Pail: Efficient kNN Search on Set-Valued Attributes

Daniel Schmitt
University of Salzburg
Austria
danielulrich.schmitt@plus.ac.at

Thomas Hütter
Software
Competence Center Hagenberg
Austria
thomas.huetter@scch.at

Nikolaus Augsten
University of Salzburg
Austria
nikolaus.augsten@plus.ac.at

ABSTRACT

We study the k-nearest neighbors (kNN) search problem on the domain of sets. Given a query set, the goal is to retrieve the k most similar sets from a collection according to a specified similarity function. Most existing solutions for set similarity queries focus on range search or top-k joins, which typically assume and exploit high similarity thresholds. We observe that existing approaches for kNN search – as well as adaptations of range search and top-k algorithms – exhibit poor performance due to low selectivity of their filtering techniques and high index traversal costs.

To address these limitations, we propose Pail, a kNN search algorithm for sets that supports a wide range of similarity functions. Pail implements the positional filter – a filter that was previously used for post-filtering of candidates returned by an index – directly into a novel index structure to effectively prune candidates. To efficiently traverse only the necessary parts of the index, Pail leverages the monotonicity of the similarity functions with respect to positional information. This traversal enables early termination by ensuring that the index is accessed in descending order of similarity upper bounds. To reduce index access overhead, we propose size grouping and eager reading of index entries that relax filter tightness for improved overall performance. Extensive experiments across diverse datasets demonstrate that Pail consistently outperforms competing algorithms by up to three orders of magnitude.

PVLDB Reference Format:
Daniel Schmitt, Thomas Hütter, and Nikolaus Augsten. Pail: Efficient kNN Search on Set-Valued Attributes. PVLDB, 19(11): 3245 - 3258, 2026.
doi:10.14778/3836663.3836686

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/DatabaseGroup/pail-knn.

1 INTRODUCTION

Set similarity search retrieves all sets similar to a given query set according to a similarity function. Due to the flexibility of sets, different applications operating on various data types can be reduced to this problem, including data cleaning and data integration [6, 14, 15, 33], web search [4], near-duplicate detection [39, 47], word-embeddings [28], and recommender systems [9, 49].

Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836686

Most prior work has focused on *range search*, which returns all sets whose similarity to the query exceeds a user-defined threshold [5, 10, 24, 25, 36, 41, 47]. However, choosing an appropriate threshold is often difficult, in particular for interactive user applications [42, 46, 51]: a high threshold may yield too few results, while a low one may include many irrelevant sets. As an example, if a user searches for {set,similarity,search} in a paper database, they expect to be presented the most relevant, e.g., 10 papers first.

Hence, we focus on *exact k-nearest neighbors (kNN) search*, which retrieves the k most similar sets to a query. This formulation requires only the parameter k, which is typically easier and more intuitive for users to specify than a similarity threshold [42].

Several approaches have been proposed for *exact kNN search on sets*. Existing methods rely on set partitioning techniques [37] or on embedding sets into low-dimensional vector spaces and performing lookups using index structures such as B- or R-trees [50, 51]. Algorithms for the related *top-k join* problem–finding the most similar k pairs of sets between two datasets–can be adapted to kNN search by joining a dataset containing only the query q with the dataset; most top-k join techniques use the prefix filter [6, 40, 46, 48]. We focus on finding the *exact* kNN result. Popular generic kNN search algorithms based on neighborhood graphs such as HNSW [23] and DiskANN [18] do not offer worst-case guarantees w.r.t. result quality [17] and are beyond the scope of this paper.

Unfortunately, existing solutions for kNN queries are highly sensitive to data characteristics such as the set size distribution, the frequency distribution of elements (called *tokens*), and the number of distinct tokens (the *universe size*). Real-world data vary greatly along these dimensions. In our experiments, a simple linear scan often outperforms specialized index-based methods. We identify two main reasons for the poor performance of current approaches:

(1) **Poor Filter Selectivity at Low Similarities:** Existing pruning techniques are designed for and perform best under high similarity thresholds, whereas kNN queries often involve low similarities.
(2) **High Filtering Overhead:** Since candidate verification is a fast linear-time operation, the heavyweight filters used by current algorithms add overhead that cannot be offset by the reduced number of candidates.

A kNN query is equivalent to a range search where the similarity threshold is defined by the k-th most similar set in the dataset: the *cutoff similarity* (not known upfront in the kNN setting). Current algorithms often struggle for low cutoffs, which is often the case in real-world datasets. Partition-based approaches [37] suffer because their partitioning schemes must produce many sparsely populated partitions, resulting in many candidates [36]. Tree-based methods [50, 51] are also affected, as low similarities require scanning large parts of the index. Both partition- and tree-based pruning

incur substantial overhead, with candidate generation dominating runtime. Adapted top-k algorithms employ lightweight pruning, but these filters are insufficiently selective at low similarities. While they perform well in the top-k setting, where cutoff similarities are higher (Jaccard ≈0.77–0.995 [40]), they produce too many candidates in the kNN setting (Jaccard ≈0.08–0.67 in our experiments).

To address the poor performance of existing methods at low similarity thresholds, we propose PAIL$^{\text{F}}$ (**P**ositional **A**lgorithm and **I**ndex for **L**ightweight filtering), a kNN algorithm for sets that builds on the idea of the positional filter [47]. Intuitively, this filter prunes a set pair if the first matching token appears too late in either set, leaving too few remaining tokens to reach the required similarity. Prior work used this filter only for post-processing, i.e., to eliminate dissimilar sets after index retrieval, which does not reduce the number of candidates (also called pre-candidates [25]). Since the filter must be evaluated for each candidate, we integrate the positional filter directly into a novel index structure that retrieves only sets that cannot be pruned by the filter. Unlike prefix filter-based methods that exploit just one dimension of positional information in the kNN setting, our approach leverages all three free dimensions to compute tighter similarity upper bounds to further reduce the number of candidates.

To increase the efficiency of generating candidates, we extend PAIL$^{\text{F}}$ to PAIL$^{\text{L}}$. While PAIL$^{\text{F}}$ uses exact positional information for candidate pruning, this can be inefficient due to many index accesses on sparse data and irregular access patterns. In PAIL$^{\text{L}}$, we relax the positional filter to reduce the number of lookups and replace point queries with efficient scans over larger regions. To enable such scans, we introduce *size grouping* and *eager horizontal reading*: the former groups index entries by set size, guided by a cost model with a proven optimal group size, while the latter improves access locality. We combine both techniques in the PAIL$^{\text{L}}$ algorithm.

In summary, our main contributions are as follows:

(1) We propose PAIL$^{\text{F}}$, a kNN algorithm leveraging the positional filter to handle low-similarity queries efficiently.
(2) PAIL$^{\text{L}}$ implements two relaxations of the positional filter that increase the overall performance: (a) *size grouping* aligns contiguous ranges of set sizes to reduce the search space; (b) *eager horizontal reading* minimizes costly index accesses.
(3) We present a cost model for index accesses with size grouping and derive the optimal number of groups based on dataset characteristics. We further show how PAIL$^{\text{L}}$ maintains effectiveness under looser positional constraints.
(4) In experiments on twelve real-world datasets, we demonstrate that PAIL$^{\text{L}}$ consistently outperforms competing algorithms, achieving speedups of 18× to 455× on average.

Omitted proofs appear in an extended version of this paper [34].

2 BACKGROUND AND ASSUMPTIONS

Similarity Search. A set r consists of tokens from a universe $\mathcal{U}$. We assume that the tokens in each set are ordered according to some global order, i.e., $r = \{r_0, r_1, ..., r_{|r|-1}\}$. As is common in set similarity, we order the tokens by inverse global token frequency [25].

Assumption on the Similarity Function. Similarity search addresses the problem of finding the sets in a dataset R that are most similar to a given query q according to a similarity function $\mathcal{S}$. In this paper, we consider the *k nearest neighbor (kNN) search problem*: Given a query q and a similarity function $\mathcal{S}$, find a subset $T \subseteq R$ such that $|T| = k$ and $\forall s \in T, \forall r \in R \setminus T: \mathcal{S}(q,s) \geq \mathcal{S}(q,r)$. We refer to the problem of k nearest neighbor search on sets as *kNN search*. We illustrate our results using the Jaccard similarity $\mathcal{J}$, defined as $\mathcal{J}(r,s) = \frac{|r \cap s|}{|r|+|s|-|r \cap s|}$, but our results generalize to *symmetric* set similarity functions [8]. A set similarity function $\mathcal{S}$ is symmetric if it can be expressed as a function $\mathcal{S}(|r|,|s|,o)$ of the set sizes $|r|$ and $|s|$ and their overlap o. Let $\Delta > 0$ be an integer. We require the following monotonicity properties [26]:

(1) **Overlap**: Higher overlaps increase similarity: $\mathcal{S}(|r|,|s|,o) \leq \mathcal{S}(|r|,|s|,o+\Delta)$.
(2) **Augmentation**: Increasing the overlap and the size of one set by the same amount increases similarity: $\mathcal{S}(|r|,|s|,o) \leq \mathcal{S}(|r|,|s|+\Delta,o+\Delta)$ and $\mathcal{S}(|r|,|s|,o) \leq \mathcal{S}(|r|+\Delta,|s|,o+\Delta)$.
(3) **Size**: Increasing a set size decreases similarity: $\mathcal{S}(|r|,|s|,o) \geq \mathcal{S}(|r|,|s|+\Delta,o)$ and $\mathcal{S}(|r|,|s|,o) \geq \mathcal{S}(|r|+\Delta,|s|,o)$.

While these monotonicity properties do not determine the relative ordering of similarity values in all cases, they suffice for PAIL and remain general enough to cover all common set similarity functions, including Overlap, Jaccard, Cosine, Dice, and Hamming [26].

ID	Set	$\mathcal{S}(q,s_i)$
q	{1,2,3,4,5}	
s_1	{x,x,3,∗}	$\frac{2}{7} \approx 0.29$
s_2	{x,1,∗,∗,∗}	$\frac{4}{6} \approx 0.67$
s_3	{1,x,x,x,x}	$\frac{1}{9} \approx 0.11$
s_4	{x,1,∗,∗,∗,x}	$\frac{4}{7} \approx 0.57$
s_5	{x,x,x,1,∗,x}	$\frac{2}{9} \approx 0.22$
s_6	{x,x,x,1,∗,∗,∗}	$\frac{4}{8} = 0.5$

Figure 1: Running example dataset and query.

Example 2.1. Consider the running example in Figure 1, which depicts a query q and a dataset consisting of sets s_1 to s_6. All sets are sorted according to some global order. We explicitly show the first token matching the query. Additional matching tokens are represented by "∗" and non-matching tokens by "x", as their concrete values are not relevant for illustrating PAIL. For example, q and s_1 have two tokens (3 and ∗) in common. Hence, the Jaccard similarity is $\mathcal{J}(q,s_1) = 2/(9-2) \approx 0.29$. The result of the kNN search for $k = 2$ is $\{s_2, s_4\}$.

Positional Filter. The positional filter bounds the overlap of two sets given the position of a token match in the globally ordered sets. We use the following variation of the positional filter applicable on the first token match of two sets [47]:

LEMMA 2.2 (LEMMA 4.2 IN [47]). *Given two sets q,s with (0-indexed) ordered tokens, if their first common token is $q_{p_q} = s_{p_s}$, then their maximal overlap is $|q \cap s| \leq \min\{|q| - p_q, |s| - p_s\}$.*

Example 2.3. Set q and s_1 have their first match at $p_q = 2$ and $p_{s_1} = 2$. Hence, $|q \cap s_1| \leq \min\{5-2, 4-2\} = 2$ and $\mathcal{J}(q,s_1) \leq \frac{2}{7}$.

