Can We Beat the Prefix Filtering? An Adaptive Framework for Similarity Join and Search

Jiannan Wang Guoliang Li Jianhua Feng

Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology (TNList), Tsinghua University, Beijing, China

wjn08@mails.thu.edu.cn; liguoliang@tsinghua.edu.cn; fengjh@tsinghua.edu.cn

ABSTRACT

As two important operations in data cleaning, similarity join and similarity search have attracted much attention recently. Existing methods to support similarity join usually adopt a prefix-filtering-based framework. They select a prefix of each object and prune object pairs whose prefixes have no overlap. We have an observation that prefix lengths have significant effect on the performance. Different prefix lengths lead to significantly different performance, and prefix filtering does not always achieve high performance. To address this problem, in this paper we propose an adaptive framework to support similarity join. We propose a cost model to judiciously select an appropriate prefix for each object. To efficiently select prefixes, we devise effective indexes. We extend our method to support similarity search. Experimental results show that our framework beats the prefix-filtering-based framework and achieves high efficiency.

Categories and Subject Descriptors: H.2.4 [Database Management]: Systems—*Textual Databases*; H.3.3 [Information Storage and Retrieval]: Information Search and Retrieval—*Search Process*

General Terms: Algorithms, Experimentation, Performance

Keywords: Prefix Filtering, Similarity Search, Similarity Join, Adaptive Framework, Cost Model

1. INTRODUCTION

As two important operations in data cleaning, similarity join and similarity search have attracted significant attention from the database community recently. Given two collections of objects, similarity join returns all *similar* object pairs. Similarity join has many real applications in data cleaning and near duplicate object detection and elimination. For example, an insurance company has two sets of customer records from two data sources. An insurance clerk wants to eliminate the duplicates from the two sets. As the two customer records may have different representations, the clerk needs to use similarity join to correlate the two sets.

Similarity search, given a collection of objects and a query object, finds all objects similar to the query object. Similarity search also has many applications in information retrieval and natural language processing. For example, as many queries issued to a search engine contain typos, search engines can use similarity search to suggest relevant queries.

To quantify similarity between objects, many similarity functions have been proposed, such as jaccard similarity, cosine similarity, dice similarity, overlap similarity, edit similarity. Given two objects, a similarity function takes as input the two objects and returns the similarity of the two objects. If the similarity is not smaller than a given threshold, the objects are taken to be similar.

Existing methods to support similarity join employ a filter-and-verification framework [4]. The basic idea is to first use an efficient filter to prune those object pairs that cannot be similar and then verify the survived object pairs by computing their real similarity. In the filter step, the prefix filtering is a dominant technique and many existing methods employ a prefix-filtering-based framework [2,4]. The prefix filtering first transforms each object to a set of elements (see Section 2.2.1). Then it sorts the elements of each object based on a global ordering, and selects a prefix set for each object based on a given similarity threshold (see Section 2.2.2). It proves that if two objects are similar, the prefix sets of the two objects must have overlap. Finally, it utilizes an inverted index to prune those object pairs whose prefix sets have no overlap (see Section 2.2.3).

We have an observation that prefix lengths have much effect on the performance. Different prefix lengths lead to significantly different performance, and the prefix filtering nearly always gets the worst performance (see Section 3). Intuitively, longer prefix lengths have larger pruning power, but involve more filtering time. On the contrary, shorter prefix lengths achieve higher filtering performance, but lead to longer verification time.

It calls for a method to adaptively select an appropriate prefix length for each object. To this end, we propose an adaptive framework to address this problem. We propose a cost model to judiciously select an appropriate prefix for each object. To efficiently select prefixes, we devise effective index structures. We develop effective pruning techniques to improve the performance. We also extend our method to support similarity search. Moreover, our method can support all of the above similarity functions. To summarize, we make the following contributions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD '12, May 20–24, 2012, Scottsdale, Arizona, USA.

Copyright 2012 ACM 978-1-4503-1247-9/12/05 ...$10.00.

$\mathcal{R}$	r_1	{ vldb, sigmod, icde, 2011, jagadish }
	r_2	{ jagadish, koudas, vldb, edbt, icde }
	r_3	{ koudas, divesh, jagadish, edbt, icde }
	r_4	{ vldb, icde, koudas, jagadish, divesh }
	r_5	{ 2011, divesh, edbt, vldb, sigmod }

$\mathcal{S}$	s_1	{ nick, koudas, 2011, vldb, sigmod }
	s_2	{ nick, vldb, icde, sigmod, edbt }
	s_3	{ koudas, divesh, sigmod, icde, edbt }
	s_4	{ icde, sigmod, 2011, jagadish, divesh }
	s_5	{ 2011, vldb, edbt, icde, jagadish }

Figure 1: Two collections of objects.

- We propose an adaptive framework to support both similarity join and similarity search.
- We develop a cost model to judiciously select an appropriate prefix for each object.
- We extend our method to support similarity search and develop effective pruning techniques.
- We have implemented our method. Experimental results on real data sets show that our framework beats the prefix filtering and achieves high performance for both similarity join and similarity search.

The rest of this paper is organized as follows. We first give the problem formulation and introduce the prefix filtering in Section 2, and then analyze the prefix-filtering-based framework theoretically and experimentally in Section 3. Our adaptive framework is proposed in Section 4. We extend our framework to support similarity search in Section 5. Experimental studies are conducted in Section 6. We review related work in Section 7 and conclude the paper in Section 8.

2. PRELIMINARIES

2.1 Problem Formulation

A similarity function is used to quantify the similarity of two objects. Given two objects r and s, a similarity function, denoted by $\mathsf{sim}(r,s)$, returns a value to represent their similarity. The larger the value, the more similar the two objects. Generally, users specify a similarity threshold θ, and two objects are *similar* if their similarity is not smaller than the threshold, i.e. $\mathsf{sim}(r,s) \geq \theta$.

In our paper, we focus on two types of objects, *sets* and *strings*, which are widely used in many real applications. If the objects are sets, we consider the following similarity functions to quantify their similarity.

DEFINITION 1. *Let r and s be two sets.*

- *Overlap similarity:* $\mathsf{sim}_o(r,s) = |r \cap s|$.
- *Dice similarity:* $\mathsf{sim}_d(r,s) = \frac{2\cdot|r\cap s|}{|r|+|s|}$.
- *Cosine similarity:* $\mathsf{sim}_c(r,s) = \frac{|r\cap s|}{\sqrt{|r|\cdot|s|}}$.
- *Jaccard similarity:* $\mathsf{sim}_j(r,s) = \frac{|r\cap s|}{|r|+|s|-|r\cap s|}$.

where $|r|(|s|)$ denotes the size of set $r(s)$.

For example, consider $r = \{$`sigmod`, `icde`, `vldb`$\}$ and $s = \{$`sigmod`, `icde`$\}$. $|r \cap s| = 2$, $|r| = 3$, and $|s| = 2$. Their overlap similarity is $\mathsf{sim}_o(r,s) = 2$, their dice similarity is $\mathsf{sim}_d(r,s) = \frac{4}{5}$, their cosine similarity is $\mathsf{sim}_c(r,s) = \frac{2}{\sqrt{6}}$ and their jaccard similarity is $\mathsf{sim}_j(r,s) = \frac{2}{3}$.

If the objects are strings, we use edit distance to quantify their similarity.

A global ordering

e_1	jagadish
e_2	koudas
e_3	nick
e_4	divesh
e_5	2011
e_6	vldb
e_7	edbt
e_8	icde
e_9	sigmod

$\mathcal{R}$	r_1	$\{e_1, e_5, e_6, e_8, e_9\}$
	r_2	$\{e_1, e_2, e_6, e_7, e_8\}$
	r_3	$\{e_1, e_2, e_4, e_7, e_8\}$
	r_4	$\{e_1, e_2, e_4, e_6, e_8\}$
	r_5	$\{e_4, e_5, e_6, e_7, e_9\}$

$\mathcal{S}$	s_1	$\{e_2, e_3, e_5, e_6, e_9\}$
	s_2	$\{e_3, e_6, e_7, e_8, e_9\}$
	s_3	$\{e_2, e_4, e_7, e_8, e_9\}$
	s_4	$\{e_1, e_4, e_5, e_8, e_9\}$
	s_5	$\{e_1, e_5, e_6, e_7, e_8\}$

Figure 2: Two collections of objects in Figure 1 after sorting elements in each object based on a global ordering.

