Distributed Similarity Joins over Top-K Rankings*

Evica Milchevski
TU Kaiserslautern (TUK)
Kaiserslautern, Germany
milchevski@cs.uni-kl.de

Sebastian Michel
TU Kaiserslautern (TUK)
Kaiserslautern, Germany
michel@cs.uni-kl.de

ABSTRACT

Top-k rankings are a commonly used technique to summarize the most important entities of a specific domain. In this work, we focus on efficiently solving the problem of similarity joins for top-k rankings, for instance, for determining users with similar affinity, or for grouping related queries in search engines, based on their results. We put forward a novel algorithmic multi-step solution, realized via Apache Spark, harnessing mathematical properties of the distance function and a preceding near-duplicate detection phase, for search-space pruning. We further show how existing state-of-the-art algorithms for set similarity joins can be adapted to handle top-k rankings and how the data partitioning internals of Spark can be used to enable efficient data processing. The experimental study over standard benchmark datasets reveals that the proposed solution outperforms the state-of-the-art competitor by up to a factor of 5, despite involving additional stages of processing.

1 INTRODUCTION

Similarity joins have been a popular research topic in the database community for more than a decade now. Previous research in this topic is concerned with solving the problem of similarity joins for sets [7, 11, 25, 28], strings [13] or the more general problem of finding the similar objects in metric space [12]. Many distributed solutions, developed for the MapReduce framework, have also been proposed. Recently, Fier et al. [10] summarized and compared these distributed solutions. In this paper, we specifically focus on solving the problem of similarity joins for top-k rankings. Top-k rankings are a very popular and widely used technique to summarize the most relevant entities from a certain domain. Fast and efficient solutions for similarity joins of top-k rankings is of great value in many contexts. For instance, the case of query suggestion or expansion in search engines based on finding similar queries by comparing their result lists, in a dating portal where we can use the preferences and affinities of users, presented in a form of top-k lists, for matchmaking, or in the case of recommender systems, where the similarity between the top sold (liked, favored) items for different clients can help in recommending products. For instance, consider Table 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.

Spearman's Footrule distance is used as a distance measure for comparing two top-k lists. Fagin et al. [9] show that there is a Spearman's Footrule adaptation for top-k rankings that is a metric. This immediately entails the use of existing metric space similarity join approaches. On the other hand, rankings can be considered as plain sets and accordingly indexed using inverted indices, that keep for each item a list of rankings where this item appears. Thus, many of the distributed algorithms that solve the problem of similarity joins for sets can be applied for top-k rankings. The best performing one, according to a recent study [10], is the algorithm proposed by Vernica et al. [24], based on the principle of prefix filtering. This approach, as shown in the study [10], also outperforms existing metric space similarity join approaches. Furthermore, Fier et al. [10] showed that the existing distributed solutions in MapReduce do not scale well, and propose that Apache Spark is used as a platform for developing new alternative solutions. In this paper, we specifically focus on studying an efficient and scalable top-k rankings similarity joins using Apache Spark [4].

	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 1: Dating portal users' favorite movies

We propose a novel approach implemented in Apache Spark that is better tailored to the properties of this platform. In contrast to MapReduce, where each stage is composed from 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. As Spearman's Footrule adaptation for top-k rankings is a metric, the algorithm uses the triangle inequality to reduce the number of candidate pairs generated. Very similar rankings are clustered together, and then, only the cluster representatives are joined, reducing the size od the data processed, and thus, finding more efficiently the join results. Through a detailed experimental study we show that our algorithm outperforms the competitor, especially for larger values of the similarity threshold θ.

1.1 Problem Statement and Setup

As **input** we are provided with a dataset $\mathcal{T}$ of rankings τ_i (Table 2). 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 different choices of k on the join performance [1]. The considered rankings do not contain any duplicate items.

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 to it. Without

*This work has been supported by the German Research Foundation (DFG) under grant MI 1794/1-1/2.

[1] Working with fixed-length rankings gives better insights into the difference of performance of the algorithms. For handling variable-length rankings, only the length boundaries for the Footrule distance, given a distance threshold, need to be computed .

ranking id	ranking content
τ_1	[2, 5, 4, 3, 1]
τ_2	[1, 4, 5, 9, 0]
τ_3	[0, 8, 5, 7, 3]

Table 2: Sample dataset $\mathcal{T}$ of top-5 rankings (items are represented by their ids).

loss of generality, in the remainder of the paper, we assume that items are also represented by their ids. The rank of an item i in a ranking τ is given as $\tau(i)$.

A distance function d quantifies the distance between two rankings—the larger the distance the less similar the rankings are. *Given a dataset of top-k rankings $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ and a distance threshold θ we want to 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, we focus on the computation of Spearman's Footrule distance, but the proposed algorithm can be applied on any distance measure that satisfies the triangle inequality. Spearman's Footrule distance is computed as a sum over the difference in position of the items in the two rankings, i.e., $F(\tau, \sigma) = \sum_{i \in D_\tau \cup D_\sigma} |\tau(i) - \sigma(i)|$. An artificial rank l for items not contained in a ranking, i.e., $\tau(i) = l$ if $i \notin D_\tau$, is considered. Consider the rankings τ_1 and τ_2, in Table 2. For a rank $l = 6$ for not-contained items, we obtain $F(\tau_1, \tau_2) = 4+1+1+5+2+2+1 = 16$. A more detailed introduction to rankings, specifically top-k rankings, distance functions, and how to handle items i that are not in a ranking τ is described in Section 3.

1.2 Contributions and Outline

The contributions of our work can be summarized as follows.