3 FRAMEWORK FOR kNN SET SIMILARITY

All existing algorithms for exact kNN (and top-k) search on sets are based on a common filter-verify framework [6, 37, 40, 42, 46, 48, 51] inspired by a kNN technique for R-trees [32]. In this section, we

Algorithm 1: Generic kNN Lookup

Input: Root $\mathcal{B}_r$, similarity fun. $\mathcal{S}$ and $\mathcal{S}_{ub}$, result size k, query q
Result: k closest points to q

```
1  H ← ∅                    // k points with highest known similarity
2  Q ← {B_r}                // priority queue ordered by decreasing S_ub
3  Loop
4      pop bucket B with highest max. similarity S_ub(q,B) from Q
5      if B was visited before then continue
6      else if S_ub(q,B) ≤ H_k then
7          return H
8      else if B is an inner bucket then
9          foreach child B' ∈ B do
10             push B' into Q with priority S_ub(q,B')
11     else                                   // B is a leaf bucket B_l
12         foreach s ∈ B_l do
13             if s was never visited before and S(q,s) > H_k then
14                 pop H_k and push s with similarity S(q,s) into H
```

generalize this framework and relate previous algorithms to our generalization.

For a kNN lookup in dataset R of query q, a heap $\mathcal{H}$ of the currently known $|\mathcal{H}| = k$ points with the highest similarity is maintained. Based on a filter, the dataset is grouped into a set of (possibly overlapping) *leaf buckets*, denoted $\mathcal{B}_l$. The filter induces a similarity bounding function $\mathcal{S}_{ub}(q, \mathcal{B}_l)$ that, given the query q and a leaf bucket $\mathcal{B}_l$, computes an upper bound on the similarity of all sets in the bucket: $\forall s \in \mathcal{B}_l : \mathcal{S}(q,s) \leq \mathcal{S}_{ub}(q, \mathcal{B}_l)$. Conceptually, the leaf buckets are traversed in descending similarity upper bound order. When a leaf bucket is accessed, the contained sets are compared to the query, and the current solution $\mathcal{H}$ is updated. Once the similarity of the k-th most similar set $\mathcal{H}_k$ in the heap $\mathcal{H}$ exceeds the similarity bound of all remaining buckets, the search is terminated and $\mathcal{H}$ is returned. Depending on filter effectivity, typically only small parts of the dataset must be inspected, but all current algorithms for exact kNN search on sets (including ours) may require linear query time in the worst case.

While we will later show that this traversal is optimal w.r.t. the number of accessed leaf buckets in Lemma 3.1, the cost of computing all similarity upper bounds of all buckets and traversing them in descending order is prohibitive if the number of buckets is high. Instead, current algorithms construct a hierarchy of *inner buckets* over the *leaf buckets* that guide the search, similar to minimum bounding rectangles in an R-tree. Hence, a bucket $\mathcal{B}$ is either (1) a leaf bucket $\mathcal{B}_l$ storing a subset of R, or (2) an inner bucket storing a set of buckets (possibly both inner and leaf buckets). The similarity bounding function $\mathcal{S}_{ub}(q,\mathcal{B})$ is extended to inner buckets such that it satisfies *monotonicity*: For an inner bucket $\mathcal{B}$ and all its children $\mathcal{B}' \in \mathcal{B}$, we have $\mathcal{S}_{ub}(q,\mathcal{B}') \leq \mathcal{S}_{ub}(q,\mathcal{B})$. We refer to the bucket hierarchy as the *bucket DAG* as its structure can be as general as a directed acyclic graph (DAG) with one designated root bucket $\mathcal{B}_r$; current algorithms use tree structures [37, 40, 46, 48, 50, 51]. A set s is called *reachable* from a bucket $\mathcal{B}$ if either $\mathcal{B}$ is a leaf bucket and $s \in \mathcal{B}$, or $\mathcal{B}$ is an inner bucket and s is reachable from some child bucket of $\mathcal{B}$. Monotonicity guarantees that for all sets s reachable from bucket $\mathcal{B}$ we have $\mathcal{S}_{ub}(q,\mathcal{B}) \geq \mathcal{S}(q,s)$.

Algorithm 1 shows how a hierarchy of buckets is explored. We leverage the inner buckets to traverse the leaf buckets in descending maximum similarity order. To this end, a priority queue Q stores the front of the traversal. Initially, Q contains the root bucket $\mathcal{B}_r$. When visiting a bucket $\mathcal{B}$, we either push its children into Q if $\mathcal{B}$ is an inner bucket (lines 8–10) or compare its stored sets with the query q if $\mathcal{B}$ is a leaf bucket, updating $\mathcal{H}$ when more similar sets are found (lines 11–14).

Xiao *et al.* [46] show that the traversal in descending similarity bound order is optimal for their particular instance of the framework. We generalize Theorem 1 from Xiao *et al.* [46]. We call a similarity bounding function $\mathcal{S}_{ub}$ *tight* if, for any bucket $\mathcal{B}$ and query q, we can construct a set s reachable from $\mathcal{B}$ such that $\mathcal{S}_{ub}(q,\mathcal{B}) = \mathcal{S}(q,s)$. The similarity bounding function based on the positional filter is tight.

Lemma 3.1 (Based on Theorem 1 in [46]). *Traversing the buckets of a bucket DAG in descending order of their maximum similarity bound, using a monotonic and tight similarity bounding function, visits the smallest possible set of leaf buckets among all traversal orders, up to tie-breaking.*

We showcase the design decision of existing algorithms following Algorithm 1 with two related algorithms:

Partition [37, 38] partitions sets into disjoint subsets; a leaf bucket contains all sets sharing a specific subset. The hierarchy of inner buckets forms a linear chain: Each inner bucket contains one leaf and one inner bucket. Hence, $|Q|$ is constant. This construction does not leave room for effective filtering: Most candidates are in the later part of the chain, but most parts of the hierarchy have to be accessed due to low cutoff similarities in kNN search (cf. Section 1).

DualTrans [51] transforms sets into vectors and indexes them using an R-tree. Leaf buckets and inner buckets directly correspond to leaf nodes and inner nodes in the R-tree. The hierarchy has a high fanout and can potentially filter many candidates by pruning high-level inner buckets. However, its high fanout can result in long queue sizes $|Q|$, leading to higher index traversal overhead.

Inner and leaf buckets can be *physical* or *virtual*. Physical buckets have materialized, pre-built content. For virtual buckets, the content is constructed on access. Both Partition and DualTrans have physical leaf buckets, but Partition has virtual inner buckets. In our approach Pail, both inner and leaf buckets are virtual.

4 POSITIONAL-BASED kNN SEARCH

We present a novel algorithm for kNN search, called Pail$^{\text{F}}$. The approach is based on the generalized framework introduced in Section 3 and presents new solutions for: (1) the construction of leaf buckets $\mathcal{B}_l$, (2) the hierarchy of inner buckets, and (3) the similarity bounding function $\mathcal{S}_{ub}$. In our construction, leaf buckets and inner buckets are virtual, i.e., they are not materialized. We discuss the construction of inner buckets in Section 4.2 and the index-based retrieval of leaf bucket content in Section 4.3.

4.1 Similarity Bounding and Monotonicities

Similarity Bounding. Using Lemma 2.2, we can bound the maximum possible similarity $\mathcal{S}_{ub}(q, p_q, s, p_s)$ of two sets q, s based on their first matching token positions p_q, p_s for any symmetric similarity

function [8], e.g., for the Jaccard similarity:

$$\mathcal{J}(q,s) = \frac{|q \cap s|}{|q|+|s|-|q \cap s|} \leq \frac{\min\{|q|-p_q, |s|-p_s\}}{|q|+|s|-\min\{|q|-p_q, |s|-p_s\}} =: \mathcal{S}_{ub}(q,p_q,s,p_s)$$

With ① Overlap, ② Augmentation, and ③ Size (cf. Section 2), we derive new inequalities for the similarity bounding function $\mathcal{S}_{ub}$.

Lemma 4.1. *For any fixed query size $|q|$, any set size $|s|$, token positions $0 \leq p_q < |q|$ and $0 \leq p_s < |s|$, the similarity bound $\mathcal{S}_{ub}$ satisfies:*

① $\mathcal{S}_{ub}(|q|,p_q,|s|,p_s) \geq \mathcal{S}_{ub}(|q|,p_q+1,|s|,p_s+1)$
② $\mathcal{S}_{ub}(|q|,p_q,|s|,p_s) \geq \mathcal{S}_{ub}(|q|,p_q+1,|s|-1,p_s)$
③ $\mathcal{S}_{ub}(|q|,p_q,|s|,p_s) \geq \mathcal{S}_{ub}(|q|,p_q,|s|+1,p_s+1)$

The monotonicities in Lemma 4.1 induce the bucket DAG in Figure 2: The children of an inner bucket $\mathcal{S}_{ub}(|q|,p_q,|s|,p_s)$ are the right sides of the inequalities of Lemma 4.1: $\mathcal{S}_{ub}(|q|,p_q+1,|s|,p_s+1)$, $\mathcal{S}_{ub}(|q|,p_q+1,|s|-1,p_s)$, and $\mathcal{S}_{ub}(|q|,p_q,|s|+1,p_s+1)$. The edges in the DAG hence correspond to the dashed arrows ⇢ in Figure 2.

Simplifying the Bucket DAG While this construction is correct, most buckets are reachable via multiple paths. As an example, bucket $\mathcal{B}(p_q+1,|s|,p_s+1)$ is reachable from the root either through a ① edge, ②-③, or ③-②. As a result, we would insert duplicate buckets into Q. To avoid duplicates, we leverage Corollary 4.2 of Lemma 4.1 to prune edges. Intuitively, the "outer" buckets have a higher similarity than the center buckets at the same level (cf. Figure 2).

Corollary 4.2. *For fixed query size $|q|$, set size $|s|$, and integers u,v,w with $|s| > u \geq v > w \geq 0$, we have the following two inequalities:*

$$\mathcal{S}_{ub}(|q|,p_q+u,|s|-v,p_s+(u-v)) \leq \mathcal{S}_{ub}(|q|,p_q+u,|s|-w,p_s+(u-w))$$

$$\mathcal{S}_{ub}(|q|,p_q+(u-w),|s|+w,p_s+u) \geq \mathcal{S}_{ub}(|q|,p_q+(u-v),|s|+v,p_s+u)$$

By Lemma 4.1 and Corollary 4.2, we prune unnecessary edges as follows: (1) For a bucket $\mathcal{B}$, edges leading to a sibling $\mathcal{B}'$ at the same level (horizontal ⇢ ② ➤ and ◀ ③ ⇢ in Figure 2) are redundant: The ① parent of $\mathcal{B}'$ has a higher value of $\mathcal{S}_{ub}$ and will enqueue $\mathcal{B}'$ before $\mathcal{B}$. (2) For each level in the depicted tree, the "outer" buckets by distance $||q|-|s||$ are visited before the buckets closer to $|s|$. Hence, having edges ◀ ③ ˊ and ˋ ② ➤ to the next level of the hierarchy in Figure 2 is only necessary for the outermost buckets. Pruning these edges transforms the bucket DAG into a bucket tree.

4.2 PAIL^F Bucket Tree

We now formalize the intuition of buckets from Section 4.1 and define leaf buckets, inner buckets, and their hierarchy.

Leaf Buckets. The positional filter (cf. Section 2) can be viewed as a four-dimensional vector space. Its four dimensions are: (1) the query set size $|q|$, (2) the query token position p_q, (3) the index set size $|s|$, and (4) the index token position p_s. For a fixed query q, the query set size $|q|$ does not change; thus, we can assume a fixed $|q|$ value. The *search space* spanned by the positional filter is therefore three-dimensional. Each dimension is bounded by the maximum set size $|s^\top| = \max_{s \in R \cup \{q\}} |s|$ of a query or index set and we have one leaf bucket $\mathcal{B}_l(p_q,|s|,p_s)$ for each combination of parameters (except $|s|=0$), i.e., $\Theta(|s^\top|^3)$ leaf buckets.

Definition 4.3. The *leaf bucket* $\mathcal{B}_l(p_q,|s|,p_s)$ associated with point $(p_q,|s|,p_s)$ in the search space contains all sets s of size $|s|$ such that the tokens at positions p_q and p_s form a match, i.e., $q_{p_q} = s_{p_s}$.