DEFINITION 2. *Let r and s be two strings. The edit distance* $\textsc{ed}(r,s)$ *between r and s is defined as the minimum number of single-character edit operations (insertions, deletions and insertions) to transform r to s. The edit similarity is defined as* $\textsc{es}(r,s) = 1 - \frac{\textsc{ed}(r,s)}{max(|r|,|s|)}$.

For example, $\textsc{ed}($`sigmod`, `sagmd`$) = 2$ and $\textsc{es}($`sigmod`,`sagmd`$) = \frac{2}{3}$. Note that the edit distance is a distance function. Different from a similarity function, the smaller the value $\textsc{ed}(r,s)$, the more similar the two objects. Therefore, given an edit-distance threshold θ, two objects are similar if and only if their edit distance is not larger than θ, $\textsc{ed}(r,s) \leq \theta$.

Next we define the SIMJOIN and SIMSEARCH queries.

DEFINITION 3 (SIMJOIN QUERY). *Given two collections of object $\mathcal{R}$ and $\mathcal{S}$, a similarity function* sim*, and a specified similarity threshold θ, a* SIMJOIN *query returns all object pairs $\langle r,s\rangle \in \mathcal{R}\times\mathcal{S}$ such that $\mathsf{sim}(r,s) \geq \theta$, i.e. $\{\langle r,s\rangle \mid \langle r,s\rangle \in \mathcal{R}\times\mathcal{S}, \mathsf{sim}(r,s) \geq \theta\}$.*

For example, given two collections of objects $\mathcal{R}$ and $\mathcal{S}$ in Figure 1, jaccard similarity sim_j and $\theta = \frac{2}{3}$, the SIMJOIN query returns object pairs $\{\langle r_1,s_4\rangle, \langle r_1,s_5\rangle, \langle r_2,s_5\rangle, \langle r_3,s_3\rangle\}$ since their jaccard similarity is not smaller than $\frac{2}{3}$, e.g. $\mathsf{sim}_j(r_1,s_4) = \frac{2}{3} \geq \frac{2}{3}$. For the other object pairs, their jaccard similarity is smaller than $\frac{2}{3}$, e.g. $\mathsf{sim}_j(r_1,s_3) = \frac{1}{4} < \frac{2}{3}$.

DEFINITION 4 (SIMSEARCH QUERY). *Given a collection of objects $\mathcal{S}$, a similarity function* sim*, a query object r, and a similarity threshold θ, a* SIMSEARCH *query returns all objects $s \in \mathcal{S}$ s.t. $\mathsf{sim}(r,s) \geq \theta$, i.e. $\{s \mid s \in \mathcal{S}, \mathsf{sim}(r,s) \geq \theta\}$.*

For example, given a collection of objects $\mathcal{S}$ in Figure 1. Suppose $r = \{$`nick`, `koudas`, `divesh`, `vldb`, `2011`$\}$ and jaccard similarity sim_j and $\theta = \frac{1}{2}$. The SIMSEARCH query returns one object $\{s_1\}$ since $\mathsf{sim}_j(r,s_1) = \frac{2}{3} \geq \frac{1}{2}$, and for any other object $s \in \mathcal{S}$, the jaccard similarity between r and s is smaller than $\frac{1}{2}$, e.g. $\mathsf{sim}_j(r,s_2) = \frac{1}{4} < \frac{1}{2}$.

2.2 Prefix-Filtering Framework

A brute-force method to answer SIMJOIN query is to first compute the similarity of each object pair and then return the pairs whose similarity is not smaller than θ. The time complexity of this method is $\mathcal{O}(cost_v \cdot |\mathcal{R}| \cdot |\mathcal{S}|)$ where $cost_v$ is the average cost of computing the similarity of an object pair. If there are a large number of objects in $\mathcal{R}$ and $\mathcal{S}$, the method becomes quite expensive. In this section, we introduce the state-of-the-art framework, namely prefix filtering [4], which can address this problem efficiently. Its basic idea is to first use an efficient filter to prune those object pairs that cannot be similar and then verify the survived object pairs by computing their real similarity. Since the number of survived object pairs is much smaller than $|\mathcal{R}| \cdot |\mathcal{S}|$, even in several orders of magnitude, the algorithms based on this framework outperform the brute-force method significantly.

2.2.1 Mapping Object to Set

The prefix-filtering framework first maps objects to sets. Then we can transform various similarity functions to the overlap similarity function on the sets. That is given a similarity function sim, a threshold θ, and two objects r, s, if $\mathsf{sim}(r,s) \geq \theta$, then the overlap similarity of the sets must be no smaller than a threshold t. Next we discuss how to map objects to the sets and how to compute the threshold t.

First, consider the set similarity functions in Definition 1. We can simply map each object to itself and the overlap threshold t can be deduced as follows.

- If $\mathsf{sim}_o(r,s) \geq \theta$, then $|r \cap s| \geq \theta$, thus $t = \lceil \theta \rceil$.
- If $\mathsf{sim}_d(r,s) \geq \theta$, then $|r \cap s| \geq \frac{\theta}{2-\theta} \cdot |r|$, thus $t = \lceil \frac{\theta}{2-\theta} \cdot |r| \rceil$.
- If $\mathsf{sim}_c(r,s) \geq \theta$, then $|r \cap s| \geq \theta^2 \cdot |r|$, thus $t = \lceil \theta^2 \cdot |r| \rceil$.
- If $\mathsf{sim}_j(r,s) \geq \theta$, then $|r \cap s| \geq \theta \cdot |r|$, thus $t = \lceil \theta \cdot |r| \rceil$.

Second, for the edit distance and the edit similarity in Definition 2, we map each object to its q-gram set. The q-gram set of a string r, denoted by $Q_\mathsf{q}(r)$, consists of all the substrings of r with length q. For example, $Q_2(\texttt{sigmod}) = \{\texttt{si}, \texttt{ig}, \texttt{gm}, \texttt{mo}, \texttt{od}\}$. Using q-gram sets, we can deduce the overlap threshold t as follows.

- If $\textsc{ed}(r,s) \leq \theta$, then $|Q_\mathsf{q}(r) \cap Q_\mathsf{q}(s)| \geq |r| + 1 - (\theta+1) \cdot \mathsf{q}$, thus $t = \lceil |r| + 1 - (\theta+1) \cdot \mathsf{q} \rceil$.
- If $\textsc{es}(r,s) \geq \theta$, then $|Q_\mathsf{q}(r) \cap Q_\mathsf{q}(s)| \geq |r| + 1 - (\frac{1-\theta}{\theta} \cdot |r| + 1) \cdot \mathsf{q}$, thus $t = \lceil |r| + 1 - (\frac{1-\theta}{\theta} \cdot |r| + 1) \cdot \mathsf{q} \rceil$.

Obviously the object pairs whose mapped sets share smaller than t common elements can be pruned. For example, consider two collections of objects in Figure 1. Suppose the jaccard-similarity threshold is $\theta = 0.8$. For the object r_1, the overlap threshold is $t = \lceil 0.8 \cdot 5 \rceil = 4$. Three object pairs $\langle r_1, s_1 \rangle$, $\langle r_1, s_2 \rangle$, and $\langle r_1, s_3 \rangle$ can be pruned since $|r_1 \cap s_1| = 3 < 4$, $|r_1 \cap s_2| = 3 < 4$, and $|r_1 \cap s_3| = 2 < 4$.

Note that these methods may result in duplicated elements in a mapped set, to avoid multi-set intersection, we append each element with an ordinary number to distinguish duplicated elements [4].

2.2.2 Prefix Filtering

Existing methods utilize a prefix-filtering technique to filter the object pairs which share smaller than t common elements. Firstly, it fixes a global ordering on the elements of all the objects. Then it sorts the elements of each object based on the global ordering. Let $Prefix(r)$ be the prefix set of r that consists of the first $|r|-t+1$ elements. It proves that if $|r \cap s| \geq t$, their prefix sets must have overlap, i.e. $Prefix(r_1) \cap Prefix(s_1) \neq \phi$. Therefore, it can filter the object pairs whose prefix sets have no overlap [4].

For example, the table on the left of Figure 2 shows a global ordering on the elements of all the objects in Figure 1. We use e_i to denote the element in the i-th position of the global ordering. Consider $r_1 = \{$ $\texttt{vldb}$, $\texttt{sigmod}$, $\texttt{icde}$, $\texttt{2011}$, $\texttt{jagadish}\}$ in Figure 1. The corresponding positions of the elements in the global ordering are $e_6 = \texttt{vldb}$, $e_9 = \texttt{sigmod}$, $e_8 = \texttt{icde}$, $e_5 = \texttt{2011}$, $e_1 = \texttt{jagadish}$. After sorting the elements according to the global ordering, we obtain $r_1 = \{e_1, e_5, e_6, e_8, e_9\}$. Similarly, we can obtain the other sorted objects as shown on the right of Figure 2. Suppose $t = 4$. Then $Prefix(r_1) = \{e_1, e_5\}$ and $Prefix(s_1) = \{e_2, e_3\}$. We can filter the pair $\langle r_1, s_1 \rangle$ based on prefix filtering since $Prefix(r_1) \cap Prefix(s_1) = \phi$.

