LES^3: Learning-based Exact Set Similarity Search

Yifan Li
York University
yifanli@eecs.yorku.ca

Xiaohui Yu
York University
xhyu@yorku.ca

Nick Koudas
University of Toronto
koudas@cs.toronto.edu

ABSTRACT

Set similarity search is a problem of central interest to a wide variety of applications such as data cleaning and web search. Past approaches on set similarity search utilize either heavy indexing structures, incurring large search costs or indexes that produce large candidate sets. In this paper, we design a learning-based exact set similarity search approach, LES^3. Our approach first partitions sets into groups, and then utilizes a light-weight bitmap-like indexing structure, called token-group matrix (TGM), to organize groups and prune out candidates given a query set. In order to optimize pruning using the TGM, we analytically investigate the optimal partitioning strategy under certain distributional assumptions. Using these results, we then design a learning-based partitioning approach called L2P and an associated data representation encoding, PTR, to identify the partitions. We conduct extensive experiments on real and synthetic datasets to fully study LES^3, establishing the effectiveness and superiority over other applicable approaches.

PVLDB Reference Format:
Yifan Li, Xiaohui Yu, and Nick Koudas. LES^3: Learning-based Exact Set Similarity Search. PVLDB, 14(11): 2073 - 2086, 2021.
doi:10.14778/3476249.3476263

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/AwesomeYifan/learning-based-set-sim-search.

1 INTRODUCTION

Given a database $\mathcal{D}$ of sets each comprised of tokens (a token can be an arbitrary string from a given alphabet Σ, a unique identifier from a known domain, etc.), a single query set Q (consisting of tokens from the same domain), and a similarity measure $Sim(*)$, the problem of *set similarity search* is to identify from $\mathcal{D}$ those sets that are within a user defined similarity threshold to the query Q (range query) or k sets that are the most similar to Q (kNN query). This operation is essential to a wide spectrum of applications, such as data cleaning [27, 66], data integration [18, 23], query refinement [57], and digital trace analysis [44]. For example, a common task in data cleaning is to perform approximate string matching to identify near duplicates of a given query string. When strings are tokenized, the task of approximate string matching becomes a set similarity search problem. Given its prevalent use, efficient set similarity search is of paramount importance. A brute-force approach to supporting set similarity search is to scan all the sets in $\mathcal{D}$ and evaluate $Sim(*)$ between Q and each set in $\mathcal{D}$ to obtain the results. When $\mathcal{D}$ is large or such operations are carried out repeatedly, however, its efficiency becomes a major concern.

Existing proposals to improve the search performance adopt a filter-and-verify framework: in the filter step, candidate sets are generated based on indexes on $\mathcal{D}$, and the candidate sets are further examined, computing the similarity between Q and each candidate set in the verify step. Depending on the indexes used in the filter step, existing methods can be categorized into two groups: inverted index-based and tree-based. Inverted index-based methods build inverted index on tokens and only fetch those sets containing (a subset of) tokens present in the query set as candidates. Tree-based methods [72, 73] transform sets to scalars [72] or vectors [73] and insert them into B+-trees or R-trees, which are then used at query processing time to quickly identify the candidate sets. As verification of a candidate set can be done very efficiently under almost all well-known set similarity measures (e.g., Jaccard, Dice, Cosine similarity) incurring a cost linear in the size of the set, optimization of the filter step is critical. Unfortunately, existing methods either utilize heavy-weight indexes that incur expensive storage consumption and excessive scanning cost during filtering [73], or employ indexes that are light-weight but with very limited pruning efficiency leading to an overly large candidate set [72]. Therefore, existing approaches mostly do not solve the set similarity search problem effectively. In fact for realistically low similarity thresholds or large result sizes, as we demonstrate in our experiments, the brute-force approach may perform much better.

In this paper, we study the problem of set similarity search, and propose a new approach named LES^3 (short for Learning-based Exact Set Similarity Search) that strives to reduce the time needed for filtering and increase the pruning efficiency of the index structure at the same time. At a high level, our approach also adopts a filter-and-verify framework; however we advocate the partitioning of the sets in $\mathcal{D}$ into non-overlapping *groups* for filtering. What differentiates our approach from existing methods is that instead of building complex index structures that could become too expensive to utilize at run-time, we introduce a light-weight index structure called *token-group matrix* (TGM); this structure is essentially a collection of bit-maps, to organize all groups, yielding comparable or higher pruning efficiency with only a fraction of storage cost and thus highly scalable. The TGM captures the association between tokens and groups, and allows us to quickly compute an upper bound on the similarity between the query set Q and any set in a given group. Such upper bounds can then be used for *pruning* unrelated groups and directing search to the most promising groups.

As the search efficiency relies on the pruning efficiency of the TGM which in turn depends on how well the sets are partitioned, we formulate the construction of TGM as an optimization problem, that aims to identify the partitioning of sets that yields the highest pruning efficiency. We first analytically model the base case in

Proceedings of the VLDB Endowment, Vol. 14, No. 11 ISSN 2150-8097.
doi:10.14778/3476249.3476263

which every token has the same probability of appearing in any set. Our developments reveal that the optimal partitioning has two properties: balance and intra-group coherence. We then design a *general partitioning objective* (*GPO*) that strives to maximize the pruning efficiency, taking both properties into consideration.

We showcase that the optimal partitioning is NP-Hard and explore the use of algorithmic and machine learning-based methods to solve the optimization problem. Recent works [22, 38] have demonstrated that machine learning techniques have solid performance in learning the cumulative distribution function (CDF) of real data sets and this property can be used in important data management tasks such as indexing [43] and sorting [39]. We establish that machine learning techniques can also be utilized to produce superior solutions to hard optimization problems, central to other important indexing tasks such as those in support of set similarity search.

Complementary to existing works [22, 43] that utilize models such as piece-wise linear regression to learn a CDF, we explore models that are much better a fit, proposing a unique ensemble learning method suitable for progressive partitioning in our setting. The main difficulty in solving the optimization problem is that depending on $\mathcal{D}$, the number of groups needed for effective pruning can be large, so it is highly challenging to train a single network that would place any given set into one of these groups. As such, we propose a new learning framework named L2P (short for Learning to Partition) to address this challenge. L2P trains a cascade of Siamese networks to hierarchically partition the database $\mathcal{D}$ into increasingly finer groups until the desired number of groups is reached, resulting in 2^i groups at level i. The loss function for the Siamese network is specifically designed to minimize the distances between sets in the same group. As the input of a Siamese network has to be a vector, we devise a novel and efficient set representation method, *path-table representation* (PTR), that specifically caters to the needs of our optimization problem and proves to be a better fit than applicable embedding techniques. Although training ML models is known to be time-consuming, as will be shown in Section 7, L2P yields better partitioning results with much shorter processing time and only a small fraction of memory usage compared with other widely-adopted partitioning methods.

We fully develop the query processing algorithms for both range search and *k*NN search based on the TGM, and conduct extensive experiments on synthetic and real data sets to study the properties of our proposal and compare it against other applicable approaches. Our results demonstrate that both the proposed set representation method and the learning framework lead to much stronger pruning efficiency than competing methods. Overall, the proposed LES^3 method significantly outperforms the baseline methods in both memory-based and disk-based settings.

In summary, we make the following main contributions.

- We propose a learning-based approach, LES^3, for exact set similarity search, which partitions the database into groups to facilitate filtering. Central to LES^3 is TGM, a light-weight yet highly effective index that provides stronger pruning efficiency with less cost than state-of-the-art indexes.
- We formally analyze the partitioning of the database into groups, casting it as an optimization problem and discussing its distinction from well-studied clustering problems.
- We devise a novel learning framework, L2P, to solve the partitioning optimization problem, which yields significantly better partitioning results while incurring a small fraction of processing time and space cost compared with traditional algorithmic methods. L2P consists of a cascade of Siamese networks, an architecture that is able to effectively learn a partition of the dataset at different granularities, with up to thousands of groups at the finest level.
- We develop a carefully designed method for set representation, PTR, taking group separation into consideration. PTR theoretically and experimentally facilitates the training of L2P. Compared with other embedding techniques, PTR is orders of magnitude faster in computing set representations, and thus is more suitable for the target application where millions or billions of sets are involved (see Section 7).
- We experimentally study the performance of LES^3, L2P, and PTR, varying parameters of interest, including the network structure, number of groups and result size. We also examine the scalability of LES^3 utilizing real world large datasets in addition to previously used set similarity benchmarks. The proposed methods significantly and consistently outperform competing methods across a large variety of settings, providing up to 5 times faster query processing and requiring up to 90% less space in typical scenarios.

The rest of the paper is organized as follows. In Section 2, we define the terminologies to be used throughout the paper. Section 3 introduces the index structure, TGM. In Section 4, we discuss how the partitioning problem can be formulated as an optimization problem. In Section 5, we propose a machine learning framework to solve the optimization problem. Section 6 presents the query processing algorithms for set similarity search. The experimental evaluation is presented in Section 7. Section 8 discusses related work, and Section 9 concludes this paper.

2 PRELIMINARIES

A *set* is an unordered collection of elements called *tokens* (we also consider multiset which may contain duplicate tokens in the paper). We use S to denote an arbitrary set and t an arbitrary token. The database $\mathcal{D}$ is a collection of sets, and all tokens form the token universe $\mathcal{T}$. Two sets are considered similar if the overlap in their tokens exceeds a user-defined threshold. Usually, such overlap is normalized to account for the size difference between sets. Examples of such similarity measures include Jaccard, Dice, and Cosine similarity. To make our discussion more concrete, we focus on Jaccard similarity, and discuss how our approach can be applied to other similarity measures in Section 3.2. Next we give the formal problem definitions.

Definition 2.1. ***k*NN Search**. Given the database of sets $\mathcal{D}$, a set similarity measure $Sim(*)$, a query set[1] Q, and a result size k, find a collection $\mathcal{R}_Q^k \subseteq \mathcal{D}$ s.t. $|\mathcal{R}_Q^k| = k$ and $\forall S \in \mathcal{R}_Q^k$, $\forall S' \in \mathcal{D} - \mathcal{R}_Q^k$, $Sim(Q, S) \geq Sim(Q, S')$.

[1]Without loss of generality, we assume throughout the paper that a query set consists of tokens existing in $\mathcal{T}$ only. The case of query sets containing tokens not in $\mathcal{T}$ can be handled similarly and is discussed in Section 3.1.

Definition 2.2. **Range Search**. Given the database of sets $\mathcal{D}$, a set similarity measure $Sim(*)$, a query set Q, and a threshold δ, find a collection $\mathcal{R}_Q^\delta \subseteq \mathcal{D}$ s.t. $\forall S \in \mathcal{R}_Q^\delta$, $Sim(Q,S) \geq \delta$, and $\forall S' \in \mathcal{D} - \mathcal{R}_Q^\delta$, $Sim(Q,S') < \delta$.

