An Efficient Partition Based Method for Exact Set Similarity Joins

Dong Deng Guoliang Li He Wen Jianhua Feng
Department of Computer Science, Tsinghua University, Beijing, China.
{dd11,wenhe13}@mails.tsinghua.edu.cn;{liguoliang,fengjh}@tsinghua.edu.cn

ABSTRACT

We study the exact set similarity join problem, which, given two collections of sets, finds out all the similar set pairs from the collections. Existing methods generally utilize the prefix filter based framework. They generate a prefix for each set and prune all the pairs whose prefixes are disjoint. However the pruning power is limited, because if two dissimilar sets share a common element in their prefixes, they cannot be pruned. To address this problem, we propose a partition-based framework. We design a partition scheme to partition the sets into several subsets and guarantee that two sets are similar only if they share a common subset. To improve the pruning power, we propose a mixture of the subsets and their 1-deletion neighborhoods (the subset of a set by eliminating one element). As there are multiple allocation strategies to generate the mixture, we evaluate different allocations and design a dynamic-programming algorithm to select the optimal one. However the time complexity of generating the optimal one is $\mathcal{O}(s^3)$ for a set with size s. To speed up the allocation selection, we develop a greedy algorithm with an approximation ratio of 2. To further reduce the complexity, we design an adaptive grouping mechanism, and the two techniques can reduce the complexity to $\mathcal{O}(s \log s)$. Experimental results on three real-world datasets show our method achieves high performance and outperforms state-of-the-art methods by 2-5 times.

1. INTRODUCTION

We study the exact set similarity join problem, which aims to find all the similar set pairs from two collections of sets. Similarity join plays an essential role in many applications, such as personalized recommendations and collaborative filtering [6,13], entity resolution [18], near duplicate detection [22], data cleaning [4], data integration [5], and machine learning [24]. The similarity functions are used to measure the similarity between two sets. Two sets are said to be similar if and only if their similarity exceeds a given threshold. In this paper, we focus on three widely-used similarity functions, `Jaccard`, `Cosine` and `Dice`. For example, Spertus et al. [16] show that `Cosine`, a.k.a L2-norm, achieves the best empirical results against the other five measures for recommending on-line communities to members on the Orkut social network.

Most of existing methods utilize the prefix-filter based framework [2,20–22]. It first sorts all the elements in each set by a global order and then takes the first few elements of each set as a prefix. It can be guaranteed that if the prefixes of two sets are disjoint, they cannot be similar. However the pruning power of the prefix filter based framework is limited, because if two dissimilar sets share a common element in the prefixes, these methods cannot prune them.

To alleviate this problem, we propose a partition-based framework for the exact set similarity join. We develop a partition scheme to partition the sets into several disjoint subsets and prove that two sets are similar only if they share a common subset. We group the sets with same size to share computation. We build inverted indexes on the subsets to efficiently perform the set similarity join. To reduce the accessed inverted list sizes, we propose to use a mixture of the subsets and their 1-deletion neighborhoods (the subset of a set by eliminating one element). As there are multiple ways to allocate the mixture, we evaluate different allocations and design a dynamic-programming method for optimal allocation selection. However the time complexity of generating the optimal one is $\mathcal{O}(s^3)$ for a set with size s. To speed up the allocation selection, we develop a greedy algorithm with an approximation ratio of 2. To further reduce the complexity, we design an adaptive grouping mechanism. The two techniques together reduce the complexity to $\mathcal{O}(s \log s)$. To summarize, we make the following contributions. (1) We propose a partition-based framework for the exact set similarity join which is fundamentally different from existing solutions (see Section 3). (2) We propose a mixture of the subsets and their 1-deletion neighborhoods to perform the exact set similarity join. We evaluate different allocations of the mixture and design an algorithm to select the optimal one (see Section 4). (3) We develop a greedy allocation selection algorithm with an approximation ratio of 2 to speed up the allocation selection and an adaptive grouping mechanism to decrease the number of groups. We reduce the complexity of allocation selection for a set with size s from $\mathcal{O}(s^3)$ to $\mathcal{O}(s \log s)$ (see Section 5). (4) Experimental results on real world datasets show our method achieves high performance and outperforms the state-of-the-art methods by 2-5 times (see Section 7).

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
Proceedings of the VLDB Endowment, Vol. 9, No. 4
Copyright 2015 VLDB Endowment 2150-8097/15/12.

2. PRELIMINARIES

2.1 Problem Formulation

Given a finite universe $\mathcal{U} = \{x_1, x_2, \ldots, x_n\}$, a set is a subset of $\mathcal{U}$, which we interchangeably refer to as a record. To measure the similarity between two records, we need a similarity function Sim. We focus on three commonly used similarity functions, Jaccard Similarity, Cosine Similarity, and Dice Similarity. For any two records $\mathcal{X}$ and $\mathcal{Y}$, the three similarity functions are respectively defined as follows.

$\texttt{Jac}(\mathcal{X},\mathcal{Y})=\frac{|\mathcal{X}\cap\mathcal{Y}|}{|\mathcal{X}\cup\mathcal{Y}|}$, $\texttt{Cos}(\mathcal{X},\mathcal{Y})=\frac{|\mathcal{X}\cap\mathcal{Y}|}{\sqrt{|\mathcal{X}||\mathcal{Y}|}}$, $\texttt{Dice}(\mathcal{X},\mathcal{Y})=\frac{2|\mathcal{X}\cap\mathcal{Y}|}{|\mathcal{X}|+|\mathcal{Y}|}$.

where $|\mathcal{X}|$ denotes the size of the record $\mathcal{X}$. Note the Dice Similarity is the same as the F-measure. Based on the definition, the similarity of two records is within $[0, 1]$. For example, considering the records in Table 1, we have $\texttt{Jac}(\mathcal{X}_1,\mathcal{X}_2)= \frac{6}{12} = 0.5$ as $|\mathcal{X}_1 \cap \mathcal{X}_2| = 6$ and $|\mathcal{X}_1 \cup \mathcal{X}_2| = 12$. Next we formally define the exact set similarity join problem.

Definition 1 (Exact Set Similarity Join). *Given two sets of records $\mathcal{R}$ and $\mathcal{S}$, a similarity function Sim and a threshold δ, the exact set similarity join finds all similar record pairs $\langle\mathcal{X},\mathcal{Y}\rangle \in \mathcal{R} \times \mathcal{S}$ such that $\texttt{Sim}(\mathcal{X},\mathcal{Y}) \geq \delta$.*

In this paper, we focus on self-join, i.e., $\mathcal{R} = \mathcal{S}$. Our techniques can be easily extended to support RS-join (see Section 6.2). We first use Jaccard Similarity as the similarity function Sim to introduce our techniques and then discuss supporting the other similarity functions in Section 6.1. Given a similarity threshold δ, two records $\mathcal{X}$ and $\mathcal{Y}$ are said to be similar if $\texttt{Jac}(\mathcal{X},\mathcal{Y}) \geq \delta$. For example, consider the dataset $\mathcal{R}$ in Table 1. Suppose the threshold $\delta = 0.73$. The exact set similarity join returns the pair $\langle\mathcal{X}_1,\mathcal{X}_5\rangle$ as $\texttt{Jac}(\mathcal{X}_1,\mathcal{X}_5) = 0.82 \geq \delta$ and all the others are not similar.

An important property of the Jaccard Similarity, which is known as size filter, is that two records $\mathcal{X}$ and $\mathcal{Y}$ are similar only if $\delta|\mathcal{Y}| \leq |\mathcal{X}| \leq \frac{|\mathcal{Y}|}{\delta}$. This is because $|\mathcal{X}| \geq |\mathcal{X} \cap \mathcal{Y}|$ and $|\mathcal{Y}| \leq |\mathcal{X} \cup \mathcal{Y}|$, which leads to $\frac{|\mathcal{X}|}{|\mathcal{Y}|} \geq \frac{|\mathcal{X}\cap\mathcal{Y}|}{|\mathcal{X}\cup\mathcal{Y}|} = \texttt{Jac}(\mathcal{X},\mathcal{Y}) \geq \delta$. Thus we have $|\mathcal{X}| \geq |\mathcal{Y}|\delta$. Similarly we can deduce $|\mathcal{X}| \leq |\mathcal{Y}|/\delta$. Let the Overlap Similarity be the size of the intersection of two records. All the three similarity functions above can be transformed to Overlap Similarity. More specifically, as $\texttt{Jac}(\mathcal{X},\mathcal{Y}) = \frac{|\mathcal{X}\cap\mathcal{Y}|}{|\mathcal{X}\cup\mathcal{Y}|} = \frac{|\mathcal{X}\cap\mathcal{Y}|}{|\mathcal{X}|+|\mathcal{Y}|-|\mathcal{X}\cap\mathcal{Y}|}$, we have $\texttt{Jac}(\mathcal{X},\mathcal{Y}) \geq \delta$ iff their Overlap Similarity $\texttt{Over}(\mathcal{X},\mathcal{Y}) = |\mathcal{X} \cup \mathcal{Y}| \geq \frac{\delta}{1+\delta}(|\mathcal{X}| + |\mathcal{Y}|)$. Similarly we can transform the other two similarity functions to Overlap Similarity.

2.2 The Prefix Filter based Methods

To solve the exact set similarity join problem, a brute-force method needs to enumerate $\mathcal{O}(|\mathcal{R}|^2)$ pairs of records and calculate their similarities, which is too expensive. The state-of-the-art methods generally utilize the prefix filter framework to prune some dissimilar record pairs and verify the survived record pairs that are not pruned. The prefix filter based methods first transform the Jaccard Similarity threshold δ to an Overlap Similarity threshold t as follows. For any two records $\mathcal{X}$ and $\mathcal{Y}$, $\texttt{Jac}(\mathcal{X},\mathcal{Y}) \geq \delta$ iff $\texttt{Over}(\mathcal{X},\mathcal{Y}) \geq \frac{\delta}{1+\delta}(|\mathcal{X}| + |\mathcal{Y}|)$. Based on the size filter $|\mathcal{Y}| \geq |\mathcal{X}|\delta$, we have $\texttt{Jac}(\mathcal{X},\mathcal{Y}) \geq \delta$ only if $\texttt{Over}(\mathcal{X},\mathcal{Y}) \geq \delta|\mathcal{X}| = t$. Then they fix a global order and sort the elements in each record by the global order. For each record $\mathcal{X}$, its prefix $\mathsf{prefix}(\mathcal{X})$ consists of its first $|\mathcal{X}| - \lceil t \rceil + 1$ elements. The prefix filter framework guarantees two records $\mathcal{X}$ and $\mathcal{Y}$ are similar only if their prefixes are not disjoint, i.e., $\mathsf{prefix}(\mathcal{X}) \cap \mathsf{prefix}(\mathcal{Y}) \neq \phi$. Next the prefix filter based methods build an inverted index based on the elements in the prefixes and any two records on the same inverted list compose a candidate pair. Finally they verify the candidate pairs. For example, we consider the dataset $\mathcal{R}$ in Table 1 and suppose the threshold $\delta = 0.73$. We suppose the global order first uses the element frequency order and second utilizes the subscript order. Then we have $\mathsf{prefix}(\mathcal{X}_1) = \{x_1, x_2, x_5\}$, $\mathsf{prefix}(\mathcal{X}_2) = \{x_{15}, x_9, x_2\}$, $\mathsf{prefix}(\mathcal{X}_3)=\{x_9, x_1, x_3\}$, $\mathsf{prefix}(\mathcal{X}_4)=\{x_8, x_{12}, x_3\}$ and $\mathsf{prefix}(\mathcal{X}_5) = \{x_1, x_2, x_3\}$. As $\mathsf{prefix}(\mathcal{X}_1) \cap \mathsf{prefix}(\mathcal{X}_2) \neq \phi$, $\langle\mathcal{X}_1, \mathcal{X}_2\rangle$ is a candidate pair. Similarly, we can get 8 candidate pairs in total, while the brute-force method has 10.

id		*The records*	*group*
1	$\mathcal{X}_1$	$\{x_1, x_2, x_5, x_6, x_7, x_{10}, x_{11}, x_{13}, x_{14}\}$	$\mathcal{R}_9$
2	$\mathcal{X}_2$	$\{x_2, x_4, x_5, x_6, x_9, x_{11}, x_{13}, x_{14}, x_{15}\}$	
3	$\mathcal{X}_3$	$\{x_1, x_3, x_6, x_7, x_9, x_{10}, x_{11}, x_{13}, x_{14}\}$	
4	$\mathcal{X}_4$	$\{x_3, x_4, x_5, x_7, x_8, x_{10}, x_{12}, x_{13}, x_{14}\}$	
5	$\mathcal{X}_5$	$\{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_{10}, x_{11}, x_{13}, x_{14}\}$	$\mathcal{R}_{11}$

Table 1: A record dataset $\mathcal{R}$

2.3 Related Work

Exact Set Similarity Joins. There have been many studies on exact set similarity joins [1,2,7,12,17,19–22]. Jiang et al. [9] conducted a comprehensive experimental study on similarity joins. Bayardo et al. [2] proposed the prefix filter as described above. Xiao et al. [22] improved the prefix filter by the position filter and the suffix filter. Moreover they deduced a tighter overlap similarity threshold $t = \frac{2\delta}{1+\delta}|\mathcal{X}|$ instead of $t = \delta|\mathcal{X}|$ and thus reduced the prefix length $|\mathcal{X}| - t + 1$. Wang et al. [20] proposed to include more elements into the prefixes to enhance the pruning power of the prefix filter and gave a cost model to achieve this. Our proposed partition based framework is fundamentally different from the prefix filter framework as we use a set of elements as a signature while the prefix filter framework utilizes a single element in the prefix as a signature.

