i, \tau_i) \leq \theta_c$. An example of a singleton cluster in Figure 6.3 is $\mathcal{C}_3 = \{\tau_6\}$.

6.3.2 Joining

In the joining phase we need to find all centroids pairs (c_i, c_j) such that $d(c_i, c_j) \leq \theta_o$. To do this, we execute the VJ algorithm over all centroids c_i, with a threshold $\theta_o = \theta + 2 * \theta_c$. However, the VJ algorithm is sensitive to the threshold value—for larger threshold values the algorithm performs worse. Thus, it could happen that, even though we are joining a dataset $\mathcal{C} \subseteq \mathcal{T}$, due to the larger threshold used, the joining phase performs worse than simply executing the VJ algorithm over the whole dataset $\mathcal{T}$. Again, note that we do not perform additional reordering of the rankings here, but the VJ algorithm is executed on the initially ordered rankings.

According to Lemma 6.3.1, using a threshold θ_o is only needed to avoid missing pairs of rankings $\{(\tau_i, \tau_j) | \tau_i \prec c_i, \tau_j \prec c_j \wedge d(\tau_i, \tau_j) \leq \theta \wedge d(c_i, c_j) > \theta\}$. Furthermore, due to the small clustering threshold, in the dataset $\mathcal{C}$ we have many centroids which are representatives of singleton clusters. For these centroids, we can actually use a smaller threshold, without missing any true result. Lemma 6.3.2 defines this:

Lemma 6.3.2 *Given a join threshold θ, a clustering threshold θ_c, and a set of centroids $\mathcal{C} = \mathcal{C}_m \cup \mathcal{C}_s$, where $\mathcal{C}_s$ is the set of centroids that represent the*

method: **Centroids Join**
input: Dataset $\mathcal{C} = \mathcal{C}_m \cup \mathcal{C}_s$, double θ, θ_c
output: all pairs (c_i, c_j) s.t. $d(c_i, c_j) \leq \theta + 2 * \theta_c$
1 p_m = get_prefix($\theta + 2 * \theta_c$,k)
2 p_s = get_prefix(θ,k)
3 grouped $\leftarrow$ transform_and_emit($\mathcal{C}_m, \mathcal{C}_s, p_m, p_s$)
4 $\mathcal{R} \leftarrow$ compute_sim(grouped,k,θ,θ_c)
return $\mathcal{R}$

Algorithm 2: Joining of centroids based on the type of the centroid.

singleton clusters and $\mathcal{C}_m = \mathcal{C} \setminus \mathcal{C}_s$ is the set of centroids representing non-singleton clusters. The following pairs of centroids need to be retrieved in order not to miss a potential join result:

$$(c_i, c_j) | d(c_i, c_j) \leq \theta + 2 * \theta_c \;\; \textit{if} \;\; c_i, c_j \in \mathcal{C}_m \tag{6.1}$$

$$(c_i, c_j) | d(c_i, c_j) \leq \theta + \theta_c \;\; \textit{if} \;\; c_i \in \mathcal{C}_m \wedge c_j \in \mathcal{C}_s \;\; \textit{or vv} \tag{6.2}$$

$$(c_i, c_j) | d(c_i, c_j) \leq \theta \;\; \textit{if} \;\; c_i, c_j \in \mathcal{C}_s \tag{6.3}$$

Lemma 6.3.2 allows us to more efficiently join the centroids. It follows that, only for the centroids $c_m \in \mathcal{C}_m$ we need to use θ_o for joining and thus, only for these centroids, we need to use a prefix based on the threshold θ_o. For the centroids $c_s \in \mathcal{C}_s$ we can actually use the prefix based on the original threshold θ. Then, when computing the distance between the candidate pairs, we keep track of the type of the centroid, and accordingly, we output the pair if it satisfies the corresponding threshold. This is outlined in Algorithm 2.

Since we propose using small values for the clustering threshold θ_c, we expect that in practice, the cardinality of $\mathcal{C}_m$ will be significantly smaller than $|\mathcal{C}|$, and thus, by applying a threshold of θ_o only for centroids $c_m \in \mathcal{C}_m$, the savings should be notable.

6.3.3 Expansion

In the last phase, the final result set is generated. For this purpose, the results from the clustering phases, $\mathcal{R}_c$, and the result from the joining phase $\mathcal{R}_j$, need to be joined together, and the generated pairs need to be verified. Depending on the joined pair from the joining phase, the expansion is done differently. The pairs where both centroids are singletons do not need to be expanded and are directly written to disc. Pairs where at least one of the rankings is not a singleton, need to be joined with the set of clusters, so that similar pairs of rankings between cluster members from different clusters, or with other singleton centroids, are generated.

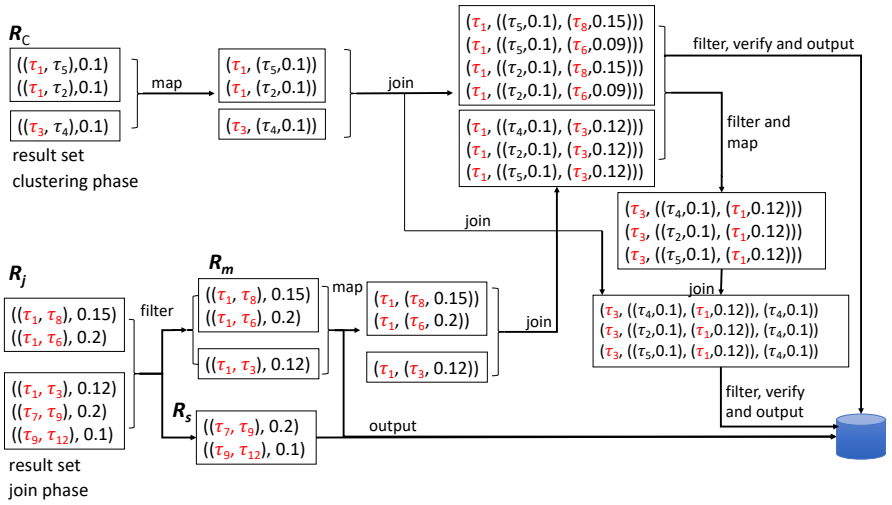


Figure 6.4: Example of computing the final result set using the result set from the joining phase and the clusters where $\theta_c = 0.1$ and $\theta = 0.2$. Cluster's centroids are marked with red.

Algorithm 3 outlines how the final result set is computed. First, the result set from the join phase, $\mathcal{R}_j$, is divided into two sets: $\mathcal{R}_s = \{(c_i, c_j)|c_i, c_j \in \mathcal{C}_s \wedge d(c_i, c_j) \leq \theta\}$ and $\mathcal{R}_m = \mathcal{R}_j \setminus \mathcal{R}_s$. $\mathcal{R}_s$ is the set of candidate pairs, where both centroids are singletons. These pairs can be directly written to disc without further processing and verification. In addition, a subset of $\mathcal{R}_m$, i.e., pairs $(c_i, c_j)|\theta_c < d(c_i, c_j) \leq \theta$, can already be included to the final results set.

Candidate pairs, where at least one centroid is not a singleton, $\mathcal{R}_m$, need to be further joined with the set of clusters $\mathcal{R}_c$, in order to find the result pairs where at least one ranking is a cluster member. These pairs are missing from $\mathcal{R}_j$, since in the joining phase the join was performed only over the centroids. To do this, first the set of clusters and the set $\mathcal{R}_j$ are transformed, so that they are brought into a format where as key we have the centroids. Next, $\mathcal{R}_m$ and $\mathcal{R}_c$ are joined into $\mathcal{R}_{R_c \bowtie R_m}$. $\mathcal{R}_{R_c \bowtie R_m}$ is used to further generate the following result pairs:

$$\mathcal{R}_{m,c} = \{(\tau_i, c_j)|(\tau_i, c_j) \leq \theta \wedge \tau_i \prec c_i \wedge (c_i, c_j) \in \mathcal{R}_j\} \tag{6.4}$$

$$\mathcal{R}_{m,m} = \{(\tau_i, \tau_j)|(\tau_i, \tau_j) \leq \theta \wedge \tau_i \prec c_i \wedge \tau_j \prec c_j \wedge (c_i, c_j) \in \mathcal{R}_j\} \tag{6.5}$$

To generate the first result set $\mathcal{R}_{m,c}$, the candidate tuples in $\mathcal{R}_{R_c \bowtie R_m}$ need to be transformed into the needed pairs and further verified if their distance is in fact smaller then θ. For pairs (τ_i, c_j), where $\tau_i \prec c_i$, we already know $d(\tau_i, c_i)$ and $d(c_i, c_j)$. Thus, using the triangle inequality we verify only those candidate pairs (τ_i, c_j) such that $d(\tau_i, c_i) + d(c_i, c_j) >= \theta$ or $(d(c_i, c_j) - d(\tau_i, c_i) <= \theta$ and the rest we can filter out since we can be certain that their distance is larger than theta.

method: **expand**
input: Dataset $\mathcal{R}_c$, $\mathcal{R}_j$, double θ, θ_c
output: all pairs (τ_i, τ_j) s.t. $d(\tau_i, \tau_j) \leq \theta$
1 $\mathcal{R}_m, \mathcal{R}_s \leftarrow$ split($\mathcal{R}_m$)
2 $\mathcal{R}_p \leftarrow$ get_partial_results($\mathcal{R}_m$,θ,θ_c)
3 $\mathcal{R}_j, \mathcal{R}_m \leftarrow$ prepare_for_join($\mathcal{R}_j$,$\mathcal{R}_m$)
4 $\mathcal{R}_{R_j \bowtie R_m} \leftarrow$ join($\mathcal{C}_m, \mathcal{R}_j, \mathcal{R}_j$)
5 $\mathcal{R}_{m,c} \leftarrow$ get_partial_results($\mathcal{R}_{R_j \bowtie R_m}$,$\theta$,$\theta_c$)
6 $\mathcal{R}_{R_j \bowtie R_m} \leftarrow$ prepare_for_join($\mathcal{R}_{R_j \bowtie R_m}$)
7 $\mathcal{R}_{(R_j \bowtie R_m) \bowtie R_j} \leftarrow$ join($\mathcal{R}_{R_j \bowtie R_m}, \mathcal{C}$)
8 $\mathcal{R}_{m,c}, \mathcal{R}_{m,m} \leftarrow$ get_partial_results($\mathcal{R}_{(R_j \bowtie R_m) \bowtie R_j}, \theta, \theta_c$)
return distinct($\mathcal{R}_p \cup \mathcal{R}_s \cup \mathcal{R}_{m,c} \cup \mathcal{R}_{m,m}$)

Algorithm 3: Computation of the final result set.

For generating the set $\mathcal{R}_{m,m}$ the set $\mathcal{R}_{R_c \bowtie R_m}$ is first transformed, so that the second centroid is set as key of the tuples, and then it is joined with the set of clusters. The joined set is then used to add pairs to the set $\mathcal{R}_{m,c}$. These will be candidate pairs from the members of the newly joined centroids to the centroids we already had in $\mathcal{R}_{R_c \bowtie R_m}$. Filtering based on the triangle inequality is applied here as well. As last step, we generate all candidate pairs (τ_i, τ_j), such that $\tau_i \prec c_i$, $\tau_j \prec c_j$ and $d(c_i, c_j) \leq \theta + 2 * \theta_c$. For these the Footrule distance is computed, and the ones where $d(\tau_i, \tau_j) \leq \theta$ are written to disc. Before writing the results to disc, the duplicates are removed.

Figure 6.4 illustrates through an example how the join of the two result sets is performed.

6.4 Repartitioning using Joins

As with all distributed algorithms, the data distribution over the partitions greatly influences the performance of the algorithms. The VJ algorithm partitions the rankings based on the items that they contain—rankings that share an item end up at the same partition. This means that in the case of a skewed data distribution, which is often the case, for the items that appear very frequently in the rankings, we would have large partitions. This problem is partially solved by the prefix filtering framework, especially for smaller values of the threshold, θ, since the most frequent items would not be included. However, as we increase the value of the threshold, θ, so does the size of the prefix, and then we can have skewed distribution of the data among the partitions. This leads to having few partitions that dominate the overall execution time of the algorithm.