Inner Buckets. For a fixed size of s, multiple leaf buckets might have identical similarity upper bounds. In particular, all leaf buckets $\mathcal{B}_l(p_q,|s|,p_s)$ and $\mathcal{B}_l(p'_q,|s|,p'_s)$ with $\min\{|q|-p_q,|s|-p_s\} = \min\{|q|-p'_q,|s|-p'_s\}$ have the same maximum overlap bound and thus share the same similarity bound (cf. Section 4.1). We thus create inner buckets $\mathcal{B}(p_q,|s|,p_s)$ for such sets of leaf buckets. The root bucket $\mathcal{B}_r$ is the inner bucket $\mathcal{B}(0,|q|,0)$.

Definition 4.4. The *inner bucket* $\mathcal{B}(p_q,|s|,p_s)$ has a number of leaf bucket children and up to three inner bucket children:

Leaf bucket children: Bucket $\mathcal{B}(p_q,|s|,p_s)$ contains all leaf buckets $\mathcal{B}_l(p'_q,|s|,p_s)$ with $p'_q \leq p_q$ and $\mathcal{B}_l(p_q,|s|,p'_s)$ with $p'_s \leq p_s$.

Inner bucket children: Bucket $\mathcal{B}(p_q,|s|,p_s)$ has up to three inner bucket children (cf. Figure 2): (1) Child ⓐ $\mathcal{B}(p_q+1,|s|,p_s+1)$, (2) Child ⓑ $\mathcal{B}(p_q+1,|s|-1,p_s)$, and (3) Child ⓒ $\mathcal{B}(p_q,|s|+1,p_s+1)$. Bucket $\mathcal{B}(p_q,|s|,p_s)$ always has Child ⓐ. If bucket $\mathcal{B}(p_q,|s|,p_s)$ satisfies $|s| \leq |q|$ and $p_s = 0$, it has an additional Child ⓑ. If bucket $\mathcal{B}(p_q,|s|,p_s)$ satisfies $|s| \geq |q|$ and $p_q = 0$, it has an additional Child ⓒ.

Note that the root bucket $\mathcal{B}(0,|q|,0)$ has three children. By Lemma 4.1, buckets satisfy monotonicity as required for Algorithm 1 (cf. Section 3). We now show the correctness of Algorithm 1 on the PAIL^F bucket tree. By Lemma 3.1, Algorithm 1 is optimal in that it accesses the minimal number of leaf buckets (and thus indexed sets) among algorithms using the positional filter for indexing.

Lemma 4.5 (Correctness). *Algorithm 1 executed on the PAIL^F Bucket Tree is correct.*

Compared to the naive approach of sorting all leaf buckets that requires an ordered traversal of $\Theta(|s^\top|^3)$ elements, our hierarchy guarantees queue sizes linear in $|s^\top|$.

Lemma 4.6 (Linear Queue Length). *At any point in Algorithm 1, the queue size is within the maximum set size: $|Q| \leq |s^\top|$.*

Example 4.7. Figure 3 and Figure 4 show the inner and leaf buckets for the example in Figure 1, respectively. Each 1×1 cell in Figure 4 is a leaf bucket. The leaf buckets contained in an inner bucket share the same color. By definition, the inner bucket $\mathcal{B}(p_q,|s|,p_s)$ contains the leaf buckets $\mathcal{B}_l(i,|s|,j)$ for all i,j with $\min\{|q|-i,|s|-j\} = \min\{|q|-p_q,|s|-p_s\}$. For example, inner bucket $\mathcal{B}(1,5,1)$ contains the leaf buckets $\mathcal{B}_l(0,5,1)$, $\mathcal{B}_l(1,5,0)$, and $\mathcal{B}_l(1,5,1)$. Out of those, only $\mathcal{B}_l(0,5,1) = \{s_2\}$ is non-empty.

Example 4.8. Consider the running example (Figure 1) of a 2NN query and the inner buckets of the PAIL^F bucket tree in Figure 3. The inner buckets have leaf buckets (cf. Figure 4), which in turn contain the sets. Figure 3 shows for each inner bucket the sets contained in its leaf buckets. By Algorithm 1, we explore the bucket tree as follows: (1) We dequeue the root $\mathcal{B}(0,5,0)$, compute $\mathcal{S}(q,s_3)$, store s_3 in $\mathcal{H}$, and enqueue $\mathcal{B}(1,4,0)$, $\mathcal{B}(1,5,1)$, and $\mathcal{B}(0,6,1)$ with $\mathcal{S}_{ub}$ 0.8, 0.67, and 0.83, respectively. (2) We dequeue $\mathcal{B}(0,6,1)$, find s_4, store it in $\mathcal{H}$, enqueue $\mathcal{B}(1,6,2)$ and $\mathcal{B}(0,7,2)$. (3) We proceed similarly for $\mathcal{B}(1,4,0)$, $\mathcal{B}(0,7,2)$, $\mathcal{B}(1,5,1)$, and $\mathcal{B}(2,3,0)$. The current result is $\mathcal{H} = \{s_2,s_4\}$. (4) The next bucket $\mathcal{B}(1,6,2)$ has $\mathcal{S}_{ub} = 0.57$. As $\mathcal{S}_{ub} \leq \mathcal{H}_k = 0.57$, we return $\mathcal{H} = \{s_2,s_4\}$ as the result. In total, we access 3 sets (s_3,s_4,s_2) and 6 inner buckets.

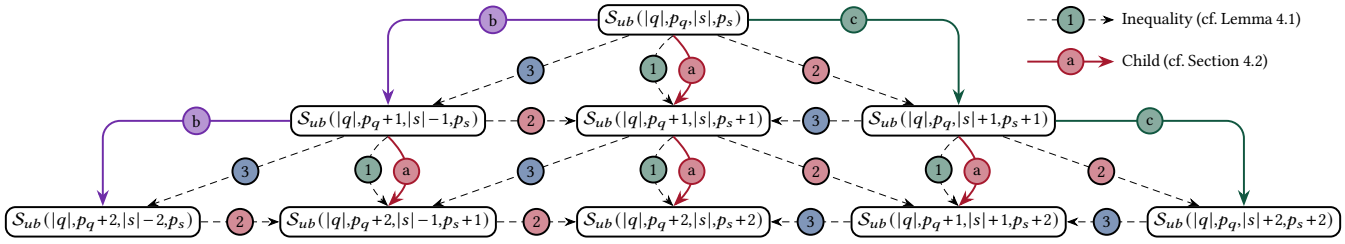

Figure 2: Application of monotonicities (assuming $|q|-p_q=|s|-p_s$).

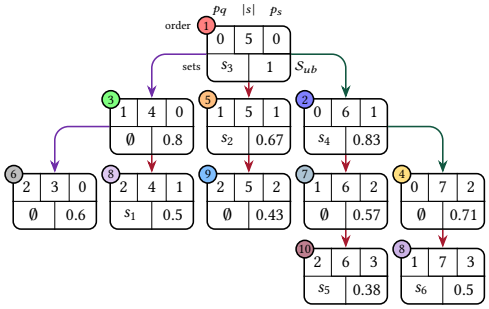

Figure 3: Example of PAIL^F bucket tree.

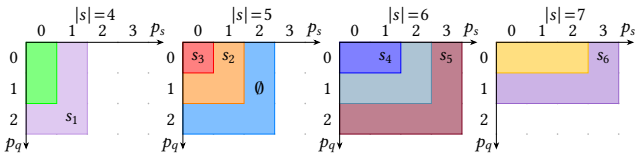

Figure 4: Example of leaf buckets.

4.3 Leaf Bucket Index

Different query sets will typically result in different leaf buckets. We do not physically store leaf buckets. Instead, we retrieve the sets contained in a leaf bucket on the fly: Leaf bucket $\mathcal{B}(p_q,|s|,p_s)$ contains all sets of size $|s|$ sharing token q_{p_q} at position p_s. As q changes, also the sets in the bucket will change. To efficiently access the relevant sets, we propose an inverted index augmented with pointers and specific orders that speed up the index access.

The leaf buckets of an inner bucket $\mathcal{B}(p_q,|s|,p_s)$ always have an ⌋-shape (cf. Figure 4) and require reading the leaf buckets $\mathcal{B}(p_q,|s|,0)$ to $\mathcal{B}(p_q,|s|,p_s)$ and $\mathcal{B}(0,|s|,p_s)$ to $\mathcal{B}(p_q-1,|s|,p_s)$, i.e., p_q+p_s+1 different buckets. The leaf buckets $\mathcal{B}(p_q,|s|,0)$ to $\mathcal{B}(p_q,|s|,p_s)$ all have the same matching token q_{p_q}, but they appear at different positions in the sets. For the remaining buckets, the matching token changes with the query positions. We first describe our physical index structure and then show efficient access methods for both cases.

Physical Index Structure. Figure 5 shows the physical index structure that implements leaf bucket lookups. At the first level of the index, sets are partitioned by their size. All sets of one size are then stored inside an inverted index that maps tokens to the sets they appear in: If set s has token t, a reference to s is stored inside the posting list indexed by t. Additionally, the posting lists are stored in increasing value of token position: If set s has token t at position p and set s' contains t at position $p'>p$, then s appears before s' in the posting list. Posting lists are sorted to increase the performance of finding all sets containing a token up to some token position.

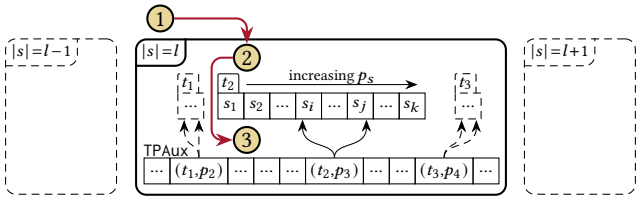

Figure 5: Physical index structure.

Additionally, each size partition contains an auxiliary data structure TPAux implemented as a hash table mapping pairs of tokens and token positions to partial posting lists corresponding to that token and token position. This data structure allows fast access to all sets containing a specific token at a specific position, avoiding the need to fetch its complete postings list.

Space Complexity. We analyze the space complexity of the PAIL^F index structure. A single token t of an input set s appears in exactly one posting list, namely the posting list of t in the partition for set size $|s|$. Additionally, we have at most one TPAux mapping for each token. Hence, memory usage is linear in the total number of tokens.

Index Access. The inner buckets of bucket $\mathcal{B}(p_q,|s|,p_s)$ form an ⌋-shape (cf. Figure 4). To find the sets inside the inner buckets, we first find the partition of the index for sets of size $|s|$ (Step ① in Figure 5). We then scan the horizontal part of ⌋ containing $\mathcal{B}(p_q,|s|,0)$ to $\mathcal{B}(p_q,|s|,p_s)$ (called *horizontal reading* of the index): These leaf buckets have the same token q_{p_q} in common with the query q and that token appears up to position p_s in the indexed set s. To find such sets, we fetch the posting list for token q_{p_q} and scan the list until we find the first entry with token position $>p_s$ (Step ② in Figure 5).

Finally, we read the vertical part of ⌋ containing $\mathcal{B}(0,|s|,p_s)$ to $\mathcal{B}(p_q-1,|s|,p_s)$ (called *vertical reading* of the index). These buckets share different tokens with q. We leverage TPAux to efficiently access the relevant parts of the different posting lists: For each token t in q up to position p_q-1, we look up the tuple (t,p_s) in TPAux to find the entrypoint into the posting list and scan it until finding a set with index position $>p_s$ (Step ③ in Figure 5). Hence, reading the inner bucket $\mathcal{B}(p_q,|s|,p_s)$ requires p_s+1 index seeks.

Example 4.9. Example 4.8 requires 9 index seeks, consisting of 5 horizontal reads and 4 vertical reads.

5 RELAXING THE POSITIONAL FILTER

Our highly-selective filtering algorithm PAIL^F (cf. Section 4) for kNN search leverages the full potential of the positional filter, i.e., its similarity upper bounds are tight w.r.t. the positional filter. Such strong filtering power is often unnecessary for real-world datasets: The time spent on filtering the result does not pay off as lightweight filters (e.g., the prefix filter) combined with fast verification are more efficient [25]. For PAIL^F, filtering can be a significant overhead for two reasons:

(1) **Sparse data:** The positional filter is computed based on set sizes, query token positions (i.e., query tokens), and index token positions. Its maximum search space is a three-dimensional cube with side length proportional to these three dimensions, all of which are bounded by the maximum set size in the dataset. If the number of different set sizes or tokens is high, searching requires accessing major parts of the search space even though each bucket only contains few candidates, i.e., verifying the candidates is less effort than finding them.
(2) **Inefficient access pattern**: Vertical reads of buckets require one index seek for each token in the query q up to some position $p_q < |q|$ (cf. Section 4.3). Horizontal reads of buckets are unaffected as only one lookup is necessary.