- We adapt existing set-based similarity join algorithm to the problem of top-k rankings. We furthermore implement and adapt this algorithm 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 paper is structured as follows. Section 2 discusses related work and, in Section 3, we present background on top-k rankings, a state-of-the-art distributed set similarity join algorithm, and Apache Spark. In Section 4, we describe how existing set-based similarity join algorithm is adapted to top-k rankings. The clustering algorithm and its components is introduced in Section 5. Section 6 proposes a Spark-based repartitioning technique. We experimentally evaluate the presented approaches in Section 7. Finally, we give the conclusion in Section 8.

2 RELATED WORK

To the best of our knowledge, the problem of similarity join for top-k rankings has not been addressed so far. As top-k rankings can also be seen as sets, we mainly focus here on explaining set-based similarity joins.

In-memory similarity join approaches: There is an ample work on computing the similarity join for sets or strings. Mann et al. summarize and compare the in-memory based approaches in [16]. 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 and applying additional filters that prune dissimilar pairs. In the verification phase the candidates are verified by computing their true similarity score. The prefix-filtering approach, initially proposed by Chaudhuri et al. [7] is the most well know algorithm for finding the similar pairs. It works by first sorting all records in the dataset in the same canonical order and then indexing only a prefix of the record with an inverted index. The size of the prefix depends on the threshold and distance, i.e., the similarity measure used. The records are usually sorted by the ascending frequency of the elements in the sets. There are many works [5, 6, 21, 25, 28] that propose improvements over the initial prefix-filtering algorithm, by introducing additional position or length-based filters, reducing the size of the prefix, introducing variable length prefixes, grouping based on the prefix, etc. Recently Wang et al. [26], motivated by the conclusions presented in [16], 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. Bouros et al. [6] propose an approach for spatio-textual similarity joins. They describe algorithms that partitions and filters the data based on the spatial (Euclidean) distance between the points in the dataset, and, in addition, they extend the prefix-filtering method by introducing grouping based on the prefix of the textual data. The grouping based on the prefix of the textual data is orthogonal to our presented approach, i.e., it can be applied in our approach in addition to (instead of) the VJ algorithm. Their method for distance based partitioning of the data space resembles our idea for clustering based on the distance threshold. However, the solution proposed in [6] works specifically for two dimensional data. Top-k rankings, on the other hand, can be interpreted as points in multidimensional space, where the dimension is determined by the size of the rankings, usually 10 or larger. In addition, our clustering approach is more general, and works for any data in metric space.

MapReduce-based similarity join approaches: To handle larger datasets, many distributed solutions for similarity join of sets have also been proposed. Recently, Fier et al. [10] summarized and compared the MapReduce-based similarity join solutions. Vernica et al. [24] present a distributed solution, referred to as VJ, based on the well known prefix filtering method. Since we use this algorithm in our implementation, we describe it in more details in Section 3. The *V-SMART* algorithm [17] adopts a different idea, by computing the ingredients of the similarity measure in a distributed manner, which are later joined to compute the final results. It works in two phases, a joining phase and a similarity phase. In the first phase, the joining phase, the partial results for each set is computed and joined to all the elements in the sets. In the similarity phase, the algorithm takes as input the output from the first phase, builds an inverted index, and then, while traversing the posting lists for each element s_i, emits pairs of sets together with information needed to compute the

final similarity. Deng et al. present MassJoin [8]. This approach is based on PassJoin [15], a main memory method for string similarity joins. The idea behind their method is to generate signatures for the sets $r \in R$, and then for each signature of r they generate signatures for $s \in S$. In order for s and r to be similar, they should share at least one signature. Additional filters are applied to reduce the number of candidate pairs generated. Rong et al. present FS-Join [20]. 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 vertically partitioned, by dividing each set into segments and then partitioning the data according to the segments. Interestingly, Fier et al. [10] came to the conclusion that the approach proposed by Vernica et al. [24] outperforms the other approaches in most scenarios. Therefore, in this work we compare our approach to the one presented in [24]. Distributed metric space approaches have also been proposed [22, 27]. Wang et al. [27] for a dataset D, partition 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. Further, they define inner and outer sets of a partition and based on that they decide the data distribution. The proposed MapReduce algorithm consists of two main stages, partitioning the data, and, in the second stage, computing the similarity join. Sarma et al. [22] propose a MapReduce method that works very well for very small distance thresholds. In fact, they evaluate their approach using only threshold values up to 0.1. The novelty in their work is that they apply several filtering techniques, both distance specific and not, which lead to having tighter partitions, and thus, fewer comparisons.

In prior work [18], we solve the problem of answering similarity range queries over top-k rankings. There, in addition to an algorithm based to the prefix-filtering framework, we also presented a so-called coarse index, that combines an inverted index with a metric index structure to reduce the number of distance function computations.

3 PRELIMINARIES

Complete rankings are considered to be permutations over a fixed domain $\mathcal{D}$. We follow the notation by Fagin et al. [9] 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 $\sigma(i) < \sigma(j)$. We consider incomplete rankings, called top-k lists in [9]. 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., $D_{\tau_1} \neq D_{\tau_2}$.

Pairwise similar rankings can be retrieved by means of distance functions, like Kendall's Tau or Spearman's Footrule distance. In this work we use Spearman's Footrule adaptation for top-k lists proposed in [9]. Spearman's Footrule distance is computed as a sum over the difference in position of the two rankings, i.e., $F(\tau, \sigma) = \sum_{e \in D_\tau \cup D_\sigma} |\tau(i) - \sigma(i)|$. An artificial rank l for items not contained in a ranking, i.e., $\tau(i) = l$ if $i \notin D_\tau$ is considered.

In this work, we assume that $\tau(i)$ takes values from 0 to $k-1$ (instead of 1 to k), and we fix the value of l to k as suggested in [9]. It is clear that this does not affect our algorithms. We further consider only rankings of same size k, thus the largest possible value of the Footrule distance is $k * (k+1)$ and occurs if two disjoint rankings are compared. The smallest distance is 0, for the compared rankings are identical. In the rest of the paper, for ease of presentation, we use normalized values for the Footrule distance and the threshold values, ranging from 0 to 1.