To solve this issue we propose an algorithm where large partitions are split into smaller sub-partitions. Then, the resulting pairs are computed for each small partition, and for each pair of sub-partitions. Algorithm 4 describes this

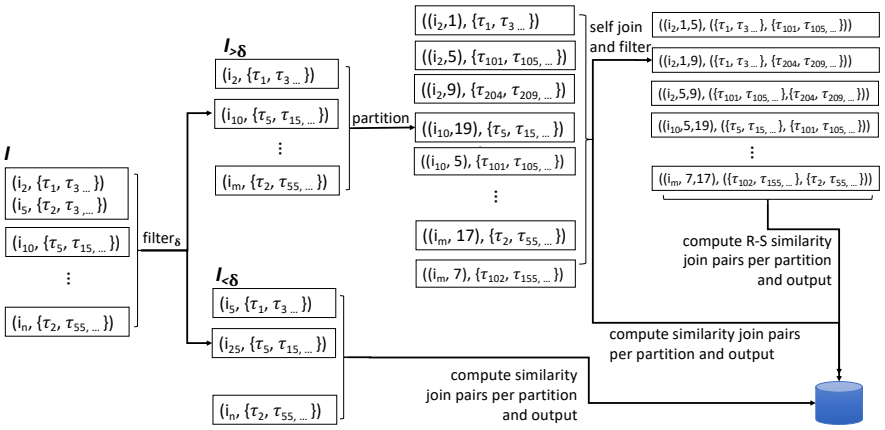


Figure 6.5: Example of repartitioning of the large partitions using a partitioning threshold δ.

procedure. First, using a user defined partitioning threshold δ we divide the inverted index into two parts, one where the partitions per item have more that δ rankings, $\mathcal{I}_{>\delta}$, and those whose partitions per item are smaller then the partitioning threshold, δ, $\mathcal{I}_{<\delta}$. This in Spark can be easily computed, since the distributed inverted index is kept in one RDD, which allows easy access to the sizes of each partition. For those partitions that are smaller then the partitioning threshold, we compute the all pair similarity join as before. The partitions larger than the partitioning threshold, $\mathcal{I}_{>\delta}$, are first split into smaller sub-partitions with at most δ rankings. This is done by assigning to each sub-partition a random number as a secondary key. To compute the final result set, we first compute the all pair similarity join over each sub-partition. Then, we self join the sub-partitions by the item id, and for those join results where the secondary key of the first join pair is smaller than the secondary key of the second join pair, we execute a R-S similarity join algorithm for the joined partitions. To redistribute the working load equal among nodes, we partition by both the primary and secondary key, i.e., by both the item id and the randomly assigned number. Figure 6.5 illustrates through an example the all pair similarity join computation in case of repartitioning.

In our experiments, we show that the performance of the algorithm does not significantly vary, when changing the partitioning threshold δ. However, the value of δ still needs to be chosen carefully, such that it is not set to a very small value, leading to too many partitions being split into many small sub-partitions. If this happens, then joining the sub partitions in step 5 of Algorithm 4 becomes too expensive, and the benefit of the repartitioning is lost. In addition, due to the use of Spark joins, choosing a very small value of δ can also lead to memory crashes of the executors.

method: **Repartitioning**
input: inverted index over $\mathcal{D}$, $\mathcal{I}$. partitioning threshold, δ
output: all pairs (τ_i, τ_j) s.t. $d(c_i, c_j) \leq \theta$
1 $\mathcal{I}_{>\delta}, \mathcal{I}_{<\delta}$ = split($\mathcal{I}$,δ)
2 $\mathcal{R}_{<\delta}$ = compute_sim($\mathcal{I}_{<\delta}$,θ,k)
3 $\mathcal{P} \leftarrow$ repartition($\mathcal{I}_{>\delta}, \delta$)
4 $\mathcal{R}_{p1} \leftarrow$ compute_sim($\mathcal{P}$,θ,k)
5 $\mathcal{R}_{p2} \leftarrow$ compute_sim(join($\mathcal{P}$,$\mathcal{P}$),θ,k)
return $\mathcal{R}_{<\delta} \cup \mathcal{R}_{p1} \cup \mathcal{R}_{p2}$

Algorithm 4: Computing the all pair similarity join with repartitioning of large partitions using a partitioning threshold δ.

6.5 Experiments

We implemented all algorithms in Spark using Scala 2.10. We deployed all algorithms on a Spark 1.6 (using YARN and HDFS) cluster running Ubuntu 14.04.5 LTS. The cluster consists of 8 nodes, each equipped with two Xeon E5-2603@ 1.6GHz/ 1.7GHz of 6 cores each, 128GB of RAM, out of which 40GB is reserved for execution of jobs by Yarn, and 4TB hard disks. All nodes are connected via a 10GBit Ethernet connection. Next, we report on the setup and the results of the experimental study.

Datasets

Due to the lack of real top-k ranking datasets, for the experiments we used datasets that are often used in previous work on similarity join of sets and were also used for performing the experimental study for distributed similarity join algorithms [FAB+18]. Specifically, we use the DBLP [DBL] and ORKU [ORK] datasets. To transform the records of these dataset into a top-k rankings, we simply take the top k tokens in the sets, and consider them as items in the rankings. Since we are working with rankings of same size, we remove records with size smaller than k. In addition the datasets are preprocessed as in [FAB+18]. Note that, while in the preprocessing step duplicates are removed from the dataset, since we cut the records to size k it can happen that we have a small amount of records with distance 0 to each other. However, this should *not* affect the performance of the algorithms, since duplicate records are *not* handled differently, i.e., the performance of the algorithms should be the same as if there were no duplicates. As we will show later on in our experimental study, what affects the performance of our algorithm is the number of records with distance smaller than θ_c.

After the preprocessing the DBLP dataset has approximately 1.2 million top-10 rankings, and ORKU has approximately 2 million top-10 rankings. Each datasets has a size of 67MB and 173MB. Since these datasets are relatively small

spark.driver.memory	12G
spark.executor.memory	8GB
spark.executor.instances	24
spark.executor.cores	5

Table 6.2: Spark parameters used for the evaluation

for a distributed setting, we also increase their size using the same method as in [VCL10, FAB+18], where the domain of the items remains the same, and the join result increases approximately linearly with the size of the dataset. We use xn to denote the number of times the dataset has been increased. For instance, "ORKUx5" represents the ORKU datasets increased 5 times.

The files in Spark are read as text files, and are directly partitioned into the number of partitions specified at input. Throughout the experiments we write the number of partitions that the data is divided into. Additionally, we show experiments that illustrate the behavior of the effect that the number of partitions has to the performance of the algorithms.

Algorithms under investigation

We investigate the performance of the following algorithms:

- The VJ adaptation for top-k rankings (VJ)
- The VJ adaptation for top-k rankings using iterators instead of inverted index (VJ-NL)
- The clustering algorithm using iterators (CL)
- The clustering algorithm with iterators and re-partitioning of the data (CL-P)

The experiments are run such that we restrict the memory utilization of Spark, and the number of cores used by an executor, based on the configuration of our cluster. Table 6.2 reports these parameters. In case we use different settings, we write these changes for the specific experiments. We report on the average wall-clock time measured in seconds over 3 runs. If an algorithm runs more than 10 hours we stop its execution.

6.5.1 Results

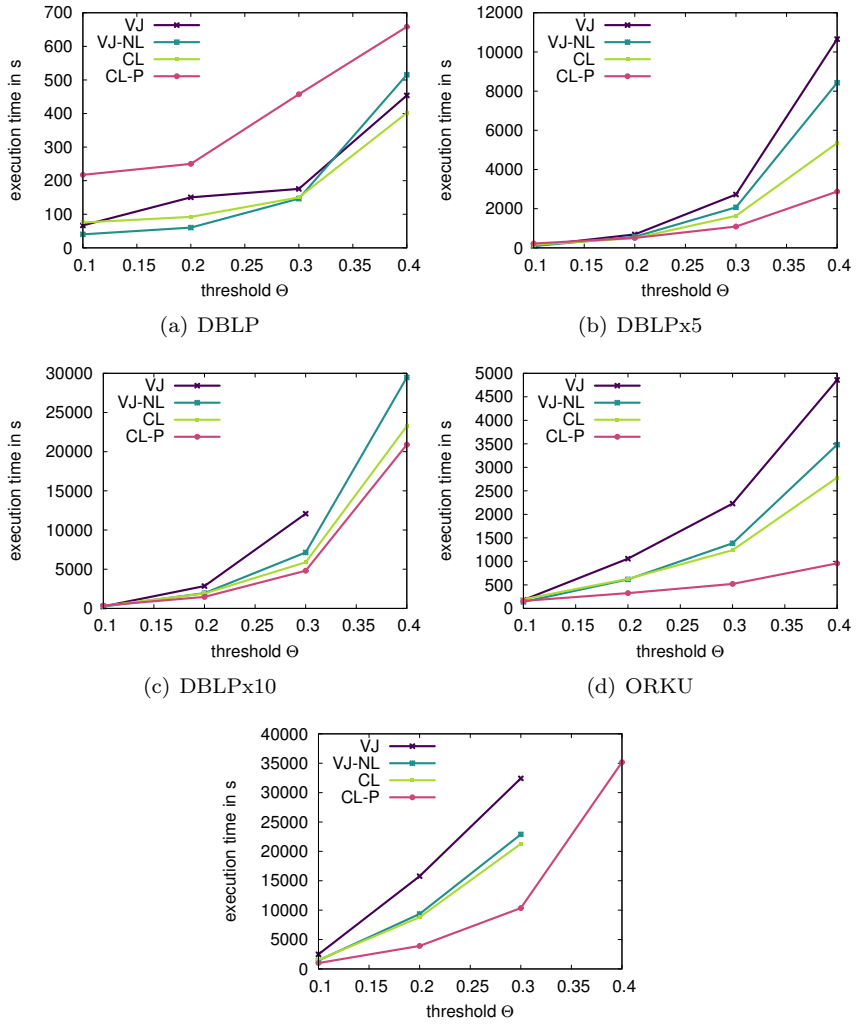
Performance Based on the Distance Threshold θ

We first evaluate and compare the performance of the above listed algorithms when we vary the distance threshold θ. Figure 6.6 reports on the performance of the four algorithms for both datasets DBLP and ORKU, for values of θ ranging from 0.1 to 0.4. We see that our algorithm outperforms the competitor algorithm

VJ for larger values of θ. Most importantly, we see that, with the exception of the DBLP dataset, each optimization that we propose brings additional performance improvement. For all algorithms, the execution time increases, as we increase the distance threshold θ, however, for our proposed algorithms, CL and CL-P, the increase in performance is smaller, especially for the latter. For instance, for the DBLPx5 dataset, the execution of the VJ algorithm for the largest threshold value, 0.4, is 100 times more expensive than when executing it for the smallest threshold value of 0.1. On the other hand, the increase in execution time for the CL and CL-P algorithms is 33 and 13 times, respectively. This can be attributed to the design of the CL algorithm. Since in the joining phase less rankings are being processed, the algorithm is not so sensitive to the dataset skewness. With the partitioning of the large partitions into smaller ones, and their redistribution among the nodes in the cluster, the CL-P algorithm shows even larger performance improvement for larger threshold values.

Furthermore, we see that for the datasets DBLPx5 (Figure 6.6(b)) and ORKU (Figure 6.6(d)) the gains in performance are the largest. Here, we can clearly see that using iterators over an inverted index is more efficient when it comes to a Spark implementation. Furthermore, we see that the largest performance benefit from our clustering algorithm are for values of θ of 0.3 and 0.4. When θ is set to 0.4, clustering combined with partitioning based on joins (CL-P) performs 5 and 3 times better than the VJ and VJ-NL algorithms, respectively, for the ORKU dataset (Figure 6.6(d)). For the DBLPx5 dataset, the CL-P algorithm outperforms the VJ and VJ-NL algorithms by almost 4 and 3 times, respectively (Figure 6.6(b)). For lower values of the partitioning threshold, i.e., when $\theta = 0.1$ or $\theta = 0.2$, the CL and CL-P algorithms either perform slightly worse than the VJ or VJ-NL, or the gain in performance is not that large. This is especially true for $\theta = 0.1$. This is due to the fact that the VJ algorithm is very efficient for a very small thresholds, since the prefix size is then small. In these cases, the overhead from the additional clustering phase in the CL approach, or partitioning for the CL-P, is larger than the benefit that we could get from it.