In this section, we introduce two relaxations of the positional filter, *size grouping* and *eager horizontal reading*, which trade upper-bound accuracy for increased lookup efficiency.

5.1 Size Grouping

By merging multiple neighboring sizes into size groups, we reduce the width of the three-dimensional cube spanning the search space. The selection of size groups has to balance two competing aspects: (1) Increasing the width of a size group decreases the total number of groups, thus reducing the overhead of candidate generation. (2) Increasing the width of a size group decreases the accuracy of the positional filter: The group's similarity upper bound is at least as high as the similarity upper bound of each individual set size inside the group. This increases the number of candidates that are retrieved from the index and must be verified.

5.1.1 Construction of Size Groups. To balance these aspects, we proceed in three steps. We first propose a *grouping scheme* that considers the contribution of a single set to the cost of the physical index structure (cf. Section 4.3). We then define a *cost model* for the evaluation of kNN queries using the index. Finally, we leverage this cost model to *optimize* the number of size groups.

Weighted Size Groups. A single set s stored inside a posting list index appears in $|s|$ different lists inside the index. Additionally, even with fast verification using early-termination [25], its verification cost grows proportionally to its size. Thus, we define the weight of a set to be its size. The full dataset R has weight $S = \sum_{s \in R} |s|$. We want to partition R into m contiguous parts such that each partition has a weight of approximately S/m. To this end, we minimize the maximum weight over all size groups. Given a weight limit w, we can decide whether a greedy allocation of set sizes into m parts is possible without exceeding w on any size group. Thus, we minimize the maximum weight by performing a binary search on w. We analyze the construction time of size groups in Section 5.3.

Cost Model. At a high level, our index structure is partitioned by set size and set token (cf. Section 4.3, Figure 5). In general, the major cost of probing the index structure are the number of list accesses (each requiring a fixed number of hash table lookups) and reading the entries of the lists. Thus, the cost $cost(q)$ of a lookup for a query set q has the following general shape:

$$cost(q) = \sum_{\text{accessed list } i} (C_o + |l_i|)$$

C_o represents the index access cost and $|l_i|$ is the length of the accessed list l_i. In our index structure, the list lengths $|l_i|$ depend on (1) the frequency of the accessed size group, (2) the frequency of the query token inside the accessed size group, (3) the effectiveness of the positional filter that in turn depends on both the cutoff similarity of the query and the granularity of the accessed size group. To model the cost of a query, we make the simplifying Assumptions 1-5.

ASSUMPTION 1. *The token distribution and the size distribution are independent. When splitting the sets stored inside a list of size $|l_i|$ into m size groups with equal weight, the expected size of each list inside the m size groups is $|l_i|/m$.*

Definition 5.1. The *modelled cost* of a single kNN lookup is

$$cost(q) = \sum_{\text{accessed list } i} k_i (C_o + |l_i|/m)$$

where l_i is the length of the i-th accessed list and k_i is the number of partitions that are read for this list. C_o is the constant cost for accessing a single partition of a list. The list length per-partition is $|l_i|/m$ (i.e., we use its expectation).

Optimizing the Cost Model. Let $\mathcal{U}$ be the universe size and $S = \sum |l_i|$ the total length of all lists. We now optimize the cost model for kNN lookups under Assumptions 1-5. We start with the ratio of read partitions k_i: k_i depends primarily on the cutoff similarity of the query set: If the cutoff similarity is high, we quickly reach a point where all unseen parts of the lists cannot be results. For low cutoff similarities, major parts of all accessed lists have to be read. We do not assume knowledge about the exact distribution of k_i and only parameterize the distribution by its expected value.

ASSUMPTION 2. *Let $\mathcal{K}_i$ be the random variable representing the number of partitions accessed for the i-th list, and let m denote the number of size groups. We read at least one partition from each accessed list, thus $1 \le \mathcal{K}_i \le m$. We assume that the expected value of $\mathcal{K}_i$ is $\mathbb{E}[\mathcal{K}_i] = 1 + \alpha(m-1)$ for some fixed access ratio $\alpha \in [0,1]$, i.e., beyond the first partition, a ratio of α of the remaining partitions is read on average.*

The accessed lists depend on the tokens in the query that are probed against the index. We consider two cases and assume a different token distribution in the query each time: In Assumption 3, we assume a self-join scenario: The probability of a token to appear in a query is the same as its frequency in the indexed dataset. In Assumption 4, we consider the scenario where the queries are unknown as in an R-S-join, and we thus assume a uniform distribution over the query tokens. We assume that the list-access distribution and the number of partitions read are independent (Assumption 5).

Assumption 3 (Self Join). *Let $\mathcal{L}_i$ be the random variable representing the i-th accessed list. The probabilities of accessing a particular posting list l are independent and follow the same distribution as the tokens in the indexed dataset, i.e.,* $\Pr[\mathcal{L}_i = l] = |l|/S$.

Assumption 4 (*R-S* Join). *Let $\mathcal{L}_i$ be the random variable for the i-th accessed list. The probability of accessing a particular posting list l is uniformly and independently distributed, i.e.,* $\Pr[\mathcal{L}_i = l] = 1/\mathcal{U}$.

Assumption 5. *The random variables for the accessed lists $\mathcal{L}_i$ and the number of accessed partitions $\mathcal{K}_i$ in each list are mean-independent.*

We now derive the optimal number of size groups m under the assumptions for self-joins (Lemma 5.2) and *R-S*-joins (Lemma 5.3). The number of size groups m depends on the lengths of the posting lists l_i, access ratio α, universe $\mathcal{U}$, and partition access cost C_o.

Lemma 5.2 (Optimality (Self Join)). *Under Assumptions 1, 2, 3 (self-join), and 5, the number of size groups m that minimizes the expected kNN lookup cost is*

$$m = \sqrt{\frac{(1-\alpha)\sum_i |l_i|^2}{\alpha C_o S}} = \sqrt{\frac{(1-\alpha)\sum_i |l_i|^2}{\alpha C_o \sum_i |l_i|}}$$

Lemma 5.3 (Optimality (*R-S* Join)). *Under Assumptions 1, 2, 4 (R-S-join), and 5, the number of size groups m that minimizes the expected kNN lookup cost is* $m = \sqrt{(1-\alpha)\bar{t}/(\alpha C_o)}$.

Corollary 5.4. *If the number of read partitions $\mathcal{K}_i$ is uniformly distributed between* 1 *and m, the number of size groups m that minimizes the expected kNN lookup is*

$$m = \sqrt{\frac{\sum_i |l_i|^2}{C_o \sum_i |l_i|}} \text{ and } m = \sqrt{\frac{\sum_i |l_i|}{C_o |\mathcal{U}|}}$$

under Assumptions 3 (self-join) and 4 (R-S-join), respectively.

We show the following complexity of constructing size groups.

Lemma 5.5 (Complexity of Size Group Construction). *The number of size groups m and optimally balanced size groups can be constructed in linear time w.r.t. the total number of tokens in R.*

Discussion. In practice, this size grouping scheme achieves three goals: (1) The number of size groups m grows with the length of the lists inside the index. The longer the list, the more size groups are created. For self joins, the skew of the tokens is considered: If the index contains many long lists, they are split into more size groups to avoid reading them in full. (2) Furthermore, the number of size groups m also grows with an increasing cutoff similarity: A high cutoff similarity allows for early termination of a lookup, resulting in few partitions being read for each list. Hence, α is low and $(1-\alpha)/\alpha$ is high. In such cases, having more size groups helps in skipping major parts of the lists. For low cutoff similarities, most parts of the lists must be read, and partitioning only adds overhead; α is high, resulting in fewer size groups. (3) The grouping scheme creates fine-grained size groups for common sizes and coarse groups for uncommon sizes. In particular, real-world datasets often show long tails of very long sets that occur infrequently [29]. Such sets are typically placed into a single group by our partitioning, avoiding wasteful index queries with almost no entries.

We will refer to a size group for sizes between l and u (both inclusive) as $\mathcal{G}_l^u$. The set of all size groups is $\mathcal{G} = \{{}_1\mathcal{G}_l^u, {}_2\mathcal{G}_l^u, \ldots, {}_m\mathcal{G}_l^u\}$; we assume that size groups are ordered by stored set size with size group ${}_1\mathcal{G}_l^u$ storing the smallest and ${}_m\mathcal{G}_l^u$ the largest sets. Indexes i of size groups ${}_i\mathcal{G}_l^u$ are hidden when only considering a single bucket. Buckets with grouped index sizes are referred to as $\mathcal{B}(p_q, \mathcal{G}_l^u, p_s)$.

5.1.2 Bucket Construction with Grouping. To leverage Algorithm 1 with size groups, we have to redesign (1) the structure of inner and leaf buckets and (2) the similarity bounding function $\mathcal{S}_{ub}$. Notably, we cannot use the correspondence of inner and leaf buckets defined in Section 4.2, where an inner bucket $\mathcal{B}(p_q, |s|, p_s)$ with $|q| - p_q = s - p_s$ corresponds to the leaf buckets $\mathcal{B}(0, |s|, p_s)$ to $\mathcal{B}(p_q, |s|, p_s)$ and $\mathcal{B}(p_q, |s|, 0)$ to $\mathcal{B}(p_q, |s|, p_s)$. In particular, all such leaf buckets have the same maximum similarity. With size groups consisting of different sizes, this cannot be guaranteed anymore.

Grouped Bucket Structure. We use the following bucket structure for Pail$^{\text{L}}$. In particular, this bucket structure is lightweight in its required index access, naturally extends the bucket structure of Pail$^{\text{F}}$, and can be extended to grouped eager horizontal reading later in Section 5.2.

Definition 5.6. The grouped leaf bucket $\mathcal{B}_l^g(p_q, \mathcal{G}_l^u, p_s)$ contains all sets s with $|s| \in [l,u]$ and $q_{p_q} = s_{p_s}$.

Definition 5.7. The grouped inner bucket $\mathcal{B}^g(p_q, \mathcal{G}_l^u, p_s)$ (with $|q| - p_q = u - p_s$) contains all leaf grouped buckets $\mathcal{B}_l^g(p'_q, \mathcal{G}_l^u, p'_s)$ such that $\min\{|q| - p'_q, u - p'_s\} = |q| - p_q$.

Grouped Similarity Bounding. We update $\mathcal{S}_{ub}$ to compute similarity upper bounds for size groups $\mathcal{B}^g(p_q, \mathcal{G}_l^u, p_s)$. Intuitively, $\mathcal{S}_{ub}$ should be the maximum over all possible choices of $|s| \in [l,u]$. We avoid computing the maximum explicitly by exploiting Case ③ of Lemma 4.1.

Lemma 5.8. *The grouped inner bucket $\mathcal{B}^g(p_q, \mathcal{G}_l^u, p_s)$ (with $|q| - p_q = u - p_s$) achieves its highest similarity bound for the lowest size $|s| \in [l,u]$ such that $|s| - |q| + p_q \geq 0$.*

Using Lemma 5.8, with $|s'| = \max\{l, |q| - p_q\}$ we replace the similarity bound function $\mathcal{S}_{ub}(|q|, p_q, |s|, p_s)$ with the grouped similarity bound function $\mathcal{S}_{ub}^g(|q|, p_q, \mathcal{G}_l^u, p_s)$ defined as:

$$\mathcal{S}_{ub}^g(|q|, p_q, \mathcal{G}_l^u, p_s) = \mathcal{S}_{ub}(|q|, p_q, |s'|, |s'| - |q| + p_q)$$

Grouped Bucket Hierarchy. The inner buckets are structured similarly as described in Section 4.2, but both index and query position have to be incremented proportionally to the distance between neighboring size groups and the distance to the query size $|q|$. Assume that we have size groups $\mathcal{G} = \{{}_1\mathcal{G}_l^u, {}_2\mathcal{G}_l^u, \ldots, {}_m\mathcal{G}_l^u\}$. The root bucket is $\mathcal{B}(0, {}_i\mathcal{G}_l^u, u - |q|)$ with $|q| \in [l,u]$.