Sarawagi et al. [12] developed a general algorithm for set joins on predicates with various similarity measures, including Jaccard Similarity and Edit Distance. Arasu et al. [1] proposed PartEnum that utilized two operations, "partition" and "enumeration", for the exact set similarity join in DBMS. Xiao et al. [21] extended the prefix filter techniques for top-k set similarity joins. Verinica et al. [17] and Deng et al. [7] proposed to perform exact set similarity join on MapReduce framework. Wang et al. [19] designed a hybrid similarity function for exact set similarity joins. Deng et al. [11] also proposed a partition based framework but for string similarity joins with Edit Distance constraints. As shown in [7], extending the techniques for set similarity joins leads a time complexity of $\mathcal{O}(s^3)$ for each record with size s, which can be very large in practice. Li et al. [10] studied efficient list-merging methods for set similarity search. All these related work are different from our work. They either solve different problems, such as top-k set similarity join and set similarity search, or solve the problem on different settings, such as MapReduce and DBMS. Our method can be easily extended to work on the MapReduce framework and DBMS (see Section 6.3).

Approximate Set Similarity Joins. Several previous work [3,8,14,23] focus on probabilistic techniques for set similarity joins. Locality Sensitive Hashing (LSH) [8] is the most popular one. An LSH scheme is a distribution on a family of hash functions operating on the sets such that two similar sets are more likely to be hashed into the same bucket. The MinHash [3] is an LSH scheme for Jaccard. Satuluri et al. [14] proposed BayesLSH, a Bayesian algorithm for candidate pruning and similarity estimation using LSH. Zhai et al. [23] proposed a probabilistic algorithm for high dimensional similarity search with very low thresholds. Shrivastava [15] proposed the asymmetric LSH for maximum inner product search. All of these methods are different from our work as they cannot find the exact answers. Moreover, they need to tune parameters which is tedious and are less effective in reducing candidate pairs at lower threshold [2].

3. PARTITION BASED FRAMEWORK

We first propose a partition strategy and then discuss how to utilize it to address the set similarity join problem.

3.1 Record Partition

We propose a partition based framework which partitions each record to several disjoint sub-records such that two records are similar only if they share a common sub-record. To achieve this goal, we need to decide (1) the number of partitions and (2) how to partition the elements into different sub-records. For example, consider the four records $\mathcal{X}_1$, $\mathcal{X}_2$, $\mathcal{X}_3$ and $\mathcal{X}_4$ in Table 1 and suppose $\delta = 0.73$. As shown in Figure 1(b) we partition them to 4 sub-records and any two of them are similar only if they share a common sub-record. Here we only have 3 candidates $\langle \mathcal{X}_1, \mathcal{X}_3 \rangle$, $\langle \mathcal{X}_1, \mathcal{X}_4 \rangle$, $\langle \mathcal{X}_3, \mathcal{X}_4 \rangle$ while the prefix filter based method has 4 candidates.

Number of Partitions. For any two records $\mathcal{X}$ and $\mathcal{Y}$, let $\mathcal{X} \Delta \mathcal{Y}$ denote their symmetric difference (the union without intersection). We have the number of different elements between them is exactly $|\mathcal{X} \Delta \mathcal{Y}|$. If $\mathtt{Jac}(\mathcal{X}, \mathcal{Y}) \geq \delta$, based on the discussion in Section 2.1, we have $\mathtt{Over}(\mathcal{X}, \mathcal{Y}) = |\mathcal{X} \cap \mathcal{Y}| \geq \frac{\delta}{1+\delta}(|\mathcal{X}| + |\mathcal{Y}|)$ and $|\mathcal{Y}| \leq |\mathcal{X}|/\delta$. Thus the number of different elements satisfies

$$\begin{aligned} |\mathcal{X} \Delta \mathcal{Y}| &= |\mathcal{X} \setminus \mathcal{Y}| + |\mathcal{Y} \setminus \mathcal{X}| \\ &\leq |\mathcal{X}| - \frac{\delta(|\mathcal{X}| + |\mathcal{Y}|)}{1+\delta} + |\mathcal{Y}| - \frac{\delta(|\mathcal{X}| + |\mathcal{Y}|)}{1+\delta} \\ &= \frac{1-\delta}{1+\delta}(|\mathcal{X}| + |\mathcal{Y}|) \leq \frac{1-\delta}{\delta}|\mathcal{X}|. \end{aligned} \quad (1)$$

Let $H_l = \lfloor \frac{1-\delta}{\delta} l \rfloor$ where $l = |\mathcal{X}|$. H_l is an upper bound of the number of different elements between $\mathcal{X}$ and any record similar to $\mathcal{X}$ as stated in Lemma 1.

Lemma 1. *Given a record $\mathcal{X}$ with size l, for any record $\mathcal{Y}$ s.t. $\mathtt{Jac}(\mathcal{X}, \mathcal{Y}) \geq \delta$, $|\mathcal{X} \Delta \mathcal{Y}| \leq H_l$.*[1]

As each different element can destroy at most 1 disjoint sub-record, to guarantee that two similar records share at least one common sub-record, for each record $\mathcal{X}$ with size l, we need to partition it to $m \geq H_l + 1$ disjoint sub-records, $\mathcal{X}^1, \mathcal{X}^2, \cdots, \mathcal{X}^m$. For example, consider a record $\mathcal{X}$ with size $l = 9$ and $\delta = 0.73$. We partition $\mathcal{X}$ to 4 sub-records. For any record $\mathcal{Y}$ similar to $\mathcal{X}$, we have $|\mathcal{X} \Delta \mathcal{Y}| \leq H_9 = 3$.

Partition Strategy. If for any record, we can partition the same element into the same disjoint sub-record, called *a homomorphism partition*, we have a good property that $|\mathcal{X} \Delta \mathcal{Y}| = \sum_{i=1}^{m} |\mathcal{X}^i \Delta \mathcal{Y}^i|$ for any two records $\mathcal{X}$ and $\mathcal{Y}$ as shown in Lemma 2. This is because, (1) for any $e \in \mathcal{X}^i \Delta \mathcal{Y}^i$, $e \in \mathcal{X} \Delta \mathcal{Y}$; and (2) for any $e' \in \mathcal{X} \Delta \mathcal{Y}$, $e' \in \mathcal{X}^i \Delta \mathcal{Y}^i$ for one and only one $i \in [1, m]$ as the sub-record $\mathcal{X}^i$ is disjoint with any $\mathcal{X}^j$ and $\mathcal{Y}^j$ where $i \neq j$.

Lemma 2. *For any records $\mathcal{X}$ and $\mathcal{Y}$, if we use an homomorphism partition to partition them into m sub-records, we have $|\mathcal{X} \Delta \mathcal{Y}| = \sum_{i=1}^{m} |\mathcal{X}^i \Delta \mathcal{Y}^i|$ where $\mathcal{X}^i$ and $\mathcal{Y}^i$ are sub-records of $\mathcal{X}$ and $\mathcal{Y}$.*

Thus for any two records $\mathcal{X}$ and $\mathcal{Y}$ with $l = |\mathcal{X}|$ and $s = |\mathcal{Y}|$, we partition them into $m \geq H_l + 1$ sub-records. If $\mathcal{X}^i \neq \mathcal{Y}^i$ for every $i \in [1, m]$, then $\mathcal{X}$ and $\mathcal{Y}$ cannot be similar as stated in Lemma 3, because $\mathcal{X}^i \neq \mathcal{Y}^i$ leads to $|\mathcal{X}^i \neq \mathcal{Y}^i| \geq 1$ and $|\mathcal{X} \Delta \mathcal{Y}| = \sum_{i=1}^{m} |\mathcal{X}^i \neq \mathcal{Y}^i| \geq m > H_l$.

Lemma 3. *For any records $\mathcal{X}$ and $\mathcal{Y}$, suppose we use a homomorphism partition to partition them into $m \geq H_l + 1$ sub-records. If $\mathcal{X}^i \neq \mathcal{Y}^i$ for every $i \in [1, m]$, then $\mathcal{X}$ and $\mathcal{Y}$ cannot be similar.*

To get a homomorphism partition, we can use a hash based method. For each element $e \in \mathcal{X}$, we put it to the $\big((hash(e) \mod m) + 1\big)^{th}$ sub-record where $hash(e)$ maps an element e to an integer. Thus any element will be partitioned into the same sub-record. However this method may introduce a skewed problem: some sub-records have many elements while some sub-records have few elements. To address this issue, we propose a new partition scheme.

First we set a universe $\mathcal{U}$, which consists of m sub-universe $\mathcal{U}^1$, $\mathcal{U}^2$, ..., $\mathcal{U}^m$ s.t. (1) $\cup_{i=1}^{m} \mathcal{U}^i = \mathcal{U}$ and (2) $\mathcal{U}^i \cap \mathcal{U}^j = \phi$ for any $1 \leq i \neq j \leq m$. For any record $\mathcal{X}$, we utilize the universe to get m sub-records $\mathcal{X}^1$, $\mathcal{X}^2$, ..., $\mathcal{X}^m$ s.t. (1) $\cup_{i=1}^{m} \mathcal{X}^i = \mathcal{X}$ and (2) $\mathcal{X}^i \subseteq \mathcal{U}^i$ for any $1 \leq i \leq m$. We use $\mathsf{scheme}(\mathcal{U}, m)$ to denote this partition scheme. For example, Figure 1(a) gives a partition scheme with $m = 4$ and Figure 1(b) shows the sub-records achieved by partitioning the records based on this partition scheme. For any records $\mathcal{X}$ and $\mathcal{Y}$, suppose we partition them based on the same partition scheme $\mathsf{scheme}(\mathcal{U}, m)$ and get the sub-records $\mathcal{X}^1$, ..., $\mathcal{X}^m$ and $\mathcal{Y}^1, \ldots, \mathcal{Y}^m$. As the sub-records $\mathcal{X}^i$ and $\mathcal{Y}^i$ are subsets of the sub-universe $\mathcal{U}^i$ and any two sub-universes are disjoint, $|\mathcal{X} \Delta \mathcal{Y}| = \sum_{i=1}^{m} |\mathcal{X}^i \Delta \mathcal{Y}^i|$ as stated in Lemma 2.

Based on Lemmas 1, 2 and 3, for any record $\mathcal{X}$ with size l and any record $\mathcal{Y}$ similar to $\mathcal{X}$, if we partition them using the same partition scheme $\mathsf{scheme}(\mathcal{U}, m)$ where $m > H_l$, they must share a common sub-record in the same slot. Otherwise, $|\mathcal{X} \Delta \mathcal{Y}| = \sum_{i=1}^{m} |\mathcal{X}^i \Delta \mathcal{Y}^i| \geq m > H_l$ which leads to $\mathtt{Jac}(\mathcal{X}, \mathcal{Y}) < \delta$. Next based on this conclusion, we introduce the partition based algorithm.

3.2 Partition-based Algorithm

The algorithm contains three steps.

- **Partitioning.** Intuitively, the shorter a sub-universe is, the higher probability two records share a common sub-record in this slot and the more candidates we will get. Thus we want the size of the sub-universes as large as possible. To this end, we use an even partition scheme which evenly partitions the universe $\mathcal{U}$ to $m = H_l + 1$ sub-universes with size $\frac{|\mathcal{U}|}{m}$ for the records with size l.[2] For example, Figures 1(a) and 1(b) respectively show the even partition scheme of the universe $\mathcal{U}$ and the sub-records generated based on it. Hereinafter, we use scheme to denote the even partition scheme. Note our techniques can work on any partition scheme. We leave the study of partition schemes as a future work.
- **Building Indexes.** To find the record pairs that share a sub-record, we can use inverted list to index the sub-record

[1] Due to space constraints, we omit the formal proof of all the lemmas.

[2] If $|\mathcal{U}|$ is not divisible by m, we set the size of the first few sub-universes as $\lceil \frac{|\mathcal{U}|}{m} \rceil$ and the size of the rest as $\lfloor \frac{|\mathcal{U}|}{m} \rfloor$.

$\mathcal{U}^1$	$\mathcal{U}^2$	$\mathcal{U}^3$	$\mathcal{U}^4$
$\{x_1, x_2, x_3, x_4\}$	$\{x_5, x_6, x_7, x_8\}$	$\{x_9, x_{10}, x_{11}, x_{12}\}$	$\{x_{13}, x_{14}, x_{15}\}$

(a) ***The even partition scheme*** $\mathsf{scheme}(\mathcal{U}, m = 4)$ ***for*** $l = 9$

id	$\mathcal{X}^1_{id}$	$\mathcal{X}^2_{id}$	$\mathcal{X}^3_{id}$	$\mathcal{X}^4_{id}$
1	$\{x_1, x_2\}$	$\{x_5, x_6, x_7\}$	$\{x_{10}, x_{11}\}$	$\{x_{13}, x_{14}\}$
2	$\{x_2, x_4\}$	$\{x_5, x_6\}$	$\{x_9, x_{11}\}$	$\{x_{13}, x_{14}, x_{15}\}$
3	$\{x_1, x_3\}$	$\{x_6, x_7\}$	$\{x_9, x_{10}, x_{11}\}$	$\{x_{13}, x_{14}\}$
4	$\{x_3, x_4\}$	$\{x_5, x_7, x_8\}$	$\{x_{10}, x_{12}\}$	$\{x_{13}, x_{14}\}$

(b) ***The sub-records of*** $\mathcal{R}_9$ ***based on*** $\mathsf{scheme}(\mathcal{U}, m = 4)$

$\mathcal{I}^1_9$	$\mathcal{I}^2_9$	$\mathcal{I}^3_9$	$\mathcal{I}^4_9$
$\{x_1, x_2\} \mapsto 1$	$\{x_5, x_6, x_7\} \mapsto 1$	$\{x_{10}, x_{11}\} \mapsto 1$	$\{x_{13}, x_{14}\} \rightarrow 1, 3, 4$
$\{x_2, x_4\} \mapsto 2$	$\{x_5, x_6\} \mapsto 2$	$\{x_9, x_{11}\} \mapsto 2$	$\{x_{13}, x_{14}, x_{15}\} \rightarrow 2$
$\{x_1, x_3\} \mapsto 3$	$\{x_6, x_7\} \mapsto 3$	$\{x_9, x_{10}, x_{11}\} \mapsto 3$	
$\{x_3, x_4\} \mapsto 4$	$\{x_5, x_7, x_8\} \mapsto 4$	$\{x_{10}, x_{12}\} \mapsto 4$	