Our goal is to accelerate the process of identifying the result collection $\mathcal{R}_Q^k$ or $\mathcal{R}_Q^\delta$ for the given k or δ. In general, the query answering process consists of a filtering step (choosing candidate sets) and a verification step (comparing candidate sets with the query set). The cost of the verification step depends directly on the pruning efficiency of the search process, which measures the proportion of sets in $\mathcal{D}$ being pruned in the filtering step.

Definition 2.3. **Pruning Efficiency (PE)**. Let $\mathcal{S}_Q$ be the collection of candidate sets for which the similarities to Q must be computed in the process of identifying $\mathcal{R}_Q^k$ or $\mathcal{R}_Q^\delta$. Then the pruning efficiency of query processing, denoted as PE, is $\frac{|\mathcal{D}|-(|\mathcal{S}_Q|-k)}{|\mathcal{D}|}$ for kNN query, or $\frac{|\mathcal{D}|-(|\mathcal{S}_Q|-|\mathcal{R}_Q^\delta|)}{|\mathcal{D}|}$ for range query.

Clearly PE falls in the range $[0, 1]$. All other things being equal, a higher PE leads to a lower verification cost. Our focus in this paper is therefore to design an approach for set similarity search that enjoys high PE and low filtering and verification cost.

3 TOKEN-GROUP MATRIX

The basic idea of our approach is to partition the sets in $\mathcal{D}$ into non-overlapping groups and index them properly, so that the search space can be pruned (i.e., certain groups can be quickly eliminated from further consideration) to speed up query processing. At the heart of our proposal is the token-group matrix (TGM), the index that records the relationship between tokens and the groups resulting from partitioning. In this section, we present the index structure and discuss its applicability across different similarity measures.

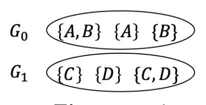
3.1 Index Structure

Assume for now, that $\mathcal{D}$ is already partitioned into n non-overlapping groups, $\mathcal{G}_1, \cdots \mathcal{G}_n$; we defer the discussion of the strategies for partitioning to the next section. The goals of the index are simplicity (so that it incurs little computational and storage overhead) and effectiveness (providing high pruning efficiency). To this end, the TGM, M, with size $n * |\mathcal{T}|$, is constructed in the following way:

$$M[g,t] = \begin{cases} 1, & \text{if } \exists S \in \mathcal{G}_g \text{ s.t. } t \in S \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

where $t \in [1, |\mathcal{T}|]$ and $g \in [1, n]$.

An example of TGM is given in Figure 1, where $\mathcal{T} = \{A, B, C, D\}$ and six sets are partitioned into two groups $\mathcal{G}_0$ and $\mathcal{G}_1$.



	A	B	C	D
G_0	1	1	0	0
G_1	0	0	1	1

Figure 1: An example of TGM

The design of the TGM is based on the observation that when deciding whether a group of sets is a candidate for a query set or not, the only information needed is the number of common tokens they share. Such information can be easily obtained by visiting some elements in M, and thus we can compute a *similarity upper bound* between a query set Q and a group of sets $\mathcal{G}_g$, which are useful in pruning the search space, as follows:

$$UB(Q, \mathcal{G}_g) = \frac{\sum_{t \in Q} M[g,t]}{|Q|} \tag{2}$$

Continuing the example above, we assume that the query set is $\{A\}$ and $\mathcal{G}_0$ and $\mathcal{G}_1$ in Figure 1 are candidates. Then the similarity bound between the query set and $\mathcal{G}_0$ is $\frac{M[\mathcal{G}_0,A]}{|\{A\}|} = 1$, and the upper bound for $\mathcal{G}_1$ is $\frac{M[\mathcal{G}_1,A]}{|\{A\}|} = 0$.

Although we assume Q contains tokens in $\mathcal{T}$ only, the case where this does not hold can be handled by letting $M[*, t'] = 0$ for $t' \notin \mathcal{T}$ in Equation (2). No further changes are required.

In the query processing step, if the upper bound of group $\mathcal{G}_g$ exceeds a threshold (can be δ in range query, or the minimal kNN similarity found so far), we compare all sets in $\mathcal{G}_g$ with the query set. The time complexity of computing the similarity bounds between the query set and all groups of sets is $O(n|Q|)$. It in general costs much less than computing the similarity between the query set and each set in $\mathcal{D}$, as the number of groups is usually orders of magnitude smaller than $|\mathcal{D}|$.

In terms of space consumption, each element in TGM is represented by a bit, and TGM is essentially a bitmap index. It is evident that M is usually a very sparse matrix as each set usually contains a very small portion of the tokens from the universe. When necessary, many existing compression techniques [58, 59] can be employed to reduce the size of M.

3.2 Applicability

Although Equation (2) is computed assuming Jaccard index as the similarity metric, TGM works with many other set similarity measures as well, including measures that do not follow the triangle inequality, such as cosine similarity.

Theorem 3.1. *For $\forall Q, S \subseteq \mathcal{T}$, let $R = Q \cap S$. TGM is applicable to set similarity search tasks with measure $Sim(*)$ if*

(1) $Sim(Q,R) \geq Sim(Q,S)$, and
(2) $\forall R' \subset R$, $Sim(Q,R) \geq Sim(Q,R')$

Proof. We prove that for an arbitrary query set Q and an arbitrary group $\mathcal{G}_g$, we can compute a similarity upper bound $UB(Q, \mathcal{G}_g)$ with TGM using Equation (2) such that $\forall S \in \mathcal{G}_g$, $UB(Q, \mathcal{G}_g) \geq Sim(Q,S)$. Let $R = \{t | t \in Q \wedge \exists S \in \mathcal{G}_g, t \in S\}$, then we know $\forall S \in \mathcal{G}_g, Q \cap S \subseteq R$. If $Q \cap S = R$, then clearly $Sim(Q,R) \geq Sim(Q,S)$; if $Q \cap S = R' \subset R$, then $Sim(Q,R) \geq Sim(Q,R') \geq Sim(Q,S)$. In either case, $Sim(Q,R)$ upper bounds $Sim(Q,S)$, and thus we can use $Sim(Q,R)$ as $UB(Q, \mathcal{G}_g)$. Since it is possible that $R = S$, in which case $Sim(Q,R) = Sim(Q,S)$, the bound $UB(Q, \mathcal{G}_g)$ is tight, even in multiset settings. $\square$

For example, let $Q = \{t_1, t_2, t_3\}$ and $Q \cap S = \{t_1, t_2\}$. Then with Jaccard similarity, the set with the maximal similarity to Q is $\{t_1, t_2\}$ and the upper bound is $\frac{2}{3}$; with cosine similarity, the set with the maximal similarity to Q is also $\{t_1, t_2\}$, but the upper bound is $\frac{2}{\sqrt{3*2}} \approx 0.82$. Note that although most similarity measures satisfy the TGM Applicability Property, some exceptions do exist. One such example is the learned metric [37] which takes two samples (e.g., images) as the input and predicts their similarity.

In what follows, we call the two properties listed in Theorem 3.1 the *TGM Applicability Property*. Note that the token universe $\mathcal{T}$ does not need to be static. We will discuss how to adapt TGM to deal with cases where $\mathcal{T}$ is dynamically changing in Section 6.

4 OPTIMIZING PARTITIONING

We analyze how to optimize partitioning to provide higher pruning efficiency. We discuss desired properties of the partitioning, and develop the objective function for the partitioning optimization problem that will guide the development of effective partitioning strategies. To make our formal analysis tractable, we make assumptions regarding the token distribution; nonetheless, as will be demonstrated by our experimental results in Section 7, the optimization objectives and strategies thus developed are also expected to perform well when the assumptions do not hold.

4.1 The Case of Uniform Token Distribution

We formally analyze the effect of partitioning on pruning efficiency when the following assumption on token distribution holds.

Definition 4.1. **Uniform Token Distribution Assumption**. The probabilities that different tokens belong to an arbitrary set are identical and independent. More specifically, $\forall t_i, t_j \in \mathcal{T}$, $\forall S \in \mathcal{D}$, $P(t_i \in s) = P(t_j \in S)$, and $P(t_i \in S | t_j \in s) = P(t_i \in S | t_j \notin S)$.

For an arbitrary query Q, the expected pruning efficiency can be computed as follows:

$$E[PE] = \sum_{g=1}^{n} |\mathcal{G}_g|(1 - UB(Q, \mathcal{G}_g)) \tag{3}$$

Given the way the TGM is constructed, we rewrite Equation (2) in the following way to ease subsequent discussion:

$$UB(Q, \mathcal{G}_g) = \frac{\sum_{t \in Q} M[t,g]}{|Q|} = \frac{|GS_g \cap Q|}{|Q|},\ GS_g = \bigcup_{S \in \mathcal{G}_g} S \tag{4}$$

Accordingly, we rewrite Equation (3) as follows:

$$E[PE] = \sum_{g=1}^{n} |\mathcal{G}_g|(1 - \frac{|GS_g \cap Q|}{|Q|}) \tag{5}$$

As we assume Q follows the same distribution as $\mathcal{D}$, $E[PE]$ over all possible Q can be estimated by the following equation:

$$\frac{\sum_{Q \in \mathcal{D}} \sum_{g=1}^{n} |\mathcal{G}_g|(1 - \frac{|GS_g \cap Q|}{|Q|})}{|\mathcal{D}|} \tag{6}$$

Since $|\mathcal{D}|$ is a constant, we keep the nominator of Equation (6) only, and adjust the order as follows:

$$\sum_{g=1}^{n} |\mathcal{G}_g| \sum_{Q \in \mathcal{D}} (1 - \frac{|GS_g \cap Q|}{|Q|}) \tag{7}$$

To ease following analysis, we define term F in Equation (8), and claim that maximizing Equation (7) (and thus maximizing the pruning efficiency) is equivalent to minimizing F:

$$F = \sum_{g=1}^{n} |\mathcal{G}_g| \sum_{Q \in \mathcal{D}} \frac{|GS_g \cap Q|}{|Q|} \tag{8}$$

We derive several properties regarding the partitioning from Equation (8) so as to design practical partitioning algorithms.