Note that, in all cases, the clustering threshold for the CL and CL-P algorithms is set to 0.03. The reason for this is explained bellow, where we study the effect that this threshold has on the performance of the algorithms. The value of the partitioning threshold δ differs depending on the dataset, and the threshold value, θ. For larger thresholds θ, we choose larger partitioning threshold δ, since we expect an increase in the size of the posting lists. Later we discuss how choosing the partitioning threshold δ affects the performance. For the smallest dataset, DBLP (Figure 6.6(a)), where the original VJ algorithm is already very efficient, the proposed optimizations lead to worse performance. The CL-P algorithm in this case always performs worse than VJ, since it brings additional overhead of repartitioning and joining already small posting lists. The CL algorithm outperforms VJ only for large values of θ. On the other hand, for the ORKUx5 dataset (Figure 6.6(e)), for $\theta = 0.4$, only the CL-P algorithm finished under 10 hours. Similarly, for the DBLPx10 dataset (Figure 6.6(c)), the VJ



(a) DBLP

(b) DBLPx5

(c) DBLPx10

(d) ORKU

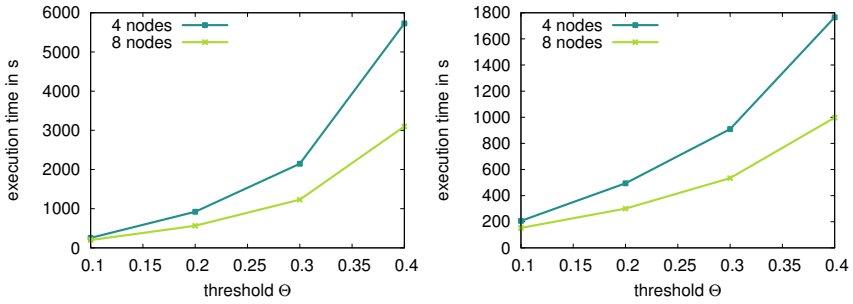
(e) ORKUx5

Figure 6.6: Comparison of different algorithms when varying θ

algorithm did not finish under 10 hours.

Scalability

To test the scalability of the proposed algorithm, we varied the number of nodes in our cluster. We executed the CL-P algorithm on a cluster with 4 nodes and with 8 nodes. For this experiment, we reduced the number of cores per executor to 3, and we did not fix the number of executors to be used, i.e., this was left to be decided by YARN, based on the cluster size. The memory restriction per executor and for the driver were kept as specified in Table 6.2. Figure 6.7 shows the performance of the CL-P algorithms for different values of the theshold θ, for the DBLPx5 and ORKU datasets. The values for the clustering threshold



(a) DBLPx5 (b) ORKU

Figure 6.7: Performance of CL-PL algorithm when varying the number of nodes in the cluster

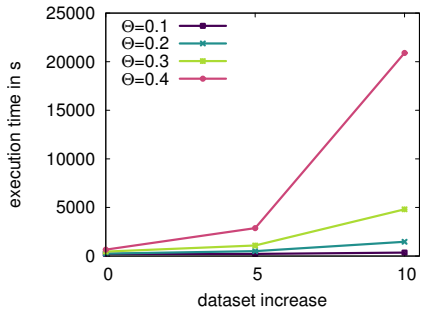
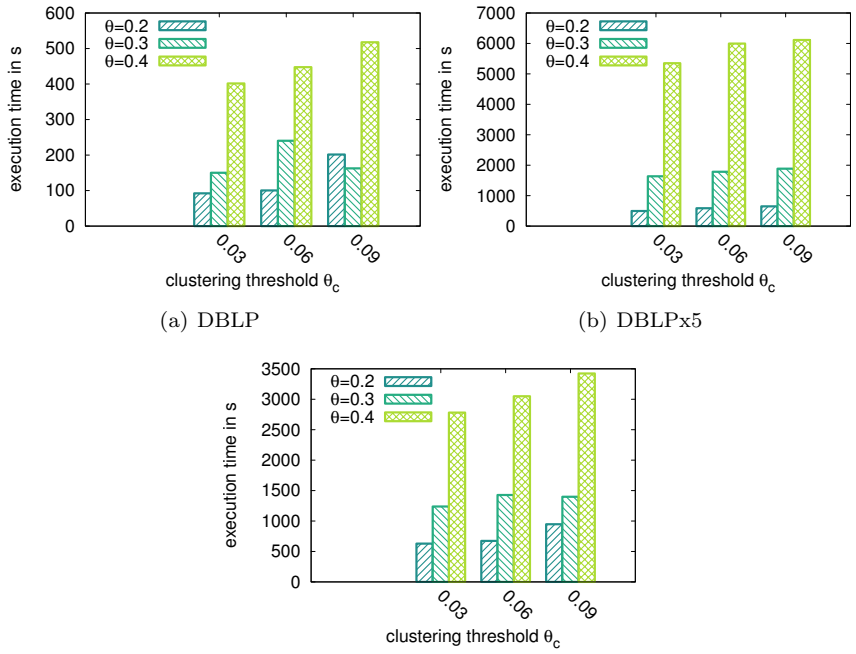


Figure 6.8: Performance of CL-PL algorithm when varying the dataset size

θ_c and the partitioning threshold δ were kept the same as for the previous experiment. We see that for both datasets, the CL-P algorithm exhibits better performance, when the number of nodes is increased. For the DBLPx5 dataset, when increasing the number of nodes from 4 to 8, the time cost decreases from 22% to 46%, and for the ORKU dataset the time savings are similar, ranging from 26% to 44%. Again, the largest performance improvement is observed for $\theta = 0.4$.

Furthermore, in Figure 6.8 we plotted the performance of the CL-P algorithm as we increase the size of the DBLP dataset. Note that the result size increases approximately linearly with the increase in the number of records. The rise of the execution time is the largest, i.e., for $\theta = 0.4$, when we increase the dataset size from x5 to x10. In this case the CL-P algorithm executes 7 times slower. However, the reason for this we see in the value of the partitioning threshold δ. We believe that with a more carefully chosen value for δ this increase in the execution time can be avoided. For all other cases of θ the decrease in performance is lower than 5 times.



(a) DBLP

(b) DBLPx5

(c) ORKU

Figure 6.9: Performance of CL algorithm when varying the clustering threshold θ_c

Effect of the Clustering Threshold θ_c

Another threshold that can have impact on the performance of the proposed clustering algorithm is the clustering threshold θ_c. Depending on the value of this threshold, the size and number of the formed clusters varies, and thus the performance of the whole algorithm. Figure 6.9 shows the performance of the CL algorithm for different values of θ_c for both datasets. We see that in almost all cases, setting $\theta_c = 0.03$ brings the best performance for the CL algorithm. This can be explained by two reasons. First, as we increase the clustering threshold θ_c, the running time of the clustering phase increases, since here we use the VJ algorithm to find the similar pairs. Second, the benefit by the additionally formed clusters does not seem to compensate for this increase in the running time. Thus, setting the clustering threshold θ_c to a very small value is the recommend choice, and in all further experiments we set θ_c to 0.03 for both CL and CL-P.

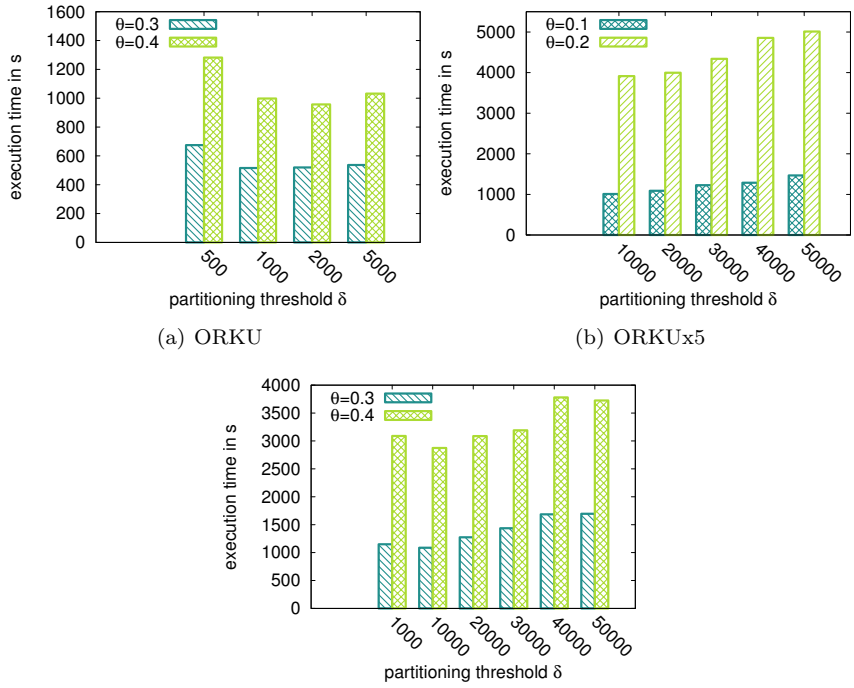
Effect of the Partitioning Threshold δ

The partitioning threshold δ is a parameter which decides which and how many posting lists need to be partitioned, and as such, it influences the performance of the CL-P algorithm. In Figure 6.10 we see the performance of the CL-P

algorithm as the partitioning threshold changes, for both datasets DBLP and ORKU and for different values of the threshold θ. For the DBLP dataset we show only the DBLPx5 increased dataset, since, as we showed in Figure 6.6(a), the DBLP dataset is small and does not benefit from the partitioning of the posting lists. For each dataset, we chose different varying ranges for the partitioning threshold, since its value is directly dependent from the size of the dataset. For ORKU (Figure 6.10(a)) we vary δ from 500 to 5000, for ORKUx5 (Figure 6.10(b)) we vary δ from 10000 to 50000 and for DBLPx5 (Figure 6.10(c)) we vary δ from 1000 to 50000. Furthermore, for ORKU and DBLPx5 we plot the performance for $\theta = 0.3$ and $\theta = 0.4$ (Figures 6.10(c) and 6.10(a), respectively), while for ORKUx5, for practical reasons, due to the large execution times when having large values of θ, we plot the performance for $\theta = 0.1$ and $\theta = 0.2$ (Figure 6.10(b)). In Figure 6.6(a) we see that the performance of CL-P is not greatly influenced by the partitioning threshold δ. Starting with small values of δ, the performance is slightly worse, due to the larger number of posting lists that need to be joined, and thus the overhead imposed by the Spark join is larger. Then, as we increase δ, the performance at first drops and reaches its minimum, and then starts to slightly increase. This is important to note, since it gives us more freedom of choosing the value for δ. Note, however, that choosing very small values can lead to either bad performance or crashes of the executors due to memory overhead caused by the joins. During our experiments execution, we experienced crashes due to memory overhead, whenever the δ value was set to an inappropriately small value, when considering the number of records being processed. On the other hand, setting δ to a very large value will not bring any performance benefit, since no postings lists will be partitioned.

Increasing the Size of the Rankings

Top-k rankings usually contain only very few items. In fact in our study [AIMS13] we showed that most of the rankings are of size 10 or 20. Therefore, in the previous experiments we focused on rankings of size 10. To see how the performance of the algorithms changes, when we have rankings of larger size, we also run experiments where $k = 25$. For this purpose we used the ORKU dataset, which contains also longer records. From the original dataset, we extracted around 1.5 million top-25 rankings, as described above. This dataset has a size of 289MB. The DBLP dataset contained only shorter records, and thus for this experiment we rely only on the ORKU dataset. Figure 6.11 shows the performance of the four algorithms when varying the distance threshold θ. While our algorithms still outperform the VJ algorithm, there are two important things to note here. First, the difference in the performance between VJ-NL and VJ is not so significant, and second CL performs almost the same as VJ-NL. This might be explained with the size of the dataset, since our clustering algorithms, CL and CL-P, perform better on larger datasets. The CL-P algorithm shows the best performance, except for $\theta = 0.1$, and is, as with rankings of size 10, least susceptible to the increase of the threshold θ. For $\theta = 0.1$, the VJ-NL



(a) ORKU

(b) ORKUx5

(c) DBLPx5

Figure 6.10: Performance of CL-P algorithm when varying the partitioning threshold δ

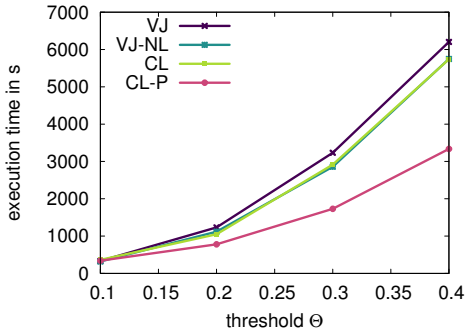
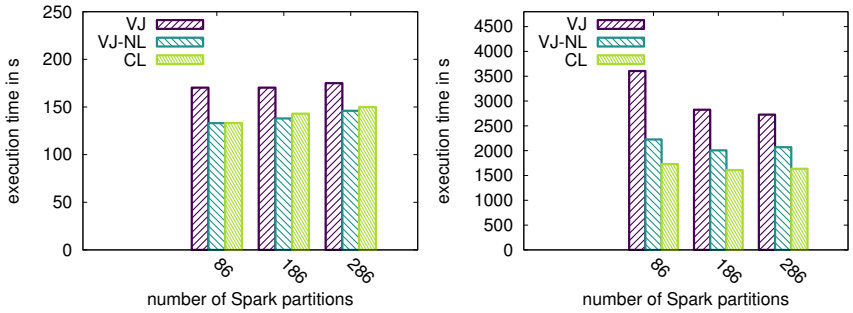


Figure 6.11: Performance of different algorithms for rankings of size 25 when varying the distance threshold θ, for the ORKU dataset.

algorithm performs slightly better than the other algorithms. The CL-P algorithm outperforms the VJ-NL algorithm for 1.5 and 1.9 times for $\theta = 0.2$, and $\theta = 0.3$ and 0.4, respectively. Note that for this experiment, for both CL and CL-P, we set $\theta_c = 0.03$ and the partitioning threshold δ, for CL-P, we set to 5000, for all values of θ.