2.2.3 Inverted Index

Note that we do not need to enumerate each object pair $\langle r, s \rangle \in \mathcal{R} \times \mathcal{S}$ to verify whether $Prefix(r) \cap Prefix(s) = \phi$ holds. Instead we use an inverted index to find the object pairs $\langle r, s \rangle \in \mathcal{R} \times \mathcal{S}$ such that $Prefix(r) \cap Prefix(s) \neq \phi$ efficiently. An inverted index maps an element to a list of objects that contain the element. Such a list of objects is called an inverted list. We first build an inverted index on the prefix-set set of objects in a collection, e.g., $\mathcal{S}$, and then enumerate objects in another collection $\mathcal{R}$. For each $r \in \mathcal{R}$, to obtain object $s \in \mathcal{S}$ such that $Prefix(r) \cap Prefix(s) \neq \phi$, we only need to merge the inverted lists of elements in $Prefix(r)$. For example, suppose $t = 4$. The table on the top of Figure 3(a) shows the prefix-set set $\{Prefix(s) \mid s \in \mathcal{S}\}$. Below is the corresponding inverted index. Consider $Prefix(r_1) = \{e_1, e_5\}$. We merge inverted lists $e_1 \rightarrow \{s_4, s_5\}$ and $e_5 \rightarrow \{s_5\}$ to obtain objects s_4 and s_5 whose prefix sets have overlap with $Prefix(r_1)$.

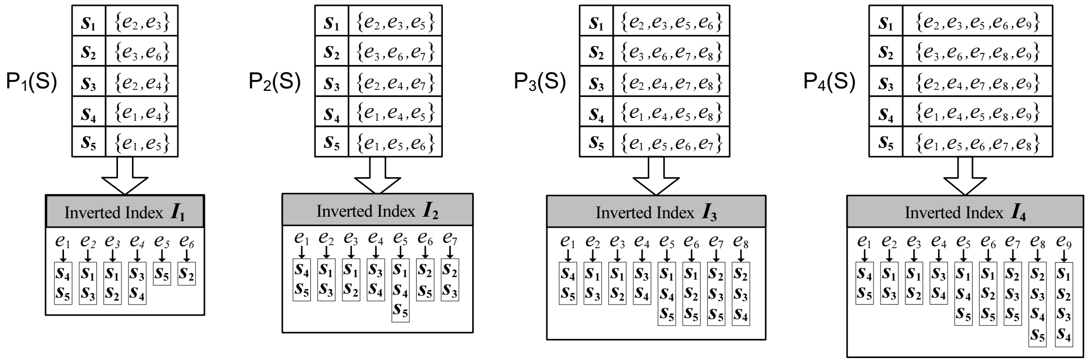
3. FIXED-LENGTH PREFIX SCHEME

Many similarity-join algorithms [2,4,19,22,25–27] have been developed based on the prefix-filtering framework. They neglect the fact that prefix lengths have significant effect on the performance. In this section, we provide a deep analysis of the prefix-filtering framework theoretically and experimentally. We conclude that the prefix-filtering framework is not effective enough and can be improved to achieve higher performance.

For ease of presentation, we first introduce some notations. Suppose the elements of each object are sorted based on a global ordering. Let $\mathcal{P}_\ell$ denote ℓ-prefix scheme. $\mathcal{P}_\ell(s)$ is defined as the ℓ-prefix set of s consisting of the first $|s|-t+\ell$ elements of s ($1 \leq \ell \leq t$). Let $\mathcal{P}_\ell(\mathcal{S}) = \{\mathcal{P}_\ell(s) \mid s \in \mathcal{S}\}$ denote the collection of ℓ-prefix sets of $\mathcal{S}$. Let $\mathcal{I}_\ell^\mathcal{S}$ denote the inverted index built on $\mathcal{P}_\ell(\mathcal{S})$, and $\mathcal{I}_\ell^\mathcal{S}(e)$ denote the inverted list of element e which consists of the objects in $\mathcal{S}$ whose ℓ-prefix sets contain e. For simplicity, if the context is clear, $\mathcal{I}_\ell^\mathcal{S}$ and $\mathcal{I}_\ell^\mathcal{S}(e)$ are abbreviated as $\mathcal{I}_\ell$ and $\mathcal{I}_\ell(e)$ respectively. Figure 3 shows four inverted indexes $\mathcal{I}_\ell$ built on $\mathcal{P}_\ell(\mathcal{S})$ for $1 \leq \ell \leq 4$.

Recall Section 2.2.2, since $Prefix(r)$ consists of the first $|r| - t + 1$ elements of r, the prefix-filtering framework essentially utilizes 1-prefix scheme (i.e. $\mathcal{P}_1$) for filtering object pairs. Next we study filter conditions using other prefix schemes. Consider two objects r and s. Suppose $|r \cap s| \geq t$. For t-prefix scheme, since $r = \mathcal{P}_t(r)$ and $s = \mathcal{P}_t(s)$, we have $|\mathcal{P}_t(r) \cap \mathcal{P}_t(s)| \geq t$. For $(t-1)$-prefix scheme, as $\mathcal{P}_{t-1}(r)$ and $\mathcal{P}_{t-1}(s)$ are respectively obtained by removing the last elements from r and s, we have $|\mathcal{P}_{t-1}(r) \cap \mathcal{P}_{t-1}(s)| \geq t-1$. Iteratively, for ℓ-prefix scheme, we have $|\mathcal{P}_\ell(r) \cap \mathcal{P}_\ell(s)| \geq \ell$. We can prune the object pairs $\langle r, s \rangle$ if $|\mathcal{P}_\ell(r) \cap \mathcal{P}_\ell(s)| < \ell$. The correctness is formalized in Lemma 1.

LEMMA 1. *For any object pair $\langle r, s \rangle \in \mathcal{R} \times \mathcal{S}$, if $|\mathcal{P}_\ell(r) \cap \mathcal{P}_\ell(s)| < \ell$, then $|r \cap s| < t$.*



(a) 1-prefix scheme (b) 2-prefix scheme (c) 3-prefix scheme (d) 4-prefix scheme

Figure 3: Inverted Indexes built on $\mathcal{P}_1(\mathcal{S})$, $\mathcal{P}_2(\mathcal{S})$, $\mathcal{P}_3(\mathcal{S})$, $\mathcal{P}_4(\mathcal{S})$ ($t = 4$)

Next we develop a framework, called FixPrefixScheme, which can use any fixed-length prefix scheme to prune object pairs based on Lemma 1. For simplicity, suppose we use the overlap similarity. Initially, FixPrefixScheme sorts the elements in each object of $\mathcal{R}$ and $\mathcal{S}$ based on the global element ordering. Then the framework builds an inverted index $\mathcal{I}_\ell$ on $\mathcal{P}_\ell(\mathcal{S})$ and utilizes the index to filter pairs $\langle r, s\rangle$ such that $|\mathcal{P}_\ell(r) \cap \mathcal{P}_\ell(s)| < \ell$. To achieve this goal, for each $r \in \mathcal{R}$, it considers the elements e in $\mathcal{P}_\ell(r)$ and retrieves their corresponding inverted lists $\mathcal{I}_\ell(e)$. For any object $s \in \mathcal{I}_\ell(e)$, its ℓ-prefix set, $\mathcal{P}_\ell(s)$, must contain element e. As $e \in \mathcal{P}_\ell(r)$, $\mathcal{P}_\ell(r)$ and $\mathcal{P}_\ell(s)$ share the common element e. And since there is no duplicated element in each object (Section 2.2.1), $|\mathcal{P}_\ell(r) \cap \mathcal{P}_\ell(s)|$ is exactly the number of inverted lists $\mathcal{I}_\ell(e)$ for $e \in \mathcal{P}_\ell(r)$ that contain the object s. We scan the inverted lists one by one and use a hash map $\mathcal{H}[s]$ to maintain the number of inverted lists that contain the object s. If $\mathcal{H}[s] \geq \ell$ holds, we take s as a candidate of r. After scanning all inverted lists, we verify the candidates by computing the real similarity.