Theorem 4.2. *In a database that satisfies the uniform token distribution assumption, the partitioning that minimizes Equation (8) produces groups with equal size (or differ by at most 1).*

Proof. We consider the special case where $\mathcal{D}$ is partitioned into two groups $\mathcal{G}_1$ and $\mathcal{G}_2$, and $|\mathcal{G}_1| \le |\mathcal{G}_2|$. The F value of such a partitioning is:

$$F = F_1 + F_2 = |\mathcal{G}_1| \sum_{Q \in \mathcal{D}} \frac{|GS_1 \cap Q|}{|Q|} + |\mathcal{G}_2| \sum_{Q \in \mathcal{D}} \frac{|GS_2 \cap Q|}{|Q|} \tag{9}$$

Next we move a set S from $\mathcal{G}_1$ to $\mathcal{G}_2$ and prove that such movement increases the F value. We know that if S is moved from $\mathcal{G}_1$ to $\mathcal{G}_2$, F_1 would decrease and F_2 would increase. And since $|\mathcal{G}_1| \le |\mathcal{G}_2|$, equivalently we can prove that $|\mathcal{G}_i| \sum_{Q \in \mathcal{D}} \frac{|GS_i \cap Q|}{|Q|}$ grows super-linearly with respect to $|\mathcal{G}_i|$, or $\sum_{Q \in \mathcal{D}} \frac{|GS_i \cap Q|}{|Q|}$ grows with $|\mathcal{G}_i|$. Given the construction of GS_i in Equation (4), this is evidently true. Therefore, the F value increases after the movement of S.

The above discussion can be naturally extended to multi-groups by moving one set from a small group to a large group each time, with the F value increasing and the pruning efficiency decreasing during the process. In conclusion, balanced partitioning results yield the highest pruning efficiency. □

Even though the optimal partitioning is expected to produce groups with almost equal sizes, evidently balance is not the only desired property, according to Equation (8). We temporarily omit the $|\mathcal{G}_g|$ in Equation (8) and discuss other properties the partitioning must satisfy in order to provide higher pruning efficiency.

Theorem 4.3. *In a database that satisfies the uniform token distribution assumption, the partitioning that minimizes the following objective provides the highest pruning efficiency:*

$$\sum_{g=1}^{n} | \bigcup_{S \in \mathcal{G}_g} S| \tag{10}$$

Proof. Given the assumption that all groups are balanced, minimizing Equation (8) is equivalent to minimizing

$$\sum_{g=1}^{n} \sum_{Q \in \mathcal{D}} \frac{|GS_g \cap Q|}{|Q|} \tag{11}$$

Since Q follows the uniform token distribution as well, which means that all tokens appear in Q with the same probability, Equation (11) is proportional to the following equation:

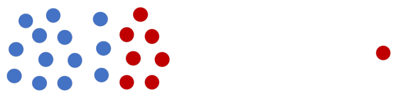
$$\sum_{g=1}^{n} |GS_g \cap \mathcal{T}| = \sum_{g=1}^{n} |GS_g| = \sum_{g=1}^{n} | \bigcup_{S \in \mathcal{G}_g} S|, \tag{12}$$

where $\mathcal{T}$ denotes the token universe.

Thus, we can maximize PE by minimizing Equation (10). □

In summary, we have the following two desired properties regarding the partitioning of database $\mathcal{D}$.

- Property 1: Groups are balanced;
- Property 2: $U = \sum_{g=1}^{n} | \bigcup_{S \in \mathcal{G}_g} S|$ is minimized.


(a) Groups produced by *GPO*

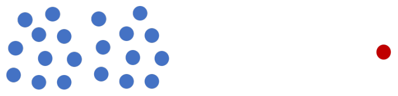

(b) Groups produced by k-medians

Figure 2: Comparison of different partitioning results

4.2 The General Case

The analysis in the preceding section depends on the uniform token distribution assumption. In real-life datasets, this assumption does not hold. However, following the same methodology to derive a formal treatment of an arbitrary set/token distribution would be challenging as a realistic mathematical model of arbitrary set/token distributions would be hard to justify. Although the two properties identified above may not be true for optimal partitioning in the general case, we draw inspirations from them and propose a heuristic objective function that strives to maximize PE.

In essence Property 2 directs that the more similar (in terms of token composition) the sets are within a group, the better. We thus design a *general partitioning objective (GPO)* we wish to minimize reflecting this property:

$$GPO = \sum_{g=1}^{n} \sum_{S_x \in \mathcal{G}_g} \sum_{S_y \in \mathcal{G}_g} (1 - Sim(S_x, S_y)), \tag{13}$$

where $Sim(*)$ can be any measures discussed in Section 3.2.

Intuitively, *GPO* aims to minimize the sum of the intra-group pair-wise distances, where *distance* is defined as $1 - Sim(*)$. This is similar to Property 2. As an example, consider two groups $\mathcal{G}_i$ and $\mathcal{G}_j$ with $|\mathcal{G}_i| = |\mathcal{G}_j|$. Assume that $Sim(*)$ is Jaccard similarity, and we are to place a new set S into one of the two groups. Then, if $\sum_{S_i \in \mathcal{G}_i}(1 - Sim(S, S_i)) < \sum_{S_j \in \mathcal{G}_j}(1 - Sim(S, S_j))$, that would mean S shares more common tokens with sets in group $\mathcal{G}_i$, and thus inserting S into group $\mathcal{G}_i$ helps to minimize U.

However, considering only Property 2 results in highly skewed partitioning results, as placing all sets in the same group provides the minimal U (which equals to $|\mathcal{T}|$). Evidently, Property 1 is used to prevent such skewed partitioning in the uniform case. Luckily, *GPO* enjoys a similar functionality: placing all sets in the same group provides $GPO = \sum_{S_x, S_y \in \mathcal{D}}(1 - Sim(S_x, S_y))$, which is the maximal possible *GPO*, and thus such a partitioning is never the optimal in terms of *GPO*. Thus, the design of *GPO* implicitly incorporates both Property 1 and Property 2.

In order to better appreciate the distinctive value of the proposed partitioning objective, we compare *GPO* with k-medians, perhaps the most popular clustering technique, and show by an example how optimizing *GPO* leads to better results. We use a database of 21 sets, and the partitioning results based on different clustering objectives are given in Figure 2. Each set is represented by a point in the plot, and to better visualize the results we replace $(1 - Sim(*))$ with Euclidean distance.

Assume that the query is to identify the nearest neighbors of all 21 points. According to the search strategy of LES^3 given in Section 3.1, all points in the same group are candidates of each other. Therefore, with the clustering results given in Figure 2(b), the total number of distance calculations is $20 * 20 + 1 * 1 = 401$, while with the partitioning results in Figure 2(a) the number is $13 * 13 + 8 * 8 = 233$. Clearly, the results based on Equation (13) have better pruning efficiency.

THEOREM 4.4. *Given a database of sets $\mathcal{D}$ minimizing GPO on $\mathcal{D}$ is NP-complete.*

PROOF. We give a brief proof by showing that minimizing *GPO* is essentially a 0-1 integer linear programming problem, which has been shown to be NP-complete [10]. More specifically, minimizing *GPO* is equivalent to solving the following optimization problem:

$$\begin{aligned} \text{maximize} \quad & \mathbf{e}_{|\mathcal{D}|} \cdot [\mathbf{A} \cdot \mathbf{A}^\intercal \odot \mathbf{D}] \cdot \mathbf{e}_{|\mathcal{D}|}^\intercal \\ \text{subject to} \quad & \mathbf{e}_n \cdot \mathbf{A}^\intercal = \mathbf{e}_{|\mathcal{D}|} \end{aligned} \tag{14}$$

where $\mathbf{A}$ is a $|\mathcal{D}| \times n$ matrix and $\mathbf{A}[x, g] = 1$ if set S_x belongs to group $\mathcal{G}_g$ and $\mathbf{A}[x, g] = 0$ otherwise, and $\mathbf{D}$ of size $|\mathcal{D}| \times |\mathcal{D}|$ denotes the distance matrix where $\mathbf{D}[x, y] = 1 - Sim(S_x, S_y)$, and $\mathbf{e}_i$ is a row vector of length i filled with ones. The goal is to find the $\mathbf{A}$ which satisfies the constraint and maximizes the objective.

The intuition behind Equation (14) can be described as follows: $\mathbf{A} \cdot \mathbf{A}^\intercal$ is a $|\mathcal{D}| \times |\mathcal{D}|$ matrix such that the value at position $[x, y]$ is 1 if S_x and S_y belong to the same group, and 0 otherwise. The element-wise product between $\mathbf{A} \cdot \mathbf{A}^\intercal$ and $\mathbf{D}$ masks out those pair-wise distances between sets belonging to different groups, and $\mathbf{e}_{|\mathcal{D}|} \cdot [\mathbf{A} \cdot \mathbf{A}^\intercal \odot \mathbf{D}] \cdot \mathbf{e}_{|\mathcal{D}|}^\intercal$ sums the remaining distances, which is the same objective as *GPO*. The constraint $\mathbf{e}_n \cdot \mathbf{A}^\intercal = \mathbf{e}_{|\mathcal{D}|}$ guarantees that each set belongs to one and only one group. Therefore, minimizing *GPO* is equivalent to solving Equation (14), which completes the proof. □

4.3 Algorithmic Approaches

In this section we propose several algorithmic approaches based on existing applicable clustering methods, which are expected to yield groups with low *GPO* values. More specifically, we design a graph cut-based approach (PAR-G), a centroid-based approach (PAR-C), and a hierarchical approach (PAR-H).

4.3.1 Graph cut-based method (PAR-G). When k or δ is fixed, it is possible to build an index structure specifically optimized for the workload. Dong et al. [19] propose a graph cut-based solution for (approximate) nearest neighbor search in $\mathbb{R}^d$ space by linking each point to its neighbors and partitioning the resulting graph into balanced subgraphs with the number of edges crossing different subgraphs minimized. Such a partitioning is shown to yield high pruning efficiency. Inspired by their approach, we design PAR-G, which takes k or δ as one of its inputs, as follows:

(1) **Similarity graph construction.** For a given k in kNN query, construct the similarity graph, $G_{\mathcal{D}}$, of $\mathcal{D}$, such that $\forall S_x \in \mathcal{D}$, there exists a corresponding vertex V_x in $G_{\mathcal{D}}$, and $\forall S_y \in \mathcal{D}$, if S_y is one of the k nearest neighbors of S_x, there is an edge between V_x and V_y in $G_{\mathcal{D}}$. For a given δ in range query, there is an edge between V_x and V_y if $Sim(S_x, S_y) \geq \delta$.
(2) **Graph cut.** Partition $G_{\mathcal{D}}$ into n balanced subgraphs while minimizing the number of edges crossing different subgraphs. This can be done with existing graph partitioners [24, 34].