(c) ***The sub-record indexes of*** $\mathcal{R}_9$

$\mathcal{D}^1_9$	$\mathcal{D}^2_9$	$\mathcal{D}^3_9$	$\mathcal{D}^4_9$
$\{x_1\} \rightarrow 1, 3$	$\{x_5\} \rightarrow 2$	$\{x_9\} \rightarrow 2$	$\{x_{13}\} \rightarrow 1, 3, 4$
$\{x_2\} \rightarrow 1, 2$	$\{x_6\} \rightarrow 2, 3$	$\{x_{10}\} \rightarrow 1, 4$	$\{x_{14}\} \rightarrow 1, 3, 4$
$\{x_3\} \rightarrow 3, 4$	$\{x_7\} \rightarrow 3$	$\{x_{11}\} \rightarrow 1, 2$	$\{x_{13}, x_{14}\} \rightarrow 2$
$\{x_4\} \rightarrow 2, 4$	$\{x_5, x_6\} \rightarrow 1$	$\{x_{12}\} \rightarrow 4$	$\{x_{14}, x_{15}\} \rightarrow 2$
	$\{x_5, x_7\} \rightarrow 1, 4$	$\{x_9, x_{10}\} \rightarrow 3$	$\{x_{13}, x_{15}\} \rightarrow 2$
	$\{x_6, x_7\} \rightarrow 1$	$\{x_9, x_{11}\} \rightarrow 3$	
	$\{x_5, x_8\} \rightarrow 4$	$\{x_{10}, x_{11}\} \rightarrow 3$	
	$\{x_7, x_8\} \rightarrow 4$		

(d) ***The 1-deletion neighborhood indexes of*** $\mathcal{R}_9$

Figure 1:The sub-records and inverted indexes of $\mathcal{R}_9$

and the records on the same inverted list share a common sub-record. To this end, we sort and group all the records in $\mathcal{R}$ based on their sizes and $\mathcal{R}_l$ contains all the records in $\mathcal{R}$ with size l which share the same partition scheme $\mathsf{scheme}(\mathcal{U}, m = H_l + 1)$. For each group $\mathcal{R}_l$, we build m inverted indexes $\mathcal{I}^i_l$ for $1 \le i \le m$ as follows. For each record $\mathcal{X} \in \mathcal{R}_l$, we partition it based on $\mathsf{scheme}(\mathcal{U}, m)$. For each sub-record $\mathcal{X}^i$, we append $\mathcal{X}.id$ to the inverted list $\mathcal{I}^i_l[\mathcal{X}^i]$ where $\mathcal{X}.id$ is the *id* of $\mathcal{X}$.

- **Finding Similar Pairs.** We discuss how to find similar record pairs using the indexes. For each record $\mathcal{X} \in \mathcal{R}$ with size s, we find its similar records using the inverted indexes. Based on the size filter, the sizes of its similar records are within $[\delta s, s]$[3]. For each possible size $l \in [\delta s, s]$, we partition $\mathcal{X}$ to $m = H_l + 1$ disjoint sub-records using the even partition scheme $\mathsf{scheme}(\mathcal{U}, m)$. Then we access the inverted list $\mathcal{I}^i_l[\mathcal{X}^i]$ and add all the records in this inverted list, who share the common sub-record $\mathcal{X}^i$ with $\mathcal{X}$, into the candidate set $\mathcal{C}$ for each $1 \le i \le m$. Finally, we verify the candidates by calculating their `Jaccard Similarity` to $\mathcal{X}$ and return the result pairs.

The pseudo code of the partition based framework is shown in Algorithm 1. It takes a dataset $\mathcal{R}$ and a similarity threshold δ as input and outputs the set of all similar record pairs. It first scans the dataset $\mathcal{R}$ and gets the universe $\mathcal{U}$ of $\mathcal{R}$ (Line 1). Then it groups and sorts the records by their sizes in ascending order (Line 2). Next for each record $\mathcal{X} \in \mathcal{R}$ with size s and each group $\mathcal{R}_l$ where $\delta s \le l \le s$, it first partitions the record $\mathcal{X}$ to $m = H_l + 1$ disjoint sub-records using $\mathsf{scheme}(\mathcal{U}, m)$ and then accesses the inverted list $\mathcal{I}^i_l[\mathcal{X}^i]$ and adds all the records in the list to candidate set $\mathcal{C}$ for each sub-record $\mathcal{X}^i$ where $1 \le i \le m$ (Lines 3 to 7). Then it verifies the candidates by calculating the `Jaccard Similarity` and adds the candidates similar to $\mathcal{X}$ to the result set $\mathcal{A}$ (Lines 8 to 9). Finally it repartitions the record $\mathcal{X}$ to $m' = H_s + 1$ disjoint sub-records using $\mathsf{scheme}(\mathcal{U}, m')$ and appends an entry $\mathcal{X}.id$ to the inverted list $\mathcal{I}^i_s[\mathcal{X}^i]$ for each sub-record $\mathcal{X}^i$ where $1 \le i \le m'$ (Lines 10). It returns $\mathcal{A}$ at last (Line 11).

Algorithm 1: The Partition Based Framework

```
Input: $\mathcal{R}$: the dataset; $\delta$: the similarity threshold;
Output: $\mathcal{A}$: $\{\langle \mathcal{X}, \mathcal{Y} \rangle \big| \mathtt{Jac}(\mathcal{X}, \mathcal{Y}) \ge \delta, \mathcal{X} \in \mathcal{R}, \mathcal{Y} \in \mathcal{R}\}$
1  retrieve the universe $\mathcal{U}$ of $\mathcal{R}$;
2  group and ascendingly sort the records by sizes;
3  foreach record $\mathcal{X} \in \mathcal{R}$ with size $s$ do
4      foreach $\delta s \le l \le s$ do
5          partition $\mathcal{X}$ to $m = H_l + 1$ sub-records using the
           even partition scheme $\mathsf{scheme}(\mathcal{U}, m)$;
6          foreach $1 \le i \le m$ do
7              add all the records in $\mathcal{I}^i_l[\mathcal{X}^i]$ into $\mathcal{C}$;
8      foreach record $\mathcal{Y}$ in $\mathcal{C}$ do
9          add $\mathcal{Y}$ into result set $\mathcal{A}$ if $\mathtt{Jac}(\mathcal{X}, \mathcal{Y}) \ge \delta$;
10     repartition $\mathcal{X}$ to $m' = H_s + 1$ sub-records using
       $\mathsf{scheme}(\mathcal{U}, m')$ and append $\mathcal{X}.id$ to $\mathcal{I}^i_s[\mathcal{X}^i]$ for each
       $1 \le i \le m'$;
11 return $\mathcal{A}$;
```

Example 1. *Consider the dataset* $\mathcal{R}$ *in Table 1. Suppose the threshold is* $\delta = 0.73$. *We use the following three steps to find similar pairs.*

- **Partitioning.** *As shown in Figure 1, for* $\mathcal{R}_9$, *as* $m = H_9 + 1 = 4$, *we evenly partition the universe* $\mathcal{U}$ *into 4 disjoint sub-universes where* $\mathcal{U}^1 = \{x_1, x_2, x_3, x_4\}$, $\mathcal{U}^2 = \{x_5, x_6, x_7, x_8\}$, $\mathcal{U}^3 = \{x_9, x_{10}, x_{11}, x_{12}\}$ *and* $\mathcal{U}^4 = \{x_{13}, x_{14}, x_{15}\}$.
- **Building Indexes.** *We partition the records in* $\mathcal{R}_9$ *into 4 sub-records based on the partition scheme, For* $\mathcal{X}_1 \in \mathcal{R}_9$, *we have* $\mathcal{X}^1_1 = \{x_1, x_2\}$, $\mathcal{X}^2_1 = \{x_5, x_6, x_7\}$, $\mathcal{X}^3_1 = \{x_{10}, x_{11}\}$ *and* $\mathcal{X}^4_1 = \{x_{13}, x_{14}\}$. *We append the id 1 of* $\mathcal{X}_1$ *to the inverted lists* $\mathcal{I}^1_9[\mathcal{X}^1_1]$, $\mathcal{I}^2_9[\mathcal{X}^2_1]$, $\mathcal{I}^3_9[\mathcal{X}^3_1]$ *and* $\mathcal{I}^4_9[\mathcal{X}^4_1]$. *Similarly we can partition the other records and build the inverted indexes.*
- **Finding Similar Pairs.** *Next we find similar record pairs. Consider the record* $\mathcal{X}_5$. *As* $|\mathcal{X}_5| = 11$ *and* $\delta|\mathcal{X}_5| = 8.25$, *we need to probe the three groups* $\mathcal{R}_9$, $\mathcal{R}_{10}$, *and* $\mathcal{R}_{11}$. *For* $\mathcal{R}_9$ *we partition* $\mathcal{X}_5$ *to* $m = H_9 + 1 = 4$ *parts based on* $\mathsf{scheme}(\mathcal{U}, m)$: $\mathcal{X}^1_5 = \{x_1, x_2, x_3, x_4\}$, $\mathcal{X}^2_5 = \{x_5, x_6, x_7\}$, $\mathcal{X}^3_5 = \{x_{10}, x_{11}\}$ *and* $\mathcal{X}^4_5 = \{x_{13}, x_{14}\}$. *We access the inverted lists* $\mathcal{I}^1_9[\mathcal{X}^1_5]$, $\mathcal{I}^2_9[\mathcal{X}^2_5]$, $\mathcal{I}^3_9[\mathcal{X}^3_5]$ *and* $\mathcal{I}^4_9[\mathcal{X}^4_5]$ *and find* $\mathcal{X}_1$ *from* $\mathcal{I}^2_9[\mathcal{X}^2_5]$, $\mathcal{X}_1$ *from* $\mathcal{I}^3_9[\mathcal{X}^3_5]$ *and* $\mathcal{X}_1, \mathcal{X}_3, \mathcal{X}_4$ *from* $\mathcal{I}^4_9[\mathcal{X}^4_5]$. *Thus we get three candidates* $\mathcal{X}_1$, $\mathcal{X}_3$, *and* $\mathcal{X}_4$. *We calculate their similarity to* $\mathcal{X}_5$. *As* $\mathtt{Jac}(\mathcal{X}_1, \mathcal{X}_5) = \frac{9}{11} \ge \delta$ *we get a result* $\langle \mathcal{X}_1, \mathcal{X}_5 \rangle$.

We prove the correctness and completeness of the partition based framework as formalized in Theorem 1.

Theorem 1. *The partition based framework satisfies (1) correctness: a record pair returned by the framework must be a similar pair; and (2) completeness: for any similar record pair the framework must return the pair as a result.*

Complexity Analysis: We first analyze the time complexity. Suppose the maximum record size in $\mathcal{R}$ is n. The time complexity of getting the universe and sorting the records

[3] Here we use a commonly used trick in self-join. For each record, we only find its similar records with smaller or equal size.

are $\mathcal{O}(|\mathcal{R}|n)$ and $\mathcal{O}(|\mathcal{R}|\log|\mathcal{R}|)$. For each record, it partitions the record for at most $n-\delta*n+1$ groups. The time complexity is $\mathcal{O}((1-\delta)n^2)$. It also needs to access the inverted lists and verify the candidates. The time complexity is $\mathcal{O}(|\mathcal{L}|+n|\mathcal{C}|)$ where $|\mathcal{L}|$ is the sum of the size of the accessed inverted lists and $|\mathcal{C}|$ is the number of candidates. The time complexity of building the inverted indexes is $\mathcal{O}(|\mathcal{R}|n)$. Thus the total time complexity of the framework is $\mathcal{O}\big(|\mathcal{R}|\log|\mathcal{R}|+|\mathcal{R}|(1-\delta)n^2+|\mathcal{L}|+n|\mathcal{C}|\big)$.

Next we analyze the space complexity. We need to store the inverted indexes. The number of entries in the inverted indexes is no more than $|\mathcal{R}|(H_n+1)$. The number of inverted lists is not larger than the number of entries in the inverted indexes. In addition, we need to store the candidates for each record, whose number is not larger than $|\mathcal{R}|$. Thus the space complexity is $\mathcal{O}(|\mathcal{R}|H_n)$.

We observe that some records may have frequent sub-records which leads to a large $|\mathcal{C}|$ and low performance. For example, in Example 1, $\mathcal{X}_5$ contains a frequent sub-record $\mathcal{X}_5^4$ which has 3 candidates. Next we discuss how to reduce the size of accessed lists $|\mathcal{L}|$, which is proportional to $|\mathcal{C}|$.

4. SUB-RECORD SELECTION

To avoid using frequent sub-records to generate candidates, we propose a sub-record selection method. Note that if two sub-records $\mathcal{X}^i$ and $\mathcal{Y}^i$ contain two (or more) different elements, we can skip a frequent sub-record $\mathcal{X}^j$ of $\mathcal{X}$ and do not check whether $\mathcal{X}^j$ equals $\mathcal{Y}^j$. This is because even if we ignore the frequent sub-record $\mathcal{X}^j$, if $\mathcal{X}$ and $\mathcal{Y}$ do not share another common sub-record, they cannot be similar (as $|\mathcal{X}\,\Delta\,\mathcal{Y}|=|\mathcal{X}^i\,\Delta\,\mathcal{Y}^i|+|\mathcal{X}^j\,\Delta\,\mathcal{Y}^j|+\sum_{k\neq i,j}|\mathcal{X}^k\,\Delta\,\mathcal{Y}^k|\geq 2+0+m-2=m$). For instance, in Example 1, as $\mathcal{X}_3^1$ and $\mathcal{X}_5^1$ contain 2 different elements, $\mathcal{X}_3^2\neq\mathcal{X}_5^2$ and $\mathcal{X}_3^3\neq\mathcal{X}_5^3$, $\mathcal{X}_3$ and $\mathcal{X}_5$ must have no less than 4 different elements. Thus we can prune this pair without checking if $\mathcal{X}_3^4$ equals $\mathcal{X}_5^4$. Similarly we can prune the pair of $\mathcal{X}_4$ and $\mathcal{X}_5$.