(1) If the bucket has the shape $\mathcal{B}^g(p_q, {}_i\mathcal{G}_l^u, p_s)$ with $|q| \in [l,u]$ and $p_q = 0$, the children are
   (a) $\mathcal{B}^g(p_q + 1, {}_i\mathcal{G}_l^u, p_s + 1)$: As before.
   (b) $\mathcal{B}^g(p_q + (|q| - u'), {}_{i-1}\mathcal{G}_{l'}^{u'}, p_s)$: We move to the next lower size group and increase the query position by the smallest distance of the size group to $|q|$, i.e., $|q| - u'$ (equivalent to $|q| - u'$ iterations of ⓑ in Figure 2).
   (c) $\mathcal{B}^g(p_q, {}_{i+1}\mathcal{G}_{l'}^{u'}, p_s + (l' - |q|))$: We move to the next higher size group and increase the dataset position by

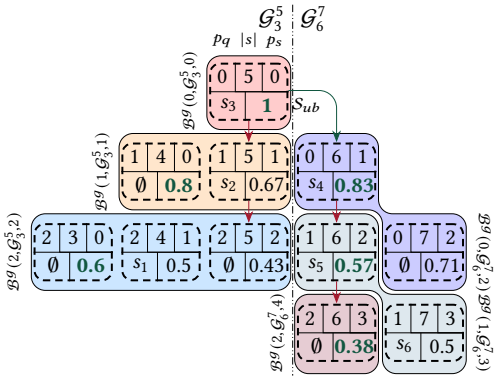

Figure 6: Example of Pail bucket tree with grouping.

the smallest distance of the size group to $|q|$, i.e., $l'-|q|$ (equivalent to $l'-|q|$ iterations of ⑤ in Figure 2).

(2) If the bucket has the shape $\mathcal{B}^g(p_{q,i}\mathcal{G}_l^u, p_s)$ with $u<|q|$, the child is the same as in Case (1)a. If additionally $p_s=0$, we also add $\mathcal{B}^g(p_q+(u-u'),_{i-1}\mathcal{G}_{l'}^{u'}, p_s)$, similar to Case (1)b.

(3) If the bucket has the shape $\mathcal{B}^g(p_{q,i}\mathcal{G}_l^u, p_s)$ with $l>|q|$, the child is the same as in Case (1)a. If additionally $p_q=0$, we also add $\mathcal{B}^g(p_{q,i+1}\mathcal{G}_{l'}^{u'}, p_s+(l'-l))$, similar to Case (1)c.

Size grouping further decreases the maximum queue size.

Lemma 5.9 (Queue Length with Size Grouping). *With size grouping and m groups, the queue size is bounded by $|Q|\leq m$ throughout the execution of Algorithm 1.*

Example 5.10. Consider the grouped Pail bucket tree (cf. Figure 6) and leaf buckets (cf. Figure 7) for the running example (cf. Figure 1). There are two size groups: $\mathcal{G}_3^5$ and $\mathcal{G}_6^7$. The buckets from the non-grouped bucket tree are drawn with dashed lines, the grouped buckets are given by the colored groups. Grouped leaf buckets are constructed by merging the respective non-grouped leaf buckets (cf. Figure 4). Note that s_5 is now contained in bucket $\mathcal{B}(1,6,2)$ as p_s of a group is determined by the highest individual p_s (in this case, $p_s=3$ as $\mathcal{B}(1,7,3)$ is part of the group). By Algorithm 1, we explore the bucket tree as follows: (1) We dequeue the root $\mathcal{B}^g(0,\mathcal{G}_3^5,0)$, finding s_3 as before. We enqueue $\mathcal{B}^g(1,\mathcal{G}_3^5,1)$ and $\mathcal{B}^g(0,\mathcal{G}_6^7,2)$ with grouped similarity bounds $\mathcal{S}_{ub}^g$ 0.8 and 0.83, respectively, as given by Lemma 5.8. (2) We continue dequeuing $\mathcal{B}^g(0,\mathcal{G}_6^7,2)$, $\mathcal{B}^g(1,\mathcal{G}_3^5,1)$, and $\mathcal{B}^g(2,\mathcal{G}_3^5,2)$, finding s_4, s_2, and s_1 , respectively. (3) Before accessing $\mathcal{B}^g(1,\mathcal{G}_6^7,3)$, Algorithm 1 terminates. In total, we access 4 sets and 4 inner buckets. Compared to Example 4.8, more sets and fewer inner buckets are accessed. Out of the 6 index seeks, 4 are horizontal and 2 are vertical reads.

5.2 Eager Horizontal Reading

Vertical reads (cf. Section 4.3) are necessary to *revisit* specific query tokens at later index positions. In Figure 7, retrieving the content for the inner bucket $\mathcal{B}^g(1,\mathcal{G}_3^5,1)$ (shaded orange) requires, among others, an index seek for the 0-th token t of the query q, searching for index sets s where $q_0=s_1$. These accesses are expensive as they require one index seek for each token and cannot be substituted by efficient scans. To reduce the impact of vertical reads, we instead propose to

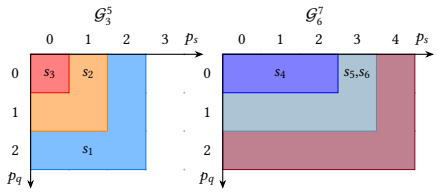

Figure 7: Example of leaf buckets with grouping.

perform longer horizontal reads that cover the otherwise necessary vertical reads. In the example of $\mathcal{B}^g(1,\mathcal{G}_3^5,1)$, the vertical read could be avoided by extending the horizontal read of $\mathcal{B}^g(0,\mathcal{G}_3^5,0)$. We now formalize this approach.

Eager Horizontal Reading without Grouping. Assume that we have just dequeued an inner bucket $\mathcal{B}(p_q,|s|,0)$. We now want to perform a horizontal read up until some index position $p_s^\top$ such that the search ends before having to read an inner bucket $\mathcal{B}(p_q,|s|,p_s)$ with $p_s>p_s^\top$. At the time of reading bucket $\mathcal{B}(p_q,|s|,0)$, we know that the search will terminate at the latest if the highest similarity upper bound $\mathcal{S}_{ub}$ is less than the current value of $\mathcal{H}_k$. As $\mathcal{S}_{ub}$ is monotonically decreasing in increasing values of p_s, we can find a p_s such that $\mathcal{S}_{ub}(|q|,p_q,|s|,p_s)<\mathcal{H}_k$ using binary search. Typical set similarity functions such as Jaccard also have closed-form expressions for p_s:

$$p_s \leq \left\lfloor |s| - \frac{\mathcal{H}_k}{1+\mathcal{H}_k}(|q|+|s|) \right\rfloor =: p_s^\top$$

Example 5.11. Consider Figure 4. When utilizing eager horizontal reading, we read more leaf buckets (i.e., 1×1 cells in the figure) to the right of a bucket than strictly necessary. The order of inner buckets remains the same as in Example 4.8 as the similarity bounds $\mathcal{S}_{ub}$ are unaffected by eager horizontal reading. Compared to Example 4.8, we differ in the following steps: (1) When reading $\mathcal{B}(0,5,0)$, we have $\mathcal{H}=\emptyset$ and $p_s^\top=\infty$. Hence, we read the full horizontal row including not only s_3, but also s_2. (2) For $\mathcal{B}(0,6,1)$, we have $\mathcal{H}=\{s_2,s_3\}$, $\mathcal{H}_k=0.11$, and $p_s^\top=4$. Hence, we read not only s_4, but also s_5. (3) For $\mathcal{B}(0,7,2)$, we could potentially read s_6, but when reading $\mathcal{B}(0,7,2)$ we have $\mathcal{H}=\{s_2,s_4\}$, $\mathcal{H}_k=0.57$, and $p_s^\top=2$. Hence, we stop scanning the index before reaching s_6. In total, we access 4 sets and 6 inner buckets. Compared to Example 4.8, more sets are accessed. However, the number of index seeks is reduced from 9 to 7 by avoiding 2 vertical reads.

Eager Horizontal Reading with Grouping. When using size grouping, we do not have access to the exact sizes of the sets stored in the index. Therefore, to achieve the required similarity bound $\mathcal{S}_{ub}>\mathcal{H}_k$, we need to consider all set sizes $|s|\in[l,u]$ in the size group $\mathcal{G}_l^u$ to bound the maximum matching position p_s. We leverage the following property of $\mathcal{S}_{ub}$ to avoid considering all set sizes.

Lemma 5.12. *For a size group $\mathcal{G}_l^u$ and a given matching position p_s, $\mathcal{S}_{ub}$ achieves its maximum for the set size upper bound u of $\mathcal{G}_l^u$.*

By Lemma 5.12, we find p_s for the upper bound size either using a binary search or using closed-form expressions (cf. Section 5.2).

5.3 Index Structure

The index structure remains the same as for Pail^F (cf. Section 4.3). When employing eager horizontal reading in Pail^L, we do not require the auxiliary data structure TPAux anymore: TPAux is required

for vertical reads in PAIL$^{\text{F}}$, but eager horizontal reading guarantees that such reads are never required in PAIL$^{\text{L}}$.

Construction Time and Space Complexity. By Lemma 5.5, size groups can be constructed in linear time. Assuming preprocessed data (a common practice in set similarity [25]), i.e., the set tokens are already sorted by some global order, indexing can also be done in linear time: We traverse all tokens of all sets by increasing token position p_s, skipping sets that have fewer than p_s tokens. This order ensures a correct order of the entries in the posting lists. Indexing a token t of set s requires looking up the size group $\mathcal{G}_l^u$ for $|s|$ and appending to the posting list of t. By implementing all lookups using hash tables, we thus achieve expected constant time to index one token. Hence, construction time and space complexity are linear in the total number of tokens over all sets.

Example 5.13. Consider Figure 7 with eager horizontal reading. As with ungrouped eager horizontal scanning, the inner bucket order is unaffected. The major difference compared to Example 5.10 is the additional read of s_5 and s_6 when accessing $\mathcal{B}^g(0,\mathcal{G}_6^7,2)$ as $p_s^\top = \infty$. In total, we read 6 candidates, 4 inner buckets, and have 4 index seeks, all of which are horizontal. This is the highest number of candidates and lowest number of index seeks among all the examples.

6 EXPERIMENTAL EVALUATION

In this section, we evaluate PAIL by studying its variants, comparing PAIL$^{\text{L}}$ – the best-performing variant on average – against current state-of-the-art algorithms and two baselines, and measuring scalability with respect to CPU cores.

6.1 Experimental Setup

We conducted all experiments on servers with one AMD EPYC 9354P CPU with 32 cores and 384 GiB memory, executing up to 8 jobs per server in parallel. PAIL and all competing algorithms were implemented in C++ and used GCC 12 with optimization (-O3) for compilation. We varied $k \in \{5,10,20,40,80\}$. To measure throughput, we randomly selected 10000 sets from each dataset as queries. We query for the $k+1$ nearest neighbors to account for every query finding itself. The same queries are used for all algorithms. For scalability experiments, the number of queries grows proportionally to the number of cores. We used a timeout of two hours; for algorithms exceeding the timeout, we report throughput over completed queries only.

Datasets. We use twelve datasets in our evaluation, selected from prior work to cover a broad range of dataset, set, and universe sizes. Previous studies have shown that these dimensions strongly influence algorithmic performance; in particular, different algorithms perform best on different datasets [36]. We refer to Mann *et al.* [25] for details on BMS, Kosarak, Livejournal, Netflix, and Orkut. For Lnonis1, we refer to Schmitt *et al.* [36]. Trackers is taken from Wang *et al.* [44]. Twitter is from Widmoser *et al.* [45]. Friendster and PMC are from Li *et al.* [21]. DBLP [7] contains metadata of publications scraped from DBLP represented as trees in XML; Movies [27] contains movie ratings from Amazon represented as trees in JSON. We convert the trees into sets by extracting all labels of the trees. Datasets are preprocessed as described by Mann *et al.* [25].