4.3.2 *Centroid-based method (PAR-C).*

Centroid-based methods [29] are iterative algorithms which at each iteration relocate an elements into a different cluster if such relocation improves the overall objective function. For our case, let $\phi(\mathcal{G}) = \sum_{S_x,S_y \in \mathcal{G}}(1 - Sim(S_x, S_y))$ be the sum of all pair-wise distances[2] in group $\mathcal{G}$, $S \in \mathcal{G}_i$ an arbitrary set, and $\Delta(S, \mathcal{G}_i, \mathcal{G}_j) = \phi(\mathcal{G}_i \setminus S) + \phi(\mathcal{G}_j \cup S) - \phi(\mathcal{G}_i) - \phi(\mathcal{G}_j)$ the decrease of *GPO* after moving S from $\mathcal{G}_i$ to $\mathcal{G}_j$ ($i, j \in [1, n]$). To be more specific, our method works as follows:

(1) **Initialization.** Randomly partition $\mathcal{D}$ into n groups;
(2) **Relocation.** For each $S \in \mathcal{D}$, suppose $S \in \mathcal{G}_i$. Find group $\mathcal{G}_j^*$ such that $\Delta(S, \mathcal{G}_i, \mathcal{G}_j^*) = \max_{S,\mathcal{G}_i,\mathcal{G}_j} \Delta(S, \mathcal{G}_i, \mathcal{G}_j)$ (denoted as "the best group"). If $\Delta(S, \mathcal{G}_i, \mathcal{G}_j^*) > 0$, relocate S from $\mathcal{G}_i$ to $\mathcal{G}_j^*$. Repeat this step until no sets are relocated in an iteration.
(3) **Simplification.** Considering the data size we deal with (see Section 7), finding "the best group" at each iteration would be too expensive. Therefore, we adopt the "first-improvement" variant [63] of the algorithm, i.e., pick the first group $\mathcal{G}_j$ with $\Delta(S, \mathcal{G}_i, \mathcal{G}_j) > 0$ rather than the best group.

4.3.3 *Divisive clustering method (PAR-D).*

Divisive clustering methods [35] start from the single cluster containing all elements and repeatedly split clusters until a desired number of clusters is reached. We reuse $\phi(\mathcal{G})$ introduced in Section 4.3.2 and use $idv\_d(S)$ to denote the sum of distances between S and all other sets in the same group as S. PAR-D works as follows:

(1) **Initialization.** Take $\mathcal{D}$ as the initial group.
(2) **Splitting.** Find group $\mathcal{G}^* = \arg\max_{\mathcal{G}_i \in \{\mathcal{G}_1,\mathcal{G}_2,\cdots\}} \phi(\mathcal{G}_i)$, where $\{\mathcal{G}_1, \mathcal{G}_2, \cdots\}$ denotes all current groups. Find set $S^* = \arg\max_{S \in \mathcal{G}^*} idv\_d(S)$. Create a new group $\mathcal{G}^{new} = \{S^*\}$. For all other sets $S' \in \mathcal{G}^*$, move S' to $\mathcal{G}^{new}$ if such movement reduces the overall *GPO*. Repeat this step until there are n groups.
(3) **Simplification.** Considering the data size we deal with, instead of finding S^*, we choose a random set in $\mathcal{G}^*$ to initialize $\mathcal{G}^{new}$, which is commonly adopted for group splitting [26].

4.3.4 *Agglomerative clustering method (PAR-A).*

Agglomerative clustering [55] works in a bottom-up fashion by initially treating each element as a cluster and repeatedly merging clusters until a desired number of clusters is reached. We reuse $\phi(\mathcal{G})$ introduced in Section 4.3.2 to denote the sum of all pair-wise distances in group $\mathcal{G}$. PAR-A works as follows:

(1) **Initialization.** Create group $\mathcal{G}_i = \{S_i\}$ for each $S_i \in \mathcal{D}$.
(2) **Merging.** Find groups $\mathcal{G}_1^*, \mathcal{G}_2^* = \arg\min_{\mathcal{G}_i,\mathcal{G}_j \in \{\mathcal{G}_1,\mathcal{G}_2,\cdots\}} \phi(\mathcal{G}_i \cup \mathcal{G}_j)$, where $\{\mathcal{G}_1, \mathcal{G}_2, \cdots\}$ denotes all current groups and $i \neq j$. Create a new group $\mathcal{G}^{new} = \mathcal{G}_1^* \cup \mathcal{G}_2^*$ and remove groups $\mathcal{G}_1^*$ and $\mathcal{G}_2^*$. Repeat this step until there are n groups.
(3) **Simplification.** Considering the data size we deal with, we adopt the heuristic that merging smaller groups (groups with smaller number of sets) usually results in smaller values of $\phi(\mathcal{G}_i \cup \mathcal{G}_j)$ and restrict that $\mathcal{G}_1^*$ is the smallest group (breaking ties randomly), and thus only $\mathcal{G}_2^*$ needs to be identified in each iteration.

[2]Note that repetitively calculating $\phi(\mathcal{G})$ during the partitioning process is computational prohibitive, and thus we approximate $\phi(\mathcal{G})$ with randomly selected sets in $\mathcal{G}$ in the experiment (Section 7.4).

As we will demonstrate in our experimental study in Section 7, these heuristic approaches do not provide satisfactory performance. The structure of the *GPO* problem objective does not resemble those targeted by well-studied clustering algorithms. In the next section, we explore the use of ML to perform such a partitioning.

5 L2P: LEARN TO PARTITION SETS INTO GROUPS

As pointed out by Bengio et al. [5], a machine learning approach to combinatorial optimization problems with well-defined objective functions, such as the Travelling Salesman Problem, has proven to be more promising than classical optimization methods with hand-wired rules in many scenarios, for the reason that it adapts solutions to the data and thus can uncover patterns in the specific problem instance as opposed to solving a general problem for every instance. It is widely agreed [4, 64] that ML-based methods are especially valuable in cases where expert knowledge of the problem domain may not be sufficient and some algorithmic decisions may not give satisfactory results. Our goal in this section, therefore, is to develop a machine learning method to optimize *GPO*.

5.1 Siamese Networks

Considering that the goal of optimizing *GPO* is to maximize the overall intra-group similarity, we adopt Siamese networks [8, 20] to solve the partitioning task. Siamese networks have been successfully utilized in deep metric learning tasks [50, 52] in computer vision, capturing both intra-class similarity and inter-class discrimination in many challenging tasks including face recognition.

We design a Siamese network as shown in Figure 3 to learn the optimal partitioning. It consists of a pair of twin networks sharing the same set of parameters working in tandem on two inputs and generate two comparable outputs.

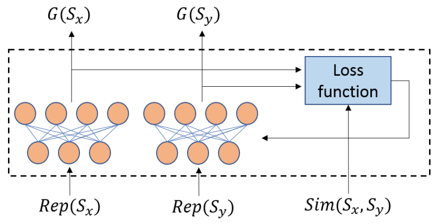


Figure 3: Siamese network

We use $Rep(S_x)$ and $Rep(S_y)$ to denote the vector representations of two sets S_x and S_y, a pair of inputs to the twin networks, and use $G(S_x)$ and $G(S_y)$ to represent their respective group assignment indicted by the outputs of the twin networks respectively. Following Equation (13) we define the loss function of the Siamese network as follows:

$$loss(S_x, S_y) = \begin{cases} (1 - Sim(S_x, S_y)), & \text{if } G(S_x) = G(S_y) \\ 0, & \text{otherwise} \end{cases} \tag{15}$$

Equation (15) minimizes the intra-group dissimilarities by summing $(1 - sim(S_x, S_y))$ as the losses, and penalizes imbalanced groups by counting pairwise dissimilarities only between sets in the same group. We use an example to illustrate how Equation (15) penalizes imbalanced partitioning. Suppose there are N sets and dissimilarity between any pair of sets is the same at d. The task is to partition these sets into 2 groups, containing N_1 and N_2 sets respectively ($N_1 + N_2 = N$). Then the overall loss is $\frac{N_1(N_1-1)d}{2} + \frac{N_2(N_2-1)d}{2} = \frac{d}{2}[N_1(N_1-1) + N_2(N_2-1)] = \frac{d}{2}(N_1^2 + N_2^2 - N) \geq \frac{d}{2}(\frac{(N_1+N_2)^2}{2} - N) = \frac{d}{2}(\frac{N^2}{2} - N)$, and the bound is tight when $N_1 = N_2$. Therefore, Equation (15) favors balanced partitioning.

By training the Siamese network with sufficient samples drawn from $\mathcal{D}$, theoretically we can minimize the overall distances between all pairs of sets which belong to the same group, and thus the Siamese network is expected to give the partitioning result in which *GPO* is minimized. Practically, however, we expect to achieve near-optimal partitioning only as the network is essentially performing local search.

5.2 Framework

Although using Siamese networks to solve the optimization problem is a promising approach, training such networks turns out to be difficult for the following reasons:

(1) When dealing with real world data, we may need to partition sets into thousands of groups. Therefore, for an input set S_x, the network needs to make prediction on which group S_x belongs to, among a collection of thousands of groups. It is well known [25] that training networks to tackle prediction problems involving thousands or more labels is challenging.

(2) What makes this task even more difficult is that unlike a classification problem, the label for each input (i.e., the optimal group) in this optimization problem is unknown, i.e., there is no ground truth regarding the labels/groups available. The only information we have is the loss if the two input sets are assigned into the same group. This makes the problem even more challenging than typical classification problems.

The inherent difficulty of utilizing Siamese networks for this problem is the dimensionality (i.e., degrees of freedom) of the output space. In response to this challenge, we propose a learning framework consisting of a cascade of Siamese models, which partitions the database in a hierarchical fashion. Each Siamese network in the framework is responsible for partitioning a group of sets into two sub-groups. The framework is illustrated in Figure 4.

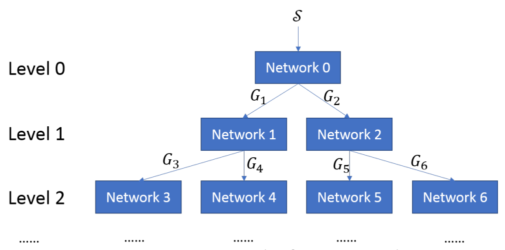


Figure 4: Cascade framework

At Level 0 of the framework, we train a Siamese network which takes each set in the entire database $\mathcal{D}$ and assigns it into one of two groups, $\mathcal{G}_1$ and $\mathcal{G}_2$, based on the loss function given in Equation (15). Then at Level 1, we train two Siamese networks working on $\mathcal{G}_1$ and $\mathcal{G}_2$ respectively in the same fashion. Thus, they partition the entire database into four groups. We continue adding more levels to the cascade framework until all groups are small enough or a pre-defined threshold on the number of levels is reached. Since each model in the cascade is specialized to partition a group of sets into only two sub-groups the resulting classification problem can be solved effectively.