To efficiently detect whether two sub-records have 2 different elements, we propose 1-deletion neighborhoods (generated by eliminating an element from the sub-records) in Section 4.1. We can use a mixture of the sub-records and their 1-deletion neighborhoods in the partition based framework instead of all sub-records. As there are multiple allocation strategies to generate the mixture, we discuss how to evaluate an allocation and propose a dynamic-programming method to choose the optimal one in Section 4.2.

4.1 The 1-Deletion Neighborhoods

Given a non-empty set $\mathcal{Z}$, its 1-deletion neighborhoods are its subsets with size of $|\mathcal{Z}|-1$. The empty set has no 1-deletion neighborhoods. Next we give the formal definition of 1-deletion neighborhoods as follows.

DEFINITION 2 (1-DELETION NEIGHBORHOODS). *Given a non-empty set $\mathcal{Z}$, its 1-deletion neighborhoods are $\mathcal{Z}\setminus\{z\}$ for each $z\in\mathcal{Z}$.*

For example, consider $\mathcal{X}_5^2=\{x_5,x_6,x_7\}$ in Example 1. Its 1-deletion neighborhoods are $\{x_5,x_6\}$, $\{x_6,x_7\}$ and $\{x_5,x_7\}$. Let $\mathsf{del}(\mathcal{X})=\{\mathcal{X}\setminus\{x\}|x\in\mathcal{X}\}$ be the set of 1-deletion neighborhoods of $\mathcal{X}$, we have $\mathsf{del}(\mathcal{X}_5^2)$=$\{\{x_5,x_6\},\{x_6,x_7\},\{x_5,x_7\}\}$.

For any two sets $\mathcal{X}$ and $\mathcal{Y}$, if $\mathcal{X}\neq\mathcal{Y}$ it is obviously that they must contain at least 1 different element, i.e., $|\mathcal{X}\,\Delta\,\mathcal{Y}|\geq 1$. Furthermore, based on Definition 2, we observe that if $\mathcal{X}\neq\mathcal{Y}$, $\mathcal{X}\notin\mathsf{del}(\mathcal{Y})$ and $\mathcal{Y}\notin\mathsf{del}(\mathcal{Y})$, they must have at least two different elements as stated in Lemma 4.

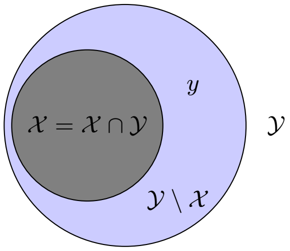

$\mathcal{X}=\mathcal{X}\cap\mathcal{Y}$
y
$\mathcal{Y}$
$\mathcal{Y}\setminus\mathcal{X}$

Figure 2: Venn diagram of $\mathcal{X}\setminus\mathcal{Y}=\phi$ and $\mathcal{Y}\setminus\mathcal{X}=\{y\}$

LEMMA 4. *For any two sets $\mathcal{X}$ and $\mathcal{Y}$, if $\mathcal{X}\neq\mathcal{Y}$, $\mathcal{X}\notin$ $\mathsf{del}(\mathcal{Y})$ and $\mathcal{Y}\notin\mathsf{del}(\mathcal{X})$, $|\mathcal{X}\,\Delta\,\mathcal{Y}|\geq 2$.*

For example, in Example 1, consider $\mathcal{X}_5^2=\{x_5,x_6,x_7\}$ and $\mathcal{X}_4^2=\{x_5,x_7,x_8\}$. As $\mathcal{X}_5^2\neq\mathcal{X}_4^2$, $\mathcal{X}_5^2\notin\mathsf{del}(\mathcal{X}_4^2)$ and $\mathcal{X}_4^2\notin\mathsf{del}(\mathcal{X}_5^2)$, $|\mathcal{X}_5^2\,\Delta\,\mathcal{X}_4^2|\geq 2$. Here, $|\mathcal{X}_5^2\,\Delta\,\mathcal{X}_4^2|=2$.

Overview. We discuss how to utilize the 1-deletion neighborhoods for the exact set similarity join. Given two records $\mathcal{X}$ and $\mathcal{Y}$ where $l=|\mathcal{X}|\leq s=|\mathcal{Y}|$, we partition them to $m=H_l+1$ sub-records, $\mathcal{X}^1,\mathcal{X}^2,\cdots,\mathcal{X}^m$, and $\mathcal{Y}^1,\mathcal{Y}^2,\cdots,\mathcal{Y}^m$ respectively. There are three cases between $\mathcal{X}^i$ and $\mathcal{Y}^i$:

Case 0: We skip $\mathcal{X}^i$ and $\mathcal{Y}^i$.
Case 1: We use $\mathcal{X}^i$ and $\mathcal{Y}^i$. If $\mathcal{X}^i=\mathcal{Y}^i$, we take $\mathcal{X}$ and $\mathcal{Y}$ as a candidate pair.
Case 2: We use $\mathcal{X}^i$, $\mathcal{Y}^i$ and their 1-deletion neighborhoods. If $\mathcal{X}^i=\mathcal{Y}^i$, $\mathcal{X}^i\in\mathsf{del}(\mathcal{Y}^i)$ *or* $\mathcal{Y}^i\in\mathsf{del}(\mathcal{X}^i)$, we take $\mathcal{X}$ and $\mathcal{Y}$ as a candidate pair.

The framework method only uses case 1. Here we consider how to use the three cases to achieve high performance. We can use a m-dimensional vector $(v_1,v_2,\ldots,v_m)$ to decide which cases we use, where $v_i\in\{0,1,2\}$ corresponding to the three cases. For example, consider the record $\mathcal{X}_5$ in Table 1 with size $s=11$ and the group $\mathcal{R}_l$ where $l=9$ and suppose $\delta=0.73$. (2,0,2,0) is a 4-dimensional vector, which denotes that we use case 2 for the first sub-record, case 0 for the second sub-record, case 2 for the third sub-record, case 0 for the fourth sub-record.

Next we consider the value of $\sum_{i=1}^m v_i$. First, suppose $\sum_{i=1}^m v_i\geq m=H_l+1$. If there is no matching for the vector, i.e., (1) for $v_i=1$, $\mathcal{X}^i\neq\mathcal{Y}^i$; (2) for $v_i=2$, $\mathcal{X}^i\neq\mathcal{Y}^i,\mathcal{X}^i\notin$ $\mathsf{del}(\mathcal{Y}^i)$ *and* $\mathcal{Y}^i\notin\mathsf{del}(\mathcal{X}^i)$, $\mathcal{X}$ and $\mathcal{Y}$ cannot be similar. This is because for case 1, there is at least one mismatch element, and for case 2, there are at least two mismatch elements, and thus there are at least $\sum_{i=1}^m v_i\geq m=H_l+1$ mismatch elements, i.e., $|\mathcal{X}\,\Delta\,\mathcal{Y}|\geq H_l+1$. Based on Lemma 1, $\mathcal{X}$ and $\mathcal{Y}$ cannot be similar.

Actually, we can provide a tighter bound than $m=H_l+1$ by using the fact that $\mathcal{X}$ and $\mathcal{Y}$ are similar only if $|\mathcal{X}\,\Delta\,\mathcal{Y}|\leq\frac{1-\delta}{1+\delta}(|\mathcal{X}|+|\mathcal{Y}|)$ as shown in Equation 1. Let $H(l,s)=\lfloor\frac{1-\delta}{1+\delta}(l+s)\rfloor$ where $|\mathcal{X}|$=l and $|\mathcal{Y}|$=s. If $\sum_{i=1}^m v_i\geq H(l,s)+1$ and there is no matching, $\mathcal{X}$ and $\mathcal{Y}$ cannot be similar.

Then, we consider $\sum_{i=1}^m v_i<H(l,s)+1$. Even there is no matching, $\mathcal{X}$ and $\mathcal{Y}$ still can be similar, as there can be up to $H(l,s)$ mismatch elements which can destroy all the selected sub-records and 1-deletion neighborhoods. Thus in this case, the method may involve false negatives.

Thus to use this method, we need to guarantee $\sum_{i=1}^m v_i\geq H(l,s)+1$. To minimize the chance of taking two records as a candidate, we want $\sum_{i=1}^m v_i$ as small as possible. Thus we only consider the vector with $\sum_{i=1}^m v_i=H(l,s)+1$. We call this kind of vector an allocation as defined below.

Algorithm 2: Deletion Neighborhood based Framework

```
  // replace Line 6 to 7 in Algorithm 1
1 generate an allocation $\mathcal{V}_l^s$;
2 foreach $1 \le i \le m$ do
3     if $v_i = 1$ then
4         add all the records in $\mathcal{I}_l^i[\mathcal{X}^i]$ into $\mathcal{C}$;
5     if $v_i = 2$ then
6         add all the records in $\mathcal{I}_l^i[\mathcal{X}^i]$, $\mathcal{D}_l^i[\mathcal{X}^i]$ and every
          $\mathcal{I}_l^i[\mathsf{del}(\mathcal{X}^i)]$ into $\mathcal{C}$;
  // right after Line 10 in Algorithm 1
7 for each $\mathcal{X}^i$ append $\mathcal{X}.id$ to every $\mathcal{D}_s^i[\mathsf{del}(\mathcal{X}^i)]$;
```

Definition 3 (Allocation). *Given two integers l and s, a vector $\mathcal{V}_l^s = (v_1, v_2, \ldots, v_m)$ is an allocation if (1)$m = H_l + 1$, (2)$v_i \in \{0, 1, 2\}$ and (3) $\sum_{i=1}^{m} v_i = H(l, s)+1$.*

We can prove that for any allocation, given two records, if there is not matching in the allocation, they cannot be similar as shown in Lemma 5.

Lemma 5. *Given two records $\mathcal{X}$ and $\mathcal{Y}$ with $m = H_l + 1$ sub-records and an allocation $\mathcal{V}_l^s$ where $l = |\mathcal{X}|$ and $s = |\mathcal{Y}|$, if there is no matching with respect to $\mathcal{V}_l^s$, $\mathtt{Jac}(\mathcal{X}, \mathcal{Y}) < \delta$.*

Algorithm. Next we adapt the partition based framework to the new filter condition. We first discuss the indexes. To utilize the new filter condition, besides the inverted index $\mathcal{I}_l^i$, we also need to build an inverted index $\mathcal{D}_l^i$ for the 1-deletion neighborhoods of the i^{th} sub-records of the records in $\mathcal{R}_l$ where $1 \le i \le m = H_l + 1$ (see Figure 1). For each record $\mathcal{X} \in \mathcal{R}_l$, we partition it into m sub-records. For each sub-record $\mathcal{X}^i$, except appending $\mathcal{X}.id$ to the inverted list $\mathcal{I}_l^i[\mathcal{X}^i]$, we also append $\mathcal{X}.id$ to every $\mathcal{D}_l^i[\mathsf{del}(\mathcal{X}^i)]$, where $\mathcal{D}_l^i[\mathsf{del}(\mathcal{X}^i)]$ denotes all the inverted lists $\mathcal{D}_l^i[\mathcal{X}^i \setminus \{x\}]$ for every $x \in \mathcal{X}^i$, so does $\mathcal{I}_l^i[\mathsf{del}(\mathcal{X}^i)]$.

Then for any record $\mathcal{X} \in \mathcal{R}$ with size s, we utilize the two indexes $\mathcal{I}$ and $\mathcal{D}$ to find its similar records. For each $l \in [\delta * s, s]$, we partition $\mathcal{X}$ to $m = H_l + 1$ disjoint sub-records using the even partition scheme $\mathsf{scheme}(\mathcal{U}, m)$ and select an allocation $\mathcal{V}_l^s = (v_1, v_2, \ldots, v_m)$ (we discuss how to select the allocation in Section 4.2). For each sub-record $\mathcal{X}^i$, if $v_i = 1$ we add all the records $\mathcal{Y}$ in $\mathcal{I}_l^i[\mathcal{X}^i]$ into candidate set $\mathcal{C}$, as $\mathcal{X}^i = \mathcal{Y}^i$. Similarly, if $v_i = 2$ we add all the records $\mathcal{Y}$ in $\mathcal{I}_l^i[\mathcal{X}^i]$, $\mathcal{D}_l^i[\mathcal{X}^i]$ and every $\mathcal{I}_l^i[\mathsf{del}(\mathcal{X}^i)]$ into $\mathcal{C}$ as they respectively indicate $\mathcal{X}^i = \mathcal{Y}^i$, $\mathcal{X}^i \in \mathsf{del}(\mathcal{Y}^i)$ and $\mathcal{Y}^i \in \mathsf{del}(\mathcal{X}^i)$. Based on Lemma 5, they are all candidates. In this way, we can adapt the partition based framework to the deletion neighborhood based framework.

The pseudo-code of the deletion neighborhood based framework is shown in Algorithm 2. Instead of directly accessing the inverted list and adding the entries to the candidate set $\mathcal{C}$, it generates an allocation $\mathcal{V}_l^s$ for each record $\mathcal{X}$ with size s and group $\mathcal{R}_l$ (Line 1). Then for each dimension v_i of the allocation $\mathcal{V}_l^s$ where $1 \le i \le m$, if $v_i = 1$, it adds all the records in the inverted list $\mathcal{I}_l^i[\mathcal{X}^i]$ into $\mathcal{C}$ (Lines 3 to 4). Otherwise if $v_i = 2$, it adds all the records in $\mathcal{I}_l^i[\mathcal{X}^i]$, $\mathcal{D}_l^i[\mathcal{X}^i]$ and $\mathcal{I}_l^i[\mathsf{del}(\mathcal{X}^i)]$ into $\mathcal{C}$ (Lines 5 to 6). In the indexing phase, besides appending $\mathcal{X}.id$ to $\mathcal{I}_l^i[\mathcal{X}^i]$, it also appends $\mathcal{X}.id$ to every inverted list of $\mathcal{D}_s^i[\mathsf{del}(\mathcal{X}^i)]$ (Line 7). The rest parts are the same as the partition based framework.