3.1 Vernica Join (VJ) Algorithm

According to a recent experimental study [10], the VJ algorithm outperforms other distributed similarity join algorithms in most cases. The algorithm is implemented in MapReduce and is based on the well known prefix filtering method. It works in several phases, each representing one map reduce job. For each phase, the authors propose several variations, however, here we describe the version which, according to their evaluation, showed the best performance.

In the first phase, all records are read and the tokens in the universe are sorted according to the increasing frequency of appearance in the sets. Then, in the next phase, the sorted tokens are loaded into the memory of the mappers and used for sorting the sets into a canonical form. Then the mappers emit a composite key consisting of the token and the size of the set, plus the whole set as value, but only those elements that belong to the prefix. For grouping the records in the reducers, only the token is used, while the size of the set is used for sorting the records by size. The latter allows utilizing size-based filters. At the reducers, for all the rankings that share at least one element together, the PPJoin+ algorithm [28] is used, to find the similar rankings. In the final phase, duplicate pairs must be removed, since the same pair can be generated at several machines.

3.2 Apache Spark

Apache Spark [4] is a general purpose platform that enables easy and fast development and execution of distributed applications. It can be considered successor of MapReduce, as it provides similar capabilities with generally better performance. Additionally, several other functionalities are provided and many libraries are built on top of its core. The parallelization of applications is easier when using Apache Spark due to the notions of RDD, *transformations* and *actions* used in the platform. RDDs are collections of elements distributed across the nodes of a cluster [14]. Once created, they are then partitioned among the available nodes of a cluster. This way, each node handles a subset of the input. 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 materialized. This allows Spark to optimize job execution, by analyzing and grouping the transformations that are performed over the RDDs.

Another important characteristic of Apache Spark is its ability to execute iterative processes, using the main memory of the nodes, in order to reduce disk I/O, thus, reducing the overall execution time of the application, leading to superior performance over MapReduce, as shown by Shi et al. [23].

4 A VJ-STYLE ALGORITHM FOR TOP-K RANKINGS

To find all pairs of similar top-k rankings for a given set of rankings $\mathcal{T}$ and a threshold θ, we can use the Vernica Join (VJ) algorithm. However, in order to be able to apply it on top-k rankings,

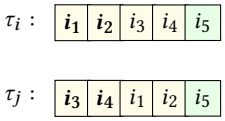


Figure 1: Example rankings with $k = 5$ and $p = 2$ with maximum Footrule distance $F(\tau_i, \tau_j) = 8$.

we need to derive the prefix size p of top-k rankings, when Spearman's Footrule distance is used to compare them[2]. There are two ways for computing the prefix size of top-k rankings, one considering the overlap of the rankings, and the other, considering their position. The latter provides slightly tighter prefix sizes than the first. However, the former allows more freedom in choosing the items in the prefix. In prior work [18], we found the minimum overlap between two rankings τ_i and τ_j, such that $F(\tau_j, \tau_j) = \theta$, as $\omega = \lfloor 0.5 * (1 + 2 * k - \sqrt{1 + 4 * \theta}) \rfloor$ and the prefix size as $p = k - \omega + 1$. We refer to this as **prefix based on overlap**. In addition, we define an ordered prefix, p_o, as:

Lemma 4.1. *For a given Spearman's Footrule distance threshold θ and a ranking length k, the* **ordered prefix** *p_o of the top-k rankings is given by the best ranked:*

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

items of the rankings.

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 overlapping, $L(p, k)$, is when the items are overlapping in the rankings, i.e., $D_{\tau_i} = D_{\tau_j}$, 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. 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 $\frac{(p*2)^2}{2}$. An example of such rankings,where $p = 2$, are the rankings τ_i and τ_j, shown in Figure 1. These rankings have the same domain, i.e., $D_{\tau_i} = D_{\tau_j} = \{i_1, i_2, i_3, i_4, i_5\}$, however, when looking only the first p items, written in bold, they have no overlap. The Footrule distance between them is $F(\tau_i, \tau_j) = 8$, the lowest that they can have when the first p items are not shared.

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. [3] □

Given the Footrule distance threshold θ, we now describe the VJ algorithm for 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 rankings and 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 the prefix size based on the overlap between the rankings should be used. 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, thus rankings are transformed to arrays of $(i_{id}, \tau(i))$ pairs. 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 index their prefixes using an inverted index. In addition, based on our previous work [19] we apply a position filter in order to filter out more candidate pairs. In [19], 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 $\frac{k*(k+1)*\theta}{2}$. That means, 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. Note that, since we work with rankings of same size k, filtering based on the length of the rankings is not applicable. Finally, before writing the final result, we remove the duplicate pairs.

4.1 Improved Memory Usage

Previous distributed approaches for similarity joins were designed and implemented in MapReduce. Spark, as a successor of MapReduce, has different characteristics than MapReduce, and thus existing approaches can be adapted to the computational properties of Spark in order to improve their performance.

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, which means that garbage collection can easily cause performance issues for Spark jobs. Thus, keeping objects in the memory of executors is not recommended, since it can lead to crashes or performance degradation when dealing with 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 [24] 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 the VJ algorithm can lead to having

[2]To use another distance measure with a prefix-filtering based algorithm, these bounds will need to be recomputed. However, our approach is flexible and any other algorithm can be used.

[3]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 we leave it as future work, since 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 $\theta \leq 0.4$. $\theta = \frac{k^2}{2}$ is around 0.45 when normalized, depending on the value of k.