The architecture of the cascade models motivates the use of a hierarchical indexing structure, which we call Hierarchical TGM (or HTGM). More specifically, assuming the level of the cascade framework is l, and $0 \leq i < j < l$, we construct TGM$_i$ and TGM$_j$ based on the partitioning results at level i and level j respectively. Suppose a group at level i, say $\mathcal{G}_g$, is partitioned into several sub-groups at level j, say $\mathcal{SG}_1, \cdots, \mathcal{SG}_m$. If for a query Q, group $\mathcal{G}_g$ can be pruned by checking TGM$_i$, then all verification operations involving groups $\mathcal{SG}_1, \cdots, \mathcal{SG}_m$ can be eliminated. The construction can be easily generalized to HTGM with h ($h > 1$) levels.

5.3 PTR: a Set Representation Method

A Siamese network accepts vectors as input and thus the sets in $\mathcal{D}$ cannot be directly fed into the network. As a result we have to build a vector representation for each set. Considering the time and space complexity of existing embedding methods such as Principal Component Analysis (PCA) or Multidimensional Scaling (MDS), they can hardly be applied to the target setting introduced in Section 7 where millions or billions of sets are involved (see comparison regarding embedding cost in Section 7.3). Besides, different from the objectives of these general-purpose embedding methods such as maximizing variance or preserving distance, our concern is to make sets containing different tokens more separable to benefit the training of the Siamese network. Intuitively, the representations that ease the training of the models are expected to bear the following property:

Definition 5.1. **Set Separation-Friendly Property**. $\forall t \in \mathcal{T}$, let $\mathcal{G}_t$ be the collection of sets containing t, and $\mathcal{G}_{\neg t}$ be the collection sets not containing t, then $\mathcal{G}_t$ and $\mathcal{G}_{\neg t}$ should to be easily separable in the representation space.

Next we discuss how to construct such representations. As the first step, we organize all tokens with a balanced binary tree such that tokens appear in leaf nodes and each leaf contains only one token. The height of the tree is thus $h = \lceil \log_2 |\mathcal{T}| \rceil$. We mark the edge from a node to its left child with 1 and the edge to its right child with 0. An example of such a tree is depicted in Figure 5.

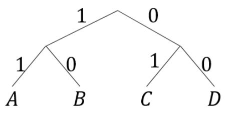


Figure 5: Tokens organized with a balanced tree

We use $path_t$ to denote the path from the root to an arbitrary token t. Since each leaf contains only one token, no two tokens share the same path. We build a path table (PT) of all tokens defined as follows:

$$PT[t, i] = \begin{cases} path_t[i], & \text{if } i \in [1, h] \\ 1 - path_t[i], & \text{if } i \in [h+1, 2h] \end{cases} \tag{16}$$

An example of PT is provided in Table 1.

We propose a method called PTR (Path Table Representation) to build a representation for a given set S as follows:

$$Rep(S)[i] = \sum_{t \in S} PT[t, i], \; i \in [1, 2h] \tag{17}$$

Table 1: An example of path table (PT)

Position	1	2	3	4
A	1	1	0	0
B	1	0	0	1
C	0	1	1	0
D	0	0	1	1

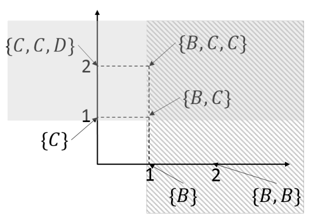


Figure 6: Separating sets

In the above example, the representation of $\{A,B,C\}$ is $[2,2,1,1]$ and the representation of $\{B,D\}$ is $[1,0,1,2]$. The second half of the path table (Positions 3 and 4) helps to reduce the chance that different sets have common representations. For example, if only the first half is used, then the representations of $\{A\}$, $\{B,C\}$, $\{A,D\}$, $\{B,C,D\}$ would all be $[1,1]$. We compare the set representations constructed on the full vs. half path tables in Section 7.3.

PTR also naturally differentiates multisets containing the same collection of tokens but with different number of occurrences. For example, $Rep(\{A\}) = [1,1,0,0]$ while $Rep(\{A,A\}) = [2,2,0,0]$.

The basic idea of the representation is to map the sets into a new space, in a way that determining collections of sets containing specific tokens can be easily performed. More specifically, set S is placed in the representation space based on the presence or absence of all tokens in S, and consequently, given a collection of tokens $\mathcal{T}_c$, we can quickly locate all sets containing $\mathcal{T}_c$. This evidently yields the set separation-friendly property. To better illustrate this, we reuse the path table in Table 1 and show that sets containing B can be separated from other sets. For better visualization, we project the representation space onto the first two dimensions (Positions 1 and 2), and keep tokens B, C, and D only in Figure 6. Clearly all sets containing token B fall into the striped area, defined by the axis aligned hyper-plane in the representation space passing from point $(1,0)$ (corresponding to $\{B\}$). Similarly all sets containing both B and C are located at the intersection of the axis aligned hyper-planes passing from $(1,0)$ and $(0,1)$ (corresponding to $\{C\}$). That way separating sets in the representation space based on token membership is conducted by determining hyper-plane intersections. We will demonstrate that such a representation is easier to learn and yields effective partitions in Section 7.

6 DEALING WITH UPDATES

Our discussions so far have assumed that the database $\mathcal{D}$ and the token universe are fixed. In some cases, however, new sets may be added to the database after the index is built, and previously unseen tokens may appear. We therefore study how updates can be handled, with a focus on TGM, as HTGM can be updated level by level in the same way.

We first discuss the case where new sets are added but the token universe remains the same. Given a new set S, we add S into the group $\mathcal{G}_g$ if the similarity upper bound between $\mathcal{G}_g$ and S is the highest among all groups. When there exist multiple groups with the same highest UB, we insert S into the group with the minimal number of sets, in line with the optimization target discussed in Section 4. After insertion, we update the TGM accordingly, i.e., for all tokens $t \in S$, we set $M[g,t] = 1$.

We now demonstrate how our approach naturally handles previously unseen tokens. This is an important feature of our solution as it is the first to deal with dynamic tokens. All previous attempts to a solution of this problem assumed a fixed token universe [72, 73]. Let S be a set containing one or more new tokens. We insert S into the database in the following two steps:

(1) Let $PS = S \cap \mathcal{T}$ be all tokens in S that have been seen previously. We find the group with the highest similarity upper bound to PS, denoted by $\mathcal{G}_g$. If $PS = \emptyset$, then $\mathcal{G}_g$ is simply the group with the minimal number of sets. S is inserted to $\mathcal{G}_g$.
(2) For any token $t_{new} \in S \setminus PS$, add a row in M corresponding to t_{new}. For all tokens $t \in S$, set $M[g,t] = 1$.

Although the partitioning in Section 4 is optimized based on existing sets and tokens, inserting new sets and tokens will not severely impact the performance of the approach, as we demonstrate in Section 7.8.

7 EXPERIMENTS

In this section, we present a thorough experimental evaluation of our approach varying parameters of interest, comparing LES^3 and its important components, L2P and PTR, with competing methods.

7.1 Settings

Environment. We run the experiments on a machine with an Intel(R) Core i7-6700 CPU, 16GB memory and a 500GB, 5400 RPM HDD (roughly 80MB/s data read rate). We use HDD for fair comparison as other disk-based methods require no random access of the data (see Section 7.6). However one could expect better performance of LES^3 when running on SDD as it incurs random access of the data by skipping some groups, especially when the number of groups is large.

Implementation. L2P is implemented with PyTorch, embedding methods in Section 7.3 and partitioning methods in Section 7.4 are implemented with Python, and TGM and the set similarity search baselines are implemented in C++ and compiled using GCC 9.3 with -O3 flag. TGM is compressed by Roaring [41], a well-performed bitmap compression technique.

Datasets. KOSARAK [2], LIVEJ [49], DBLP [1], and AOL [53] are three popular datasets used for set similarity search problems and we adapt them for this reason. We also include a social network dataset from Friendster[70] (denoted by FS), where each user is treated as a set with his/her friends being the tokens; and a dataset from PubMed Central journal literature [33] (denoted by PMC), where each sentence is treated as a set with the words being the tokens[3]. Table 2 presents a summary of the statistics on these datasets. Considering the size of FS and PMC, we utilize them for disk-based evaluation in Section 7.6 to examine the scalability of LES^3.

[3]with basic data cleaning operations such as stop-words removal.

Table 2: Dataset statistics

Dataset	$\lvert\mathcal{D}\rvert$	Set size			$\lvert\mathcal{T}\rvert$
		Max	Min	Avg	
KOSARAK	990,002	2,498	1	8.1	41,270
LIVEJ	3,201,202	300	1	35.1	7,489,073
DBLP	5,875,251	462	2	8.7	3,720,067
AOL	10,154,742	245	1	3.0	3,849,555
FS	65,608,366	3,615	1	27.5	65,608,366
PMC	787,220,474	2,597	1	8.8	22,923,401

Evaluation. Following previous studies [14, 46, 73], we adopt Jaccard similarity as the metric in our experimental evaluation. We stress however that any similarity measures satisfying the TGM applicability property introduced in Section 3.2 can be adopted in our framework with highly similar results as those reported below. For each experiment, we randomly select 10K sets in the corresponding dataset as the queries and report the average search time. Unless otherwise specified, the indexing structure (TGM) and the data are memory-resident. We conduct disk-based evaluation in Section 7.6. We compare TGM with HTGM in Section 7.7. We select n (number of groups) for each dataset which results in the shortest query latency. The influence of n is studied in Section 7.5.

Network and Loss Function. We consider Multi-Layer Perceptron (two hidden layers, each consisting of eight neurons) and Sigmoid activation function for L2P training in the experiment and leave the investigation of other networks as future work. Clearly the network has one neuron at the output layer. Let O_x be the output on input set S_x. If $O_x < 0.5$, then S_x belongs to the first group; if $O_x \geq 0.5$, then S_x belongs to the second group.

The loss function given in Equation (15) clearly describes the learning objective. However, it is difficult to train the network with that loss function as its gradient is 0 for most outputs (to be exact, the gradient is $(1 - sim(S_x, S_y))$ when $O_x = O_y = 0.5$, and is 0 elsewhere). For efficient training, we use the following surrogate loss function, which leads to the same global optimum as Equation (15) while introducing non-zero gradients:

$$loss'(S_x, S_y) = \begin{cases} W(O_x, O_y)(1 - Sim(S_x, S_y)), & \text{if } V(O_x, O_y); \\ 0, & \text{otherwise;} \end{cases} \tag{18}$$

where $W(O_x, O_y) = (0.5 - |O_x - O_y|)$, and $V(O_x, O_y) = [(O_x \geq 0.5 \wedge O_y \geq 0.5) \vee (O_x < 0.5 \wedge O_y < 0.5)]$.