Table 1 summarizes the main characteristics of the datasets. Additionally, we also show the average cutoff similarity for $k \in \{5,80\}$.

Table 1: Dataset characteristics.

Dataset	#Sets	Set Size		Universe	k cut. sim.	
		avg.	99.9 pct.		5	80
BMS	$3.2{\cdot}10^5$	9.3	61	1657	0.52	0.40
DBLP	$2.0{\cdot}10^6$	26.1	48	$9.7{\cdot}10^6$	0.60	0.56
Friendster	$3.7{\cdot}10^7$	48.8	507	$6.5{\cdot}10^7$	0.09	0.03
Kosarak	$6.1{\cdot}10^5$	11.9	378	41270	0.48	0.38
Livejournal	$3.1{\cdot}10^6$	36.4	223	$7.5{\cdot}10^6$	0.24	0.19
Lnonis1	$8.2{\cdot}10^6$	20.3	39	44103	0.31	0.27
Movies	$8.5{\cdot}10^6$	23.1	28	$1.7{\cdot}10^7$	0.67	0.64
Netflix	$4.8{\cdot}10^5$	209.5	2515	17770	0.27	0.22
Orkut	$2.7{\cdot}10^6$	119.7	1000	$8.7{\cdot}10^6$	0.13	0.08
PMC	$5.5{\cdot}10^8$	16.5	76	$2.8{\cdot}10^7$	0.36	0.28
Trackers	$6.0{\cdot}10^6$	21.8	931	$2.7{\cdot}10^7$	0.26	0.17
Twitter	$4.6{\cdot}10^6$	12.4	31	$6.1{\cdot}10^5$	0.37	0.29

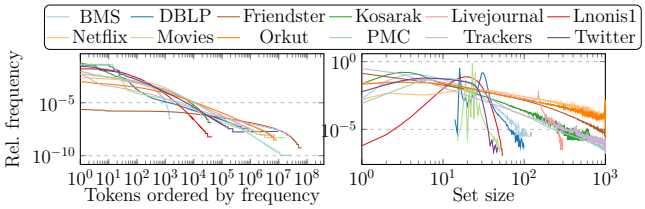

Figure 8: Token and size distributions of all datasets.

This value is of particular interest for kNN lookups as it also represents the difficulty of a dataset: kNN lookups are often similar in performance to range lookups performed for the cutoff similarity, and different algorithms tend to respond differently to particularly high or low cutoff similarities. Figure 8 shows the token frequency and set size distributions of all datasets.

Algorithms. We evaluate PAIL in four variations in our ablation study: (1) PAIL$^{\text{F}}$ is the algorithm proposed in Section 4 using the **F**ull positional filter and exact set sizes. (2) PAIL$^{\text{L}}$ is the **L**ightweight algorithm proposed in Section 5 using both size grouping and eager horizontal reading. (3) PAIL$^{\text{E}}$ only uses **E**ager horizontal reading. (4) PAIL$^{\text{G}}$ only uses size **G**rouping. For variations with grouping, we use the grouping procedure of Section 5.1 for self-joins with overhead constant $C_o = 32$ (one index seek $\approx$ 32 list reads) and assuming a uniform distribution of read partitions (cf. Corollary 5.4).

We compare our approach to different existing solutions and two baselines: LSCAN performs a simple linear scan to find the k nearest neighbors. It always reads the full dataset. LSCAN^{++} improves upon the bruteforce baseline by scanning the dataset in increasing size difference to the query and terminating once no remaining non-scanned set can be more similar than the current result. It also batches same-size queries to increase cache efficiency, with batch sizes between 4 and 32. PARTITION and PALLOC are partition-based approaches [37, 38]. Conceptually, both algorithms split each set into an infinite number of partitions and find candidates by matching partitions. In addition to exact matches of partitions used in PARTITION, PALLOC can search for matches in a Hamming radius of 1 inside each partition and uses a cost model to decide the next exploration step. DUALTRANS [51] transforms the sets into a fixed-dimensional

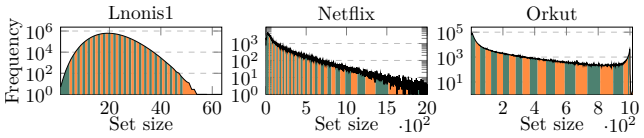

Figure 9: Distribution of size groups.

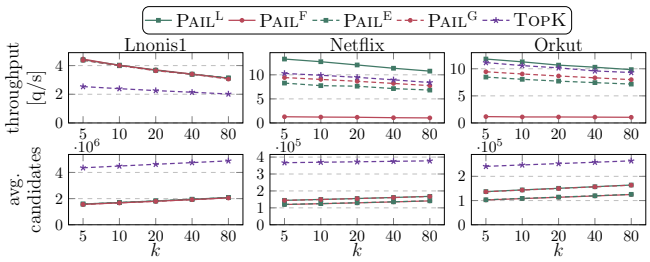

Figure 10: Throughput and candidates of PAIL variants.

vector space and uses an R-tree for indexing and querying. As suggested by the authors, we use two 16-dimensional vectors for set representation. We store 64 keys inside each node and bulk-load the R-tree [13, 19]. LES3 [21] partitions the datasets into groups represented by the union of their constituent sets. We use the authors' implementation [22] with their $0.5\% \cdot |R|$ partition heuristic and replace verification with the fast routine of Mann *et al.* [25], significantly increasing performance. TOPK [46] is a top-k set similarity join algorithm based on the prefix filter. We adapt it to the kNN setting. Our adaptation of TOPK can be viewed as a simplified form of PAIL without positional index information and only a single size group containing all sets, i.e., it uses a one-dimensional search space. TOPK has not appeared as a baseline in previous work focusing on kNN lookups. All algorithms are instances of Algorithm 1 and were implemented in C++. For scalability experiments, we leverage data parallelism and split the queries between all available cores using Intel oneAPI TBB [1].

6.2 Evaluation of Variants

We first analyze PAIL^F, PAIL^L, PAIL^E, and PAIL^G on datasets that stress different dataset characteristics. (1) A small universe size leads to long posting lists and weak token-based filtering [36]. (2) A large universe size makes individual tokens highly selective but increases the index-access cost of positional filtering. (3) Highly skewed size distributions reduce the effectiveness of size grouping compared to uniform or multi-modal distributions. We therefore select the following three datasets: (1) Lnonis1 is a large dataset with a small universe size and tightly clustered mid-sized sets, making prefix filtering difficult [36]. (2) Netflix is a small dataset with large sets, a small universe size, and a wide range of set sizes. It has long posting lists but allows effective size filtering and grouping. (3) Orkut is a large dataset with large sets, a large universe size, and a bathtub-shaped size distribution. Individual tokens prune well, but fine-grained positional filtering is costly.

Size Grouping. PAIL^F/PAIL^E and TOPK represent opposite extremes of size grouping, creating either one group per set size or a single group for all set sizes. PAIL^L and PAIL^G optimize the number of groups using the grouping procedure described in Section 5.1. For self-joins, the token distribution and the total number of tokens determine the

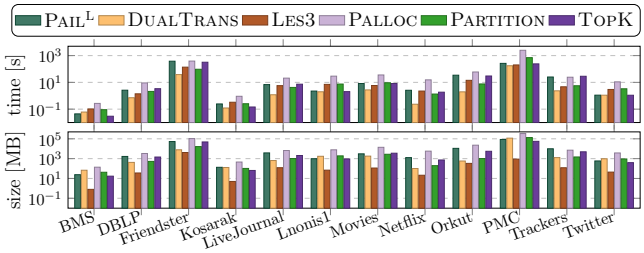

Figure 11: Indexing time and size of competing algorithms.

number of groups. Lnonis1, Netflix, and Orkut are split into 46, 42, and 22 groups, respectively. Lnonis1 and Netflix have small universes and long posting lists, causing PAIL^L and PAIL^G to use fine-grained partitions; on Lnonis1, nearly one group is created for each set size.

Figure 9 shows the absolute frequency of each set size and the size groups for all considered datasets; alternating colors represent groups. Netflix has a long tail of low-frequency, low-weight sizes common in real-world datasets [29]; Orkut has a *bathtub* distribution with frequencies first decreasing, then increasing near size 1000, and a few outliers up to size 40425. PAIL^L builds size groups of approximately equal weight: Netflix has fine-grained groups for common sizes and coarse groups for uncommon sizes: The last group ranges from 2238 to 17653. Orkut has more uniform size-group widths, except for the last group containing all outliers and spanning from 3053 to 40425.

Throughput and Candidates. We compare the variants based on their throughput and average candidates for a single lookup in Figure 10. In general, the number of candidates follows the order PAIL$^F \lesssim$ PAIL$^E <$ PAIL$^G \lesssim$ PAIL^L. Clearly, PAIL^F has the lowest and PAIL^L the highest number of candidates as they represent no relaxations and both relaxations of the positional filter. Between eager horizontal scanning (PAIL^E) and size grouping (PAIL^G), only size grouping has a major effect on the number of candidates with an increase of about 20%. Eager horizontal scanning has almost no effect on the candidates, suggesting that the positional bounds based on $\mathcal{H}_k$ keep the additional reads from eager horizontal reading minimal (cf. Section 5.2).

On Lnonis1, all PAIL variants perform similarly: The small universe size and clustered size distribution of Lnonis1 require many size groups for precise filtering, not leaving room for relaxations. As the range of set sizes is low, PAIL^F has low overhead.

The results on Netflix and Orkut are similar: PAIL^F performs worse than all other variants. The reason is the sparsity of the search space and the inefficient index access pattern of PAIL^F: For Netflix with k=80, a single read of an inverted index finds 0.04 entries on average. This is significantly lower than the average read sizes of 1.9, 8.7, and 261 of PAIL^G, PAIL^E, and PAIL^L, respectively. As the number of candidates is similar, PAIL^F requires vastly more index reads to collect the candidates, negatively affecting the runtime on datasets where the positional filter is highly effective.

Summary of the evaluation of variants. PAIL^F shows the worst and PAIL^L the best throughput on average. Across all twelve datasets, PAIL^F only rarely outperforms PAIL^L; most gains are below 5%, with a maximum of 22% on Movies. Both eager horizontal scanning and size grouping are required on sparse datasets to decrease the overhead of index accesses. The increase in candidates of size grouping is modest while eager horizontal scanning has virtually no downside.

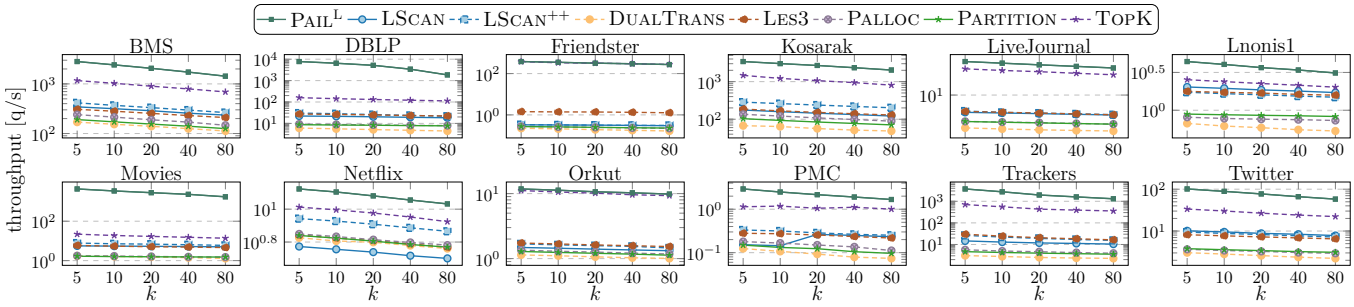

Figure 12: Throughput of competing algorithms.