(a) DBLP (b) DBLPx5

Figure 6.12: Performance of the VJ, VJ-NL and CL algorithms when varying the number of Spark partitions, for $\theta = 0.3$, for DBLP and DBLPx5.

Varying the number of Spark partitions

The general recommendation when executing Spark jobs is to set the number of partitions to be at least four times as the number of executors running. In our setting, this means that the general recommendation is to have at least 100 partitions. Figure 6.12 shows the performance of different algorithms (VJ, VJ-NL and CL) for different number of partitions. For this experiments the partitioning threshold θ is fixed to 0.3. We see that, for both DBLP and DBLPx5, the performance does not change much as we increase the number of partitions. In fact, we see that whether the performance increases or decreases, as we increase the number of partitions, depends on the size of the dataset. For the smaller dataset, DBLP, the best performance is observed when the number of partitions is set to 86, and then the performance slightly decreases. For DBLPx5, on the other hand, we have the best performance of both CL and VJ-NL for 186 partitions. Figure 6.13 shows the performance of the CL-P algorithm when changing the number of partitions. For CL-P we used a larger span of the number of partitions, from 286 to 686. Since here we additionally repartition the large partitioning into smaller ones, we believed that using a larger number of partitions is more appropriate for this approach However, as we can see from Figure 6.13, the performance is again not greatly influenced by the change of the number of the partitions. In fact, there is also a slight drop in the performance in the initial increase in the number of partitions, from 286 to 486. In all of the experiments presented before, the number of partitions was set to 286.

Lessons Learned

The proposed clustering algorithms, CL and CL-P, outperform the adaptation of the state-of-the-art algorithm for similarity joins over sets, VJ, for higher values of the distance threshold θ. For small values of θ the VJ algorithm is very efficient on its own, and thus, the benefits introduced by the additional stages of the CL approach, do not seem to pay off. This is also the case for

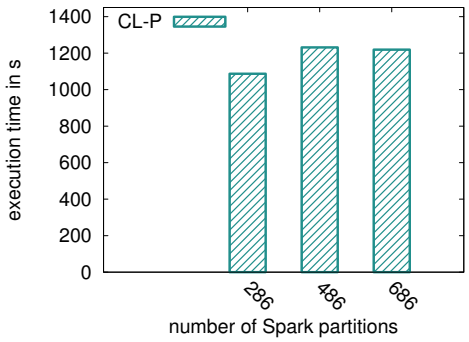


Figure 6.13: Performance of the CL-P algorithm when varying the number of Spark partitions, for $\delta = 10000$ and $\theta = 0.3$, for the DBLPx5 dataset.

small datasets. However, more importantly, for larger datasets, the CL and CL-P approaches seem to bring larger performance improvements over the VJ algorithm. Additionally, the both seem to be less susceptible to the increase of the distance threshold. This seems to be especially true for the CL-P algorithm, in particular, when the partitioning threshold is chosen right. Furthermore, our approach is more appropriate for handling datasets with skewed distribution, as first, the dataset is reduced for the joining phase, and second, large posting lists are split into smaller ones and processed in parallel. For choosing the partitioning threshold, δ, statistics like the number of records in the dataset, and the size of the vocabulary, or item domain, can be used. Although, we showed that the performance of the CL-P algorithm does not change drastically with the alteration of this threshold, still its value should be appropriately chosen, since assigning very small values leads to memory crashes. This is the drawback of our solution—since we rely on Spark joins, memory crashes are possible, especially where the result set is larger. Regarding the clustering threshold, we showed that using a very small values, like $\theta_c = 0.03$, leads to the best performance of the CL algorithm, regardless of the dataset.

6.6 Summary

In this chapter, we addressed distributed similarity join processing techniques for a datasets of top-k rankings. First, we adapted existing state-of-the-art set-based algorithm for distributed similarity joins for processing of top-k rankings. Next, we presented an approach which synthesizes this state-of-the-art distributed similarity join algorithm with the advantages of metric-space, distance-based, filtering. It works in several stages, where each can be independently configured from each other. Furthermore, our algorithms are designed and implemented in Apache Spark, as suggested by a recent experimental study. By a comprehensive performance evaluation using two real-world datasets, we showed that the presented approach exhibits better performance than the competitor, Vernica Join.

Chapter 7

Class-Constrained k-Nearest Neighbor (CCk-NN)

7.1 Introduction

Another typical similarity query is the k-Nearest-Neighbor (k-NN) query. As described in Chapter 2, a classical k-Nearest-Neighbor (k-NN) query returns k objects with the smallest distances to a query object q. In this chapter, we address a variation of this type of similarity query, i.e., we address the problem of processing k-NN queries where the result objects should additionally be constrained to specific user-defined classes, C_q. We refer to this problem as a Class-Constrained k-Nearest Neighbor (CCk-NN). The work presented in this chapter is based on our own publication at WebDB 2018 [MNM18].

This problem can be applied to many settings where the searched objects can have different attributes, however, it is especially common in the domain of geospatial data where we have objects of different classes, like sightseeing, recreational, restaurant, administrative and also more fine-grained types such as cuisine of a restaurant. For instance, consider Alice, who is visiting London. While out for sightseeing, she would like to have lunch in a French restaurant. She is traveling on budget, so the restaurant should be inexpensive. To assist Alice, an application would need to find the nearest k French restaurants, that have good reviews but are also not expensive. The application of *CCk-NN* is not restricted to geospatial data, but it is the most illustrative one.

To solve the CCk-NN problem, we could index the dataset using an index structure suitable for k-NN search, for instance, an R-tree [Gut84]. To process our k-NN query with class constraints, first a k'-NN query is executed, where $k' > k$. Then, it is checked whether the results fulfill all the classes in C_q. If not enough results are found, the query is executed once again with a larger

k', ideally in an incremental fashion. This solution has already been presented by Hjaltason and Samet [HS99]. The drawback of this approach is that many unnecessary comparisons have to be made, when the classes in C_q have low selectivity, i.e., only a small percentage of objects fall into the classes in C_q. Another possibility is creating an inverted index that maps a class to all objects that fall into this class. Then, instead of going through all objects, we could directly determine those objects that satisfy the classes C_q upfront. Then, we need to compute the distance to the query for all of them, and return the k objects with the smallest distance.

In this paper, similarly as in our work presented in Chapter 5, we present a solution which combines an inverted index with a spatial index structure. An inverted index is used to find the objects that satisfy the classes in C_q, but instead of comparing all these objects to the query, an index structure is used to find the k closest objects to q. One major challenge that arises from our indexing approach is the question for how many and for which classes, or combination of classes, do we create sub indices. Since one object can belong to several classes, it would also be indexed by multiple indices. The number of sub indices that can be created, and thus the memory and construction time overhead, quickly increase as the set of classes increases. The focus of this work is to explore and in particular optimize the potential of having inverted indices of different granularity, in the sense that not only single classes but pairs, triplets, etc. of classes are used as keys to determine the corresponding objects.

7.1.1 Problem Statement

The problem addressed in this chapter can be defined as follows:

Given a set of objects $O = \{o_1 \dots o_n\}$, where each object $o_i \in O$ has a set C_{o_i} of associated classes from a global set of classes $C = \{c_1, \dots, c_j\}$, and a distance function d, the task is, for a user specified query q and query classes $C_q \subseteq C$ to find k objects $O_R \subseteq O$ such that $\forall o_r \in O_R : C_{o_r} \supseteq C_q$ and there is *no* object $o_s \in O$ and $o_r \in O_R$ such that $d(q, o_s) < d(q, o_r)$ and $C_{o_s} \supseteq C_q$.

If there is more than one class constraint given ($|C_q| > 1$), then they are all applied, thus they have a conjunctive meaning. The result set can be smaller than k in the case when less than k objects satisfy the classes in C_q. Each class is of the form attribute:value, for instance color:blue.

As with standard k-NN search, we aim at having as few object comparisons as possible, low memory usage, fast index creation time, while having low query response time.

In prior work of our group [dSTM18], this problem was already presented and it was shown that creating an index for all combination of classes can lead to performance gains, however, it also leads to significantly increased memory and construction time overhead. It is clear that materializing all combination of classes is not a feasible when having larger global set of classes, C, as it requires possibly generating $2^{|C|}$ sub indices. In this work, we specifically address this.

7.1.2 Contributions and Outline

The contributions of this work are as follows:

- We propose an index, coined MISP, to efficiently solve the CCk-NN. In MISP, objects that belong to the same class, or combination of classes, are indexed together using a separate similarity-search index like an R-tree, which we call sub indices. In addition, MISP integrates an inverted index to easily match the classes, or combination of classes, to the sub indices.
- We present a cost model which estimates the performance and index size of the MISP index. Furthermore, we sketch an algorithm and discuss how a configuration of MISP can be chosen that would perform best under limited memory resources.
- Using both a synthetic and a real dataset, we first experimentally evaluate the correctness of the proposed cost model, and, second, we evaluate the performance gain of the proposed index against existing approaches under different configuration setups.

Section 7.2 presents our combined index, how it is constructed and queried. In Section 7.3, we provide a cost model that estimates the performance of the index depending on the number of sub indices created. In Section 7.4, we show a detailed experimental evaluation with both synthetic and real dataset, where we also show the accuracy of our cost model. Finally, in Section 7.5, we summarize the work presented in this chapter.

7.2 Multi-Key Inverted Index with Smart Posting Lists (MISP)

In this work, similarly to the work presented in Chapters 5 and 6, we combine an inverted index with an existing spatial indexing technique, into a new index, named Multi-Key Inverted Index with Smart Posting Lists (MISP). MISP combines the benefits of both inverted indices and k-NN index structures. The idea behind this mixture, for this specific problem setting, is that we can easily find the data objects that belong to the classes we are searching for, and then, find the most similar k ones without comparing them all to q. Figure 7.1 illustrates this index. Instead of having data stored in plain posting lists, we index them using iDistance [JOT$^+$05], an index structure for searching data in metric space that allows incremental k-NN searching—but any other index structure for k-NN search can be used instead. The MISP index can also have combinations of classes as keys in the inverted index. Thus, only the objects that satisfy all of the key's classes are indexed in the corresponding iDistance sub index. We call these m-key sub indices, where m is the number of keys for this sub index. Figure 7.1 illustrates a simplified MISP index for the Alice example

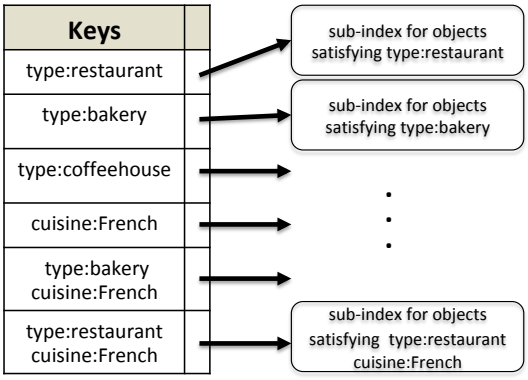


Figure 7.1: Multi-Key Inverted Index with Smart Posting Lists

presented in Section 7.1. This MISP index has indexed in 1-key sub indices all objects that are of type restaurant, coffeehouse, or bakery, as well as objects with cuisine French. In addition, via 2-key sub indices, both French restaurants and French bakeries are indexed. Hence, when Alice is searching for the nearest French restaurant, we can directly search for the nearest French restaurant using the created 2-key sub index—instead of searching all the restaurants and then pruning the ones which are not French.

However, there is also the drawback that a French restaurant will be indexed three times: once for restaurants, once for objects with French cuisine, and once in the sub index for French restaurants. However, when searching for French coffeehouses, we have the opposite situation: During search, we have to access either the sub index containing all coffeehouses, or the one for French cuisine, but we do not have an additional storage overhead. If we do not expect many queries for French coffeehouses, this would be the preferred solution.