Initialization. Models at the first few levels of the Cascade framework deal with a large number of sets and incur long training time. To improve the training efficiency, we first sort all sets based on the minimal token contained in each set, and then partition all sets into 128 groups such that each group contains consecutive $|\mathcal{D}|/128$ sets, inspired by the idea of imposing sequential constraint to clustering tasks [62]. Since we always build TGM on the partitioning results at level 10 or higher which may contain thousands of groups, such initialization has minor impact on the performance but greatly reduces the training time. Note that initialization is not performed for the sampled dataset used in Section 7.3 due to its small size.

Training. For the Siamese network partitioning an arbitrary group, we randomly select 40,000 pairs of sets in the group to generate training samples, relatively small compared to the data size. It is observed that further increasing the number of training samples do not significantly improve the pruning efficiency of the partitioning results. We stop partitioning a group if it contains less than 50 sets, and thus the number of groups at level i may be less than 2^i. The batch size is set to 256, the number of epochs is set to 3 (except for Section 7.2 which reports the learning curves), and Adam is used as the optimizer. The same sampling-and-training procedure is repeated for each model in the cascade framework, starting from level 0.

7.2 Model Convergence and Training Cost

In this section we report the learning curves and the training costs. We observe that different models in the cascade framework introduced in Section 5.2 yield similar learning curves, and thus we present the training loss of a random model at level 0 for each dataset (note that there are 128 models at level 0, see Section 7.1, paragraph Initialization). The training losses and costs are presented in Figure 7.

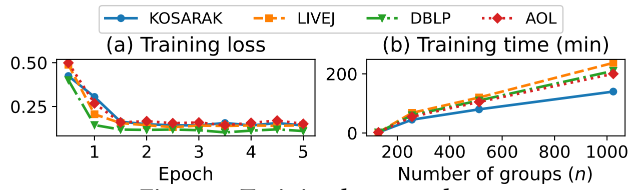


Figure 7: Training losses and costs

As is clear from Figure 7(a), on all datasets used in the experiment, the training loss decreases rapidly and the model converges after approximately two epochs, attesting to the efficiency of the model training process. Also, as can be observed from Figure 7(b), the training cost grows linearly with respect to the number of groups, making LES^3 scalable for large datasets. Besides, models at the same level of the cascade framework can be trained in parallel to further reduce the training cost, which is an interesting direction for future investigation.

7.3 PTR vs. Set Representation Techniques

We compare PTR with other applicable set representation techniques. More specifically, we choose PCA [32], a widely-used linear embedding method, MDS [12], a representative non-linear embedding approach, and Binary Encoding [28], an efficient categorical data embedding technique. We also include the variant of PTR constructed on the first half of the path table (see Section 5.3), denoted by PTR-half. Considering the complexity of PCA and MDS, we conduct experiments on sampled KOSARAK (sample ratio of 5%). We report the representation construction time of each method and the query answering time using the resulting partitioning results for kNN query ($k = 10$) and range query ($\delta = 0.7$) in Figure 8; similar trends are observed on other datasets and queries.

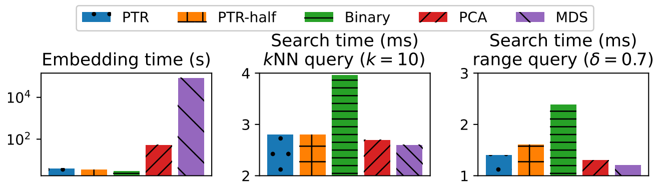


Figure 8: Comparison of representation techniques

As can be observed from Figure 8, compared with PCA and MDS, PTR incurs much lower embedding time (10 to 20,000 times faster) while results in similar search time; compared with Binary Encoding and PTR-half, PTR leads to faster query answering with comparable embedding cost. Binary Encoding assigns unique representations to different sets without considering set characteristics (e.g., tokens contained therein), and thus can hardly achieve any Set Separation-Friendly Property. PTR-half, as discussed in Section 5.3, suffers from the risk that different sets may have common representations, and consequently these (dissimilar) sets are partitioned into the same group as they are not separable in the representation space, and the resulting search time thus is slightly longer than that of PTR. The major advantage of PTR is that it integrates the Set Separation-Friendly Property introduced in Section 5.3 into set representations by allowing sets consisting of different tokens to be easily separable by axis-aligned hyper-planes in the embedding space, and thus eases the training of the downstream Siamese networks.

7.4 L2P vs. Algorithmic Approaches

We compare the learning-based partitioning approach, L2P, to the algorithmic methods introduced in Section 4.3, namely the graph cut-based method (PAR-G), centroid-based method (PAR-C), divisive clustering method (PAR-D), and agglomerative clustering method (PAR-A), in terms of partitioning cost, including time cost and space cost, and query answering time.

For PAR-G, we adopt PaToH [9], a graph partitioning tool known to be efficient and performing well, to cut the graph. We report the cost of different methods in partitioning KOSARAK into 1024 groups and the query answering time for kNN with $k = 10$ in Figure 9; similar trends are observed on other datasets and queries. Note that the partitioning time of L2P includes model training time and inference time (the time required to assign a set into a group), and the partitioning time of PAR-G consists of the kNN graph construction time and the graph cut time. PAR-G is specially optimized for $k = 10$ and the construction of its kNN graph is accelerated by LES^3.

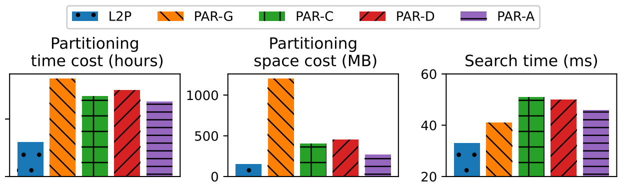


Figure 9: Comparison of partitioning methods

As depicted in Figure 9, L2P provides the fastest search while only incurs a small fraction of partitioning time and space cost compared to competitors (saving 80% partitioning time and 99% space compared with PAR-G). The reason why L2P incurs less partitioning time and space overhead is that, as described in Section 5.1 and Section 7.1, by training the models on a small portion of data, L2P is better positioned to approach the optimal partitioning where the *GPO* is minimized, while other techniques work on the entire dataset and require (sometimes repetitively) computing the *GPO* of arbitrary groups (or pairs) of sets. Besides, only model parameters and the training samples in a mini batch have to be saved in memory for L2P, with minimal storage overhead, while other techniques require materializing a large amount of intermediate partitioning results (and the entire kNN graph for PAR-G) in memory, incurring prohibitive space consumption.

By directly optimizing the *GPO* which integrates the two desired properties of a partitioning with higher pruning efficiency, L2P is able to outperform PAR-G, the objective of which is minimizing the number of edges in the similarity graph crossing different sub-graphs rather than *GPO*. PAR-C, PAR-D, and PAR-A, although also aim to optimize the *GPO*, suffer from severe local optimality problems: a set is moved to a group only if such movement reduces the overall *GPO*, while in many cases movements temporarily increasing the *GPO* must be allowed to determine a global optimum. For example, let $S_i \in \mathcal{G}_i$, $S_j \in \mathcal{G}_j$ be two sets. Assume that moving S_i to $\mathcal{G}_j$ and moving S_j to $\mathcal{G}_i$ when individually carried out both increase the *GPO*, and consequently S_i remains in $\mathcal{G}_j$ and S_j in $\mathcal{G}_i$. However, swapping S_i and S_j may reduce the overall *GPO* and thus leads to better partitioning. Such swapping cannot be achieved based on the strategy followed by PAR-C and PAR-D. Similarly, the strategy of PAR-A does not allow the merge of groups temporarily increasing the overall *GPO*, which however may be necessary in identifying a global optimum.

7.5 Sensitivity to Number of Groups and k

We test the performance of LES^3 in terms of query processing time on kNN queries, varying the number of groups n and the result size k. The results are presented in Figure 10.

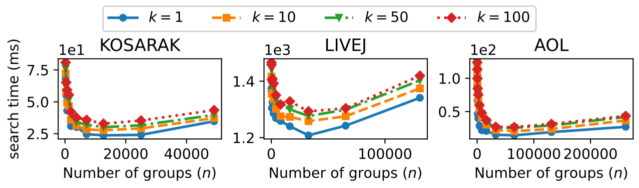


Figure 10: Sensitivity to the number of groups and result size

Increasing n accelerates query answering, regardless of the result size. This is because with more groups, as indicated by Equation (13), the overall pruning efficiency of LES^3 can be improved, meaning fewer candidates have to be checked. Increasing n benefits search time up to a point. In particular we observe a diminishing return behavior with respect to search performance as n increases further. The reason is that, with a sufficiently large number of groups, sets are already well-separated, and further increasing n brings no significant change to the pruning efficiency but incurs higher index (TGM) scan cost. Moreover, search time increases for larger k, which is consistent with our analysis in Equation (**??**), as in general a larger k in kNN search is analogous to a smaller δ in range search and thus the pruning efficiency is lower.

While determining the optimal number of groups for partitioning is a known NP-hard problem [31], we empirically observe from the experiments that setting the number of groups at approximately $0.5\%|\mathcal{D}|$ leads to the lowest search time, where $|\mathcal{D}|$ is the number of sets in the corresponding dataset.

7.6 LES^3 vs. Set Similarity Search Baselines

In this section, we compare LES^3 with existing set similarity search approaches to answering kNN and range queries in memory-based

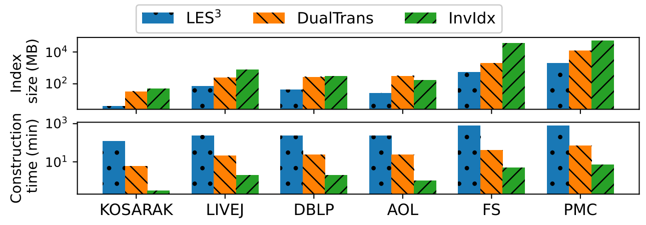


Figure 11: Index size and construction time

and disk-based settings respectively. Among tree-based set similarity search approaches to date, DualTrans [73] provides the fastest query processing. For inverted index-based methods, we adopt the method proposed in [67] (denoted by InvIdx), which yields the state-of-the-art performance for set similarity join tasks. Note that we exclude methods requiring index construction during query time [13, 14, 69] as the index construction cost is much higher than the query cost. Since inverted index-based methods are designed for range queries and do not naturally support kNN queries, we modify the query answering algorithm of InvIdx for kNN queries as follows. (1) Given a query set Q and a result size k, start with threshold $\delta = 1.0$ and use InvIdx to find candidate sets from $\mathcal{D}$ whose similarity with Q exceeds δ, denoted by C. (2) Identify the temporary kNN results from C, denoted by $\mathcal{R}_k$. If the minimal similarity between any set in $\mathcal{R}_k$ and Q exceeds δ, terminate. Otherwise, decrease δ by z, use InvIdx to find candidates sets with the new δ, update C accordingly, and repeat the step. (3) Upon termination, $\mathcal{R}_k$ is guaranteed to be the kNN to Q, as the similarity between any sets in $\mathcal{D} \setminus C$ and Q does not exceed the current δ. The value of z is tuned for faster query answering.