Example 1. *Consider two collections of objects, $\mathcal{R}$ and $\mathcal{S}$ in Figure 2. Given an overlap threshold $t = 4$ and 2-prefix scheme $\mathcal{P}_2$, we show how* FixPrefixScheme *utilizes $\mathcal{P}_2$ to find $\langle r, s\rangle \in \mathcal{R} \times \mathcal{S}$ s.t. $|r \cap s| \geq 4$. Firstly, we build an inverted index $\mathcal{I}_2$ on $\mathcal{P}_2(\mathcal{S})$ (See Figure 3(b)). Then we enumerate each $r \in \mathcal{R}$ and find its similar objects in $\mathcal{S}$. Consider $r_1 = \{e_1, e_5, e_6, e_8, e_9\} \in \mathcal{R}$. To obtain similar objects of r_1, we consider its 2-prefix set $\mathcal{P}_2(r_1) = \{e_1, e_5, e_6\}$ that consists of the first $|r_1| - t + \ell = 3$ elements of r_1. We retrieve the inverted lists from $\mathcal{I}_2$, $\mathcal{I}_2(e_1) = \{s_4, s_5\}$, $\mathcal{I}_2(e_5) = \{s_1, s_4, s_5\}$, $\mathcal{I}_2(e_6) = \{s_2, s_5\}$, corresponding to the elements in $\mathcal{P}_2(r_1)$. Since s_4 appears in $\mathcal{I}_2(e_1)$ and $\mathcal{I}_2(e_5)$, we have $\mathcal{H}[s_4] = 2$. As $\mathcal{H}[s_4] \geq \ell = 2$ holds, s_4 is a candidate of r_1. Similarly, we can compute $\mathcal{H}[s_5] = 3$, $\mathcal{H}[s_1] = 1$, and $\mathcal{H}[s_2] = 1$, thus s_5 is also a candidate. Next we verify the candidates by computing $|r_1 \cap s_4|$ and $|r_1 \cap s_5|$, and comparing them with the threshold $t = 4$. As $|r_1 \cap s_4| \geq 4$ and $|r_1 \cap s_5| \geq 4$, s_4 and s_5 are similar objects of r_1.*

Obviously the prefix-filtering framework ($\ell = 1$) is a special case of FixPrefixScheme framework. Next we prove that the prefix-filtering framework cannot always have good performance theoretically and experimentally.

Theoretical Analysis. We analyze the time cost of FixPrefixScheme framework using different prefix schemes. The framework mainly includes the following two steps [1].

- Filter. For each object $r \in \mathcal{R}$, FixPrefixScheme needs to scan the inverted list of each elements $e \in \mathcal{P}_\ell(r)$, the total filter cost is $\sum_{r\in\mathcal{R}} \sum_{e\in\mathcal{P}_\ell(r)} |\mathcal{I}_\ell(e)|$.
- Verification. Let $\mathcal{C}_\ell(r)$ denote the candidate set of r which consists of the objects that appear in at least ℓ inverted lists of the elements in $\mathcal{P}_\ell(r)$ and $cost_v(r)$ denote the average cost of verifying a candidate r. For all objects $r \in \mathcal{R}$, the total verification cost is $\sum_{r\in\mathcal{R}} cost_v(r) \cdot |\mathcal{C}_\ell(r)|$.

By adding the two cost [2], we obtain the total cost of FixPrefixScheme using ℓ-prefix scheme, i.e.

$$\Theta_\ell = \Big(\sum_{r\in\mathcal{R}} \sum_{e\in\mathcal{P}_\ell(r)} |\mathcal{I}_\ell(e)|\Big) + \Big(\sum_{r\in\mathcal{R}} cost_v(r) \cdot |\mathcal{C}_\ell(r)|\Big). \quad (1)$$

Obviously, Θ_1 is the cost of prefix filtering. For the cost of longer prefix schemes, i.e. Θ_ℓ ($\ell > 1$), the filter cost increases since both $\mathcal{P}_\ell(r)$ and $\mathcal{I}_\ell(e)$ increase, while the verification cost decreases since $\mathcal{P}_\ell$ has a more powerful filter condition than $\mathcal{P}_1$ which can lead to fewer candidates (as proved in Lemma 2). Therefore, Θ_ℓ ($\ell > 1$) may involve smaller costs than Θ_1.

Lemma 2. *For any $r \in \mathcal{R}$, $\mathcal{C}_1(r) \supseteq \mathcal{C}_2(r) \supseteq \cdots \supseteq \mathcal{C}_t(r)$.*

Experimental Analysis. We also conduct an experiment on `DBLP-Set` data set (The data set description is in Section 6) to compare the running time of FixPrefixScheme using different prefix schemes. Figure 4 reports the results. The x-axis denotes the overlap threshold which is varied from 8 to 13. We can see that 1-prefix scheme (prefix-filtering) performs the worst among all prefix schemes. For example, when the overlap threshold is $t = 8$, FixPrefixScheme with 1-prefix scheme consumed 10882s while FixPrefixScheme with other prefix schemes took less than 3000s. Another observation is that prefix schemes have a great effect on the performance of FixPrefixScheme. For instance, for threshold $t = 10$, the performance of FixPrefixScheme with different prefix schemes varies from 373s (3-prefix scheme) to 4563s (1-prefix scheme).

From the experiments and the theoretical analysis, we have a conclusion that a fixed prefix scheme may not always achieve the highest performance. To achieve the highest performance, we need to dynamically select the prefix length. More importantly, we do not need to fix the prefix length for all objects. Instead we can select different prefix lengths for different objects. To this end, we propose

[1] We ignore the cost of sorting elements in each object and building an inverted index since the former remains the same for any prefix scheme and the latter is much smaller than other steps.

[2] We suppose all operations have the same unit cost for ease of presentation.

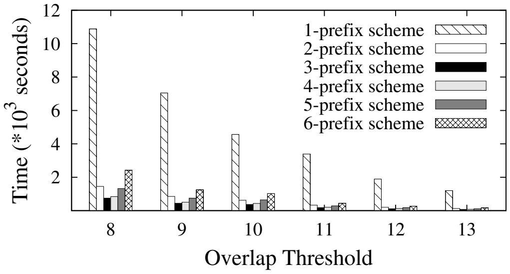


Figure 4: Running time of FixPrefixScheme using different prefix schemes on DBLP-Set data set.

an adaptive framework to judiciously select variable-length prefix schemes for different objects in Section 4.

4. ADAPTIVE FRAMEWORK FOR SimJoin

In this section, we first present a variable-length prefix scheme in Section 4.1. Then in Section 4.2, we propose an adaptive framework to select appropriate prefixes, and give two challenges that arise in our framework. Finally, we present effective methods in Sections 4.3 and 4.4 to address these two problems respectively.

4.1 Variable-Length Prefix Scheme

Instead of fixing the same prefix scheme for all objects, we adaptively select a variable-length prefix scheme for each object $r \in \mathcal{R}$. We call this method AdaptPrefixScheme. Suppose we use the ℓ_r-prefix scheme for object r. The total cost of AdaptPrefixScheme is

$$\Theta = \sum_{r \in \mathcal{R}} \Theta_{\ell_r}(r) = \sum_{r \in \mathcal{R}} \big(\mathrm{F}_{\ell_r}(r) + \mathrm{V}_{\ell_r}(r)\big) \qquad (2)$$

where $\mathrm{F}_{\ell_r}(r)$ is the filter cost

$$\mathrm{F}_{\ell_r}(r) = \sum_{e \in \mathcal{P}_{\ell_r}(r)} |\mathcal{I}_{\ell_r}(e)|, \qquad (3)$$

and $\mathrm{V}_{\ell_r}(r)$ is the verification cost

$$\mathrm{V}_{\ell_r}(r) = cost_v(r) \cdot |\mathcal{C}_{\ell_r}(r)|. \qquad (4)$$

As FixPrefixScheme is a special case of AdaptPrefixScheme, AdaptPrefixScheme performs better than FixPrefixScheme. In this paper we study how to select the best prefix scheme for each object in order to achieve the highest performance. We use the following example to illustrate our basic idea.