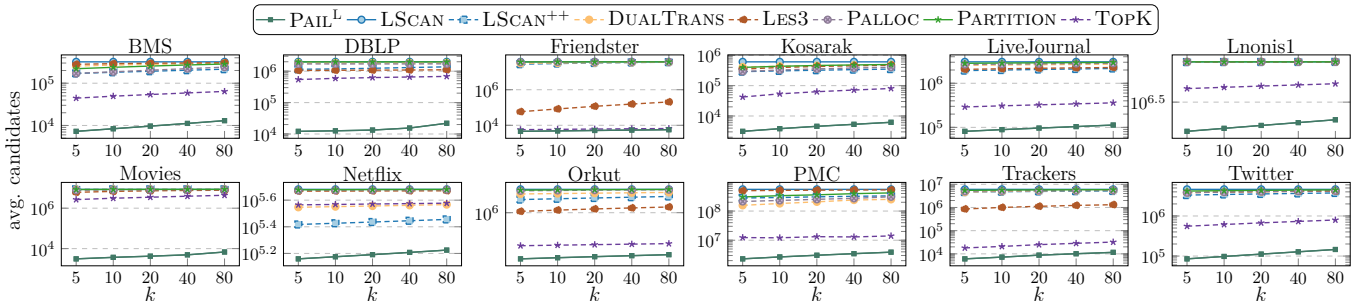

Figure 13: Average candidates for each query of competing algorithms.

6.3 Comparison with Existing Solutions

We now compare the performance of PAIL^L (the best-performing variant) to the existing solutions described in Section 6.

Indexing Time and Size. Figure 11 shows the indexing time and index size of all competing algorithms; LSCAN and LSCAN^{++} are (virtually) indexless and are therefore omitted. PAIL^L has similar indexing time (+8%) and index size (+47%) to TOPK across all datasets. Both store one index entry per token, with the additional overhead of PAIL^L stemming from its multiple size groups. DUALTRANS is the fastest algorithm to index, followed by TOPK (+100%) and PAIL^L (+116%). The set-partitioning-based methods PARTITION and PALLOC are both slower and require more memory than PAIL^L on average. LES3 computes and stores representatives only for large groups of sets and therefore requires the least memory of all algorithms.

Query Performance. Figure 12 shows the throughput and Figure 13 the average number of candidates per query. DUALTRANS and the partition-based algorithms PARTITION and PALLOC are outperformed by baselines (LSCAN and LSCAN^{++}) on all datasets and settings because they fail to filter enough sets: Compared to LSCAN, they reduce candidates by at most 50% in the best cases (BMS and Kosarak, $k=5$) and only 17% on average, while their index lookup and cost-model overhead further increases the gap. We therefore focus our subsequent discussion on PAIL^L, TOPK, LES3, and LSCAN^{++}.

On average over all datasets and values of k, PAIL^L is faster than LSCAN^{++}, TOPK, and LES3 by a factor of 148, 18.4, and 96, and reduces candidates to 0.1%, 1.3%, and 0.5%, respectively. PAIL^L performs particularly well on datasets with a large universe size like Movies, Friendster, and Trackers, where PAIL^L outperforms LSCAN^{++} and LES3 by factors of ≥1634, ≥24, and ≥36, respectively: The positional filter in PAIL^L prunes between 99.9% (Movies) and 99.99% (Friendster) of the candidates considered by LSCAN^{++}. Notably, the large universe of PMC and the small universe of Lnonis1 illustrate this effect: Although PMC contains 67× more sets than Lnonis1, PAIL^L achieves similar absolute throughput on both datasets.

Compared to our closest competitor, TOPK, PAIL^L shows near identical performance in terms of throughput and candidates on datasets with large universe sizes and low cutoff similarities, including Friendster, Orkut, and Livejournal in particular. Due to the low cutoff similarity, a major part of all size groups have to be considered for PAIL^L while a single token still has high pruning power due to the universe size even without considering its position. As TOPK can be viewed as PAIL^L without size grouping and positional filtering, their similar performance is expected on those datasets. On datasets that have large universe sizes and small-to-medium-sized sets, PAIL^L shows its best performance compared to TOPK, outperforming it by 791×, 42×, and 6.0× on Movies, DBLP and Twitter, respectively. This is caused by the multi-modal distribution of set sizes in Movies and DBLP, combined with their high cutoff similarities that favor PAIL^L. Due to its one-dimensional search space lacking size-based filtering, TOPK must verify 400× and 35× more candidates, respectively. On the remaining datasets, PAIL^L is faster than TOPK by 96% on average as its positional-based filtering and size grouping prunes 76% of TOPK's candidates. On Lnonis1, no algorithm is particularly effective in reducing the candidates compared to LSCAN due to Lnonis1's difficult characteristics of a small universe and clustered set sizes [36]. Only PAIL^L reduces the candidates by ≈80%, managing to outperform LSCAN^{++}, LES3, and TOPK by factors of 2.31, 2.21, and 1.6, respectively.

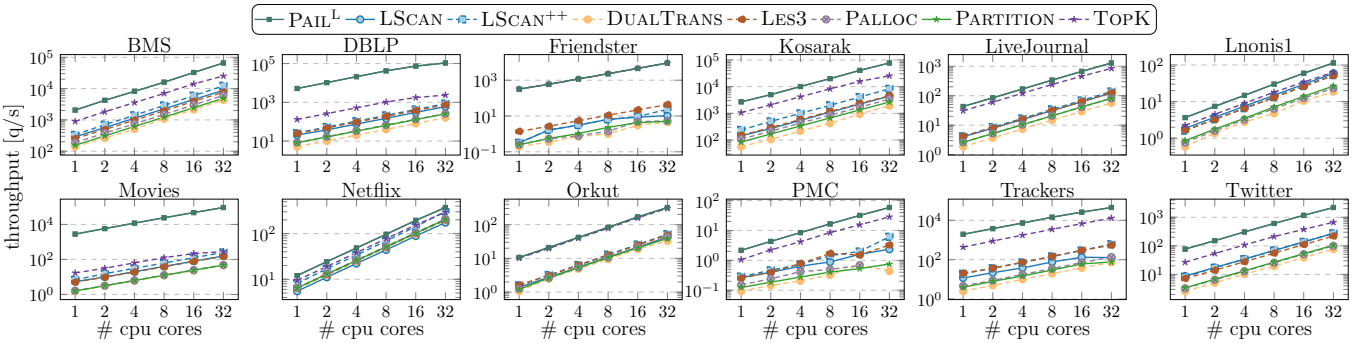

Figure 14: Scalability of competing algorithms with increasing CPU cores, $k = 20$.

6.4 Evaluation of Scalability

We evaluate scalability with respect to CPU cores in Figure 14. Most algorithms scale reasonably, reaching 28–32× their single-core performance on 32 cores. All algorithms are cache-intensive and -sensitive because the L3 cache is shared across cores: On the largest dataset PMC, all algorithms except LScan^{++}, Pail^{L}, and TopK scale poorly, achieving only 4–13× speedup on 32 cores. LScan^{++} uses batching, while Pail^{L} and TopK are selective enough to keep the memory small, achieving speedups of 21.5, 27.1, and 27.1, respectively.

7 RELATED WORK

We briefly summarize related work on set similarity and cover both search and join techniques for kNN, top-k, and the related field of range queries that often utilizes similar ideas.

Range Search and Join. There are two major approaches for exact set similarity range search and joins: (1) variations of the prefix-filter [6] and (2) algorithms based on the partition-and-enumeration framework [3]. Mann *et al.* [25] provide an experimental survey on numerous variations of prefix-filter based algorithms [4–6, 24, 41, 47]. More recent advancements using the prefix filter include techniques for grouping index entries and probing signatures [44], methods for efficiently enumerating subsets as signatures [11], and approaches that adaptively combine multiple algorithms based on dataset characteristics [35, 36, 45]. The partition-and-enumeration framework has also seen numerous implementations, including optimizing the selection of partitions and index-side enumeration [10], generalized query-side enumeration and optimization [31], and candidate pruning using combined partitions [30]. Both methods are complementary and work well for different dataset characteristics [36].

kNN and top-k Join. Xiao *et al.* [46] first use the prefix filter in the context of top-k set similarity joins. They introduce the event-based framework (that is closely related to the tree search algorithms of Roussopoulos *et al.* [32]) and propose multiple enhancements for self joins. SETJoin [40] enhances Xiao *et al.*'s [46] work by using a tighter similarity bound and utilizing additional filters. Furthermore, Yang *et al.* [48] probe multiple set positions for the same event to reduce the cost of heap operations. Sun *et al.* [37] describe an algorithm for range, kNN, and top-k joins based on the partition filter of Deng [10]. Zhang *et al.* [50] utilize a B-tree to index a numerical representation of sets and perform both range and kNN search queries. Sets are encoded as numbers by partitioning them and counting the number of tokens in each partition; these token counts are then combined into a single number. Zhang *et al.* [51] further enhance this approach by directly storing the token counts as a vector and indexing the vectors in an R-tree. Li *et al.* [21] use machine learning to partition the dataset into groups and use the group representatives for similarity estimation. We implemented all kNN and relevant top-k search algorithms and found Pail^{L} to outperform them significantly.

Approximate Similarity Search. Numerous algorithms for approximate similarity search have been proposed, including methods based on locality sensitive hashing [2, 16], learning-to-hash [43], and tree-based approaches using partitions [20]. Most recently, neighborhood graphs [12, 18, 23] have been the focus of attention for practical approximate kNN queries. However, these algorithms do not have guarantees and require parameter tuning to achieve high quality results. In this work, we focus on exact similarity search.

8 CONCLUSION

In this paper, we studied the k nearest neighbor search problem on the set domain. We identified two shortcomings in current approaches: Low filter selectivity on low similarity queries that are common for kNN search and inefficient candidate generation. To address these problems, we proposed Pail^{F} that leverages the positional filter in its index structure. We showed monotonicities in the positional filter and exploit them for early termination in the index traversal. We identified inefficiencies in the index accesses of Pail^{F} that we overcame in Pail^{L} with two relaxations of the positional filter: (1) Size grouping reduces the sparsity of the index, and (2) eager horizontal scanning reduces the number of index accesses. Experimental results show that Pail^{L} outperforms competing algorithms on real-world datasets with diverse characteristics by 18× to 455× on average.

ACKNOWLEDGMENTS

This research was funded in whole, or in part, by the Austrian Science Fund (FWF) 10.55776/P34962 and the Austrian ministries BMIMI, BMWET and the State of Upper Austria in the frame of the SCCH COMET competence center INTEGRATE (FFG 892418). For open access purposes, the author has applied a CC BY public copyright license to any author-accepted manuscript version arising from this submission.

REFERENCES

[1] 2006–2026. Intel oneAPI Threading Building Blocks. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html. Accessed: 2026-07-15.

[2] Alexandr Andoni and Piotr Indyk. 2008. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM* 51, 1 (2008), 117–122. https://doi.org/10.1145/1327452.1327494

[3] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. 2006. Efficient Exact Set-Similarity Joins. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim (Eds.). ACM, 918–929. http://dl.acm.org/citation.cfm?id=1164206

[4] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. 2007. Scaling up all pairs similarity search. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy (Eds.). ACM, 131–140. https://doi.org/10.1145/1242572.1242591

[5] Panagiotis Bouros, Shen Ge, and Nikos Mamoulis. 2012. Spatio-textual similarity joins. *Proc. VLDB Endow.* 6, 1 (2012), 1–12. https://doi.org/10.14778/2428536.2428537

[6] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. 2006. A Primitive Operator for Similarity Joins in Data Cleaning. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang (Eds.). IEEE Computer Society, 5. https://doi.org/10.1109/ICDE.2006.9

[7] Mohamed L Chouder, Stefano Rizzi, and Rachid Chalal. 2019. EXODuS: exploratory OLAP over document stores. *Information Systems* 79 (2019), 44–57.