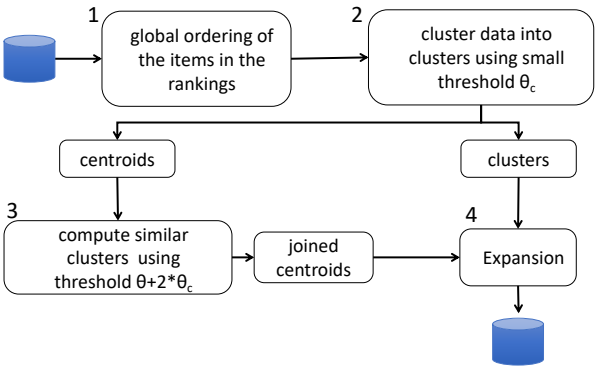


Figure 2: Overall architecture. The algorithm has four main phases: ordering, clustering, joining and expansion.

both 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 walk through the rankings in a nested loop fashion. For each ordered pair of rankings,(τ_i, τ_j), where $\tau_i \prec \tau_j$ 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.

5 APPROACH

Driven by the idea that similar rankings should have similar result sets, we propose a novel approach having a pre-processing step, where very similar rankings are grouped. Then, only one representative ranking from the clusters, called centroid, is considered in the next similarity join phase. The idea is that by doing this, the number of records being joined is reduced and, thus, the execution time of the main joining phase is reduced too—which is actually the most expensive part of similarity join algorithms. Since Spark is suitable for iterative processing, adding an additional phase should be acceptable. However, this pre-processing phase should be very efficient, such that we do not end up with having a higher overhead than real benefit. Another key observation is that the Footrule distance is a metric and, thus, 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 an approach consisting of four main phases: Ordering, Clustering, Joining, and Expansion, depicted in Figure 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.

Clustering: The second phase of our approach is forming clusters, such that similar rankings will belong to the same cluster, C_i. First, a similarity join algorithm is executed, to find the similar rankings that need to be grouped together. Our experiments revealed that VJ is the most efficient one to be used here, which supports the findings by Fier et al. [10]. In principle, however, any similarity algorithm could be employed at this stage. Then, clusters are formed such that the pairwise distance between each member of the cluster and its representative is at most θ_c. We will refer to θ_c as the *clustering threshold*. In contrast to other similarity join algorithms in metric space, where clusters have different radius, the radius of all clusters formed by our approach is bounded by the clustering threshold, θ_c. We write $\tau_i \prec c_i$ to denote that ranking τ_i belongs to the cluster, 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, independently from the algorithm used in the clustering phase. Due to the aforementioned reason, we implement the VJ 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 results is computed, we first discuss the correctness of the proposed algorithm.

Lemma 5.1. *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 5.1 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 5.1 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.

5.1 Clustering

When forming the clusters the following points need to be considered: ***(i)*** To ensure correctness, the radius of all the clusters should be the same, i.e., for any ranking $\tau_i \in C_i$, represented by a ranking c_i, $d(\tau_i, c_i) \leq \theta_c$. ***(ii)*** The clustering method should be

very efficient, otherwise the cost of the clustering would overweight its benefit. ***(iii)*** 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 [22, 27], 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.

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, $\mathcal{T}$. 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, i.e., the one with a smaller id, 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 5.3. 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 3, the following clusters would be formed $C_1 = \{\tau_1, \tau_2, \tau_5\}$, $C_2 = \{\tau_3, \tau_4\}$ 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 $d(\tau_i, \tau_j) \leq 2 * \theta_c$, and thus, members of the same clusters can directly be written to disk as partial results, as long as $\theta_c * 2 < \theta$.

By creating the clusters in this way, all of the aforementioned requirements are satisfied. The radius of all the clusters is the same and both forming 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 overlap would negatively impact the performance of the clustering and the expansion phase.

As input to the next, joining phase, we union the set of centroids C_m that contains all centroids representatives of clusters with at least two members, i.e., $|C| \geq 2$ with the set of centroids C_s that represent the singleton clusters, i.e., $|C| = 1$. The set C_s is derived from the original dataset, by finding those rankings $\tau_i \in \mathcal{T}$ such that there is no other ranking $\tau_j \in \mathcal{T}$, such that,

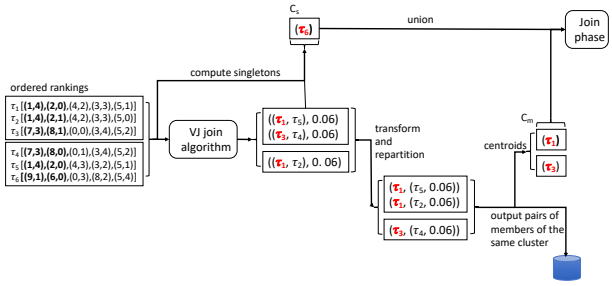


Figure 3: Example of how clusters are formed and centroids (marked with red) are chosen, where $\theta_c = 0.1$.

$d(\tau_i, \tau_i) \leq \theta_c$. An example of a singleton cluster in Figure 3 is $C_3 = \{\tau_6\}$.

Example 5.2. Figure 3 shows through an example the creation of the clusters, for $\theta_c = 0.1$. The items in the rankings $\tau_1, \ldots \tau_6$ have already been sorted by increasing order of their frequency. For instance, in τ_1 item 1 with position $\tau_1(1) = 4$ is placed on the first position, as it appears three times in the rankings (ties are arbitrarily broken). After running a similarity join algorithm with distance threshold $\theta_c = 0.1$ on these rankings, the pairs (τ_1, τ_5), (τ_1, τ_2) and (τ_3, τ_4). In the following step clusters $C_1 = \{\tau_1, \tau_2, \tau_5\}$, $C_2 = \{\tau_3, \tau_4\}$ with centroids τ_1 and τ_3, respectively are formed. Furthermore, the ranking τ_6 forms a singleton cluster since it does not belong to any of the formed clusters C_1, C_2.