Example 2. *Consider the example in Figure 2. Given overlap similarity and the threshold 4, for each $r \in \mathcal{R}$, we respectively utilize different prefix schemes to find objects $s \in \mathcal{S}$ s.t. $|r \cap s| \geq 4$, and compute the corresponding cost.*

Consider the object $r_4 = \{e_1, e_2, e_4, e_6, e_8\} \in \mathcal{R}$. If we use 1-prefix scheme, then $\mathcal{P}_1(r_4) = \{e_1, e_2\}$. We retrieve $\mathcal{I}_1(e_1) = \{s_4, s_5\}$ and $\mathcal{I}_1(e_2) = \{s_1, s_3\}$ from the inverted index in Figure 3(a). We obtain $F_1(r_4) = |\mathcal{I}_1(e_1)| + |\mathcal{I}_1(e_2)| = 4$. As s_1, s_3, s_4, s_5 at least appear in one inverted list, the candidate set is $\mathcal{C}_1(r_4) = \{s_1, s_3, s_4, s_5\}$. Since we need $|r| + |s|$ cost to verity $|r \cap s| \geq 4$, we have $cost_v(r_4) = |r_4| + |s| = 10$, thus $V_1(r_4) = cost_v(r_4) \cdot |\mathcal{C}_1(r_4)| = 40$. The total cost of using 1-prefix scheme is $\Theta_1(r_4) = F_1(r_4) + V_1(r_4) = 44$.

If we use 2-prefix scheme for r_4, then $\mathcal{P}_2(r_4) = \{e_1, e_2, e_4\}$. We retrieve $\mathcal{I}_2(e_1) = \{s_4, s_5\}$, $\mathcal{I}_2(e_2) = \{s_1, s_3\}$ and $\mathcal{I}_2(e_4) = \{s_3, s_4\}$ from the inverted index in Figure 3(b). We have $F_2(r_4) = |\mathcal{I}_2(e_1)| + |\mathcal{I}_2(e_2)| + |\mathcal{I}_2(e_4)| = 6$. As s_3 and s_4 appear in at least two inverted lists, the candidate set is $\mathcal{C}_2(r_4) = \{s_3, s_4\}$, thus $V_2(r_4) = cost_v(r_4) \cdot |\mathcal{C}_2(r_4)| = 20$. The cost of using 2-prefix scheme is $\Theta_2(r_4) = F_2(r_4) + V_2(r_4) = 26$. Similarly $\Theta_3(r_4) = 9$ and $\Theta_4(r_4) = 13$. As $\Theta_3(r_4)$ is minimum, 3-prefix scheme is optimal for r_4.

Table 1 shows $\Theta_\ell(r)$ for all objects $r \in \mathcal{R}$. We can see different objects have various optimal prefix schemes. For example, it is optimal for r_1 to select 1-prefix scheme while for r_2, 2-prefix scheme can lead to the minimum cost. If all objects are required to select the same scheme, the minimum total cost is $\Theta_3 = \sum_{r \in \mathcal{R}} \Theta_3(r) = 100$. But if we can select an optimal prefix scheme for each object, the minimum cost will be $\Theta_1(r_1) + \Theta_2(r_2) + \Theta_3(r_3) + \Theta_3(r_4) + \Theta_4(r_5) = 82$.

Table 1: $\Theta_\ell(r)$ for all objects $r \in \mathcal{R}$.

	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
$\Theta_\ell(r_1)$	**23**	27	31	36
$\Theta_\ell(r_2)$	44	**16**	20	24
$\Theta_\ell(r_3)$	44	26	**19**	23
$\Theta_\ell(r_4)$	44	26	**9**	13
$\Theta_\ell(r_5)$	33	27	21	**15**
Θ_ℓ	188	122	100	111

4.2 Overview of Our Framework

We present an overview of our AdaptPrefixScheme framework. Figure 5 gives the pseudo-code. The framework first builds an index on $\mathcal{S}$ (Line 2). Then it enumerates objects in $\mathcal{R}$ (Line 3). For each $r \in \mathcal{R}$, the framework automatically selects an appropriate prefix scheme $\mathcal{P}_\ell$ for r rather than using a fixed one (Line 4). Next it uses the selected prefix to filter objects in $\mathcal{S}$ and obtain a candidate set of the survived objects (Line 5). Finally, the framework verifies the candidates and returns similar object pairs (Line 6).

Algorithm 1: AdaptPrefixScheme ($\mathcal{R}$, $\mathcal{S}$, t)

```
Input:  R, S : two collections of objects
           t : an overlap threshold
Output: O : all pairs of objects ⟨r, s⟩ such that |r ∩ s| ≥ t
1 begin
2   Build an index that can support variable-length prefix
    schemes on S ;
3   for each r ∈ R s.t. |r| ≥ t do
4       Select a prefix scheme P_ℓ for r;
5       Utilize P_ℓ to filter objects and get candidates;
6       Verify the candidates and add results to O;
7 end
```

Figure 5: AdaptPrefixScheme framework.

In our framework, there are two challenges to select variable-length prefix schemes for objects. The first one is how to use the selected prefix scheme to do filtering and the other one is how to select the prefix scheme for an object.

We first consider the first challenge. Consider two objects r_i and r_j. Suppose r_i selects ℓ_{r_i}-length prefix scheme and r_j selects ℓ_{r_j}-length prefix scheme. Then r_i needs to use the inverted index $\mathcal{I}_{\ell_{r_i}}$ to do filtering while r_j needs to use the inverted index $\mathcal{I}_{\ell_{r_j}}$ to do filtering. To address this issue, a naive method is to build inverted indexes for all prefix schemes, i.e. $\mathcal{I}_1, \mathcal{I}_2, \cdots, \mathcal{I}_t$. Obviously, this method is expensive in terms of indexing time and space. In Section 4.3, we study how to build effective indexes to support effective filtering for variable-length prefix schemes.

Next we consider the second challenge. Given an object r, a straightforward method enumerates each possible prefix scheme $\mathcal{P}_\ell(r)$ ($\ell \in [1,t]$), then estimates the value of $\Theta_\ell(r)$, denoted by $\widehat{\Theta_\ell(r)}$, and finally select $\mathcal{P}_{\ell_o}(r)$ such that $\widehat{\Theta_{\ell_o}(r)}$ is minimum, i.e. $\ell_o = \arg\min_{\ell\in[1,t]} \widehat{\Theta_\ell(r)}$. However, this method neglects the estimation cost. Let $\mathrm{E}_\ell(r)$ denote the estimation cost for estimating $\Theta_\ell(r)$. The total estimation cost to select the optimal prefix scheme is $\sum_{\ell\in[1,t]} \mathrm{E}_\ell(r)$. If the estimation cost is expensive, it will be rather time-consuming to estimate the cost for all prefix schemes.

In addition, to estimate $\Theta_\ell(r)$, we need to estimate the candidate-set size. That is given a group of inverted lists, we need to estimate the number of elements that appear in at least ℓ inverted lists. The VSOL estimator [17] which is proposed to estimate the selectivity of approximate string queries can be applied to address this problem. The technique computes min-wise signatures for each inverted list, and utilizes these signatures to estimate the number of elements. However the cost of computing signatures is very high. For SimJoin queries, this cost should be added to the similarity-join cost. Therefore, it is necessary to develop an estimation approach to avoid such expensive signature-computation step. To address these issues, we propose an efficient method in Section 4.4.

4.3 Delta Inverted Indexes

In this section, we propose delta inverted indexes to support effective filtering using variable-length prefix schemes. Recall the inverted index $\mathcal{I}_\ell$. Given an element e, the inverted list $\mathcal{I}_\ell(e)$ keeps the objects whose ℓ-prefix set contains e. Similarly, $\mathcal{I}_{\ell+1}(e)$ keeps the objects whose $(\ell+1)$-prefix set contains e. Obviously $\mathcal{I}_\ell(e) \subseteq \mathcal{I}_{\ell+1}(e)$. To save space, we only keep the different objects between $\mathcal{I}_\ell(e)$ and $\mathcal{I}_{\ell+1}(e)$. Let $\triangle\mathcal{I}_1(e) = \mathcal{I}_1(e)$ and $\triangle\mathcal{I}_{\ell+1}(e)(1 \le \ell \le t-1)$ denote the delta inverted list of e between $\mathcal{I}_\ell(e)$ and $\mathcal{I}_{\ell+1}(e)$, that is $\triangle\mathcal{I}_{\ell+1}(e) = \mathcal{I}_{\ell+1}(e) - \mathcal{I}_\ell(e)$. Thus we build delta inverted indexes $\triangle\mathcal{I}_1, \cdots, \triangle\mathcal{I}_t$ to replace $\mathcal{I}_1, \cdots, \mathcal{I}_t$.