[8] Tobias Christiani and Rasmus Pagh. 2017. Set similarity search beyond MinHash. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, Hamed Hatami, Pierre McKenzie, and Valerie King (Eds.). ACM, 1094–1107. https://doi.org/10.1145/3055399.3055443

[9] Abhinandan Das, Mayur Datar, Ashutosh Garg, and Shyamsundar Rajaram. 2007. Google news personalization: scalable online collaborative filtering. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy (Eds.). ACM, 271–280. https://doi.org/10.1145/1242572.1242610

[10] Dong Deng, Guoliang Li, He Wen, and Jianhua Feng. 2015. An Efficient Partition Based Method for Exact Set Similarity Joins. *Proc. VLDB Endow.* 9, 4 (2015), 360–371. https://doi.org/10.14778/2856318.2856330

[11] Dong Deng, Yufei Tao, and Guoliang Li. 2018. Overlap Set Similarity Joins with Theoretical Guarantees. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 905–920. https://doi.org/10.1145/3183713.3183748

[12] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *Proc. VLDB Endow.* 12, 5 (2019), 461–474. https://doi.org/10.14778/3303753.3303754

[13] Yván J. García, Mario Alberto López, and Scott T. Leutenegger. 1998. A Greedy Algorithm for Bulk Loading R-Trees. In *ACM-GIS '98, Proceedings of the 6th international symposium on Advances in Geographic Information Systems, November 6-7, 1998, Washington, DC, USA*, Robert Laurini, Kia Makki, and Niki Pissinou (Eds.). ACM, 163–164. https://doi.org/10.1145/288692.288723

[14] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. 2001. 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*, Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass (Eds.). Morgan Kaufmann, 491–500. http://www.vldb.org/conf/2001/P491.pdf

[15] Mauricio A. Hernández and Salvatore J. Stolfo. 1998. Real-world Data is Dirty: Data Cleansing and The Merge/Purge Problem. *Data Min. Knowl. Discov.* 2, 1 (1998), 9–37. https://doi.org/10.1023/A:1009761603038

[16] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*, Jeffrey Scott Vitter (Ed.). ACM, 604–613. https://doi.org/10.1145/276698.276876

[17] Piotr Indyk and Haike Xu. 2023. Worst-case Performance of Popular Approximate Nearest Neighbor Search Implementations: Guarantees and Limitations. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.), Vol. 36. 66239–66256. http://papers.nips.cc/paper_files/paper/2023/hash/d0ac28b79816b51124fcc804b2496a36-Abstract-Conference.html

[18] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc., 13748–13758. https://proceedings.neurips.cc/paper_files/paper/2019/file/09853c7fb1d3f8ee67a61b6bf4a7f8e6-Paper.pdf

[19] Scott T. Leutenegger, Jeffrey Edgington, and Mario Alberto López. 1997. STR: A Simple and Efficient Algorithm for R-Tree Packing. In *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997, Birmingham, UK*, W. A. Gray and Per-Åke Larson (Eds.). IEEE Computer Society, 497–506. https://doi.org/10.1109/ICDE.1997.582015

[20] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2020. Approximate Nearest Neighbor Search on High Dimensional Data - Experiments, Analyses, and Improvement. *IEEE Trans. Knowl. Data Eng.* 32, 8 (2020), 1475–1488. https://doi.org/10.1109/TKDE.2019.2909204

[21] Yifan Li, Xiaohui Yu, and Nick Koudas. 2021. LES3: Learning-based exact set similarity search. *Proc. VLDB Endow.* 14, 11 (2021), 2073–2086. https://doi.org/10.14778/3476249.3476263

[22] Yifan Li, Xiaohui Yu, and Nick Koudas. 2021. LES3: Learning-based exact set similarity search (Source Code). https://github.com/AwesomeYifan/learning-based-set-sim-search. Source code accompanying the paper. Accessed: 2026-05-05.

[23] Yury A. Malkov and Dmitry A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 42, 4 (2020), 824–836. https://doi.org/10.1109/TPAMI.2018.2889473

[24] Willi Mann and Nikolaus Augsten. 2014. PEL: Position-Enhanced Length Filter for Set Similarity Joins. In *Proceedings of the 26th GI-Workshop Grundlagen von Datenbanken, Bozen-Bolzano, Italy, October 21st to 24th, 2014 (CEUR Workshop Proceedings)*, Friederike Klan, Günther Specht, and Johann Gamper (Eds.), Vol. 1313. CEUR-WS.org, 89–94. https://ceur-ws.org/Vol-1313/paper_16.pdf

[25] Willi Mann, Nikolaus Augsten, and Panagiotis Bouros. 2016. An Empirical Evaluation of Set Similarity Join Techniques. *Proc. VLDB Endow.* 9, 9 (2016), 636–647. https://doi.org/10.14778/2947618.2947620

[26] Willi Mann, Nikolaus Augsten, Christian S. Jensen, and Mateusz Pawlik. 2025. SWOOP: top-k similarity joins over set streams. *VLDB J.* 34, 1 (2025), 13. https://doi.org/10.1007/S00778-024-00880-X

[27] Jianmo Ni, Jiacheng Li, and Julian J. McAuley. 2019. Justifying Recommendations using Distantly-Labeled Reviews and Fine-Grained Aspects. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Association for Computational Linguistics, 188–197. https://doi.org/10.18653/V1/D19-1018

[28] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, Alessandro Moschitti, Bo Pang, and Walter Daelemans (Eds.). ACL, 1532–1543. https://doi.org/10.3115/V1/D14-1162

[29] Casper Petersen, Jakob Grue Simonsen, and Christina Lioma. 2016. Power Law Distributions in Information Retrieval. *ACM Trans. Inf. Syst.* 34, 2 (2016), 8:1–8:37. https://doi.org/10.1145/2816815

[30] Jianbin Qin and Chuan Xiao. 2018. Pigeonring: A Principle for Faster Thresholded Similarity Search. *Proc. VLDB Endow.* 12, 1 (2018), 28–42. https://doi.org/10.14778/3275536.3275539

[31] Jianbin Qin, Chuan Xiao, Yaoshu Wang, Wei Wang, Xuemin Lin, Yoshiharu Ishikawa, and Guoren Wang. 2021. Generalizing the Pigeonhole Principle for Similarity Search in Hamming Space. *IEEE Trans. Knowl. Data Eng.* 33, 2 (2021), 489–505. https://doi.org/10.1109/TKDE.2019.2899597

[32] Nick Roussopoulos, Stephen Kelley, and Frédéic Vincent. 1995. Nearest Neighbor Queries. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, Michael J. Carey and Donovan A. Schneider (Eds.). ACM Press, 71–79. https://doi.org/10.1145/223784.223794

[33] Sunita Sarawagi and Alok Kirpal. 2004. Efficient set joins on similarity predicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, Gerhard Weikum, Arnd Christian König, and Stefan Deßloch (Eds.). ACM, 743–754. https://doi.org/10.1145/1007568.1007652

[34] Daniel Schmitt, Thomas Hütter, and Nikolaus Augsten. 2026. *PAIL: Efficient kNN Search on Set-Valued Attributes.* Technical Report. University of Salzburg. https://github.com/DatabaseGroup/pail-knn/blob/master/paper/paper-extended.pdf

[35] Daniel Ulrich Schmitt, Thomas Hütter, and Nikolaus Augsten. 2025. Extensible and Robust Evaluation of Similarity Queries. *Proc. VLDB Endow.* 18, 11 (2025), 3868–3882. https://www.vldb.org/pvldb/vol18/p3868-schmitt.pdf

[36] Daniel Ulrich Schmitt, Daniel Kocher, Nikolaus Augsten, Willi Mann, and Alexander Miller. 2023. A Two-Level Signature Scheme for Stable Set Similarity Joins. *Proc. VLDB Endow.* 16, 11 (2023), 2686–2698. https://doi.org/10.14778/3611479.3611480

[37] Ji Sun, Zeyuan Shang, Guoliang Li, Zhifeng Bao, and Dong Deng. 2019. Balance-Aware Distributed String Similarity-Based Query Processing System. *Proc. VLDB Endow.* 12, 9 (2019), 961–974. https://doi.org/10.14778/3329772.3329774

[38] Ji Sun, Zeyuan Shang, Guoliang Li, Dong Deng, and Zhifeng Bao. 2017. Dima: A Distributed In-Memory Similarity-Based Query Processing System. *Proc. VLDB Endow.* 10, 12 (2017), 1925–1928. https://doi.org/10.14778/3137765.3137810

[39] Martin Theobald, Jonathan Siddharth, and Andreas Paepcke. 2008. SpotSigs: robust and efficient near duplicate detection in large web collections. In *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2008, Singapore, July 20-24, 2008*, Sung-Hyon Myaeng, Douglas W. Oard, Fabrizio Sebastiani, Tat-Seng Chua, and Mun-Kew Leong (Eds.). ACM, 563–570. https://doi.org/10.1145/1390334.1390431

[40] Hongya Wang, Lihong Yang, and Yingyuan Xiao. 2020. SETJoin: a novel top-k similarity join algorithm. *Soft Comput.* 24, 19 (2020), 14577–14592. https://doi.org/10.1007/S00500-020-04807-W

[41] Jiannan Wang, Guoliang Li, and Jianhua Feng. 2012. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman (Eds.). ACM, 85–96. https://doi.org/10.1145/2213836.2213847

[42] Jianhua Wang, Jianye Yang, and Wenjie Zhang. 2021. Top-k Tree Similarity Join. In *CIKM '21: The 30th ACM International Conference on Information and Knowledge Management, Virtual Event, Queensland, Australia, November 1 - 5, 2021*, Gianluca Demartini, Guido Zuccon, J. Shane Culpepper, Zi Huang, and Hanghang Tong (Eds.). ACM, 1939–1948. https://doi.org/10.1145/3459637.3482304

[43] Jingdong Wang, Ting Zhang, Jingkuan Song, Nicu Sebe, and Heng Tao Shen. 2018. A Survey on Learning to Hash. *IEEE Trans. Pattern Anal. Mach. Intell.* 40, 4 (2018), 769–790. https://doi.org/10.1109/TPAMI.2017.2699960

[44] Xubo Wang, Lu Qin, Xuemin Lin, Ying Zhang, and Lijun Chang. 2019. Leveraging set relations in exact and dynamic set similarity join. *VLDB J.* 28, 2 (2019), 267–292. https://doi.org/10.1007/S00778-018-0529-2

[45] Manuel Widmoser, Daniel Kocher, Nikolaus Augsten, and Willi Mann. 2023. MetricJoin: Leveraging Metric Properties for Robust Exact Set Similarity Joins. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. IEEE, 1045–1058. https://doi.org/10.1109/ICDE55515.2023.00085

[46] Chuan Xiao, Wei Wang, Xuemin Lin, and Haichuan Shang. 2009. Top-k Set Similarity Joins. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, Yannis E. Ioannidis, Dik Lun Lee, and Raymond T. Ng (Eds.). IEEE Computer Society, 916–927. https://doi.org/10.1109/ICDE.2009.111

[47] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. 2011. Efficient similarity joins for near-duplicate detection. *ACM Trans. Database Syst.* 36, 3 (2011), 15:1–15:41. https://doi.org/10.1145/2000824.2000825

[48] Zhong Yang, Bolong Zheng, GuoHui Li, Xi Zhao, Xiaofang Zhou, and Christian S. Jensen. 2020. Adaptive Top-k Overlap Set Similarity Joins. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 1081–1092. https://doi.org/10.1109/ICDE48307.2020.00098

[49] Yongfeng Zhang, Guokun Lai, Min Zhang, Yi Zhang, Yiqun Liu, and Shaoping Ma. 2014. Explicit factor models for explainable recommendation based on phrase-level sentiment analysis. In *The 37th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '14, Gold Coast , QLD, Australia - July 06 - 11, 2014*, Shlomo Geva, Andrew Trotman, Peter Bruza, Charles L. A. Clarke, and Kalervo Järvelin (Eds.). ACM, 83–92. https://doi.org/10.1145/2600428.2609579

[50] Yong Zhang, Xiuxing Li, Jin Wang, Ying Zhang, Chunxiao Xing, and Xiaojie Yuan. 2017. An Efficient Framework for Exact Set Similarity Search Using Tree Structure Indexes. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*. IEEE Computer Society, 759–770. https://doi.org/10.1109/ICDE.2017.127

[51] Yong Zhang, Jiacheng Wu, Jin Wang, and Chunxiao Xing. 2020. A Transformation-Based Framework for KNN Set Similarity Search. *IEEE Trans. Knowl. Data Eng.* 32, 3 (2020), 409–423. https://doi.org/10.1109/TKDE.2018.2886189