Example 2. *Following the same setting as Example 1, we partition the records in $\mathcal{R}_9$ to $m = H_9 + 1 = 4$ sub-records. We build the inverted indexes for their 1-deletion neighborhoods as shown in Figure 1 (c). Next we find the result pairs. Consider $\mathcal{X}_5$ and $\mathcal{R}_9$. Suppose we select the allocation $\mathcal{V}_9^{11} = (2, 0, 2, 0)$ for them. For $\mathcal{X}_5^1$, as $v_1 = 2$, we access $\mathcal{I}_9^1[\mathcal{X}_5^1]$, $\mathcal{D}_9^1[\mathcal{X}_5^1]$ and every $\mathcal{I}_9^1[\mathsf{del}(\mathcal{X}_5^1)]$ and get no candidate. For $\mathcal{X}_5^2$, as $v_2 = 0$, we do nothing. For $\mathcal{X}_5^3$, as $v_3 = 2$, we access $\mathcal{I}_9^3[\mathcal{X}_5^3]$, $\mathcal{D}_9^3[\mathcal{X}_5^3]$ and every $\mathcal{I}_9^3[\mathsf{del}(\mathcal{X}_5^3)]$ and get $\mathcal{X}_1$ and $\mathcal{X}_3$. We add $\mathcal{X}_1$ and $\mathcal{X}_3$ to $\mathcal{C}$. For $\mathcal{X}_5^4$, as $v_4 = 0$ we skip it. Finally we verify the records in $\mathcal{C}$ and find a result pair $\langle \mathcal{X}_1, \mathcal{X}_5 \rangle$.*

We prove the correctness and completeness of the deletion neighborhood based framework as formalized in Theorem 2.

Theorem 2. *The deletion neighborhood based framework satisfies correctness and completeness.*

Space Complexity: We analyze the space complexity of the inverted indexes. For each record, it generates at most $m = H_n + 1$ sub-records and n 1-deletion neighborhoods where n is the maximum record size in $\mathcal{R}$. Thus there are $\mathcal{O}(|\mathcal{R}|n)$ entries in the inverted lists. As the number of inverted lists is no more than the number of entries in them, the space complexity is $\mathcal{O}(|\mathcal{R}|n)$ which is relatively small.

Note that the partition based framework is a special case of the deletion neighborhood based framework when the allocation is assigned as $\mathbf{1}$, i.e., a vector with all ones. Actually given a record and a group there are many kinds of allocations, and a nature question is how to evaluate the allocation and select the optimal one. Next we answer this question.

4.2 Optimal Allocation Selection

We first discuss how to evaluate an allocation. As shown in the deletion neighborhood based framework, based on the allocation we need to access the inverted lists $\mathcal{I}_l^i[\mathcal{X}^i]$, $\mathcal{D}_l^i[\mathcal{X}^i]$ or $\mathcal{I}_l^i[\mathsf{del}(\mathcal{X}^i)]$ for each record $\mathcal{X}$ and group $\mathcal{R}_l$. The shorter the inverted list we access, the less time we take and the less candidates we get. Thus we want to minimize the total size of the inverted lists we access. To this end, given a record $\mathcal{X}$ with size s and a group $\mathcal{R}_l$, let

$$c_0^i = 0;\ \ c_1^i = \big|\mathcal{I}_l^i[\mathcal{X}^i]\big|;\ \ c_2^i = \big|\mathcal{I}_l^i[\mathcal{X}^i]\big| + \big|\mathcal{D}_l^i[\mathcal{X}^i]\big| + \big|\mathcal{I}_l^i[\mathsf{del}(\mathcal{X}^i)]\big|$$

where $1 \le i \le m = H_l + 1$, we define the cost of the allocation $\mathcal{V}_l^s = (v_1, v_2, \ldots v_m)$ as $\mathsf{cost}(\mathcal{V}_l^s) = \sum_{i=1}^{m} c_{v_i}^i$ which is the size of the inverted lists the allocation needs to access. Our objective is to select the allocation with the minimum cost. Next we formulate the optimal allocation selection problem.

Definition 4. (Optimal Allocation Selection) *Given a record $\mathcal{X}$ with size s and a group $\mathcal{R}_l$, the optimal allocation selection problem is to select an allocation $\mathcal{V}_l^s$ with the minimum cost.*

For example, the costs for the record $\mathcal{X}_5$ and the group $\mathcal{R}_9$ are shown in Figure 4(a). The optimal allocation is $\mathcal{V}_9^{11} = (2, 1, 1, 0)$ with a cost of $\mathsf{cost}(\mathcal{V}_9^{11}) = c_2^1 + c_1^2 + c_1^3 + c_0^4 = 2$.

Next we introduce a dynamic programming method to solve the optimal allocation selection problem. Let $\mathsf{cost}(i, j)$ be the minimum value of $\sum_{k=1}^{i} c_{v_k}^i$ such that $\sum_{k=1}^{i} v_k = j$ where $v_k \in \{0, 1, 2\}$ and $\mathcal{V}(i, j)$ denotes the corresponding vector $(v_1, v_2, \ldots, v_i)$. Then the optimal allocation selection problem is to calculate the minimum cost $\mathsf{cost}(m, H(l, s)+1)$ and the optimal allocation $\mathcal{V}(m, H(l, s) + 1)$. To calculate $\mathsf{cost}(i, j)$, considering the value of the last dimension v_i of $\mathcal{V}(i, j)$, which can be 0, 1 or 2, we have

$$\mathsf{cost}(i, j) = min \begin{cases} \mathsf{cost}(i-1, j) + c_0^i \\ \mathsf{cost}(i-1, j-1) + c_1^i \\ \mathsf{cost}(i-1, j-2) + c_2^i \end{cases}$$

which is equivalent to

i	c_0^i	c_1^i	c_2^i
1	0	0	0
2	0	1	3
3	0	1	2
4	0	3	4

(a) ***The costs***

i	$\nabla_0^i = c_1^i - c_0^i$	$\nabla_1^i = c_2^i - c_1^i$
1	0	0
2	1	2
3	1	1
4	3	1

(b) ***The cost increments***

cost	-1	0	1	2	3	4	*concatenated vectors*
0	∞	0	∞	∞	∞	∞	$\mathcal{V}(0,0) = ()$
1	∞	0	0	0	∞	∞	$\mathcal{V}(1,2) = (\mathit{2})$
2	∞	0	0	0	1	3	$\mathcal{V}(2,2) = (2, \mathit{0})$
3	∞	0	0	0	1	2	$\mathcal{V}(3,4) = (2, 0, \mathit{2})$
4	∞	0	0	0	1	2	$\mathcal{V}(4,4) = (2, 0, 2, \mathit{0})$

(c) ***The*** **cost** ***matrix for*** $\mathcal{X}_5$ ***and*** $\mathcal{R}_9$

Figure 3: An optimal allocation selection example

$$\mathsf{cost}(i,j) = \min_{v \in \{0,1,2\}} \mathsf{cost}(i-1, j-v) + c_v^i. \qquad (2)$$

Moreover let

$$v_{\min} = \arg\min_{v \in \{0,1,2\}} \mathsf{cost}(i-1, j-v) + c_v^i, \qquad (3)$$

$v_{\min}$ is the i^{th} (the last) dimension of $\mathcal{V}(i,j)$ and we have $\mathcal{V}(i,j) = \mathcal{V}(i-1, j-v_{\min}) \oplus v_{\min}$ where $\oplus$ concatenates $v_{\min}$ to the end of $\mathcal{V}(i-1, j-v_{\min})$. Next we discuss the initialization. For each $0 \le i \le m$, $\mathsf{cost}(i,0) = 0$ as we can set all $v_k = 0$ for $1 \le k \le i$ and $\mathcal{V}(i,0) = \mathbf{0}^i$, which is an i dimensional zero vector whose elements are all 0. For all $1 \le j \le H(l,s)+1$, we set $\mathsf{cost}(0,j) = \infty$ and for all $0 \le i \le m$, we set $\mathsf{cost}(i,-1) = \infty$ as they are undefined, i.e., there does not exist these kinds of allocations. With the initialization and the recursive formula we can easily solve the optimal allocation selection problem by calculating $\mathsf{cost}(m, H(l,s)+1)$ and $\mathcal{V}(m, H(l,s)+1)$ is the optimal allocation.

The pseudo-code of the optimal allocation selection algorithm OptimalSelection is shown in Algorithm 3. It takes the two kinds of inverted indexes $\mathcal{I}_l$ and $\mathcal{D}_l$, a record $\mathcal{X}$ with size s and a group $\mathcal{R}_l$ as input and outputs the optimal allocation $\mathcal{V}_l^s$ for the record $\mathcal{X}$ and the group $\mathcal{R}_l$. It first calculates all the costs c_0^i, c_1^i and c_2^i using $\mathcal{X}^i$, $\mathcal{I}_l^i$ and $\mathcal{D}_l^i$ for $1 \le i \le m = H_l + 1$ (Line 1). Then it initializes a cost matrix cost and an allocation matrix $\mathcal{V}$ by setting $\mathsf{cost}(i,0) = 0$, $\mathsf{cost}(i,-1) = \infty$ and $\mathcal{V}(i,0) = \mathbf{0}^i$ for all $0 \le i \le H_l + 1$ and $\mathsf{cost}(0,j) = \infty$ for all $1 \le j \le H(l,s)+1$ (Lines 2 to 5). Then for each $1 \le i \le H_l + 1$ and $1 \le j \le H(l,s)+1$, it chooses the $v_{\min}$ from $\{0,1,2\}$ which leads to minimum cost, i.e., it sets $v_{\min} = \arg\min_{v \in \{0,1,2\}} \mathsf{cost}(i-1, j-v) + c_v^i$ (Line 8). Next based on the selected $v_{\min}$, it fills $\mathsf{cost}(i,j)$ as $\mathsf{cost}(i-1, j-v_{\min}) + c_{v_{\min}}^i$ and appends $v_{\min}$ to $\mathcal{V}(i-1, j-v_{\min})$ to form $\mathcal{V}(i,j)$ (Lines 9 to 10). Finally, it returns $\mathcal{V}(H_l+1, H(l,s)+1)$ as the optimal allocation $\mathcal{V}_l^s$ (Line 11).

Example 3. *Consider $\mathcal{X}_5$ and $\mathcal{R}_9$. Following the same setting as Example 1, the costs for them are shown in Figure 3(a). As $H_9 + 1 = 4$ and $H(9,11)+1 = 4$, we first initialize $\mathsf{cost}(i,0) = 0$ and $\mathcal{V}(i,0) = \mathbf{0}^i$ for $0 \le i \le 4$ and $\mathsf{cost}(i,-1) = \mathsf{cost}(0,j) = \infty$ for $0 \le i \le 4$ and $1 \le j \le 4$. Note that $\mathcal{V}(0,0) = \mathbf{0}^0 = ()$. Then we fill the matrix cost based on Equation 2 and the results are shown in Figure 3. Next we concatenate the vectors. For $i = 1$ and $j = 2$, the values of $\mathsf{cost}(i-1, j-v) + c_v^i$ for $v = 0, 1$ and 2 are respectively ∞, ∞ and 0. Thus $v_{\min} = 2$ and $\mathcal{V}(1,2) = \mathcal{V}(1-1, 2-2) \oplus 2 = (2)$. Similarly we can concatenate the other elements and finally get the optimal allocation $\mathcal{V}(4,4) = (2,0,2,0)$ whose cost is $\mathsf{cost}(4,4) = 2$.*

Algorithm 3: OptimalSelection($\mathcal{I}_l, \mathcal{D}_l, \mathcal{X}, s, \mathcal{R}_l$)

```
Input: $\mathcal{I}_l$ and $\mathcal{D}_l$: the two kinds of inverted indexes;
       $\mathcal{X}$: A record; $s$: the size of $\mathcal{X}$; $\mathcal{R}_l$: the group.
Output: The optimal allocation for $\mathcal{X}$ and $\mathcal{R}_l$.
1  get the costs $c_0^i$, $c_1^i$ and $c_2^i$ using $\mathcal{I}_l^i$, $\mathcal{D}_l^i$ and $\mathcal{X}^i$;
2  for $0 \le i \le H_l + 1$ do
3      $\mathsf{cost}(i,-1) = \infty$, $\mathsf{cost}(i,0) = 0$ and $\mathcal{V}(i,0) = \mathbf{0}^i$;
4  for $1 \le j \le H(l,s)+1$ do
5      $\mathsf{cost}(0,j) = \infty$;
6  for $1 \le i \le H_l + 1$ do
7      for $1 \le j \le H(l,s)+1$ do
8          $v_{\min} = \arg\min_{v \in \{0,1,2\}} \mathsf{cost}(i-1, j-v) + c_v^i$;
9          $\mathsf{cost}(i,j) = \mathsf{cost}(i-1, j-v_{\min}) + c_{v_{\min}}^i$;
10         $\mathcal{V}(i,j) = \mathcal{V}(i-1, j-v_{\min}) \oplus v_{\min}$;
11 return $\mathcal{V}(H_l+1, H(l,s)+1)$;
```

Time Complexity: For each record $\mathcal{X} \in \mathcal{R}$ with size s and a group $\mathcal{R}_l$, the time complexity of the dynamic programming algorithm is $\mathcal{O}((H_l+1) * (H(l,s)+1))$. As $l \in [\delta * s, s]$, the time complexity for each record $\mathcal{X}$ is $\mathcal{O}(\sum_{l=\delta*s}^{s}(H_l+1)(H(l,s)+1)) = \mathcal{O}(s^3)$, which is very expensive. We speedup the allocation selection in Section 5.

Note that we can extend the deletion neighborhoods definition to q-deletion neighborhoods and use it in the deletion neighborhood based framework. However the number of q-deletion neighborhoods increases exponentially with q (which is $\binom{|\mathcal{X}|}{q}$ for a set $\mathcal{X}$). This not only leads to large overhead of storing the q-deletion neighborhoods but also requires a lot of time to generate them. Thus we focus on 1-deletion neighborhoods in this paper and leave the q-deletion neighborhood based techniques as a future work.