7.2.1 Index Creation

MISP is built layer-by-layer in a *bottom-up* manner. First, we create the 1-key sub indices. For correctness reasons, in order not to miss any potential result object, this layer is completely built, i.e., there is one sub index for each class $c \in C$. This layer represents a foundation for the next layers and also serves as *fallback* for every query, because every data object must fall into one of these indices. For instance, if we are given a query q with two class constraints, restaurants and French, $|C_q| = 2$, it can happen that there is no sub index created for this combination of classes. Then, to find the results, we can query the single-key sub indices, either the one for restaurants or for French objects depending on its size. For the next layer(s), the 2-key sub indices (and 3-key sub indices, etc.), we can decide, if we want to build all possible ones. We combine every 1-key sub index with every other. In order to speed up construction, we use the layer below as basis for calculating the possible combinations. The

drawback of this approach is that we have to have all possible combinations from the layer below, to make every possible combination in the current level. If we do not build the level of $m-1$ fully, we will miss some combinations in the m-key sub indices. The maximum m for which we create m-key sub indices we refer to as the *depth of the MISP index*. The depth of the MISP index in Figure 7.1 is two.

7.2.2 Querying

Querying the MISP index is slightly different than querying the inverted index. Instead of querying all sub indices for each class in C_q, and then merging the results, we query only the sub index for the most selective class (combination of classes) in C_q, and the sub index is incrementally searched until we find k nearest objects that satisfy all the constraining classes. This is possible since each object in the index has a set of classes that it belongs to, C_{o_i}, attached to it. In order to find the most selective class, during construction time of the index, we gather statistics for how many elements each sub index has indexed and, we keep class co-occurrence statistics. When at query time we are given a set of classes C_q where we have a combination of classes that do not occur together in the dataset, we terminate the search early, as there are no possible results.

When we have MISP index with larger depth, finding the most selective sub index imposes a small overhead, because in order to find it, we first create the power set of the query's classes, that is, 2^{C_q}. Then, for each element of this power set, we check if we have the corresponding sub index and, if so, its size.

Therefore, in order to reduce the overhead, while increasing our chances to "hit" a selective multi-key sub index, we create only those multi-key indices of the classes that contain the most elements in the dataset D. The idea behind this is that we assume that the issued queries would follow the same class distribution as our dataset D. In order to find the most popular classes, we use the class co-occurrence statistics mentioned above. If the queries are not following the dataset characteristics, one would require a query log for analysis, but the idea would remain the same.

Since an object can take on multiple classes, the drawback of MISP is that its memory consumption as well as construction and maintenance overhead increases with the number of sub indices created. Furthermore, the time needed for searching for the best most selective sub index increases, as we have to check for every m-combination of class constraints, if they exist and if yes, how many objects are in this sub index. Even when creating the multi-key sub indices only for the most popular classes, the number of possible combinations of sub indices, computed as $\sum_1^c \binom{c}{i}$, drastically increases with the number of possible classes (i.e., $|C| = c$). To address this issue, in the next section, we present a cost model that finds the cutting point at which the creation of additional sub indices would lead only to negligible performance gains.

7.3 Cost Model

In the following, we propose a cost model that estimates the cost for querying and the cost for storing the MISP index. As the MISP index employs an inverted index, the cost model has some resemblance with the cost model devised for estimating the performance of the Coarse index, described in Chapter 5. However, the two indicies differ in the way they are queried, which, in turn, highly impacts the query performance.

7.3.1 Cost for Querying

In order to understand the cost for querying MISP, we compute the expected number of distance function calls, for a given query q and a set of constraining classes C_q. For this, we need to estimate the size of the sub index, the probability of hitting an index structure at some index level m, and the cost for querying the specific sub index structure used. However, since our goal is to devise a general and simple cost model, independent of the underlying index structure used, we devise a cost model where we assume that all indexed objects in the queried sub index are evaluated via a full scan. This gives us an estimation of the upper bound of the cost for querying the MISP index. Furthermore, for simplicity, we assume that the data is uniformly distributed over the global set of classes C, and we assume that the size of the dataset $|D| = d$ and of the global set of classes $|C| = c$ is known. We also assume that the average expected size of the set of constraining classes $|C_q| = a_q$ is known. We start by estimating the cost for querying a MISP index with depth 1, which is basically an inverted index where each sub index contains elements from only one class and we build up our cost model from there.

Let Y be a random variable representing the number of objects, y_i, in the sub index for the i-th class c_i. $E[Y]$ is then the expected number of data points in this index. When the objects in the dataset belong to more than one class, they will be inserted into every sub index they belong to. Therefore, to estimate the expected size of the sub indices, we need to estimate the average count of classes that an object belongs to, avg_{c_o}. This can be computed as $avg_{c_o} = \sum_i \frac{C_{o_i}}{|O|}$. The total number of stored objects in the MISP index can be estimated as $d \cdot avg_{c_o}$. Then $y_i = \frac{d \cdot avg_{c_o}}{c}$ and $P(y_i) = \frac{1}{c}$ as we are always querying only one sub index in MISP. Thus, the expected size of the queried sub index can be computed as:

$$E[Y_{InvSeq}] = \sum_{i=1}^{c} y_i \cdot P(y_i) = \sum_{i=1}^{c} \frac{d \cdot avg_{c_o}}{c} \cdot \frac{1}{c} = \frac{d \cdot avg_{c_o}}{c} \tag{7.1}$$

Thus, in the case of having a sub index only for each individual class, the upper bound on the number of distance function computations can be computed as:

$$E[Y_{InvSeq}] = \frac{d \cdot avg_{c_o}}{c} \tag{7.2}$$

Next, we extend the cost model to estimate the cost of a MISP index of depth 2, where in addition to the sub indices for each class, the MISP index also contains sub index for each pair of classes in C. In this case, we need to take into account the number of additional sub indices created. This can be calculated as:

$$max_{extra} = \binom{c}{2} = \frac{1}{2}c(c-1) \tag{7.3}$$

Since in practice the number of max_{extra} sub indices can be very large, we actually would create only $l, l \leq max_{extra}$ sub indices. To calculate the expected number of items in such an index, we first need to estimate the expected size of a 2-key sub index. Similarly to the expected size of the single-key sub indices, we have $P(y_i) = \frac{1}{l}$ and:

$$y_i = \underbrace{\frac{d \cdot avg_{c_o}}{c}}_{\text{first "entry"}} \cdot \underbrace{\frac{avg_{c_o} - 1}{c-1}}_{\substack{\text{matching} \\ \text{second "entry"}}}$$

Then:

$$E[Y_{2-key}] = \sum_{i=1}^{l} y_i \cdot P(y_i) = \sum_{i=1}^{l} \frac{d \cdot avg_{c_o}}{c} \cdot \frac{avg_{c_o} - 1}{c-1} \cdot \frac{1}{l} \tag{7.4}$$

Next, we need to estimate the probability, that we hit such 2-*key sub index*. The probability depends on the number of class constraints $|C_q|$ provided at query time. If we have only one class constraint on the query, we can not hit a 2-*key sub index* at all. If we have built all possible 2-*key sub indices* and $|C_q| \geq 2$, we have a 100% chance that we will hit such sub index. This is estimated with the following formula:

$$P(2-key) = \begin{cases} 0 & a_q = 1 \\ 1 & l = max_{extra} \wedge a_q \geq 2 \\ \frac{l}{max_{extra}} & a_q = 2 \\ 1 - B_{n,p,k}(\binom{a_q}{2}, \frac{l}{max_{extra}}, 0) & a_q > 2 \end{cases} \tag{7.5}$$

$B_{n,p,k} = \binom{n}{k} p^k (1-p)^{n-k}$ is the probability to hit exactly k times in a Bernoulli process with n trials and p is the success probability. In our case, we have a Bernoulli process where n is the number of pair of classes that we can build with the classes in C_q. Then, we want to know, which are the chances that these would match at least one of the 2-key sub indices that are created. This is expressed by the complementary probability of hitting exact zero times.

The final expectation of data point in the accessed index is:

$$\begin{aligned} E[Y_{MISP_{2-key}}] = P(2-key) \cdot E[Y_{2-key}] \\ + (1 - P(2-key)) \cdot E[Y_{InvSeq}] \end{aligned} \tag{7.6}$$

Estimating the cost at any level m can be done accordingly. The number of m-key sub indices, under the assumption that all indices at each level are created, is $l_m = \binom{c}{m}$, and the total number of sub indices would be $l_{total} = \sum_{i=1}^{m} l_i$. The number of items can be computed as:

$$y_i(m) = \frac{d \cdot avg_{c_o}}{c} \cdot \frac{avg_{c_o} - 1}{c-1} \cdots \frac{avg_{c_o} - (m-1)}{c-(m-1)} = d \cdot \prod_{i=0}^{m-1} \frac{avg_{c_o} - i}{c-i} \tag{7.7}$$

and similarly $P(y_i) = \frac{1}{l_m}$. Thus $E[Y_{m\text{-}indexSeq}]$ can be computed as:

$$E[Y_{m\text{-}key}] = \frac{d}{l_m} \cdot \prod_{i=0}^{m-1} \frac{avg_{c_o} - i}{c-i} \tag{7.8}$$

Next, we need to calculate the probability of hitting such an index. Since we are always favoring the sub indices for keys created as a combination of more class constraints, we estimate the probability of hitting some level i, where $1 \leq i \leq m$, recursively as:

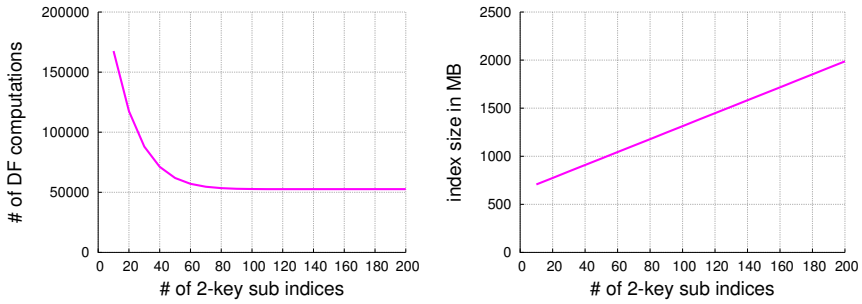
$$prop(i) = \left(1 - \sum_{j=i+1}^{m} prop(i)\right) \cdot (1 - B_{n,p,k}(\binom{a_q}{i}, \frac{l_i}{l_{i_{max}}}, 0)) \tag{7.9}$$

The first part of Equation 7.9 estimates the probability that we have not yet hit a level having more class constraints than the current level i. The second part is the generalization of Equation 7.5. It is still a series of Bernoulli processes, however, this time we estimate to hit at least one of the combination of the classes in C_q with the probability of the number of sub indices build at level i, l_i, divided by the possible sub indices at this level, $l_{i_{max}}$. To get the estimated number of distance function calls when we have the general case of MISP index, we sum up the numbers:

$$E[Y_{MISP}(m) = \sum_{i=1}^{m} prop(i) \cdot E[Y_{i\text{-}key}] \tag{7.10}$$

By plugging a value for the number of created sub indices l at level m, l_m, in Equation 7.10 we can estimate the number of distance function computations depending on the number of sub indices created at each level of the MISP index. Thus, Equation 7.10 should ideally help us not only in choosing the depth of the index, but also the number of sub indices created at level m.

According to Equation 7.6, Figure 7.2(a) shows the estimation of the number of distance function computations when varying the number of 2-key sub indices



(a) (b)

Figure 7.2: Cost estimation (#df computations (a) and index size in MB (b)) for 2-key MISP index when varying the number of 2 key sub indices created ($d = 1,000,000$, $c = 20$, $avg_{c_o} = 5$ and $a_q = 5$).

for a dataset with the following parameters: $d = 1,000,000$, $c = 20$, $avg_{c_o} = 5$ and $a_q = 5$. As expected, the number of distance function computations is decreasing as we increase the number of sub indices created. However, we can see that the rate at which the value is decreasing is dropping, and after some point, in this case around $l = 60$, the decrease in distance function computations becomes insignificant. One way of choosing the number of sub indices to be created at level m would be to calculate the percent decrease and then stop when this becomes lower than some threshold value t. Below, we propose an algorithm for choosing the number of sub indices to be created that also takes into account the estimated size of the index.

7.3.2 Estimating the Size of the Index

The size of the MISP index can be easily estimated using the equations from before. Note that in this approximation, the occupied space for the data itself will be left out, because it will always be constant for one data set, independent of the employed indexing method. This means, when comparing two different index structures, only the difference of their overhead is relevant. We try to formulate this as general as possible, as this overhead depends heavily on the implementation.