In addition, we also include a brute-force approach, i.e., computing the similarity between the query set and all other sets to derive the results, for completeness of comparison.

In Figure 11, we show the index size and index construction time for all methods. It is clear that the indexing structure of LES^3, namely TGM, is much more lightweight, requiring up to 90% less space than DualTrans and InvIdx. The major time cost of constructing TGM comes from the model training, which however is a preprocessing step incurring only a one-time cost and can be further reduced as discussed in Section 7.2.

In Figure 12 we compare the performance of the four methods in a memory-based setting. We observe that LES^3 outperforms competitors for both kNN queries and range queries, accelerating the query answering by 2 to 20 times. DualTrans incurs longer search time as it uses an R-tree to organize all sets, with each set being represented with a d-dimensional vector (d can be tuned for faster pruning). When the value of d is small, sets containing different tokens cannot be clearly separated based on their representations, while when the value of d is large, using R-tree to organize the vectors incurs high overlap between the bounding boxes of nodes on the R-tree, as previous research indicates [30]. Besides, scanning the R-tree is expensive, which is not worthwhile considering that set similarity (e.g., Jaccard similarity) can usually be computed efficiently. While InvIdx provides comparable performance with LES^3 for range queries with large δ, it incurs greater search latency for kNN queries, especially when the average set size is large (e.g., on KOSARAK and LIVEJ). The reason is that, with InvIdx filtering operations need to be repeated for each candidate set (or multiple candidates with some common characteristics), and larger set size and kNN queries both enlarge the number of candidates, leading to sub-par query performance.

In contrast, we use TGM to compute the upper bounds between a query set and a group of sets; obtaining all bounds requires only $O(|S| * |\mathcal{G}|)$ time, which is relatively cheap. Although the search time of LES^3 increases for range query as δ decreases, LES^3 provide much faster query answering under a wide range of δ.

We compare the performance of the four methods in the disk-based setting in Figure 13. Note that for DualTrans and InvIdx, only the part of the index that is necessary to the query answering, such as R-nodes on the search path and inverted indexes related to the query set, is retrieved into memory to reduce I/O cost. We observe that LES^3 generally provides faster search compared with competitors, accelerating the query answering by 2 to 10 times. The reasons why LES^3 incurs lower search time are: (1) Sets sharing no or very few common tokens with the query set can be easily pruned without being retrieved into memory; and (2) Since sets in the same group are checked jointly during the searching process; materializing a group of sets continuously on disk minimizes the data transfer delay. DualTrans and InvIdx, on the contrary, incur longer search latency and are outperformed by the Brute-force method for a wide range of k and δ. Besides the drawbacks discussed above in the memory-based setting, the search strategies of DualTrans and InvIdx incur repetitive retrieval of data with random disk access, which results in higher I/O cost (more pages retrieved, higher seek and rotation overhead, etc.), making them less efficient in the disk-based setting.

7.7 TGM vs. HTGM

We evaluate the performance of TGM and HTGM to determine whether building a hierarchical index pays off. Intuitively, whether it benefits the query processing largely depends on the similarity distribution. For example, in cases where very few sets share common tokens, one can prune a large number of candidates using the matrices at the first few levels of HTGM, avoiding scanning the larger matrices at finer levels. However, in cases where most sets are similar, the small matrices at the first few levels of HTGM may provide no pruning efficiency at all. We assume that the similarity between sets in $\mathcal{D}$ can be modeled by a power-law distribution $P[sim = v] \sim v^{-\alpha}$, where $P[sim = v]$ denotes the probability that the similarity between any two sets is v, $v \in [0, 1]$, $\alpha \in [1, \infty)$. We generate multiple synthetic databases consisting of 20,000 sets and 20,000 tokens each, by varying the value of α. We train a cascade model with 9 levels (including level 0). We use the partitioning results at level 8 (256 groups) to build the TGM, and use the partitioning results at level 5 (32 groups) and level 8 to build the HTGM. We compare HTGM and TGM from two aspects. First, the index access cost, measured by the number of columns in the HTGM or TGM that are checked when processing the query. Second, the computational cost, measured by the number of similarity calculations. We measure the ratio of cost between HTGM and TGM, and the results are shown in Figure 14. It is evident that HTGM outperforms TGM when the value of α is large, i.e., most sets are dissimilar. This is in line with the discussions in Section 7.7.

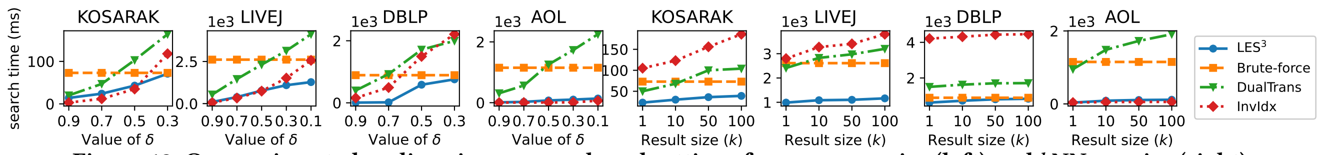


Figure 12: Comparison to baselines in memory-based settings for range queries (left) and kNN queries (right)

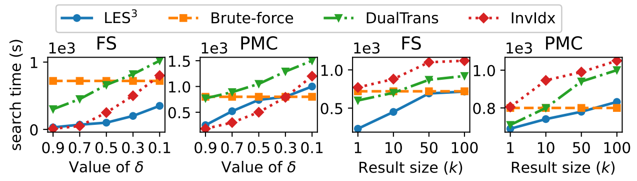


Figure 13: Comparison to baselines in disk-based settings for range queries (left) and kNN queries (right)

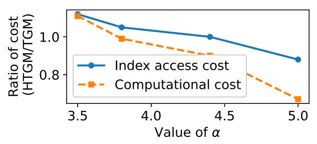


Figure 14: TGM vs. HTGM

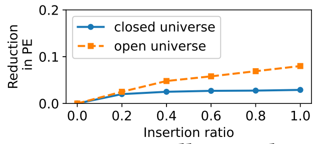


Figure 15: Handling updates

7.8 Handling Updates

We evaluate the performance of the proposed approach under updates. Two cases are considered: (1) *closed universe*, meaning the new sets to be inserted contain only tokens from the original database, and (2) *open universe*, where the new sets may contain previously unseen tokens. Let $\mathcal{D}$ be the original database, $\mathcal{D}^{closed}$ be the collection of new sets to be inserted under a closed universe, and $\mathcal{D}^{open}$ be the collection of new sets to be inserted under an open universes. For the experiment, we set insertion ratio ($|\mathcal{D}^{closed}|/|\mathcal{D}|$ and $|\mathcal{D}^{open}|/|\mathcal{D}|$) in range $[0, 1]$, and half of the tokens in $\mathcal{D}^{open}$ are from $\mathcal{D}$ and half are new. We compute the decrease in pruning efficiency after insertion compared to obtaining a partitioning from scratch (namely running L2P) on $\mathcal{D} \cup \mathcal{D}^{closed}$ or $\mathcal{D} \cup \mathcal{D}^{open}$ (referred to as *re-build*). We give the results for kNN query with $k = 10$ on KOSARAK in Figure 15; the experiments on the other datasets show similar trends.

Figure 15 depicts the percentage of pe reduction compared to re-build. The pruning efficiency decreases slightly as more new sets are inserted into the database. Insertions under an open universe have a higher impact on performance. The reason is that the tokens from the same universe mainly follow a similar distribution and the partition results obtained on the original data are still sufficient, while there is no prior knowledge regarding the distribution of new tokens. We observe, however, that the overall pruning efficiency is resistant to insertions (experiencing a decrease by at most 8%), which attests to the robustness of the proposed approach.

8 RELATED WORK

The problem of processing set similarity queries, including set similarity search [36, 72, 73] and set similarity joins [14, 15, 21, 46, 67], has attracted remarkable research interest recently. Zhang et al. [72, 73] propose to transform sets into scalars or vectors with the relative distance between sets preserved, and organize the transformed sets with B+-trees or R-trees, which facilitate the use of tree-based branch-and-bound algorithms for similarity search. The major drawback of their work is that, as shown in the experiments, the tree structure can easily grow larger than the original data and thus using the index for filtering incurs a significant cost, especially when the index and the data are stored externally. Most prior research in the area of set similarity join focuses on threshold-join queries and follows the filter-and-verify framework. In the filter step, existing methods mainly adopt (1) prefix-based filters [7, 65, 69], based on the observation that if the similarity between two sets exceeds δ, then they must share common token(s) in their prefixes of length m; and (2) partition-based filters [3, 13, 14, 68], which partition a set into several subsets so that two sets are similar only if they share a common subset. Set similarity queries in distributed environments [11, 47] and approximate queries [60, 61] are beyond the scope of this paper but represent promising directions for further investigation.

Indexing is an important and well-studied problem in data management and recent works have utilized machine learning to learn a CDF or to partition the data space for traditional database indexing [17, 19, 40, 42, 43, 51, 56, 71]. In this paper, we complement recent work by studying the applicability of machine learning techniques to assist index construction for set similarity search problems. Our results show that the proposed methods offer vast advantages over traditional techniques.

Embedding sets and other entities consisting of discrete elements has been well-studied. The most natural way to represent such data types is n-hot encoding, but the resulting vectors are often very long and sparse. Dimensionality reduction techniques are used to compress the encoding vectors with different focuses: maximizing variances [54], preserving distances [6], solving the crowding problem [45], etc. Recent advances in document embedding, e.g., word2vec [48], BERT [16], also provide new perspectives to construct representations of sets. Compared to these methods, the PTR proposed in Section 5.3 utilizes a very efficient method to produce relatively short representations and is optimized for the specific problem at hand.

9 CONCLUSIONS

In this paper, we have studied the problem of *exact set similarity search*, and designed LES^3, a filter-and-verify approach for efficient query processing. Central to our proposal is TGM, a simple yet effective structure that strikes a balance between index access cost and effectiveness in pruning candidate sets. We have revealed the desired properties of optimal partitioning in terms of pruning efficiency under the uniform token distribution assumption. We develop a learning-based approach, L2P, utilizing a cascade of Siamese networks to identify partitions. A novel set representation method, PTR, is developed to cater to the requirements of network training. The experimental results have demonstrated the superiority of LES^3 over other applicable approaches.