5.2 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, as almost all similarity join algorithms, 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 $C \subseteq \mathcal{T}$, 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 5.1, 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 C we have many centroids which are representatives of singleton clusters. For these centroids, we can avoid unnecessary computation, by using a smaller threshold, without missing any true result. Lemma 5.3 defines this:

LEMMA 5.3. *Given join threshold θ and clustering threshold θ_c, and a set of centroids $C = C_m \cup C_s$, where C_s is the set of centroids that represent the singleton clusters and $C_m = C \setminus C_s$ 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:*

$$\{(c_i, c_j) \mid d(c_i, c_j) \leq \theta + 2 * \theta_c \;\; if \;\; c_i, c_j \in C_m\} \quad (1)$$

$$\{(c_i, c_j) \mid d(c_i, c_j) \leq \theta + \theta_c \;\; if \;\; c_i \in C_m \wedge c_j \in C_s \;\; or \; v.v.\} \quad (2)$$

$$\{(c_i, c_j) \mid d(c_i, c_j) \leq \theta \;\; if \;\; c_i, c_j \in C_s\} \quad (3)$$

method: **Centroids Join**

input: Dataset $C = C_m \cup C_s$, 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 ← transform_and_emit(C_m, C_s, p_m, p_s)
4 $\mathcal{R}$ ← compute_sim(grouped, k, θ, θ_c)

return $\mathcal{R}$

Algorithm 1: Joining of centroids based on the type of the centroid.

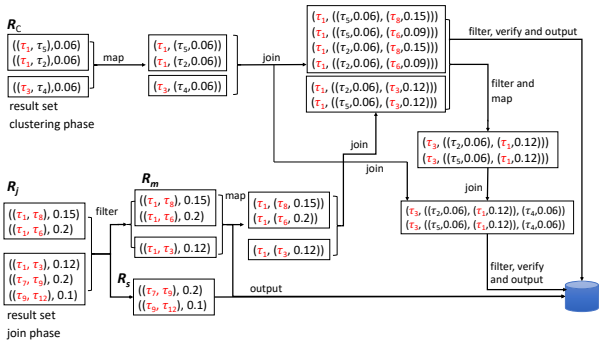


Figure 4: Example of computing the final result set using the result set from the joining phase and the clusters where $\theta_c = 0.1$ and $\theta = 0.2$. Cluster's centroids are marked with red.

Lemma 5.3 allows us to more efficiently join the centroids. It follows that, only for the centroids $c_m \in C_m$ 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 $c_s \in C_s$, 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 1.

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

5.3 Expansion

In the last phase, the final result set is generated. For this purpose, the results from the clustering phase, $\mathcal{R}_c$, 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 pairs 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.

Algorithm 2 outlines how the final result set is computed. First, the result set from the join phase, $\mathcal{R}_j$, is divided into two sets: $\mathcal{R}_s = \{(c_i, c_j) | c_i, c_j \in C_s \wedge d(c_i, c_j) \leq \theta\}$ 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 $\mathcal{R}_m$,

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 $\mathcal{R}_m, \mathcal{R}_s$ ← split($\mathcal{R}_m$)
2 $\mathcal{R}_p$ ← get_partial_results($\mathcal{R}_m$, θ, θ_c)
3 $\mathcal{R}_j, \mathcal{R}_m$ ← prepare_for_join($\mathcal{R}_j$, $\mathcal{R}_m$)
4 $\mathcal{R}_{R_j \bowtie R_m}$ ← join($C_m, \mathcal{R}_j, \mathcal{R}_j$)
5 $\mathcal{R}_{m,c}$ ← get_partial_results($\mathcal{R}_{R_j \bowtie R_m}$, θ, θ_c)
6 $\mathcal{R}_{R_j \bowtie R_m}$ ← prepare_for_join($\mathcal{R}_{R_j \bowtie R_m}$)
7 $\mathcal{R}_{(R_j \bowtie R_m) \bowtie R_j}$ ← join($\mathcal{R}_{R_j \bowtie R_m}, C$)
8 $\mathcal{R}_{m,c}, \mathcal{R}_{m,m}$ ← get_partial_results($\mathcal{R}_{(R_j \bowtie R_m) \bowtie R_j}$, θ, θ_c)

return distinct($\mathcal{R}_p \cup \mathcal{R}_s \cup \mathcal{R}_{m,c} \cup \mathcal{R}_{m,m}$)

Algorithm 2: Computation of the final result set.

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 $\mathcal{R}_{R_c \bowtie R_m}$. Then, $\mathcal{R}_{R_c \bowtie R_m}$ is used to generate the following result pairs:

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

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 $|d(c_i, c_j) - d(\tau_i, c_i)| \leq \theta$ and the remaining ones we can filter out since we can be certain that their distance is larger than θ.

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 disk. Before writing the results to disc, the duplicates are removed.

Example 5.4. Figure 4 illustrates the expansion through an example. As results from the clustering phase, we have tuples (τ_1, τ_5), (τ_1, τ_2), and (τ_3, τ_4). The centroids of these clusters are τ_1 and τ_3—the clusters are the same as in the aforementioned example. The join results $\mathcal{R}_j$ are split into $\mathcal{R}_s = \{(\tau_7, \tau_9), (\tau_9, \tau_{12})\}$, where none of the rankings in the pairs is a centroid and $\mathcal{R}_m = \{(\tau_1, \tau_8), (\tau_1, \tau_6), (\tau_1, \tau_3)\}$, where at least one ranking in the pair is a centroid. Pairs in $\mathcal{R}_s$ are directly written to disk. Tuples in $\mathcal{R}_c$ and $\mathcal{R}_m$ are transformed such that the centroids, τ_1 and τ_3 are placed as keys of the tuples. They are joined and $\mathcal{R}_m, c =$

$\{(\tau_5, \tau_8), (\tau_5, \tau_6), (\tau_2, \tau_8), (\tau_2, \tau_6), (\tau_2, \tau_3), (\tau_5, \tau_3)\}$ are verified. Then we take only those pairs in $\mathcal{R}_{R_c \bowtie R_m}$ where two rankings are centroids, in the example the last two elements of $\mathcal{R}_{R_c \bowtie R_m}$. For these, we switch the places of the centroids τ_1 and τ_3 so that the members of the second cluster could be joined with members of the first cluster, (τ_4, τ_5) and (τ_4, τ_2). These pairs need to be verified if their distance is smaller than θ.