5. ALLOCATION SELECTION

As finding the optimal allocation is rather expensive, we speed up the allocation selection. We first propose a heap-based greedy algorithm in Section 5.1, which is a 2 factor approximation algorithm. Furthermore, we design an adaptive record grouping mechanism to reduce the number of groups probed by each record in Section 5.2. The two techniques reduce the time complexity from $\mathcal{O}(s^3)$ to $\mathcal{O}(s \log s)$.

5.1 Greedy Algorithm

The goal of finding an allocation is to compute a vector $(v_1, v_2, \cdots, v_m)$ such that $\sum_{i=1}^{m} v_i = H(l,s)+1$ and $\sum_{i=1}^{m} c_v^i$ is as small as possible. To achieve this goal, we can devise a greedy algorithm as follows. Given a record $\mathcal{X}$ with size s and a group $\mathcal{R}_l$, we first initialize an $m = H_l+1$ dimensional zero vector $\mathcal{V} = \mathbf{0}^m$. Then we repeatedly compare the cost increment of increasing v_i by 1, where the cost increment is $c_1^i - c_0^i$ for $v_i = 0$ and $c_2^i - c_1^i$ for $v_i = 1$[4], and select the smallest one for $i \in [1,m]$. In other words, we greedily "*choose a dimension with the minimum cost increment*" and increases

[4] Note we cannot increase $v_i = 2$.

```
Algorithm 4: GreedySelection($\mathcal{I}_l, \mathcal{D}_l, \mathcal{X}, s, \mathcal{R}_l$)
  Input: $\mathcal{I}_l$ and $\mathcal{D}_l$: the two kinds of inverted indexes;
         $\mathcal{X}$: A record; $s$: the size of $\mathcal{X}$; $\mathcal{R}_l$: the group.
  Output: $\mathcal{V}_l^s$ : an allocation for $\mathcal{X}$ and $\mathcal{R}_l$.
1 get the cost increments $\nabla_0^i$ and $\nabla_1^i$ by $\mathcal{I}_l^i$,$\mathcal{D}_l^i$ and $\mathcal{X}^i$;
2 set all $v_i = 0$ for $\mathcal{V}$=$(v_1, v_2, \dots, v_m)$ where $m$=$H_l$+1;
3 build a min-heap $\mathcal{M}$ over $\langle v_i, \nabla_{v_i}^i \rangle$ for all $1 \le i \le m$;
4 foreach $1 \le j \le H(l, s) + 1$ do
5     pop $\mathcal{M}$ to get the $\langle v_i, \nabla_{v_i}^i \rangle$ with minimum $\nabla_{v_i}^i$;
6     increase $v_i$ by 1, which is the $i^{th}$ dimension of $\mathcal{V}$;
7     if $v_i = 1$ then push $\langle v_i = 1, \nabla_{v_i}^i \rangle$ into $\mathcal{M}$
8 return $\mathcal{V}$;
```

the selected dimension v_i by 1, where the cost increment for $v_i \in \{0, 1\}$ is $\nabla_{v_i}^i = c_{v_i+1}^i - c_{v_i}^i$. For example, the cost increments for the records in $\mathcal{R}_9$ are shown in Figure 3 (b). After we make $H(l, s)+1$ increments, $\mathcal{V}$ is an allocation which satisfies $\sum_{i=1}^m v_i = H(l, s) + 1$ and $v_i \in \{0, 1, 2\}$ for $1 \le i \le m$ and we return $\mathcal{V}$ as the allocation $\mathcal{V}_l^s$. To select the smallest increment, we can utilize a heap. Next we propose the heap-based greedy method. The pseudo-code of the greedy method GreedySelection is shown in Algorithm 4.

Given a record $\mathcal{X}$ with size s and a group $\mathcal{R}_l$, we first compute the cost increments of v_i where $v_i \in \{0, 1\}$, using $\mathcal{I}_l^i$, $\mathcal{D}_l^i$ and $\mathcal{X}^i$ for $1 \le i \le m = H_l + 1$ (Line 1). Then we initialize an $m = H_l + 1$ dimensional vector $\mathcal{V} = (v_1, v_2, \dots, v_m)$ and set $v_i = 0$ for all $1 \le i \le m$ (Line 2). Next, we build a min-heap $\mathcal{M}$ where each node in $\mathcal{M}$ is a tuple $\langle v_i, \nabla_{v_i}^i \rangle$ where $1 \le i \le m$ (Line 3). Then we pop the heap $H(l, s)+1$ times, which always pops out the node with minimum cost increment $\nabla_{v_i}^i$ (Lines 4 to 5). For each popped out node $\langle v_i, \nabla_{v_i}^i \rangle$, we increase v_i, which is the i^{th} dimension of $\mathcal{V}$, by one. Then if $v_i = 1$, we need to update its cost increment. To this end we push in a new node $\langle v_i = 1, \nabla_{v_i}^i \rangle$ into $\mathcal{M}$. If $v_i = 2$, we cannot increase v_i anymore, thus we do not push new nodes into $\mathcal{M}$ (Lines 6 to 7). After $H(l, s) + 1$ nodes are popped, we return $\mathcal{V}$ as the allocation $\mathcal{V}_l^s$ (Line 8).

Example 4. *Consider $\mathcal{X}_5$ and $\mathcal{R}_9$. Following the same setting as Example 1, the cost increments are shown in Figure 3. We first initialize a 4 dimensional vector $\mathcal{V} = (0, 0, 0, 0)$ and build a min-heap of four nodes $\langle v_1, \nabla_0^1 = 0 \rangle$, $\langle v_2, \nabla_0^2 = 1 \rangle$, $\langle v_3, \nabla_0^3 = 1 \rangle$ and $\langle v_4, \nabla_0^4 = 3 \rangle$ as shown in Figure 4. Note that we omit the v_i in all the nodes in the figure. We pop the heap $H(l, s) + 1 = 4$ tims. For the first pop up, we get the node $\langle v_1, \nabla_0^1 \rangle$. We increase v_1 by 1 and have $\mathcal{V} = (1, 0, 0, 0)$. As $v_1 = 1$ we push in a new node $\langle v_1, \nabla_1^1 = 0 \rangle$. Then for the second pop up, we get the newly pushed in node $\langle v_1, \nabla_1^1 \rangle$. We increase v_1 by 1 and have $\mathcal{V} = (2, 0, 0, 0)$. As $v_i = 2$ we do not push in new node. For the third pop up[5] we get the node $\langle v_2, \nabla_0^2 \rangle$. We increase v_2 by one and have $\mathcal{V} = (2, 1, 0, 0)$. As $v_2 = 1$ we push in a new node $\langle v_2, \nabla_1^2 \rangle$ to the heap. For the fourth pop up we get the node $\langle v_3, \nabla_0^3 \rangle$ and increase v_3 by one. Thus we have $\mathcal{V} = (2, 1, 1, 0)$ and we return it as the allocation $\mathcal{V}_l^s$, whose cost is 3.*

We can prove that the cost of the allocation selected by the heap-based algorithm is no larger than 2 times of the cost of the optimal allocation as stated in Lemma 6.

Lemma 6. *The heap-based algorithm is a 2-approximation algorithm for the optimal allocation selection problem.*

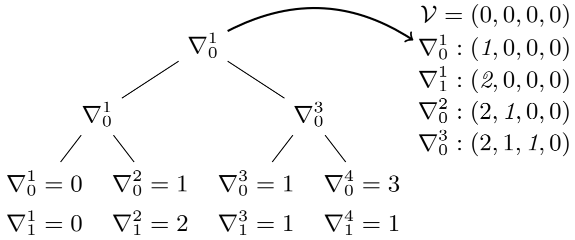


Figure 4: A greedy allocation selection example

Time Complexity. The time complexity of the greedy allocation selection is $\mathcal{O}\big((H(l, s)+1)\log(H_l+1)\big) = \mathcal{O}(s \log s)$ for a record with size s and a group $\mathcal{R}_l$. However based on the size filter, we need to probe $s-\delta s+1$ groups for the record and for each group we need to perform a greedy allocation selection. Thus the time complexity of allocation selection for one record with size s is $\mathcal{O}(s^2 \log s)$ which is still expensive. Next we propose an adaptive grouping mechanism to reduce the number of probed groups for each record.

5.2 Adaptive Grouping Mechanism

If we aggregate all the records with sizes within the range $\big[\frac{l_{\min}}{\delta^{k-1}}, \frac{l_{\min}}{\delta^k}\big)$ as a group $\mathcal{R}_k^\delta$ where $l_{\min}$ is the smallest record size in $\mathcal{R}$, we only need to probe at most 2 different groups for any record $\mathcal{X}$. This is because based on the size filter the sizes of its similar records are within $[\delta s, s]$ where $s = |\mathcal{X}|$. Without loss of generality, suppose $\mathcal{X} \in \mathcal{R}_k^\delta$, i.e., $\frac{l_{\min}}{\delta^{k-1}} \le s < \frac{l_{\min}}{\delta^k}$. Then we have $\frac{l_{\min}}{\delta^{k-2}} \le \delta s < \frac{l_{\min}}{\delta^{k-1}}$, which indicates all its similar records are in the group $\mathcal{R}_{k-1}^\delta$ and $\mathcal{R}_k^\delta$. Thus we only need to probe these two groups to find its similar records for the record $\mathcal{X}$. Next we formalize our idea.

Let $l_{k+1} = \frac{l_k}{\alpha} + 1$ where $l_1 = l_{\min}$ and $\alpha \in [\frac{1}{2}, 1]$ (we will discuss how to set α later). The adaptive grouping mechanism aggregates all the records in $\mathcal{R}$ with sizes within the size range $\big[l_k, \frac{l_k}{\alpha}\big]$ as a group $\mathcal{R}_k^\alpha$. Next we integrate the adaptive grouping mechanism into our algorithm.

We first show how to build the inverted indexes $\mathcal{I}$ and $\mathcal{D}$. For the group $\mathcal{R}_k^\alpha$, we evenly partition the universe $\mathcal{U}$ into $m = H_{l_k} + 1$ disjoint sub-universes and accordingly partition the records $\mathcal{X} \in \mathcal{R}_k^\alpha$ into m sub-records. For each sub-record $\mathcal{X}^i$ we append $\mathcal{X}.id$ to all the inverted lists $\mathcal{I}_{l_k}^i\big[\mathcal{X}^i\big]$ and $\mathcal{D}_{l_k}^i\big[\mathsf{del}(\mathcal{X}^i)\big]$ where $1 \le i \le m$. Then we find similar record pairs using the indexes. For each record $\mathcal{Y}$ with size s we probe all the groups $\mathcal{R}_k^\alpha$ where its size range $[l_k, \frac{l_k}{\alpha}]$ overlaps $[\delta s, s]$, which indicates $\mathcal{X}$ may have similar record in the group $\mathcal{R}_k^\alpha$. For each of these groups $\mathcal{R}_k^\alpha$, we partition the record $\mathcal{Y}$ to $m = H_{l_k}+1$ sub-records based on $\mathcal{U}_{l_k}$. Then we invoke the greedy allocation selection or the optimal allocation selection to select an allocation $\mathcal{V} = (v_1, v_2, \dots, v_m)$ where $m = H_{l_k} + 1$. Note that for the greedy allocation selection we need to pop the heap $H(\frac{l_k}{\alpha}, s) + 1$ times and for the optimal allocation selection the number of columns in the cost matrix and vector matrix is $H(\frac{l_k}{\alpha}, s) + 1$ as the group $\mathcal{R}_k^\alpha$ contains records with size as large as $\frac{l_k}{\alpha}$. Thus the selected allocation $\mathcal{V}$ satisfies $\sum_{i=1}^m v_i = H(\frac{l_k}{\alpha}, s) + 1$. Next we find similar records by $\mathcal{V}$, which is the same as the deletion neighborhood based framework.

We discuss how to set α. Obviously α cannot be larger than 1 as if $\alpha > 1$, the size range $[l_k, \frac{l_k}{\alpha}]$ of the group $\mathcal{R}_k^\alpha$ is empty. Actually when $\alpha = 1$, the grouping mechanism degenerates to the original grouping strategy which builds one group for each record size. Also α should be larger than 0. However we find α cannot infinitely close to 0. The

[5] Note ties are broken arbitrarily.

`Sim`	$H(l,s)$	H_l	Size Filter
`Jaccard`	$\lfloor \frac{1-\delta}{1+\delta}(s+l) \rfloor$	$\lfloor \frac{1-\delta}{\delta} l \rfloor$	$[l\delta, \frac{l}{\delta}]$
`Cosine`	$\lfloor s+l-2\delta\sqrt{sl} \rfloor$	$\lfloor \frac{1-\delta^2}{\delta^2} l \rfloor$	$[\delta^2 l, \frac{l}{\delta^2}]$
`Dice`	$\lfloor (1-\delta)(s+l) \rfloor$	$\lfloor 2\frac{1-\delta}{\delta} l \rfloor$	$[\frac{\delta}{2-\delta} l, \frac{2-\delta}{\delta} l]$

Table 2: The parameters in the partition framework

intuition is that when α is close to 0, the size range of the group is rather wide and the number of partitions is not large enough for the records with very large size in the group.

Formally, consider a record with size s and a group $\mathcal{R}_k^\alpha$ probed by this record. On the one hand, $[s\delta, s]$ and $[l_k, \frac{l_k}{\alpha}]$ overlap, i.e., $s \geq l_k$ and $s\delta \leq \frac{l_k}{\alpha}$. Thus we have $l_k \leq s \leq \frac{l_k}{\alpha\delta}$. On the other hand, the allocation $\mathcal{V} = (v_1, v_2, \ldots, v_m)$ satisfies $\sum_{i=1}^m v_i = H(\frac{l_k}{\alpha}, s)+1$ and $v_i \in \{0,1,2\}$ where $m = H_{l_k} + 1$. As $v_i \leq 2$, we have $\sum_{i=1}^m v_i \leq 2m$. Thus we have $H(\frac{l_k}{\alpha}, s) + 1 \leq 2(H_{l_k} + 1)$. To make this inequality true for any s and l_k with the restriction $l_k \leq s \leq \frac{l_k}{\alpha\delta}$, we need $\alpha \geq \frac{1}{2}$. Thus the value range of α is $[\frac{1}{2}, 1]$. Similarly for the partition based framework which always sets the allocation as $\mathbf{1}^m$ we can deduce the domain of α, which is $[1, 1]$. This means we cannot integrate the adaptive grouping mechanism into the partition based framework.