For estimating the size of the MISP index we again need the expected size of the sub indices, the size of $|C|$, and the overhead imposed by the index structure. Let X be a random variable representing the size of the sub indices. Without loss of generality, we assume that in the index we only store pointers to the data objects. For the MISP index with depth one we would have:

$$E[X_{MISP_1}] = c \cdot (oh_{tree} + pointer + string + integer) + c \cdot \frac{d \cdot a_d}{c} \cdot oh_{node} \quad (7.11)$$

where the string and the pointers take into account for storing the key in the inverted index, and a pointer to each sub index, and oh_{tree} and oh_{node} account

input: $|D|$, $|C|$, a_q, avg_{c_o}, $step$
1 **while** $m \leq a_q$ **do**
2 $l_i = 0$; increase(m,1)
3 **while** improvement(L,M) > cost(L,M) **do**
4 increase(l_i,$step$)
5 M=est_size(m, l_i);
6 L=est_df_computations(m, l_i)
7 **return** L, m

Algorithm 5: Algorithm for determining the depth m and the number of sub indices per level for the MISP index by accounting the performance gained versus the memory consumed for the created sub indices.

for the overhead of each sub index and each node in the sub index which we consider them to be trees.

Generalizing this cost model to level m is straightforward, keeping in mind that we can reuse the estimations from the previous subsection. We have:

$$E[X_{MISP_m}] = (l_m \cdot (oh_{tree} + pointer + i \cdot string + integer) + l_m \cdot ((d \cdot \prod_{i=0}^{m-1} \frac{a_d - i}{c - i} + 1) \cdot oh_{node}) \tag{7.12}$$

where $1 \leq l_m \leq \binom{c}{m}$ is the number of sub indices created at level m. To get the total size of the MISP index we just need to sum up the costs from each level. Figure 7.2(b) shows the estimated cost for a 2-key MISP index for a dataset with $d = 1,000,000$, $c = 20$, $avg_{c_o} = 5$ and $a_q = 5$. Furthermore, as overhead for the tree and node we have used $oh_{tree} = 704b$, $oh_{node} = 1024b$, $pointer = 256b$, $string = 576b$ and $integer = 128b$, which account to our Java implementation. As expected, the size of the index increases linearly with the number of sub indices added (cf., Figure 7.2(b)).

7.3.3 Overall Cost

To put the two costs together one needs to consider the trade-off between the performance gain induced by having more sub indices and the price of increased memory consumption. At some point, as it can be seen in Figure 7.2, the reduction in distance function computations becomes marginal, while the memory consumption steadily increases. At this point, the price paid, in terms of memory consumption, for creating more sub indices is larger than the benefit gained.

Algorithm 5 sketches this idea. For each level m, the distance function computations and the index size is estimated, for varying number of sub indices

created. While the benefit from having these indices is higher than the cost paid, we continue adding more indices. Once this stopping condition applies, the number of sub indices that should be created for the current level is marked, and we continue with the estimations for the next level m.

Depending on the use case scenario, as cost paid can be considered also the construction time, or the overhead imposed for updates. Considering that when adding more sub indices we also increase the redundancy of objects in the MISP index, the cost for updates needs to be considered. While the construction time can be deduced form the size of the index, the cost for updates needs more detailed investigation. However, estimating the cost for updates, and how exactly the trade-off between the benefit gained and the cost paid can be computed is left for future work.

7.4 Experiments

All experiments were executed on a desktop machine running Ubuntu 16.4 with an Intel Core^{TM} i5-4570 CPU and 8 GB of RAM (5 GB used by the Java VM). We implemented the different indexing strategies using Java 8.

Datasets

For the experiments we used both a synthetic and a real dataset. The synthetic dataset consist of two dimensional points where the value of every dimension is fixed between 0 and 100. The dataset is created by first randomly picking points that act as the pivot of the partitions. Then points in each partition are added by moving the points away from the pivot. The direction, and the distance is chosen randomly, however, it does not exceed the radius of the partition, provided at input. Important to note is that the data objects are distributed uniformly over the classes. For the experiments in this paper, we have generated a dataset with the parameters: $|O| = 1,000,000$, $|C| = 20$, $|C_{o_i}| \leq 5$, $|C_q| \leq 5$.

For the real-world dataset, we used geospatial data provided by OpenStreetMap (OSM). We used the available data from within the boundaries of Germany[1]. For the experiments we focus on Points of Interest (POIs), and the rest of the data, e.g., streets, railroads or buildings, were dropped. We were left with about 1.3 million data entries. The data in the OSM dataset is described by a set of user specified tags whose values are freely inputed by users. The tags we cleaned and adapted as classes of the data objcets. To each data object we assigned at most six classes, thus $\forall o_i \in O : |C_{o_i}| \leq 6$: amenity, shop, city, hasName, hasOperator, hasOpeningHours. The first two classes can take many values. *Amenity* can take more than 8500 and *shop* more than 10000 different values. For city the values are only sometimes there. The last three we made them as boolean classes, an object either belongs to this class or not. For instance, a data object with the tags "amenity=pub, name=Wladi Rockstock,

[1] `http://download.geofabrik.de/europe/germany.html`, downloaded on 2016-11-20

opening_hours=Tu-Su 18:00+" would get the classes "amenity:pub", "hasName" and "hasOpeningHours". The queries that we use for the experiments are randomly chosen from the data.

Indexes under Investigation

We have focused on measuring the performance of the following indexing strategies:

- Sequential scan (Seq), as a baseline approach
- *iDistance* [JOT+05] having all the data objects indexed.
- Inverted index using single classes as keys (*InvSeq*).
- Additionally, for understanding the accuracy of the cost model, we also look at MISP-seq, that is, MISP where the elements in the inverted index are evaluated by sequential scan (MIPS-seq).
- Our combined index, MISP, where iDistance is used as sub index (MISP).

Performance Measures

To get some insights into the performance of the different indexing strategies we measured the following: *(i)* index creation time, *(ii)* memory consumption, *(iii)* wallclock time, and *(iv)* number of distance function (df) computations. We report the time (df computations) needed to execute 100 queries, with $k = 20$.

7.4.1 Results

Validity of the Cost Model

In order to assess the validity of our cost model, we conduct an experiment where we measure the performance of the MISP index as we vary the number of 2-key and 3-key sub indices created. For this experiment, we used the synthetic dataset described above. We also measured the performance for a version of the MISP index, where the sub indices are plain posting lists, denoted as MISP-seq. We did this because this better reflects the cost model, which estimates the distance function comparisons to all the objects in a sub index.

As we can see in Figure 7.3, the cost model provides an accurate estimation of the real performance and size of the MISP index for level 2. The performance of the MISP index where as sub index we use the iDistance index is lower than the one estimated with the cost model. However, what is important is that the performance improvement when adding more sub indices follows the same trend. For the estimation for the number of sub indices at level 3 (Figure 7.3(c)), we have created only 60 sub indices at level 2, as this seems to be the point at which the improvement in distance function computations becomes marginal.

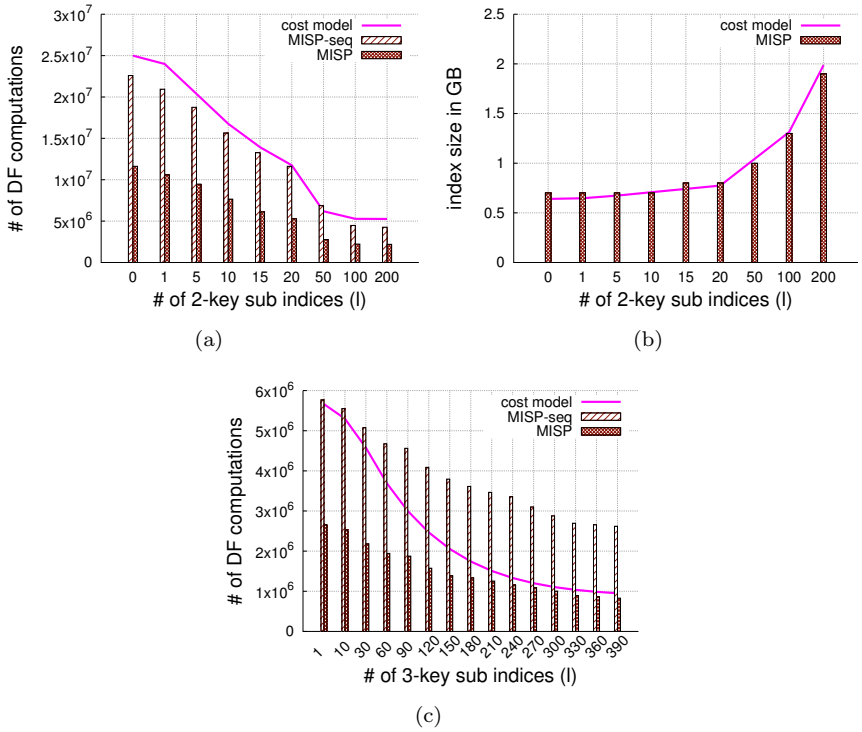


Figure 7.3: Comparison of the cost model estimation with the experimentally measured df computations (a) and index size (b) when changing the number of 2-key and 3-key (c) sub indices (synthetic dataset).

We see that the cost model, at first, overestimates the decrease in the number of distance function computations, but then at around 200 sub indices, the cost model starts to more realistically capture the decline in distance function computations of the MISP index.

Comparison of Different Indexing Techniques

Next, using the OSM dataset, we compare the MISP index against the inverted index and the iDistance. We also plot the result when having a plain sequential scan, as a baseline. To see whether introducing 3-key sub indices would lead to more performance benefits, we have also created a version of the MISP index with the best 500 2-key iDistance sub indices, and the objects belonging to the remaining 2-key combinations indexed in a corresponding posting list, instead of using an iDistance index. In addition to this we also created 500 3-key iDistance sub indices. This MISP index version is denoted as MISP2+3 in Figures 7.4 and 7.5 .

Figure 7.4 shows the memory consumption and the creation time of the MISP where we have created the best 500 2-key sub indices, compared to the iDistance the inverted index and a sequential scan. The reason for creating 500 sub indices

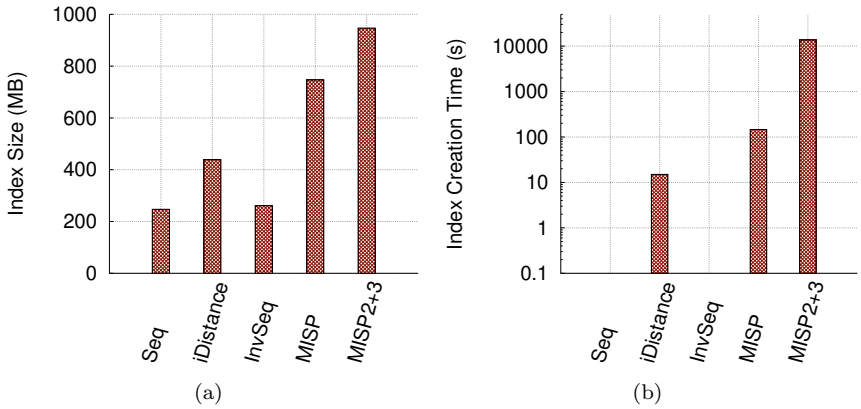


Figure 7.4: Memory consumption (a) and creation time (log-scale) (b) of different indexing strategies (OSM dataset).

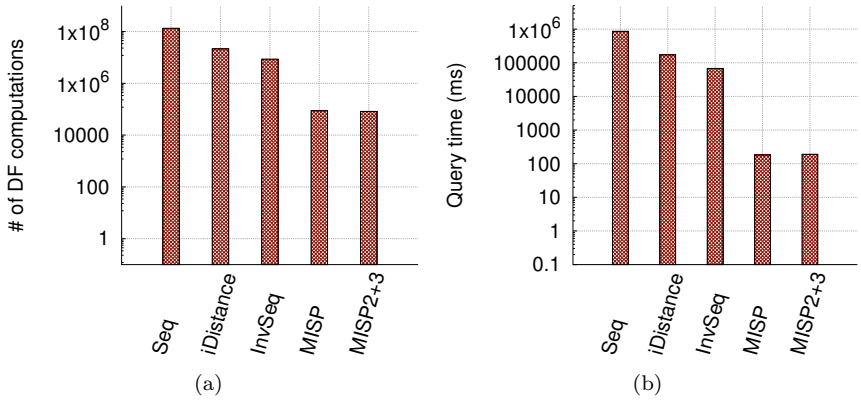
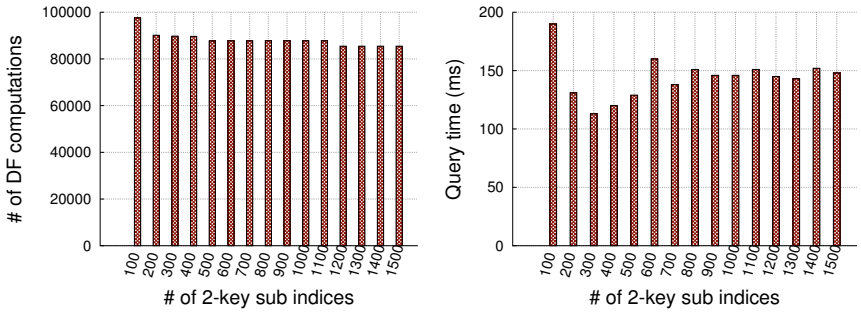


Figure 7.5: Log-scale plots of df computations (a) and query execution time (b) different indexing strategies (OSM dataset).

is explained below. As expected, the creation time and memory consumption of the MISP index is worse compared to the two baselines. Especially for the creation time, we see that creating additional sub indicies presents quite some overhead, especially when we have created all 2-key sub indices and 500 3-key sub indices (MISP2+3). However, even though we are working with relatively large dataset of 1.3 million objects, the memory consumption is still reasonable, i.e., under 1GB.