REFERENCES

[1] DBLP. http://dblp.uni-trier.de/

[2] KOSARAK. http://fimi.uantwerpen.be/data/

[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*. VLDB Endowment, 918–929.

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[5] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. 2020. Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research* (2020).

[6] Ingwer Borg and Patrick Groenen. 2003. Modern multidimensional scaling: Theory and applications. *Journal of Educational Measurement* 40, 3 (2003), 277–280.

[7] Panagiotis Bouros, Shen Ge, and Nikos Mamoulis. 2012. Spatio-textual similarity joins. *Proceedings of the VLDB Endowment* 6, 1 (2012), 1–12.

[8] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. 1994. Signature verification using a" siamese" time delay neural network. In *Advances in neural information processing systems*. 737–744.

[9] Ümit V Çatalyürek and Cevdet Aykanat. 2011. Patoh (partitioning tool for hypergraphs). In *Encyclopedia of Parallel Computing*. Springer, 1479–1487.

[10] Stephen A Cook. 1971. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*. 151–158.

[11] Akash Das Sarma, Yeye He, and Surajit Chaudhuri. 2014. Clusterjoin: A similarity joins framework using map-reduce. *Proceedings of the VLDB Endowment* 7, 12 (2014), 1059–1070.

[12] Vin De Silva and Joshua B Tenenbaum. 2004. *Sparse multidimensional scaling using landmark points*. Technical Report. Technical report, Stanford University.

[13] Dong Deng, Guoliang Li, He Wen, and Jianhua Feng. 2015. An efficient partition based method for exact set similarity joins. *Proceedings of the VLDB Endowment* 9, 4 (2015), 360–371.

[14] 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*. 905–920.

[15] Dong Deng, Chengcheng Yang, Shuo Shang, Fan Zhu, Li Liu, and Ling Shao. 2019. LCJoin: set containment join via list crosscutting. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 362–373.

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[17] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. 2020. ALEX: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 969–984.

[18] Xin Luna Dong and Theodoros Rekatsinas. 2018. Data integration and machine learning: A natural synergy. In *Proceedings of the 2018 International Conference on Management of Data*. 1645–1650.

[19] Yihe Dong, Piotr Indyk, Ilya Razenshteyn, and Tal Wagner. 2020. Learning Space Partitions for Nearest Neighbor Search. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rkenmREFDr

[20] Jingfan Fan, Xiaohuan Cao, Zhong Xue, Pew-Thian Yap, and Dinggang Shen. 2018. Adversarial similarity network for evaluating image alignment in deep learning based registration. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer, 739–746.

[21] Raul Castro Fernandez, Jisoo Min, Demitri Nava, and Samuel Madden. 2019. Lazo: A cardinality-based method for coupled estimation of jaccard similarity and containment. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1190–1201.

[22] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Fiting-tree: A data-aware index structure. In *Proceedings of the 2019 International Conference on Management of Data*. 1189–1206.

[23] Chang Ge, Yinan Li, Eric Eilebrecht, Badrish Chandramouli, and Donald Kossmann. 2019. Speculative distributed CSV data parsing for big data analytics. In *Proceedings of the 2019 International Conference on Management of Data*. 883–899.

[24] Lars Gottesbüren, Michael Hamann, Sebastian Schlag, and Dorothea Wagner. 2020. Advanced Flow-Based Multilevel Hypergraph Partitioning. In *18th International Symposium on Experimental Algorithms, SEA 2020, June 16-18, 2020, Catania, Italy (LIPIcs)*, Simone Faro and Domenico Cantone (Eds.), Vol. 160. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 11:1–11:15. https://doi.org/10.4230/LIPIcs.SEA.2020.11

[25] Maya R Gupta, Samy Bengio, and Jason Weston. 2014. Training highly multiclass classifiers. *The Journal of Machine Learning Research* 15, 1 (2014), 1461–1492.

[26] Antonin Guttman. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. 47–57.

[27] Marios Hadjieleftheriou, Xiaohui Yu, Nick Koudas, and Divesh Srivastava. 2008. Hashed samples: selectivity estimators for set similarity selection queries. *Proceedings of the VLDB Endowment* 1, 1 (2008), 201–212.

[28] Jiawei Han, Micheline Kamber, and Jian Pei. 2011. Data mining concepts and techniques third edition. *The Morgan Kaufmann Series in Data Management Systems* 5, 4 (2011), 83–124.

[29] John A Hartigan and Manchek A Wong. 1979. Algorithm AS 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28, 1 (1979), 100–108.

[30] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. 604–613.

[31] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2013. *An introduction to statistical learning*. Vol. 112. Springer.

[32] Ian T Jolliffe and Jorge Cadima. 2016. Principal component analysis: a review and recent developments. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 374, 2065 (2016), 20150202.

[33] PubMed Central Journal. .. https://www.ncbi.nlm.nih.gov/pmc/tools/openftlist/.

[34] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. 1999. Multilevel hypergraph partitioning: applications in VLSI domain. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 7, 1 (1999), 69–79.

[35] Leonard Kaufman and Peter J Rousseeuw. 2009. *Finding groups in data: an introduction to cluster analysis*. Vol. 344. John Wiley & Sons.

[36] Jongik Kim and Hongrae Lee. 2012. Efficient exact similarity searches using multiple token orderings. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 822–833.

[37] Sungyeon Kim, Minkyo Seo, Ivan Laptev, Minsu Cho, and Suha Kwak. 2019. Deep metric learning beyond binary supervision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2288–2297.

[38] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*. 489–504.

[39] Ani Kristo, Kapil Vaidya, Ugur Çetintemel, Sanchit Misra, and Tim Kraska. 2020. The Case for a Learned Sorting Algorithm. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1001–1016.

[40] Harald Lang, Alexander Beischl, Viktor Leis, Peter Boncz, Thomas Neumann, and Alfons Kemper. 2020. Tree-Encoded Bitmaps. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 937–967.

[41] Daniel Lemire, Owen Kaser, Nathan Kurz, Luca Deri, Chris O'Hara, François Saint-Jacques, and Gregory Ssi-Yan-Kai. 2018. Roaring bitmaps: Implementation of an optimized software library. *Software: Practice and Experience* 48, 4 (2018), 867–895.

[42] Linwei Li, Kai Zhang, Jiading Guo, Wen He, Zhenying He, Yinan Jing, Weili Han, and X Sean Wang. 2020. BinDex: A Two-Layered Index for Fast and Robust Scans. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 909–923.

[43] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A Learned Index Structure for Spatial Data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2119–2133.

[44] Yifan Li, Xiaohui Yu, and Nick Koudas. 2019. Top-k queries over digital traces. In *Proceedings of the 2019 International Conference on Management of Data*. 954–971.

[45] Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, Nov (2008), 2579–2605.

[46] Willi Mann, Nikolaus Augsten, and Panagiotis Bouros. 2016. An empirical evaluation of set similarity join techniques. *Proceedings of the VLDB Endowment* 9, 9 (2016), 636–647.

[47] Ahmed Metwally and Christos Faloutsos. 2012. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *arXiv preprint arXiv:1204.6077* (2012).

[48] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[49] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and Analysis of Online Social Networks. In *Proceedings of the 5th ACM/Usenix Internet Measurement Conference (IMC'07)*. San Diego, CA.

[50] Jonas Mueller and Aditya Thyagarajan. 2016. Siamese recurrent architectures for learning sentence similarity. In *thirtieth AAAI conference on artificial intelligence*.

[51] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-dimensional Indexes. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 985–1000.

[52] Paul Neculoiu, Maarten Versteegh, and Mihai Rotaru. 2016. Learning text similarity with siamese recurrent networks. In *Proceedings of the 1st Workshop on Representation Learning for NLP*. 148–157.

[53] Greg Pass, Abdur Chowdhury, and Cayley Torgeson. 2006. A picture of search. In *Proceedings of the 1st international conference on Scalable information systems*. 1–es.

[54] Karl Pearson. 1901. LIII. On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of*

Science 2, 11 (1901), 559–572.

[55] Lior Rokach and Oded Maimon. 2005. Clustering methods. In *Data mining and knowledge discovery handbook*. Springer, 321–352.

[56] Alexandre Sablayrolles, Matthijs Douze, Cordelia Schmid, and Hervé Jégou. 2019. Spreading vectors for similarity search. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=SkGuG2R5tm

[57] Mehran Sahami and Timothy D Heilman. 2006. A web-based kernel function for measuring the similarity of short text snippets. In *Proceedings of the 15th international conference on World Wide Web*. 377–386.

[58] David Salomon. 2004. *Data compression: the complete reference*. Springer Science & Business Media.

[59] David Salomon and Giovanni Motta. 2010. *Handbook of data compression*. Springer Science & Business Media.

[60] Venu Satuluri and Srinivasan Parthasarathy. 2011. Bayesian locality sensitive hashing for fast similarity search. *arXiv preprint arXiv:1110.1328* (2011).

[61] Sebastian Schelter and Jérôme Kunegis. 2016. Tracking the trackers: A large-scale analysis of embedded web trackers. In *Tenth International AAAI Conference on Web and Social Media*.

[62] Tibor Szkaliczki. 2016. clustering. sc. dp: Optimal clustering with sequential constraint by using dynamic programming. *R JOURNAL* 8, 1 (2016), 318–327.

[63] Matus Telgarsky and Andrea Vattani. 2010. Hartigan's method: k-means clustering without voronoi. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. 820–827.

[64] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. In *Advances in neural information processing systems*. 2692–2700.

[65] 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 2012 ACM SIGMOD International Conference on Management of Data*. 85–96.

[66] Pei Wang and Yeye He. 2019. Uni-Detect: A Unified Approach to Automated Error Detection in Tables. In *Proceedings of the 2019 International Conference on Management of Data*. 811–828.

[67] Xubo Wang, Lu Qin, Xuemin Lin, Ying Zhang, and Lijun Chang. 2019. Leveraging set relations in exact and dynamic set similarity join. *The VLDB Journal* 28, 2 (2019), 267–292.

[68] Chuan Xiao, Wei Wang, Xuemin Lin, and Haichuan Shang. 2009. Top-k set similarity joins. In *2009 IEEE 25th International Conference on Data Engineering*. IEEE, 916–927.

[69] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. 2011. Efficient similarity joins for near-duplicate detection. *ACM Transactions on Database Systems (TODS)* 36, 3 (2011), 1–41.

[70] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems* 42, 1 (2015), 181–213.

[71] Cong Yue, Zhongle Xie, Meihui Zhang, Gang Chen, Beng Chin Ooi, Sheng Wang, and Xiaokui Xiao. 2020. Analysis of Indexing Structures for Immutable Data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 925–935.

[72] 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 *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 759–770.

[73] 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.