6 REPARTITIONING USING JOINS

Naturally, the way data is distributed across partitions/machines greatly influences the performance of distributed 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 real-world data, items that appear very frequently cause very large partitions. This problem is partially solved by the prefix filtering framework, especially for smaller values of θ, since the most frequent items would not be included. However, as we increase the value of θ, the size of the prefix increases, leading to skewed a distribution of data across the partitions, thus, having few partitions that dominate the overall execution time of the algorithm.

To tackle 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 3 describes this 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, $\mathcal{I}_{<\delta}$. In Spark, this 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 than the partitioning threshold, we compute the 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 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 better handle the increased load due to data replication and to redistribute the working load equally among nodes, we partition by both the primary and secondary key, i.e., by both the item id and the randomly assigned number and increase the number of partitions.

Example 6.1. Figure 5 illustrates through an example the similarity join computation in case of repartitioning. In this example, the posting list for items $i_2, i_{10}, \ldots i_m$ have size larger than δ and thus are split into smaller partitions. For instance, the posting list for item i_2 is split into three smaller lists with keys $(i_1, 1)$, $(i_1, 5)$, and $(i_1, 9)$. In order to keep the correctness of the algorithms, in addition to generating the pairs for each of these lists, they are self joined, and an R-S join algorithm over the joined posting lists (with keys $(i_1, 1, 5)$, $(i_1, 1, 9)$, and $(i_1, 5, 9)$) is performed.

Choosing the Partitioning Threshold δ. In our experiments, we show that the performance of the algorithm does not significantly vary, when changing the partitioning threshold δ. However, the partitioning threshold still needs to be chosen carefully, such that

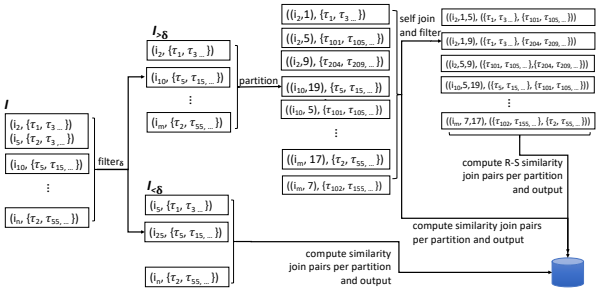


Figure 5: Example of repartitioning of the large partitions using a partitioning threshold δ.

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

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 3 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.

As a general guidance for choosing the value of the parameter δ an estimation for the size of the posting lists can be used. In our previous work on similarity search for top-k rankings [18], we devised a formula for estimating this:

$$E[\textit{index list length}] = \sum_i n * f(i; s, v')^2 \quad (4)$$

where n is the number of rankings indexed, and $f(i; s, v')$ is the frequency of the item at rank i, when the items follow a Zipf's distribution with skewness parameter s. v' is the distinct number of items in the prefix of the rankings.

7 EXPERIMENTS

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.

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 joins for sets and were also used for performing the experimental study for distributed similarity join algorithms [10]. Specifically, we use the DBLP [1] and ORKU [2] datasets. To transform the records of these dataset into top-k

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

Table 3: Spark parameters used for the evaluation

rankings, we simply take the first 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 [10], without the sorting of the records. 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 are 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 for a distributed setting, we also increase their size using the same method as in [10, 24], where the domain of the items remains the same, and the join result increases approximately linearly with the size of the dataset. We use suffix 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 adaptation of VJ to top-k rankings in Spark (VJ)
- The adaptation of VJ to 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)

Based on general recommendations for running Spark jobs, which suggest to not run 'tiny' or 'fat' executors, we assign 5 cores per executor. Then, based on the total number of cores and the available memory of the nodes in the cluster, we set the other execution parameters, reported in Table 3. The memory assigned to the executers also corresponds to the amount assigned to the reducers in a previous experimental study [10]. 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.

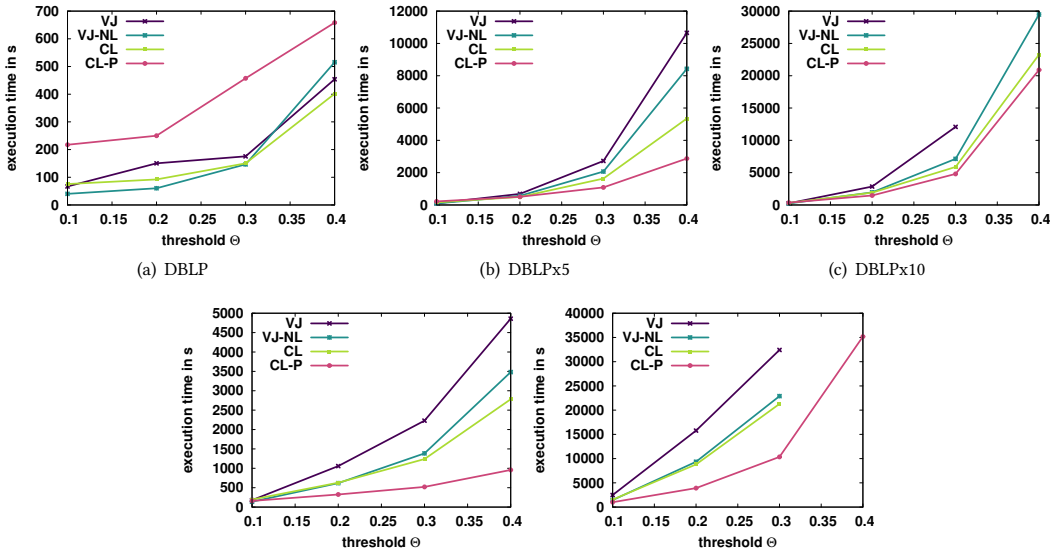
7.1 Results

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 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 too much affected by the skewness of the dataset. 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(b)) and ORKU (Figure 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. Additionally, 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(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(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(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(e)), for $\theta = 0.4$, only the CL-P algorithm finished under 10 hours. Similarly, for the DBLPx10 dataset (Figure 6(c)), the VJ algorithm did not finish under 10 hours.