Then we discuss how to build the delta inverted indexes. Initially, delta inverted indexes are empty. Then for each object $s \in \mathcal{S}$, we visit its elements based on the global element ordering. If the element e is in 1-prefix set of s, we insert s into $\triangle\mathcal{I}_1(e)$; otherwise, we insert s into $\triangle\mathcal{I}_\ell(e)$ such that ℓ-prefix set contains e but $(\ell-1)$-prefix set does not. Since each element in $\mathcal{S}$ is at most added into one delta inverted index, the space complexity is $\mathcal{O}(\sum_{s\in\mathcal{S}} |s|)$. As the time complexity of inserting an element to a list is $\mathcal{O}(1)$, the time complexity is $\mathcal{O}(\sum_{s\in\mathcal{S}} |s|)$.

Example 3. *Consider the collection $\mathcal{S}$ in Figure 2 and suppose $t = 4$. To build delta inverted indexes on $\mathcal{S}$,we first initialize four empty inverted indexes, i.e., $\triangle\mathcal{I}_1$, $\triangle\mathcal{I}_2$, $\triangle\mathcal{I}_3$ and $\triangle\mathcal{I}_4$. Then we insert $s_1, s_2, \cdots, s_5$ into the indexes. Suppose $s_1, \cdots, s_4$ have been inserted. Figure 6 shows the process of inserting $s_5 = \{e_1, e_5, e_6, e_7, e_8\}$. Since the 1-prefix set of s_5 is $\{e_1, e_5\}$, we insert s_5 into $\triangle\mathcal{I}_1(e_1)$ and $\triangle\mathcal{I}_1(e_5)$ respectively. Since e_6 is in 2-prefix set but not in 1-prefix set, we insert s_5 into $\triangle\mathcal{I}_2(e_6)$. Similarly, we insert s_5 into $\triangle\mathcal{I}_3(e_7)$ and s_5 into $\triangle\mathcal{I}_4(e_8)$.*

Next we discuss how to use delta inverted indexes to do filtering. Suppose we want to find the candidates of an object r w.r.t ℓ-prefix scheme. If we use inverted indexes, we need to merge the inverted lists $\mathcal{I}_\ell(e)$ for $e \in \mathcal{P}_\ell(r)$, and find the objects that appear in at least ℓ lists. In terms of delta inverted indexes, since $\mathcal{I}_\ell(e)$ is divided into $\triangle\mathcal{I}_1(e), \cdots, \triangle\mathcal{I}_\ell(e)$, we

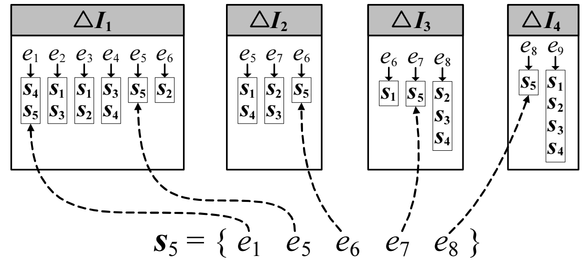


Figure 6: Delta inverted indexes built on the collection $\mathcal{S}$ in Figure 2 ($t = 4$).

need to merge the delta inverted lists $\triangle\mathcal{I}_i(e)$ for $1 \le i \le \ell$ and $e \in \mathcal{P}_\ell(r)$. If we use a hash-based method to merge lists, the method using the delta inverted index has the same time complexity with that using the inverted index $\mathcal{I}_\ell$, i.e., $\sum_{e\in\mathcal{P}_\ell(r)} \sum_{i\in[1,\ell]} |\triangle\mathcal{I}_i(e)| = \sum_{e\in\mathcal{P}_\ell(r)} |\mathcal{I}_\ell(e)|$.

4.4 Adaptively Selecting Prefix Scheme

To select an optimal prefix of an object, the brute-force method which estimates all possible prefix lengths and selects the best one is very expensive as discussed in Section 4.2. To address this issue, we propose a cost-based method to select an appropriate prefix for an object.

We have an observation that with the increase of the prefix length, the overall cost (the sum of the filter cost and verification cost) usually first increases and then decrease. For example, in Figure 4, when the overlap threshold is 8, the running time of FixPrefixScheme first increases with prefix lengths from 1 to 3, and then decreases with prefix lengths from 3 to 6. This is because with the increases of prefix lengths, the filtering time increases and the verification time decreases. Thus there is a tradeoff between the filtering cost and verification cost. Based on this observation, we compare the ℓ-prefix scheme with the $(\ell+1)$-prefix scheme from $\ell = 1$ to $t-1$. If the $(\ell+1)$-prefix scheme is not better than the ℓ-prefix scheme, we stop the algorithm and select the ℓ-prefix scheme as r's prefix scheme; otherwise, we continue to compare the $(\ell+1)$-prefix scheme and the $(\ell+2)$-prefix scheme.

To decide which one is better between ℓ-prefix and $(\ell+1)$-prefix, we compute the total cost of selecting them as r's prefix scheme. If the ℓ-prefix scheme is selected, we need to estimate $\Theta_i(r)$ for each $i \in [1, \ell+1]$, thus the total cost will be $\Theta_\ell(r) + \sum_{i\in[1,\ell+1]} \mathrm{E}_i(r)$. Similarly, if the $(\ell+1)$-prefix scheme is selected, the total cost will be $\Theta_{\ell+1}(r) + \sum_{i\in[1,\ell+2]} \mathrm{E}_i(r)$. Obviously, if $\Theta_\ell(r) < \Theta_{\ell+1}(r) + \mathrm{E}_{\ell+2}(r)$, the ℓ-prefix scheme is better as it takes less cost; otherwise, the $(\ell+1)$-prefix scheme is better. We can see if the algorithm finally selects the ℓ_e-prefix scheme as r's prefix scheme, it only estimate $\Theta_i(r)$ for each $i \in [1, \ell_e+1]$ rather than for all possible prefix schemes (i.e. $i \in [1,t]$). Next, we discuss how to effectively estimate $\Theta_\ell(r)$ and give the estimation cost $\mathrm{E}_{\ell+2}(r)$.

The cost $\Theta_\ell(r)$ consists of the filter cost and the verification cost. Based on Equation 3, we can easily get the filter cost by adding up the lengths of inverted lists of the elements in r's ℓ-prefix set, i.e. $\mathrm{F}_\ell(r) = \sum_{e\in\mathcal{P}_\ell(r)} |\mathcal{I}_\ell(e)|$. As we use the delta inverted indexes, we need add up the lengths of delta inverted lists, i.e. $\mathrm{F}_\ell(r) = \sum_{e\in\mathcal{P}_\ell(r)} \sum_{1\le i\le\ell} |\triangle\mathcal{I}_i(e)|$. For ease of presentation, we use $\Phi_\ell(r)$ to denote the set of delta inverted lists to be merged for ℓ-prefix scheme in the filter step of r, i.e., $\Phi_\ell(r) = \{\triangle\mathcal{I}_i(e) \mid e \in \mathcal{P}_\ell(r), 1 \le i \le \ell\}$. So the filter cost for ℓ-prefix scheme can be equivalently denoted by $\mathrm{F}_\ell(r) = \sum_{\triangle\mathcal{I}(e)\in\Phi_\ell(r)} |\triangle\mathcal{I}(e)|$. Note we do not need to compute the filter cost for ℓ-prefix scheme from scratch, since we have already gotten the filter cost for $(\ell-1)$-prefix scheme, and in the filter step, the set of delta inverted lists to be merged for ℓ-prefix scheme is a superset of the set of those to be merged for $(\ell-1)$-prefix scheme. Let $\triangle\Phi_\ell(r)$ denote the set of additional delta inverted lists to be merged for ℓ-prefix scheme comparing to $(\ell-1)$-prefix scheme, i.e., $\triangle\Phi_\ell(r) = \Phi_\ell(r) - \Phi_{\ell-1}(r)$. Then we have $\mathrm{F}_\ell(r) = \mathrm{F}_{\ell-1}(r) + \sum_{\triangle\mathcal{I}(e)\in\triangle\Phi_\ell(r)} |\triangle\mathcal{I}(e)|$. Therefore, we can obtain $\mathrm{F}_\ell(r)$ by only computing $\sum_{\triangle\mathcal{I}(e)\in\triangle\Phi_\ell(r)} |\triangle\mathcal{I}(e)|$ with $|\triangle\Phi_\ell(r)|$ cost.