Figure 7.5 shows the distance function computations and the query execution time for different indexing techniques. As we can see in Figure 7.5(a) by using the MISP index with 500 2-key sub indices we have significantly lower number of distance function computations compared to the iDistance index or the inverted index. Accordingly, the MISP index has the best query execution time as well (Figure 7.5(b)). It is interesting to notice that the inverted index outperforms the iDistance index for this dataset. We can see that adding the additional 3-key



(a) (b)

Figure 7.6: df computations (a) and query time (b) of MISP index for different number of 2-key sub indices (OSM dataset).

sub indices only slightly reduces the number of distance function computations. There is even a slight degradation in the query execution time due to overhead for finding the best index. Note, that Figure 7.5 is in logarithmic scale.

Varying the Number of 2-Key Sub Indices

Next we evaluate the performance of the MISP index using the OSM dataset when varying the number of 2-key sub indices. Since this dataset has a very large set of possible classes, we first investigate the performance of the MISP index in the case when we vary the number of only 2-key sub indices created. Only for 2-key sub indices in this dataset there are $196,284$ possible combinations. Note that all the 1-key sub indices are also present in the MISP index, and we create the n best 2-key sub indices as described in Section 7.2. Figure 7.6(a) shows that for the real dataset the distance function computation follow a similar trend as with the synthetic dataset. At first, there is a drastic drop in the number of distance function computations, which then starts to slowly decrease. However, in Figure 7.6(b) we see that as we increase the number of 2-key sub indices there is an added overhead from finding the best index, and thus the query execution time at first improves, but then starts to slightly deteriorate. At the moment this overhead is not taken into account in our cost model.

We see that for 500 2-key sub indices the df computations only marginally improve, and there is no significant performance degradation. That is why in the previous experiment only the best 500 sub indices were created.

Performance Based on the Depth of MISP Index

To better investigate the performance of the MISP index, when having an index with different depth, i.e., when increasing the value of m, we performed an experiment using the synthetic dataset. We test the performance of the MISP index for values $2 \leq m \leq 5$. However, to reduce the overhead, on each level we create only the 100 best m-key sub indices. The performance of the MISP index,

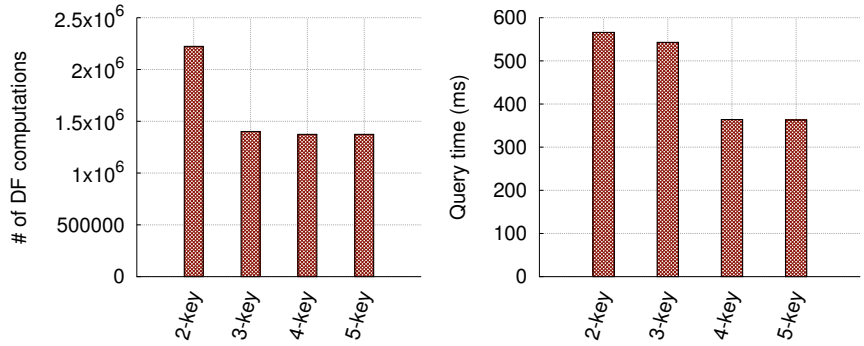


Figure 7.7: Comparison of df computations (a) and query execution time (b) when different levels of m-key sub indices are build (synthetic dataset).

as it can be seen in Figure 7.7, does not improve much as we add additional layers. We have measured that the construction time increases significantly with adding layers of m-key sub indices. The memory consumption, on the other hand, shows not to be such an issue.

Lessons Learned

From the presented experiments we can conclude that the MISP index clearly outperforms the inverted index and the iDistance index. The query execution benefits of the MISP index can already be seen when having sub indices at level 2. Adding more levels to the index, as expected, brings additional performance improvement, however, it comes with the expense of higher memory consumption and index creation time. In fact, the index creation cost seems to be more of an issue that the memory consumption, when it comes to adding more sub indices. This is because the creation of each iDistance index is expensive. An interesting observation is that adding more sub indices at some level is not always beneficial, when it comes to the performance, especially when the cardinality of the global set of classes C is very large. In this case, the cost for searching the best index to be queried, seems to outweigh the benefit of having these indices. Finally, and most importantly, we could see that the presented cost model correctly captures the performance and memory consumption cost of the MISP index, and thus, could be used for estimating the number of sub indices that would bring the best performance time versus memory consumption trade off, for some dataset D, with an associated set of global classes C.

7.5 Summary

In this work, we proposed and analyzed an indexing strategy for solving the Class-Constrained k-NN problem. As a solution we proposed an index that

combines an inverted index with a similarity search index. We specifically use the iDistance, but any other index can be used. To solve the problem imposed by the proposed indexing strategy, of how many and which sub indices do we need to create to get the best performance, while reducing the overhead, we proposed an experimentally validated cost model and discussed how the user can find the point at which the performance gain will be minor when creating more sub indices. Furthermore, in an experimental study we show that our indexing strategy performs better over existing indexing techniques.

Chapter 8

Conclusion and Outlook

In this thesis we addressed the problem of similarity search over top-k rankings and class-constrained objects. All of the presented solutions combine in some way an inverted index with distance-based filtering.

First, we have presented some theoretical bounds for top-k rankings, which allow the adaptation of existing state-of-the-art set-based similarity search algorithms to our problem setting. Next, the Coarse index, which unites an inverted index and a BK-tree, in order to efficiently process similarity range queries over top-k rankings has been presented. The Coarse index first indexes the data using a BK-tree, and uses this tree to divide the data into partitions with fixed radius. The centroid of each partition is then indexed using an inverted index. However, in order to achieve the best performance, this index blend requires some tunning, which dictates the granularity of the partitions and the size of the data indexed with an inverted index. Thus, we have also presented a theoretical cost model, which, based on the data distribution, enables choosing the performance sweet spot of the Coarse index. We further presented optimizations over the inverted index, that can either be used independently or in combination with the Coarse index, to even further reduce the time cost needed for processing similarity range queries. Furthermore, we have presented an iterative, distributed, algorithm, implemented in Apache Spark, for efficiently answering similarity join queries over top-k rankings, which reuses the concepts introduced for solving the similarity range problem, and applies them for answering similarity join queries in a distributed setting.

In addition, we have presented the MISP index structure, which is specifically designed to efficiently handle a special type of the k nearest neighbors query, where class constraints are also introduced, the class-constrained k nearest neighbors query (CCk-NN). Similarly as the Coarse index, MISP combines and inverted index with a spatial index structure, specifically the iDistance—an inverted index is used to index the objects per class, or combination of classes. Then, instead of placing the objects in regular posting lists, MISP indexes them with the iDistance index. This allows both fast access to the objects that satisfy

the query class constraints and efficient execution of the k-NN query. As the combination of classes, and thus, the number of possible iDistance sub indices, explodes combinatorially, we have presented a cost model that assesses the performance and memory consumption of the MISP index, with the goal of finding the number of sub indices and the combination of classes needed in order to achieve the best trade-off between the performance gained and the cost paid in memory consumption or construction time.

The presented similarity search approaches over top-k rankings are designed to work only over same size rankings. Extending these solutions to work for rankings of different sizes would be a possible future work. In Chapter 4 we have already sketched some of the bounds needed for this extension, however, applying and evaluating these in practice still needs to be done. Another interesting direction of future work would be to apply the proposed approaches for similarity search over sets when using an appropriate distance measure that also satisfies the triangle inequality, e.g., the Jaccard distance. It would be interesting to see how the proposed approaches perform against set-based approaches using existing benchmarks.

As for the MISP index, as future work the presented cost model could be extended to actually propose concrete number of sub indices that need to be created, so that some performance-overhead trade-off is achieved. For this, approaches like the weighted sum method, the weighted metric method or the ϵ-constrained method, which work with competing costs, could be used—one cost being the cost for querying the index, and the other one, the construction (memory consumption) cost.

List of Figures

List of Algorithms

List of Tables

Bibliography

[AIMS13] Foteini Alvanaki, Evica Ilieva, Sebastian Michel, and Aleksandar Stupar. Interesting event detection through hall of fame rankings. In Kristen LeFevre, Ashwin Machanavajjhala, and Adam Silberstein, editors, *Proceedings of the 3rd ACM SIGMOD Workshop on Databases and Social Networks, DBSocial 2013, New York, NY, USA, June, 23, 2013*, pages 7–12. ACM, 2013.

[BCG02] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM Trans. Database Syst.*, 27(2):153–187, 2002.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[BK73] Walter A. Burkhard and Robert M. Keller. Some approaches to best-match file searching. *Commun. ACM*, 16(4):230–236, 1973.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Inf.*, 1:173–189, 1972.

[BMS07] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling up all pairs similarity search. In Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy, editors, *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, pages 131–140. ACM, 2007.

[BÖ97] Tolga Bozkaya and Z. Meral Özsoyoglu. Distance-based indexing for high-dimensional metric spaces. In Joan Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA.*, pages 357–368. ACM Press, 1997.

[BYCMW94] Ricardo A. Baeza-Yates, Walter Cunto, Udi Manber, and Sun Wu. Proximity matching using fixed-queries trees. In Maxime Crochemore and Dan Gusfield, editors, *CPM*, volume 807 of *Lecture Notes in Computer Science*, pages 198–212. Springer, 1994.

[CCJW13] Lisi Chen, Gao Cong, Christian S. Jensen, and Dingming Wu. Spatial keyword query processing: An experimental evaluation. *PVLDB*, 6(3):217–228, 2013.

[CGK06] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. A primitive operator for similarity joins in data cleaning. In Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang, editors, *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 5. IEEE Computer Society, 2006.

[Chi94] Tzi-cker Chiueh. Content-based image indexing. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 582–593. Morgan Kaufmann, 1994.

[CJW09] Gao Cong, Christian S. Jensen, and Dingming Wu. Efficient retrieval of the top-k most relevant spatial web objects. *PVLDB*, 2(1):337–348, 2009.

[CMN01] Edgar Chávez, José L. Marroquín, and Gonzalo Navarro. Fixed queries array: A fast and economical data structure for proximity searching. *Multimedia Tools Appl.*, 14(2):113–135, 2001.

[CN05] Edgar Chávez and Gonzalo Navarro. A compact space decomposition for effective metric indexing. *Pattern Recognition Letters*, 26(9):1363–1376, 2005.

[CNBM01] Edgar Chávez, Gonzalo Navarro, Ricardo A. Baeza-Yates, and José L. Marroquín. Searching in metric spaces. *ACM Comput. Surv.*, 33(3):273–321, 2001.

[CPZ97] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 426–435. Morgan Kaufmann, 1997.

[CSD] Apache Casandra. `http://cassandra.apache.org`. Accessed: 26.03.2019.

[DBL] DBLP dataset. `https://www.cs.sfu.ca/~jnwang/projects/adapt/`. Accessed: 26.03.2018.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In Eric A. Brewer and Peter Chen,

editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150. USENIX Association, 2004.

[DLH+14] Dong Deng, Guoliang Li, Shuang Hao, Jiannan Wang, and Jianhua Feng. Massjoin: A mapreduce-based method for scalable string similarity joins. In Isabel F. Cruz, Elena Ferrari, Yufei Tao, Elisa Bertino, and Goce Trajcevski, editors, *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 340–351. IEEE Computer Society, 2014.

[dSTM18] Jéssica Andressa de Souza, Agma J. M. Traina, and Sebastian Michel. Class-constraint similarity queries. In Hisham M. Haddad, Roger L. Wainwright, and Richard Chbeir, editors, *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*, pages 549–556. ACM, 2018.

[FAB+18] Fabian Fier, Nikolaus Augsten, Panagiotis Bouros, Ulf Leser, and Johann-Christoph Freytag. Set similarity joins on mapreduce: An experimental survey. *PVLDB*, 11(10):1110–1122, 2018.