We prove the correctness and completeness of the deletion neighborhood based framework with adaptive grouping mechanism as formalized in Theorem 3.

THEOREM 3. *The deletion neighborhood based framework with adaptive grouping mechanism satisfies correctness and completeness when* $\alpha \in [\frac{1}{2}, 1]$.

Next we analyze the number of groups probed by a record in the adaptive grouping mechanism. Consider a record with size s, we need to probe all the groups with size ranges overlap $[\delta s, s]$. Among these probed groups, suppose $\mathcal{R}_t^\alpha$ is the first one and $\mathcal{R}_{t'}^\alpha$ is the last one. Thus the number of the probed groups is $t' - t + 1$. Next we deduce the upper bound of $t' - t + 1$. Based on the adaptive grouping mechanism, we have $l_k = \frac{l_{k-1}}{\alpha} + 1 = \frac{l_{k-2}}{\alpha^2} + \frac{1}{\alpha} + 1 = \cdots = \frac{l_{min}}{\alpha^{k-1}} + \frac{\alpha^{2-k} - \alpha}{1-\alpha}$. After transformation we have $k = -\log_\alpha \frac{(1-\alpha)l_k+1}{(1-\alpha)(l_{\min}+\alpha)}$ which is monotonically decreasing with l_k. On the other hand as the size range $[l_k, \frac{l_k}{\alpha}]$ overlaps $[\delta s, s]$, we have $l_k \leq s \leq \frac{l_k}{\alpha\delta}$, that is $\alpha\delta s \leq l_k \leq s$. Thus we have the number of the probed groups $t' - t + 1 \leq \log_\alpha \frac{(1-\alpha)\delta s+1}{(1-\alpha)s+1} + 1 \leq \log_\alpha \delta + 1$.

Time Complexity: As the number of probed groups by each record is no more than $\log_\alpha \delta + 1$, the time complexity of allocation selection for a record with size s is $\mathcal{O}(s \log s \log_\alpha \delta)$. We can always set $\alpha = \delta$ and the time complexity becomes $\mathcal{O}(s \log s)$ when $\delta \geq 0.5$ which is true almost all the time[6].

6. DISCUSSION

6.1 Supporting the Other Similarity Functions

In our techniques, only three parameters, $H(l,s)$, H_l and the size filter depend on the similarity functions. Similar to the deduction of the `Jaccard Similarity`, we can also deduce these parameters for `Cosine Similarity` and `Dice Similarity` and the details are shown in Table 2. Based on these parameters, we can easily extend our techniques for the other two similarity functions.

6.2 Supporting RS-Join

Given two datasets $\mathcal{R}$ and $\mathcal{S}$, we first build the inverted indexes of sub-records and 1-deletion neighborhoods for one dataset, e.g., $\mathcal{R}$. Then for each record in another dataset, e.g., $\mathcal{S}$, we can use the indexes to find all of its similar records in $\mathcal{R}$ in a same way as self-join, i.e., using the allocation selection techniques to select an allocation for this record, probing the inverted indexes based on the allocation to fetch candidates and verifying the candidates to get all the records in $\mathcal{R}$ similar to this record. We can also apply the adaptive grouping mechanism to build the indexes for efficiently finding similar records.

Dataset $\mathcal{R}$	$\lvert\mathcal{R}\rvert$	$l_{\min}$	$l_{\max}$	$\bar{l}$	$\lvert\mathcal{U}\rvert$
DBLP	873,524	6	1,538	94.1	44,798
TWEETS	2,000,000	1	70	21.6	1,713,437
MOVIELENS	138,493	20	9,254	144.4	26,744

Table 3: The dataset details

6.3 Extending to the MapReduce Framework

Our techniques can be easily extended to work on the MapReduce framework. We first show how to adapt the partition-based framework to work on MapReduce. We only need to utilize a single MapReduce round. For each record $\mathcal{X}$ with size l, we first partition it to H_l+1 *indexing* sub-records (to build the inverted indexes). Then for each $s \in [l * \delta, l]$, we partition $\mathcal{X}$ to $H_s + 1$ *probing* sub-records to probe the inverted indexes and find records similar to $\mathcal{X}$. A record $\mathcal{X}$ with indexing sub-records and a record $\mathcal{Y}$ with probing sub-records are similar only if they share a common sub-record $\mathcal{X}^i = \mathcal{Y}^i$ under the same partition scheme $\mathcal{U}_l$ where $l = |\mathcal{X}|$. Thus in the MapReduce framework we use the combination $(\mathcal{X}^i, i, l)$ as the key, and use the combination of $(\mathcal{X}, flag)$ as the value where $flag = 0$ indicates that the key-value pair comes from an indexing sub-record and $flag = 1$ indicates that the key-value pair comes from a probing sub-record.

In the map phase, for each record $\mathcal{X}$, we first partition it to $H_l + 1$ indexing sub-records and emit a key-value pair $\langle(\mathcal{X}^i, i, l), (\mathcal{X}, 0)\rangle$ for each sub-record $\mathcal{X}^i$. For each $s \in [l * \delta, l]$, we partition $\mathcal{X}$ to $H_s + 1$ probing sub-records and emit a key-value pair $\langle(\mathcal{X}^i, i, s), (\mathcal{X}, 1)\rangle$ for each sub-record $\mathcal{X}^i$.

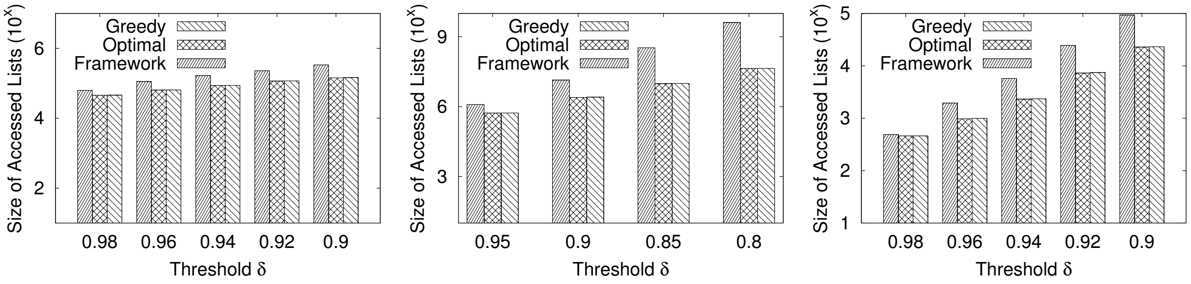
In the reduce phase, for key $(\mathcal{X}^i, i, l)$, let $list(\mathcal{X}, flag)$ denote the list of values with this key, which share the same sub-record $\mathcal{X}^i$. We split the list to two lists $\mathcal{L}_0$ and $\mathcal{L}_1$ based on $flag$ where $\mathcal{L}_0$ contains all the records with indexing sub-records $\mathcal{X}^i$ and $\mathcal{L}_1$ contains all the records with probing sub-records $\mathcal{X}^i$. Each pair of records $\langle\mathcal{X}, \mathcal{Y}\rangle$ in $\mathcal{L}_0 \times \mathcal{L}_1$ is a candidate pair. As some record pairs may share multiple common sub-records, there may be duplicate candidates in different reducers. To avoid verifying duplicate candidates, for a candidate pair $(\mathcal{X}, \mathcal{Y})$ of key $(\mathcal{X}^i, i, l)$, we first check whether they have the same sub-record before the i-th sub-record. If so (i.e., $\exists j < i$, s.t. $\mathcal{X}^j = \mathcal{Y}^j$), we do not need to verify the pair; otherwise, we verify this pair. In this way, we can only verify each candidate pair once.

7. EXPERIMENTS

We have conducted experiments to evaluate our method, and our experimental goal was to evaluate our proposed techniques and compare the efficiency and scalability with state-of-the-art methods, `PPJoin+` [22] and `AdaptJoin` [20].Although there are other algorithms, such as `All-Pair` [2], `Flamingo` [10] `PartEnum` [1], and `BayesLSH` [14] for set similarity joins, previous studies [9,20] have shown they cannot outperform the two state-of-the-art methods. Thus we only reported the comparison results of our method with `PPJoin+` and `AdaptJoin`. We got the codes from the authors.

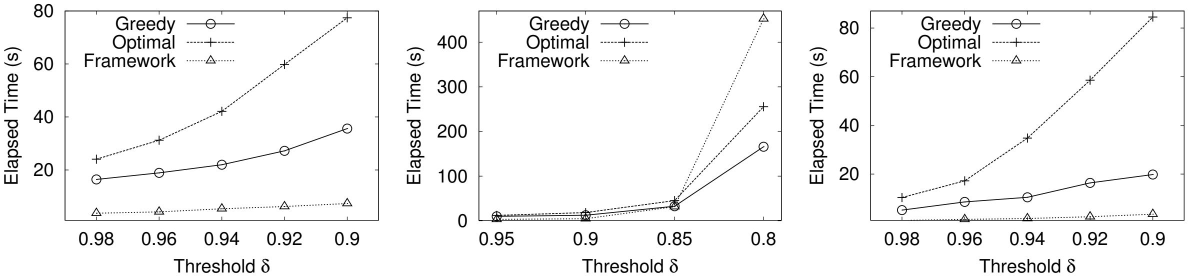
All the algorithms were implemented in C++ and compiled using GCC 4.8.2 with -O3 flag. All the experiments were conducted on a Ubuntu machine with 24 Intel Xeon X5670 2.93GHz processors and 64 GB memory.

[6] In practice δ is usually between 0.8 and 1.



(a) DBLP (Avg Size = 94) (b) Tweets (Avg Size = 22) (c) MovieLens (Avg Size = 144)

Figure 5: Sizes of Accessed Lists: Evaluating Different Allocation Selection Methods.



(a) DBLP (Avg Size = 94) (b) Tweets (Avg Size = 22) (c) MovieLens (Avg Size = 144)

Figure 6: Efficiency: Evaluating Different Allocation Selection Methods.

Dataset: We used three real-world datasets in our experiments. 1) DBLP was a computer-science bibliographic dataset. Each record was a set of 3-grams by tokenizing the bibliography. 2) TWEETS was a tweet dataset. Each record was a tweet and each element was a word tokenized by space. 3) MOVIELENS was a movie rating dataset. Each record corresponded to a user, which was a set of movies the user has rated. Table 3 showed the details of the datasets.

7.1 Evaluating Allocation Selection

We evaluated different allocation selection methods. We implemented the following three algorithms. (1) The partition based framework, `Framework`, which always set the allocation $\mathcal{V}_l^s = \mathbf{1}^m$ for any record with size s and group $\mathcal{R}_l$ where $m = H_l + 1$. (2) The dynamic programming algorithm for optimal allocation selection, denoted by `Optimal`. (3) The min-heap based algorithm for greedy allocation selection, denoted by `Greedy`. We did not utilize the adaptive grouping mechanism for any of them. According to Sections 3, 4.2 and 5.1, their time complexities of allocation selection were respectively $\mathcal{O}(s^2)$[7], $\mathcal{O}(s^3)$ and $\mathcal{O}(s^2 \log s)$ where s was the record size. As the running time of `Optimal` and `Greedy` grew rapidly with the decreases of the threshold, we only reported the results with a relatively large threshold in this section to finish the experiments in reasonable time. We first compared the size of accessed inverted lists. Figure 5 shows the results. Note the y-axis was in log scale.

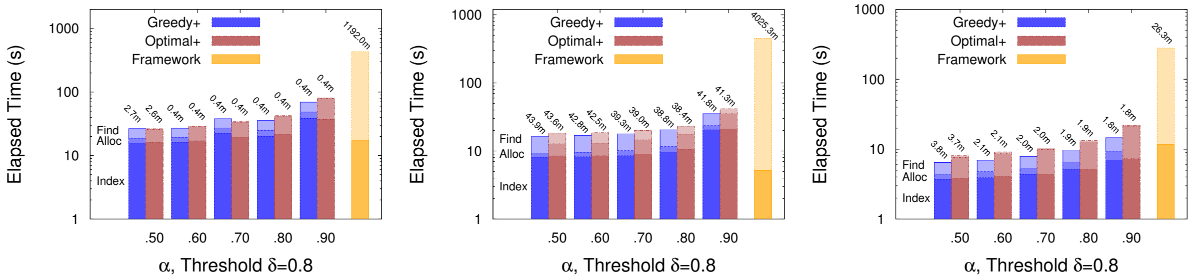
We could see that `Optimal` achieved the smallest sizes of accessed lists and the sizes of accessed lists of `Greedy` were slightly larger than that of `Optimal`. `Framework` had the largest sizes of accessed lists and were 10 to 100 times of those of `Greedy` and `Optimal` and got worse with the decreases of the threshold. For example, on TWEETS dataset, for $\delta = 0.8$, the sizes of accessed lists for `Framework`, `Greedy` and `Optimal` were respectively $4*10^9$, $4.4*10^7$ and $4.3*10^7$, because `Framework` did not use the 1-deletion neighborhoods and thus could only utilize a fixed allocation while `Optimal` selected the optimal allocation with minimum accessed list sizes. `Greedy` was a 2 factor approximation algorithm for optimal allocation selection whose accessed list sizes was no more than twice of the optimal allocation. The experimental results were consistent with our theoretical analysis.

We also compared the elapsed time of the three methods. The results are shown in Figure 6. We can see `Greedy` consistently outperformed `Optimal`, because `Greedy` had lower time complexity than `Optimal` but similar size of accessed lists. When the threshold was larger than 0.9, the `Framework` was better than `Greedy` and `Optimal`, which was consistent with the time complexity analysis of allocation selection, because with large threshold and without adaptive grouping mechanism, the allocation selection time $\big(\mathcal{O}(s^2)$ for `Framework`, $\mathcal{O}(s^3)$ for `Optimal` and $\mathcal{O}(s^2 \log s)$ for `Greedy`$\big)$ dominated the running time. However when the threshold got smaller than 0.9, `Greedy` and `Optimal` outperformed `Framework`, as the size of accessed inverted list and the candidates of `Framework` grew exponentially with the decrease of δ and the verification dominated the running time.