In order to get the verification cost w.r.t an object r, we need to estimate the average cost to verify a candidate and the candidate-set size, i.e., $cost_v(r)$ and $|\mathcal{C}_\ell(r)|$. To estimate $cost_v(r)$, consider a candidate s and overlap similarity. Since the elements in each object have been sorted based on the global ordering, we can use Merge-Join algorithm to compute $|r \cap s|$, thus the cost of verifying a candidate is $|r|+|s|$, which is only related to the length of a candidate. So we compute the cost corresponding to every possible length of a candidate and use the average of these cost as the estimator of $cost_v(r)$. Based on this idea, we can obtain $\widehat{cost_v}(r) = \frac{\sum_{|s|=|s|_l}^{|s|_u}(|r|+|s|)}{|s|_u-|s|_l+1} = |r| + \frac{|s|_u+|s|_l}{2}$ for overlap similarity, where $|s_u|$ and $|s_l|$ are respectively the upper-bound and the lower-bound of $|s|$. Using the similar idea, we can obtain $\widehat{cost_v}(r)$ for other similarity functions as shown in Table 2.

Table 2: The estimation of the average cost of verifying a candidate s w.r.t an object r for different similarity functions. ($\theta^* = \lfloor\theta\rfloor$ for edit distance; otherwise for edit similarity, $\theta^* = \lfloor\frac{(1-\theta)\cdot|r|}{\theta}\rfloor$)

SimFunc	$\|s\|_l$	$\|s\|_u$	Verify $\langle r,s\rangle$	$\widehat{cost_v}(r)$
$\mathsf{sim}_o(r,s)$	$\lceil\theta\rceil$	$\max_{s\in\mathcal{S}} \|s\|$	$\|r\|+\|s\|$	$\|r\| + \frac{\|s\|_u+\|s\|_l}{2}$
$\mathsf{sim}_d(r,s)$	$\lceil\frac{\theta}{2-\theta}\cdot\|r\|\rceil$	$\lfloor\frac{2-\theta}{\theta}\cdot\|r\|\rfloor$		
$\mathsf{sim}_c(r,s)$	$\lceil\theta^2\cdot\|r\|\rceil$	$\lfloor\frac{\|r\|}{\theta^2}\rfloor$		
$\mathsf{sim}_j(r,s)$	$\lceil\theta\cdot\|r\|\rceil$	$\lfloor\frac{\|r\|}{\theta}\rfloor$		
$\mathrm{ED}(r,s)$	$\lceil\|r\|-\theta\rceil$	$\lfloor\|r\|+\theta\rfloor$	$(2\theta^*+1)\cdot\min(\|r\|,\|s\|)$	$(2\theta^*+1)\cdot\frac{\|s\|_u^2-\|s\|_l^2+\|r\|+\|s\|_l}{2\cdot(\|s\|_u-\|s\|_l+1)}$ [3]
$\mathrm{ES}(r,s)$	$\lceil\theta\cdot\|r\|\rceil$	$\lfloor\frac{\|r\|}{\theta}\rfloor$		

Next we discuss how to estimate candidate-set size, $|\mathcal{C}_\ell(r)|$. We first estimate candidate-set size w.r.t 1-prefix scheme, $|\mathcal{C}_1(r)|$ (Section 4.4.1), then estimate candidate-set size w.r.t 2-prefix scheme $|\mathcal{C}_2(r)|$ (Section 4.4.2). Finally we extend our method to estimate candidate-set size w.r.t ℓ-prefix scheme $|\mathcal{C}_\ell(r)|$ $(\ell > 2)$ (Section 4.4.3).

4.4.1 Estimating candidate-set size w.r.t 1-prefix scheme

We estimate candidate-set size w.r.t 1-prefix scheme, $|\mathcal{C}_1(r)|$, to decide which one between 1-prefix scheme and 2-prefix scheme is better. If 1-prefix scheme is better, it will be selected as r's prefix scheme. If we use 1-prefix scheme, we need to merge the lists in $\Phi_1(r)$. That is, inserting the objects of each list in $\Phi_1(r)$ into a hash map and find the objects that appear in at least one list. If 2-prefix scheme is better, the selected prefix scheme must be longer than 1-prefix scheme. Suppose ℓ_e-prefix scheme is selected as r's prefix scheme $(\ell_e \geq 2)$. If we use ℓ_e-prefix scheme, we need to merge the lists in $\Phi_{\ell_e}(r)$. That is, inserting the objects of each list in $\Phi_{\ell_e}(r)$ into a hash map and find the objects that appear in at least ℓ_e lists. Since $\Phi_1(r) \subseteq \Phi_{\ell_e}(r)$, when comparing 1-prefix scheme and 2-prefix scheme, no matter which prefix scheme is better, the lists in $\Phi_1(r)$ must be merged. Therefore, we can merge the lists in $\Phi_1(r)$ to get the real value of $|\mathcal{C}_1(r)|$ before comparing 1-prefix scheme and 2-prefix scheme. Example 4 illustrates the method to estimate $|\mathcal{C}_1(r)|$.

EXAMPLE 4. *For example, consider $r_1 = \{e_1, e_5, e_6, e_8, e_9\}$ in Figure 2. To estimate $|\mathcal{C}_1(r_1)|$, we first get $\Phi_1(r_1) = \{\triangle\mathcal{I}_1(e_1), \triangle\mathcal{I}_1(e_5)\}$. As shown in Figure 6, $\triangle\mathcal{I}_1(e_1) = \{s_4, s_5\}$ and $\triangle\mathcal{I}_1(e_5) = \{s_5\}$. Based on our analysis above, no matter which prefix scheme is selected, we need to merge $\triangle\mathcal{I}_1(e_1)$ and $\triangle\mathcal{I}_1(e_5)$, therefore we can obtain $\mathcal{C}_1(r_1) = \{s_4, s_5\}$ by merging these two lists. Then we get the real value $|\mathcal{C}_1(r_1)| = 2$.*

4.4.2 Estimating candidate-set size w.r.t 2-prefix scheme

In this section, we focus on estimating candidate-set size w.r.t 2-prefix scheme, $|\mathcal{C}_2(r)|$, which is the number of objects that appear in at least two lists in $\Phi_2(r)$. As $\mathcal{C}_1(r)$ has been computed (discussed in Section 4.4.1), we can utilize $\mathcal{C}_1(r)$ to estimate $|\mathcal{C}_2(r)|$. Since $\mathcal{C}_1(r) \supseteq \mathcal{C}_2(r)$ (See Lemma 2), we only need to check for each $s \in \mathcal{C}_1(r)$ whether $s \in \mathcal{C}_2(r)$ holds. We divide $\mathcal{C}_1(r)$ into two disjoint sets, $\mathcal{C}_1^{=}(r)$ and $\mathcal{C}_1^{>}(r)$, where $\mathcal{C}_1^{=}(r)$ denotes the set of objects that appear in *only one* list in $\Phi_1(r)$, and $\mathcal{C}_1^{>}(r)$ denotes the set of objects that appear in *more than one* list in $\Phi_1(r)$. For each object $s \in \mathcal{C}_1^{>}(r)$, since $\Phi_1(r) \subseteq \Phi_2(r)$, s must appear in at least two lists in $\Phi_2(r)$, thus $s \in \mathcal{C}_2(r)$. For each object $s \in \mathcal{C}_1^{=}(r)$, if s appears in the lists in $\triangle\Phi_2(r)$, s must appear in at least two lists in $\Phi_2(r) = \Phi_1(r) + \triangle\Phi_2(r)$, i.e. $s \in \mathcal{C}_2(r)$; otherwise, $s \notin \mathcal{C}_2(r)$. Therefore, as shown in Equation 5, $|\mathcal{C}_2(r)|$ can be computed based on $\mathcal{C}_1(r)$.

$$|\mathcal{C}_2(r)| = |\mathcal{C}_1^{>}(r)| + \Big|\mathcal{C}_1^{=}(r) \cap \bigcup_{\triangle\mathcal{I}(e)\in\triangle\Phi_2(r)} \triangle\mathcal{I}(e)\Big|. \quad (5)$$

For instance, consider $\mathcal{C}_1(r_1) = \{s_4, s_5\}$ in Example 4. We show how to compute $|\mathcal{C}_2(r_1)|$ based on $\mathcal{C}_1(r_1)$. $\mathcal{C}_1^{>}(r_1) = \{s_5\}$ since s_5 appears in more than one list in $\Phi_1(r_1)$, i.e. $\triangle\mathcal{I}_1(e_1)$ and $\triangle\mathcal{I}_1(e_5)$. For the objects in $\mathcal{C}_1^{>}(r_1)$, they must belong to $\mathcal{C}_2(r_1)$ (i.e. $s_5 \in \mathcal{C}_2(r_1)$). $\mathcal{C}_1^{=}(r_1) = \{s_4\}$ since s_4 appears in only one list in $\Phi_1(r_1)$ (i.e. $\triangle\mathcal{I}_1(e_5)$). For the objects in $\mathcal{C}_1^{=}(r_1)$, we need check whether they appear in the lists in $\triangle\Phi_2(r_1) = \{\triangle\mathcal{I}_1(e_6), \triangle\mathcal{I}_2(e_1), \triangle\mathcal{I}_2(e_5), \triangle\mathcal{I}_2(e_6)\}$. As $s_4 \in \triangle\mathcal{I}_2(e_5)$, $\Big|\mathcal{C}_1^{=}(r) \cap \bigcup_{\triangle\mathcal{I}(e)\in\triangle\Phi_2(r_1)} \triangle\mathcal{I}(e)\Big| = 1$. Based on Equation 5, we obtain $|\mathcal{C}_2(r)| = 2$.