[FGT92] Philippe Flajolet, Danièle Gardy, and Loÿs Thimonier. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Discrete Applied Mathematics*, 39(3):207–229, 1992.

[FKS03] Ronald Fagin, Ravi Kumar, and D. Sivakumar. Comparing top k lists. *SIAM J. Discrete Math.*, 17(1):134–160, 2003.

[FLN03] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.

[GIJ+01] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Approximate string joins in a database (almost) for free. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 491–500, 2001.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In Beatrice Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, pages 47–57. ACM Press, 1984.

[Ham50] Richard W Hamming. Error detecting and error correcting codes. *The Bell system technical journal*, 29(2):147–160, 1950.

[HBS] Apache HBASE. `https://hbase.apache.org`. Accessed: 26.03.2019.

[HDP] Apache Hadoop. `https://hadoop.apache.org`. Accessed: 26.03.2019.

[HM03] Sven Helmer and Guido Moerkotte. A performance study of four index structures for set-valued attributes of low cardinality. *VLDB J.*, 12(3):244–261, 2003.

[HS99] Gísli R. Hjaltason and Hanan Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.

[IMS13] Evica Ilieva, Sebastian Michel, and Aleksandar Stupar. The essence of knowledge (bases) through entity rankings. In Qi He, Arun Iyengar, Wolfgang Nejdl, Jian Pei, and Rajeev Rastogi, editors, *22nd ACM International Conference on Information and Knowledge Management, CIKM'13, San Francisco, CA, USA, October 27 - November 1, 2013*, pages 1537–1540. ACM, 2013.

[Jac12] Paul Jaccard. The distribution of the flora in the alpine zone. 1. *New phytologist*, 11(2):37–50, 1912.

[JLFL14] Yu Jiang, Guoliang Li, Jianhua Feng, and Wen-Syan Li. String similarity joins: An experimental evaluation. *PVLDB*, 7(8):625–636, 2014.

[JOT+05] H. V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. idistance: An adaptive b^+-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, 30(2):364–397, 2005.

[JP08] Christopher B. Jones and Ross S. Purves. Geographical information retrieval. *International Journal of Geographical Information Science*, 22(3):219–228, 2008.

[JS08] Edwin H. Jacox and Hanan Samet. Metric space similarity joins. *ACM Trans. Database Syst.*, 33(2), 2008.

[KKWZ15] Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. *Learning Spark: Lightning-Fast Big Data Analytics*. 1st edition, 2015.

[KM83] Iraj Kalantari and Gerard McDonald. A data structure and an algorithm for the nearest point problem. *IEEE Trans. Software Eng.*, 9(5):631–634, 1983.

[KW17] Holden Karau and Rachel Warren. *High performance Spark: best practices for scaling and optimizing Apache Spark.* " O'Reilly Media, Inc.", 2017.

[LDWF11] Guoliang Li, Dong Deng, Jiannan Wang, and Jianhua Feng. PASS-JOIN: A partition-based method for similarity joins. *PVLDB*, 5(3):253–264, 2011.

[MAB16] Willi Mann, Nikolaus Augsten, and Panagiotis Bouros. An empirical evaluation of set similarity join techniques. *PVLDB*, 9(9):636–647, 2016.

[Mam03] Nikos Mamoulis. Efficient processing of joins on set-valued attributes. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 157–168. ACM, 2003.

[MAM15] Evica Milchevski, Avishek Anand, and Sebastian Michel. The sweet spot between inverted indices and metric-space indexing for top-k-list similarity search. In Gustavo Alonso, Floris Geerts, Lucian Popa, Pablo Barceló, Jens Teubner, Martín Ugarte, Jan Van den Bussche, and Jan Paredaens, editors, *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015.*, pages 253–264. OpenProceedings.org, 2015.

[MDB] MongoDB. `https://www.mongodb.com/`. Accessed: 26.03.2019.

[MM16] Evica Milchevski and Sebastian Michel. Quantifying likelihood of change through update propagation across top-k rankings. In Pitoura et al. [PMK+16], pages 660–661.

[MNM18] Evica Milchevski, Fabian Neffgen, and Sebastian Michel. Processing class-constraint K-NN queries with MISP. In *Proceedings of the 21st International Workshop on the Web and Databases, Houston, TX, USA, June 10, 2018*, pages 2:1–2:6. ACM, 2018.

[MRS10] Christopher Manning, Prabhakar Raghavan, and Hinrich Schütze. Introduction to information retrieval. *Natural Language Engineering*, 16(1):100–103, 2010.

[MTI] M-tree implementation. `https://github.com/erdavila/M-Tree`. Accessed: 01.12.2013.

[NVZ92] Hartmut Noltemeier, Knut Verbarg, and Christian Zirkelbach. Monotonous bisector* trees - A tool for efficient partitioning of complex scenes of geometric objects. In Burkhard Monien and Thomas Ottmann, editors, *Data Structures and Efficient Algorithms, Final Report on the DFG Special Joint Initiative*, volume 594 of *Lecture Notes in Computer Science*, pages 186–203. Springer, 1992.

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

[ORK] ORKU dataset. `http://ssjoin.dbresearch.uni-salzburg.at/datasets.html`. Accessed: 01.12.2018.

[O'S06] Mícheál O'Searcoid. *Metric spaces*. Springer Science & Business Media, 2006.

[Pal18] Koninika Pal. *Mining and Querying Ranked Entitites*. PhD thesis, Kaiserslautern University of Technology, Germany, 2018.

[PM16] Kiril Panev and Sebastian Michel. Reverse engineering top-k database queries with PALEO. In Pitoura et al. [PMK+16], pages 113–124.

[PMK+16] Evaggelia Pitoura, Sofian Maabout, Georgia Koutrika, Amélie Marian, Letizia Tanca, Ioana Manolescu, and Kostas Stefanidis, editors. *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016*. OpenProceedings.org, 2016.

[PMM16] Kiril Panev, Evica Milchevski, and Sebastian Michel. Computing similar entity rankings via reverse engineering of top-k database queries. In *32nd IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2016, Helsinki, Finland, May 16-20, 2016*, pages 181–188. IEEE Computer Society, 2016.

[PMMP16] Kiril Panev, Sebastian Michel, Evica Milchevski, and Koninika Pal. Exploring databases via reverse engineering ranking queries with PALEO. *PVLDB*, 9(13):1525–1528, 2016.

[RLS+17] Chuitian Rong, Chunbin Lin, Yasin N. Silva, Jianguo Wang, Wei Lu, and Xiaoyong Du. Fast and scalable distributed set similarity joins for big data analytics. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 1059–1070. IEEE Computer Society, 2017.

[RLW+13] Chuitian Rong, Wei Lu, Xiaoli Wang, Xiaoyong Du, Yueguo Chen, and Anthony K. H. Tung. Efficient and scalable processing of string similarity join. *IEEE Trans. Knowl. Data Eng.*, 25(10):2217–2230, 2013.

[Sam84] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, 1984.

[Sam06] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[SHC14] Akash Das Sarma, Yeye He, and Surajit Chaudhuri. Clusterjoin: A similarity joins framework using map-reduce. *PVLDB*, 7(12):1059–1070, 2014.

[SK04] Sunita Sarawagi and Alok Kirpal. Efficient set joins on similarity predicates. In Gerhard Weikum, Arnd Christian König, and Stefan Deßloch, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 743–754. ACM, 2004.

[SKRC10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In Mohammed G. Khatib, Xubin He, and Michael Factor, editors, *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*, pages 1–10. IEEE Computer Society, 2010.

[SPK] Apache Spark. `https://spark.apache.org`. Accessed: 26.03.2019.

[SQM+15] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Özcan. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *PVLDB*, 8(13):2110–2121, 2015.

[TBV+11] Manolis Terrovitis, Panagiotis Bouros, Panos Vassiliadis, Timos K. Sellis, and Nikos Mamoulis. Efficient answering of set containment queries for skewed item distributions. In Anastasia Ailamaki, Sihem Amer-Yahia, Jignesh M. Patel, Tore Risch, Pierre Senellart, and Julia Stoyanovich, editors, *EDBT 2011, 14th International Conference on Extending Database Technology, Uppsala, Sweden, March 21-24, 2011, Proceedings*, pages 225–236. ACM, 2011.

[TPVS06] Manolis Terrovitis, Spyros Passas, Panos Vassiliadis, and Timos K. Sellis. A combination of trie-trees and inverted files for the indexing of set-valued attributes. In Philip S. Yu, Vassilis J. Tsotras, Edward A. Fox, and Bing Liu, editors, *Proceedings of the 2006 ACM CIKM International Conference on Information and Knowledge Management, Arlington, Virginia, USA, November 6-11, 2006*, pages 728–737. ACM, 2006.

[Uhl91] Jeffrey K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Inf. Process. Lett.*, 40(4):175–179, 1991.

[VCL10] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using mapreduce. In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 495–506. ACM, 2010.

[WLF12] Jiannan Wang, Guoliang Li, and Jianhua Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 85–96. ACM, 2012.

[WMP13] Ye Wang, Ahmed Metwally, and Srinivasan Parthasarathy. Scalable all-pairs similarity search in metric spaces. In Inderjit S. Dhillon, Yehuda Koren, Rayid Ghani, Ted E. Senator, Paul Bradley, Rajesh Parekh, Jingrui He, Robert L. Grossman, and Ramasamy Uthurusamy, editors, *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013*, pages 829–837. ACM, 2013.

[WQL+17] Xubo Wang, Lu Qin, Xuemin Lin, Ying Zhang, and Lijun Chang. Leveraging set relations in exact set similarity join. *PVLDB*, 10(9):925–936, 2017.

[XWLY08] Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. Efficient similarity joins for near duplicate detection. In Jinpeng Huai, Robin Chen, Hsiao-Wuen Hon, Yunhao Liu, Wei-Ying Ma, Andrew Tomkins, and Xiaodong Zhang, editors, *Proceedings of the 17th International Conference on World Wide Web, WWW 2008, Beijing, China, April 21-25, 2008*, pages 131–140. ACM, 2008.

[Yia93] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In Vijaya Ramachandran, editor, *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA.*, pages 311–321. ACM/SIAM, 1993.

[ZADB06] Pavel Zezula, Giuseppe Amato, Vlastislav Dohnal, and Michal Batko. *Similarity Search - The Metric Space Approach*, volume 32 of *Advances in Database Systems*. Kluwer, 2006.

[ZSAR98] Pavel Zezula, Pasquale Savino, Giuseppe Amato, and Fausto Rabitti. Approximate similarity retrieval with m-trees. *VLDB J.*, 7(4):275–293, 1998.

[ZXW+05] Yinghua Zhou, Xing Xie, Chuang Wang, Yuchang Gong, and Wei-Ying Ma. Hybrid index structures for location-based web search. In Otthein Herzog, Hans-Jörg Schek, Norbert Fuhr, Abdur Chowdhury, and Wilfried Teiken, editors, *Proceedings of the 2005 ACM CIKM International Conference on Information*

and Knowledge Management, Bremen, Germany, October 31 - November 5, 2005, pages 155–162. ACM, 2005.

Evica Milchevski

Education

11/2013 – 08/2019 **PhD in Computer Science**, *TU Kaiserslautern (11/2014 – 08/2019), Germany,*
International Max Planck Research School for Computer Science (IMPRS-CS Scholarship), Saarland University (11/2013 – 10/2014), Germany.
Thesis: Similarity Search Algorithms over Top-k Rankings and Class-Constrained Objects.

10/2011 – 09/2013 **Masters in Computer Science**, *Saarland University, Saarbrücken, Germany*, **GPA:** 1.1 out of 1.0.
Thesis: Analyzing and Creating Top-K Entity Rankings.
Grade of Master's Thesis: 1.0 out of 1.0

10/2005 – 03/2010 **Bachelor of Computer Science**, *Faculty of Natural Sciences and Mathematics - Institute of Informatics, University "St. Cyril and Methodius", Skopje, Macedonia*, **GPA:** 9.28 out of 10.
Thesis: Horizontal and Vertical Partitioning of Data in Databases.
Grade of Bachelor's Thesis: 10 out of 10

Work Experience

12/2012 – 09/2013 **Part Time Research Assistant (HiWi)**, *DFKI GmbH, Germany.*
Main activities and responsibilities:
- Developing Eclipse plugins

03/2009 – 09/2011 **Junior Analyst/Developer**, *T-Mobile Macedonia, Skopje, Macedonia.*
Main activities and responsibilities:
- Developing ETL procedures utilizing PL/SQL
- Administration of existing ETL processes
- Database design for new systems

Languages

Macedonian	Native
English	Level C1 by CEFR
German	Level B2 by CEFR