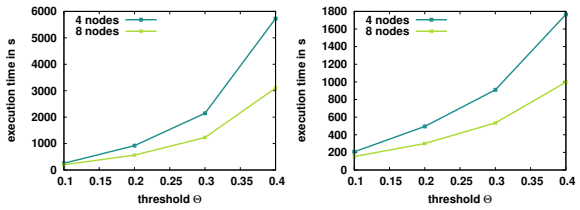
Scalability. To test the scalability of the proposed algorithm, we varied the number of nodes in our cluster. We executed the CL-P algorithm on a cluster with 4 nodes and with 8 nodes. For



(a) DBLP (b) DBLPx5 (c) DBLPx10

(d) ORKU (e) ORKUx5

Figure 6: Comparison of different algorithms when varying θ



(a) DBLPx5 (b) ORKU

Figure 7: Performance of CL-PL algorithm when varying the number of nodes in the cluster (DBLPx5 and ORKU).

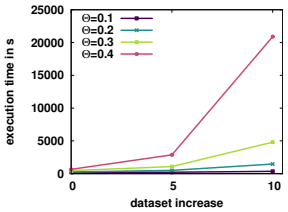


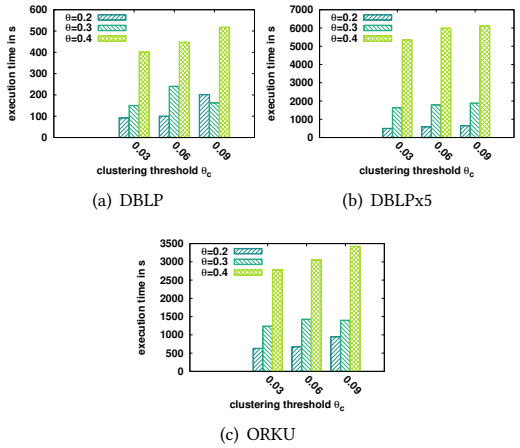
Figure 8: Performance of CL-P algorithm when varying the dataset size for the DBLP dataset.

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 3. Figure 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 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 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.

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 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.

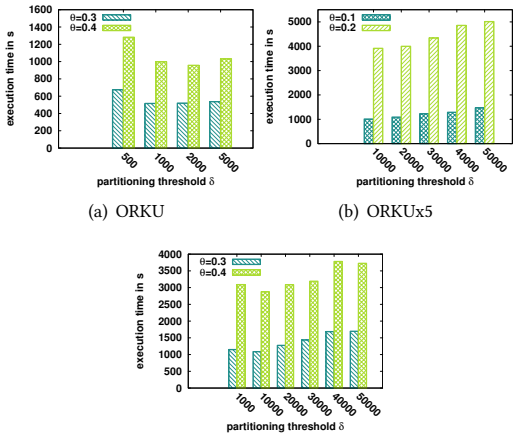
Effect of the Partitioning Threshold δ. The partitioning threshold δ is a parameter which decides which and how many posting lists need to be partitioned, and as such, it influences the performance of the CL-P algorithm. In Figure 10 we see the performance of the CL-P algorithm as the partitioning threshold 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(a), the DBLP dataset is small and does not benefit from the partitioning of the posting lists. For each dataset, we chose different varying



(a) DBLP (b) DBLPx5

(c) ORKU

Figure 9: Performance of CL algorithm when varying the clustering threshold θ_c



(a) ORKU (b) ORKUx5

(c) DBLPx5

Figure 10: Performance of CL-P algorithm when varying the partitioning threshold δ

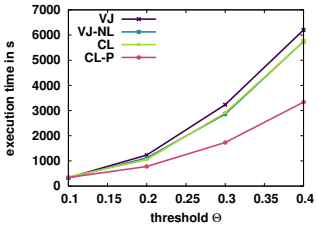
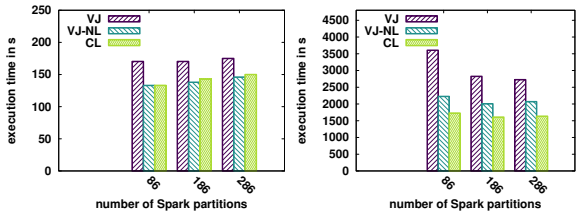


Figure 11: Performance of different algorithms for rankings of size 25 when varying the distance threshold θ (ORKU)

ranges for the partitioning threshold, since its value is directly dependent from the size of the dataset. For ORKU (Figure 10(a)) we vary δ from 500 to 5000, for ORKUx5 (Figure 10(b)) we vary δ from 10000 to 50000 and for DBLPx5 (Figure 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 10(c) and 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 10(b)). In Figure 6(a) we see that the performance of CL-P is not widely 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 [3] 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 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 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.

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 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 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



(a) DBLP (b) DBLPx5

Figure 12: Performance of VJ, VJ-NL and CL when varying the number of Spark partitions, $\theta = 0.3$ (DBLP and DBLPx5).