7.2 Evaluating the Adaptive Grouping

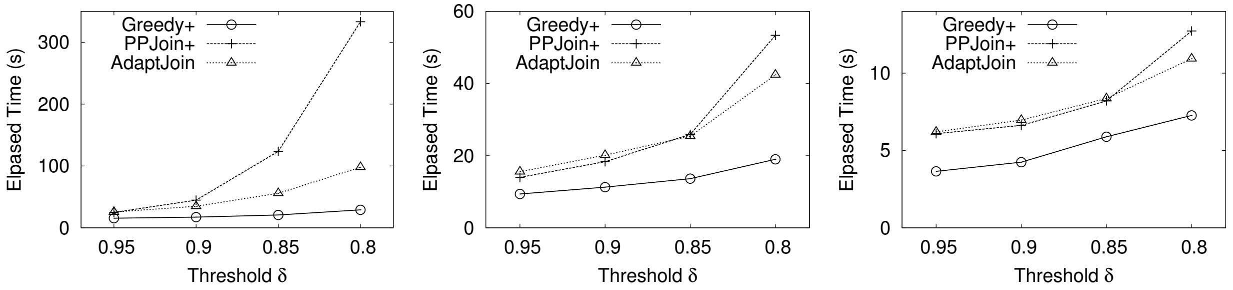
We evaluated the adaptive grouping mechanism. We varied the value of α from 0.5 to 0.9 and reported the elapsed time and the sizes of the accessed inverted lists of `Greedy+` (`Greedy` with adaptive grouping) and `Optimal+` (`Optimal` with adaptive grouping). We did not report the results for $\alpha > 0.9$ as the running time were rather long that could not finish in reasonable time. We also reported the results of `Framework`. Figure 7 showed the results. The numbers above the bars were their sizes of accessed lists. We split the elapsed time to three parts: the indexing time, the allocation selection time and the finding similar pairs time. Note the pre-processing steps (getting the universe, sorting the dataset and grouping the records) took less than 0.5 second. We can see that with the increases of α, the elapsed time for `Greedy+` and `Optimal+` both increased. Moreover they both outperformed `Framework`. For the same α, `Greedy+` was better than `Optimal+`. For example, on MOVIELENS dataset with $\delta = 0.8$, the elapsed time when α varied from 0.5 to 0.9 were 6s, 7s, 8s, 10s, and 14s for `Greedy+` and 8s, 9s, 10s, 14s, and 23s for `Optimal+`, while the elapsed time for `Framework` was 286s. This was because the time complexity for allocation selection of `Greedy+`, $\mathcal{O}(s \log s \log_\alpha \delta)$, was lower than

[7] The record partitioning was also included in allocation selection.



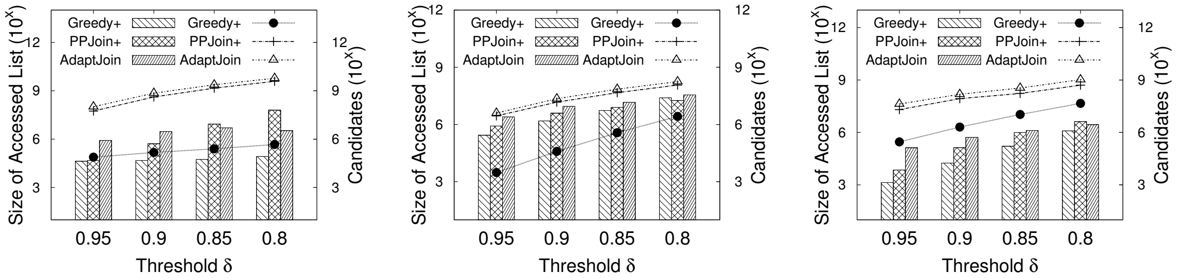
(a) DBLP (Avg Size = 94) (b) Tweet (Avg Size = 22) (c) MovieLens (Avg Size = 144)

Figure 7: Evaluating The Adaptive Grouping Mechanism by Varying α.



(a) DBLP (Avg Size = 94) (b) Tweet (Avg Size = 22) (c) MovieLens (Avg Size = 144)

Figure 8: Efficiency: Comparison with State-of-the-art Studies.



(a) DBLP (Avg Size = 94) (b) Tweet (Avg Size = 22) (c) MovieLens (Avg Size = 144)

Figure 9: Sizes of Accessed Inverted Lists: Comparison with State-of-the-art Studies.

that of `Optimal+`, which was $\mathcal{O}(s^2 \log_\alpha \delta)$. In addition, the sizes of accessed lists of `Greedy+` were at most twice of that of `Optimal+`. Moreover when α was small, their time complexities were both not higher than that of `Framework`, which was $\mathcal{O}(s^2)$ and they accessed much smaller number of entries in the inverted lists than `Framework` as shown in the figure. The sizes of accessed lists of `Greedy+` and `Optimal+` slightly changed with the increases of α. This was because they could always choose a good or even optimal allocation for different records and groups. As `Greedy+` achieved the best performance, in the following experiments we only reported the results of `Greedy+`.

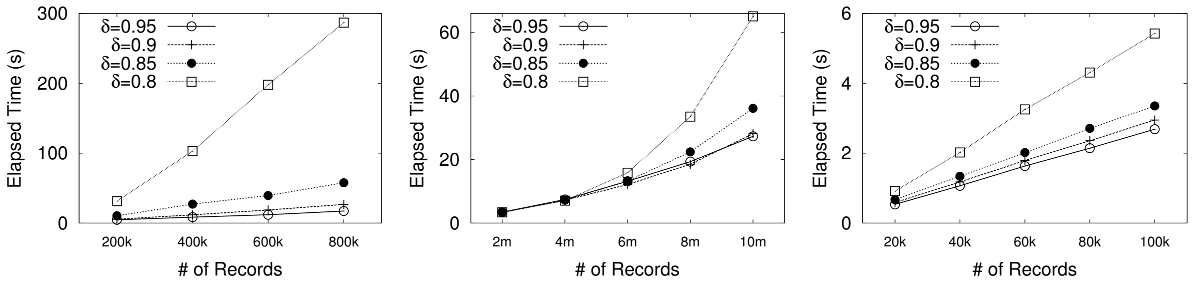
7.3 Comparison with Existing Methods

We compared our method `Greedy+`, which utilized the deletion neighborhood based framework with adaptive grouping mechanism, with the state-of-the-art methods `PPJoin+` and `AdaptJoin`. We reported the elapsed time, sizes of accessed inverted lists and candidate number. The results were shown in Figures 8 and 9. We can see that for efficiency our method `Greedy+` achieved the best performance and outperformed existing methods by 2 to 5 times. For example on the DBLP dataset with `Jaccard Similarity` threshold $\delta = 0.8$, the elapsed time for `Greedy+` was 20s while `PPJoin+` and `AdaptJoin` took 330s and 100s respectively. This was because `Greedy+` had more powerful pruning technique than `AdaptJoin` and `PPJoin+`, and the greedy allocation selection and the adaptive grouping mechanism decreased the time complexity of the pruning techniques to $\mathcal{O}(s \log s)$. Our method `Greedy+` also achieved the smallest sizes of accessed inverted lists and candidate numbers. The bars in Figure 9 showed the candidate numbers and the points showed the size of accessed lists. `AdaptJoin` accessed more number of entries in the inverted lists but with less number of candidates than `PPJoin+`. This was because both `PPJoin+` and `AdaptJoin` utilized the prefix filter framework where each element corresponded to an inverted lists while each inverted list corresponded to a subset in `Greedy+`. Thus the size of the inverted lists was much shorter than that of `PPJoin+` and `AdaptJoin`. Moreover with the greedy allocation selection, `Greedy+` can further reduce the number of accessed entries. `PPJoin+` included a few more elements into prefix to prune dissimilar pairs thus had a larger accessed inverted lists size and a smaller number of candidates.

We also compared the index sizes. On the MovieLens dataset with threshold $\delta = 0.8$, the index sizes for `AdaptJoin`, `PPJoin+` and `Greedy+` were respectively 80.1 MB, 16.1 MB and 87.2 MB. The `Greedy+` had the largest index size as it needs to insert both the sub-records and the 1-deletion neighborhoods to the index. `PPJoin+` had the smallest index size as it only inserted the prefixes to the index while `AdaptJoin` inserted every element in the record to the index.

7.4 Scalability

We evaluated the scalability of our method `Greedy+` with two different similarity functions, the `Jaccard Similarity`



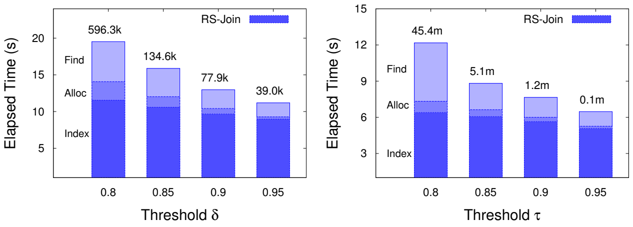
(a) Jaccard on DBLP (Avg Size = 94) (b) Jaccard on Tweet (Avg Size = 22) (c) Cosine on MovieLens (Avg Size = 144)

Figure 10: Scalability with Different Similarity Functions.

and the Cosine Similarity, by varying the number of records in the dataset. Figure 10 showed the results. We could see that our method scaled very well. For example, on the TWEETS dataset with Jaccard Similarity threshold $\delta = 0.9$, we varied the number of records from 2 million to 10 million. The elapsed time were respectively 3s, 7s, 12s, 18s and 28s. This was attributed to our effective filter conditions and allocation selections.

7.5 Evaluating RS-Join

We extended Greedy+ to support the RS-join as discussed in Section 6.2. For each dataset, we randomly split it to two parts. Figure 11 showed the elapsed time and the number of candidates. Our algorithms still achieved high performance for RS-join, because our method built indexes for one dataset and utilized the indexes to find similar pairs.



(a) RS-Join on DBLP (b) RS-Join on Tweet

Figure 11: Evaluating RS-Join.

8. CONCLUSION

We proposed a partition based framework for exact set similarity join. We designed a partition scheme to partition the sets into subsets. We generated the 1-deletion neighborhoods for the subsets. Then we built inverted indexes for the 1-deletion neighborhoods and the subsets. For each set we accessed the inverted lists of some of its subsets and 1-deletion neighborhoods to find similar sets. We studied how to evaluate different allocations of the subsets and 1-deletion neighborhoods and developed a dynamic-programming method to select the optimal one. To accelerate the allocation selection, we designed a greedy algorithm with 2 approximation ratio. We proposed an adaptive grouping mechanism to further speedup the allocation selection. These techniques improved the allocation complexity from $\mathcal{O}(s^3)$ to $\mathcal{O}(s \log s)$ for a set with size s. Experiments showed our method outperformed state-of-the-art studied.

Acknowledgement: This work was supported by the National Grand Fundamental Research 973 Program of China (2015CB358700), and the National Natural Science Foundation of China (61422205, 61472198), Beijing Higher Education Young Elite Teacher Project(YETP0105), Tsinghua-Tencent Joint Laboratory for Internet Innovation Technology, "NExT Research Center", Singapore (WBS:R-252-300-001-490), Huawei, Shenzhou, FDCT/116/2013/A3, MYRG105(Y1-L3)-FST13-GZ, National 863 Program of China (2012AA012600), and Chinese Special Project of Science and Technology (2013zx01039-002-002).

9. REFERENCES

[1] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.

[2] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.

[3] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations (extended abstract). In *ACM STOC*, pages 327–336, 1998.

[4] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, pages 5–16, 2006.

[5] W. W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *ACM SIGMOD*, pages 201–212, 1998.

[6] A. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: scalable online collaborative filtering. In *WWW*, pages 271–280, 2007.

[7] D. Deng, G. Li, S. Hao, J. Wang, and J. Feng. Massjoin: A mapreduce-based method for scalable string similarity joins. In *ICDE*, pages 340–351, 2014.

[8] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *ACM STOC*, pages 604–613, 1998.

[9] Y. Jiang, G. Li, J. Feng, and W. Li. String similarity joins: An experimental evaluation. *PVLDB*, 7(8):625–636, 2014.

[10] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.

[11] G. Li, D. Deng, J. Wang, and J. Feng. Pass-join: A partition-based method for similarity joins. *PVLDB*, 5(3):253–264, 2011.

[12] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD Conference*, pages 743–754, 2004.

[13] B. M. Sarwar, G. Karypis, J. A. Konstan, and J. Riedl. Item-based collaborative filtering recommendation algorithms. In *WWW*, pages 285–295, 2001.

[14] V. Satuluri and S. Parthasarathy. Bayesian locality sensitive hashing for fast similarity search. *PVLDB*, 5(5):430–441, 2012.

[15] A. Shrivastava and P. Li. Asymmetric LSH (ALSH) for sublinear time maximum inner product search (MIPS). In *NIPS*, pages 2321–2329, 2014.

[16] E. Spertus, M. Sahami, and O. Buyukkokten. Evaluating similarity measures: a large-scale study in the orkut social network. In *ACM SIGKDD*, pages 678–684, 2005.

[17] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD*, pages 495–506, 2010.

[18] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. Crowder: Crowdsourcing entity resolution. *PVLDB*, 5(11):1483–1494, 2012.

[19] J. Wang, G. Li, and J. Feng. Fast-join: An efficient method for fuzzy token matching based string similarity join. In *ICDE*, pages 458–469, 2011.

[20] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *SIGMOD Conference*, pages 85–96, 2012.

[21] C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k set similarity joins. In *ICDE*, pages 916–927, 2009.

[22] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, pages 131–140, 2008.

[23] J. Zhai, Y. Lou, and J. Gehrke. ATLAS: a probabilistic algorithm for high dimensional similarity search. In *ACM SIGMOD*, pages 997–1008, 2011.

[24] X. Zhu and A. B. Goldberg. *Introduction to Semi-Supervised Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2009.