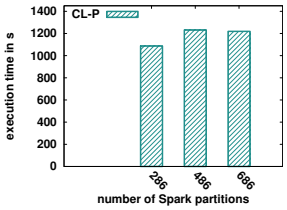


Figure 13: Performance of CL-P when varying the number of Spark partitions, $\delta = 10000$, $\theta = 0.3$ (DBLPx5).

286 to 686. Since here we additionally repartition the large partitioning into smaller ones, we believe that using a larger number of partitions is more appropriate for this approach However, as we can see from Figure 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 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, they 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, as discussed in Section 3. For choosing the clustering threshold, as a rule of thumb, we suggest choosing a very small value, namely to set θ_c to be smaller than 0.05. A drawback of our solution is that, since we rely on Spark joins, it can run out of memory, especially where the result set is large.

8 CONCLUSION AND OUTLOOK

In this paper, we addressed distributed similarity join processing techniques for a datasets of top-k rankings. As a distance for comparing the rankings, we specifically considered Spearman's Footrule adaptation to top-k rankings. The presented approach synthesizes existing state-of-the-art, set-based, 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. In the future, we plan to extend our approach to sets where the Jaccard distance is used as a distance measure.

REFERENCES

[1] DBLP Dataset. http://dbgroup.cs.tsinghua.edu.cn/wangjn/projects/adapt/. Accessed: 26.03.2018.
[2] ORKU Dataset. http://ssjoin.dbresearch.uni-salzburg.at/datasets.html. Accessed: 01.12.2018.
[3] F. Alvanaki, E. Ilieva, S. Michel, and A. Stupar. Interesting event detection through hall of fame rankings. In *DBSocial*, pages 7–12, 2013.
[4] Apache Spark [n.d.]. https://spark.apache.org. Accessed: 26.03.2019.
[5] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In WWW 2007, Banff, Alberta, Canada, pages 131–140..
[6] Panagiotis Bouros, Shen Ge, and Nikos Mamoulis. Spatio-textual similarity joins. *PVLDB* 6, 1 (2012), 1–12.
[7] S. Chaudhuri, V. Ganti, and R. Kaushik. A Primitive Operator for Similarity Joins in Data Cleaning. In ICDE 2006, Atlanta, GA, USA, page 5, 2006.
[8] D. Deng, G. Li, S. Hao, J. Wang, and J. Feng. MassJoin: A mapreduce-based method for scalable string similarity joins. In ICDE 2014, IL, USA, pages 340–351, 2014.
[9] R. Fagin, R. Kumar, and D. Sivakumar. Comparing Top k Lists. *SIAM J. Discrete Math.*, 17(1):134–160, 2003.
[10] F. Fier, N. Augsten, P. Bouros, U. Leser, and J. Freytag. Set Similarity Joins on MapReduce: An Experimental Survey. *PVLDB*, 11(10):1110–1122, 2018.
[11] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate String Joins in a Database (Almost) for Free. In *VLDB 2001, Roma, Italy*, pages 491–500.
[12] E. H. Jacox and H. Samet. Metric space similarity joins. *ACM Trans. Database Syst.*, 33(2), 2008.
[13] Y. Jiang, G. Li, J. Feng, and W. Li. String Similarity Joins: An Experimental Evaluation. *PVLDB*, 7(8):625–636, 2014.
[14] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia. *Learning Spark: Lightning-Fast Big Data Analytics.* 1st edition, 2015.
[15] G. Li, D. Deng, J. Wang, and J. Feng. PASS-JOIN: A Partition-based Method for Similarity Joins. *PVLDB*, 5(3):253–264, 2011.
[16] W. Mann, N. Augsten, and P. Bouros. An Empirical Evaluation of Set Similarity Join Techniques. *PVLDB*, 9(9):636–647, 2016.
[17] A. Metwally and C. Faloutsos. V-SMART-Join: A Scalable MapReduce Framework for All-Pair Similarity Joins of Multisets and Vectors. *PVLDB*, 5(8):704–715, 2012.
[18] E. Milchevski, A. Anand, and S. Michel. The Sweet Spot between Inverted Indices and Metric-Space Indexing for Top-K-List Similarity Search. In *EDBT 2015, Brussels, Belgium.*, pages 253–264.
[19] K. Panev, E. Milchevski, and S. Michel. Computing similar entity rankings via reverse engineering of top-k database queries. In ICDE Workshops 2016, Helsinki, Finland, May 16-20, 2016, pages 181–188.
[20] C. Rong, C. Lin, Y. N. Silva, J. Wang, W. Lu, and X. Du. Fast and Scalable Distributed Set Similarity Joins for Big Data Analytics. In *ICDE 2017, San Diego, CA, USA*, pages 1059–1070.
[21] C. Rong, W. Lu, X. Wang, X. Du, Y. Chen, and A. K. H. Tung. Efficient and Scalable Processing of String Similarity Join. *IEEE Trans. Knowl. Data Eng.*, 25(10):2217–2230, 2013.
[22] A. D. Sarma, Y. He, and S. Chaudhuri. ClusterJoin: A Similarity Joins Framework using Map-Reduce. *PVLDB*, 7(12):1059–1070, 2014.
[23] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan. Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics. *PVLDB*, 8(13):2110–2121, 2015.
[24] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using MapReduce. In *SIGMOD 2010, Indianapolis, IN, USA, pages 495–506.*
[25] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In SIGMOD 2012, Scottsdale, AZ, USA, pages 85–96.
[26] X. Wang, L. Qin, X. Lin, Y. Zhang, and L. Chang. Leveraging Set Relations in Exact Set Similarity Join. *PVLDB*, 10(9):925–936, 2017.
[27] Y. Wang, A. Metwally, and S. Parthasarathy. Scalable all-pairs similarity search in metric spaces. *KDD 2013, Chicago, IL, USA*, pages 829–837.
[28] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW 2008, Beijing, China*, pages 131–140.