In order to use Equation 5 to estimate $|\mathcal{C}_2(r)|$, there are two issues that need to be addressed:

1. How to efficiently compute $|\mathcal{C}_1^{>}(r)|$;
2. How to efficiently and effectively estimate $\Big|\mathcal{C}_1^{=}(r) \cap \bigcup_{\triangle\mathcal{I}(e)\in\triangle\Phi_2(r)} \triangle\mathcal{I}(e)\Big|$.

The first one can be easily addressed. Recall the algorithm of estimating $|\mathcal{C}_1(r)|$ in Section 4.4.1. During the process of merging the lists in $\Phi_1(r)$, we maintain a hash map $\mathcal{H}$ with $\mathcal{H}[s]$ storing the number of processed lists that contain object s. Initially, $|\mathcal{C}_1^{>}(r)| = 0$. When finding $\mathcal{H}[s] = 2$ holds, $|\mathcal{C}_1^{>}(r)| = |\mathcal{C}_1^{>}(r)| + 1$. After processing all lists in $\Phi_1(r)$, we return $|\mathcal{C}_1^{>}(r)|$.

[3] This equation is deduced from $(2\theta^*+1)\cdot\frac{\sum_{|s|=|s|_l}^{|s|_u}\min(|r|,|s|)}{|s|_u-|s|_l+1}$

Next we study the second problem. We have an interesting observation that none of lists in $\triangle\Phi_2(r)$ have overlaps. Thus the union of $\triangle\mathcal{I}(e) \in \triangle\Phi_2(r)$ is actually equal to the *multiset* union of $\triangle\mathcal{I}(e) \in \triangle\Phi_2(r)$. Lemma 3 proves the correctness of this observation.

Lemma 3. *Given a collection of S and delta inverted indexes $\triangle\mathcal{I}_1, \cdots, \triangle\mathcal{I}_{\ell+1}$ built on $\mathcal{S}$, for any $r \in \mathcal{R}$, we have*

$$\bigcup_{\triangle\mathcal{I}(e)\in\triangle\Phi_{\ell+1}(r)} \triangle\mathcal{I}(e) = \biguplus_{\triangle\mathcal{I}(e)\in\triangle\Phi_{\ell+1}(r)} \triangle\mathcal{I}(e)$$

Based on Lemma 3, we only need to estimate

$$\Big|\mathcal{C}_1^{=}(r) \ \cap \biguplus_{\triangle\mathcal{I}(e)\in\triangle\Phi_2(r)} \triangle\mathcal{I}(e)\ \Big|. \tag{6}$$

If the context is clear, $\biguplus_{\triangle\mathcal{I}(e)\in\triangle\Phi_2(r)} \triangle\mathcal{I}(e)$ is abbreviated as $\biguplus \triangle\mathcal{I}(e)$ for ease of notation. Given an object $s \in \biguplus \triangle\mathcal{I}(e)$, the conditional probability of $s \in \mathcal{C}_1^{=}(r)$ holds is

$$\mathbb{P}\Big(s \in \mathcal{C}_1^{=}(r) \ \Big|\ s \in \biguplus \triangle\mathcal{I}(e)\Big) = \frac{\big|\mathcal{C}_1^{=}(r) \ \cap \biguplus \triangle\mathcal{I}(e)\ \big|}{\big|\biguplus \triangle\mathcal{I}(e)\big|}. \tag{7}$$

To estimate the conditional probability, consider K sampled objects, $(s^1, s^2, \cdots, s^K)$, which are randomly selected with replacement from $\biguplus \triangle\mathcal{I}(e)$. For any s^i ($i \in [1, K]$), the probability of $s^i \in \mathcal{C}_1^{=}(r)$ holds is equal to the conditional probability in Equation 7, thus an unbiased estimator of the conditional probability is

$$\widehat{\mathbb{P}}\Big(s \in \mathcal{C}_1^{=}(r) \ \Big|\ s \in \biguplus \triangle\mathcal{I}(e)\Big) = \frac{1}{K}\sum_{i=1}^{K} \mathbf{1}_{\mathcal{C}_1^{=}(r)}(s^i), \tag{8}$$

where $\mathbf{1}_{\mathcal{C}_1^{=}(r)}(s^i) = 1$ if $s^i \in \mathcal{C}_1^{=}(r)$ holds, and 0 otherwise.

Note that for a random object s^i, it is very efficient to check whether $\mathbf{1}_{\mathcal{C}_1^{=}(r)}(s^i) = 1$ holds. This is because when estimating $|\mathcal{C}_1(r)|$, we maintain a hash map $\mathcal{H}$ for the objects in $\mathcal{C}_1(r)$, and $\mathbf{1}_{\mathcal{C}_1^{=}(r)}(s^i) = 1$ (i.e. $s^i \in \mathcal{C}_1^{=}(r)$) iff. $\mathcal{H}[s^i] = 1$.

Based on Equations 7 and 8, an unbiased estimator of $\big|\mathcal{C}_1^{=}(r) \ \cap \biguplus \triangle\mathcal{I}(e)\big|$ is

$$\frac{1}{K}\sum_{i=1}^{K} \mathbf{1}_{\mathcal{C}_1^{=}(r)}(s^i) \cdot \ \Big|\biguplus \triangle\mathcal{I}(e)\Big|. \tag{9}$$

Therefore, based on Equations 5 and 9, an unbiased estimator of $\mathcal{C}_2(r)$ is

$$|\widehat{\mathcal{C}_2(r)}| = |\mathcal{C}_1^{>}(r)| + \frac{1}{K}\sum_{i=1}^{K} \mathbf{1}_{\mathcal{C}_1^{=}(r)}(s^i) \cdot \ \Big|\biguplus \triangle\mathcal{I}(e)\Big|. \tag{10}$$

Next we show how to compute this equation efficiently. We first compute $\big|\biguplus \triangle\mathcal{I}(e)\big|$ by adding up the length of each $\triangle\mathcal{I}(e) \in \triangle\Phi_2(r)$. Then we select K random objects from $\biguplus \triangle\mathcal{I}(e)$. To achieve this goal, consider a *virtual* list of objects obtained by joining all delta lists $\triangle\mathcal{I}(e) \in \triangle\Phi_2(r)$. Given a random position in the virtual list, we can return the corresponding object with $\mathcal{O}\big(|\triangle\Phi_2(r)|\big)$ cost. The cost can be improved to $\mathcal{O}\big(\log|\triangle\Phi_2(r)|\big)$ by binary search but requires an extra $\mathcal{O}\big(|\triangle\Phi_2(r)|\big)$ initialization cost. After getting K random objects $(s^1, \cdots, s^K)$, we compute the number of random objects such that $\mathcal{H}[s^i] = 1$ holds, i.e., $\sum_{i=1}^{K} \mathbf{1}_{\mathcal{C}_1^{=}(r)}(s^i)$. Finally, we can obtain $|\widehat{\mathcal{C}_2(r)}|$ based on Equation 10. Example 5 illustrates how to estimate $|\mathcal{C}_2(r)|$.

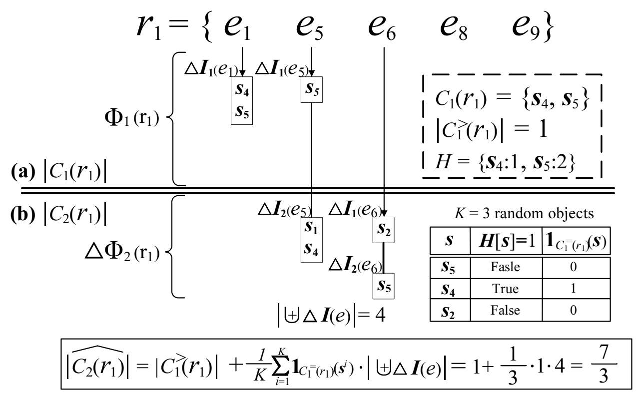


Figure 7: An illustration of estimating $|\mathcal{C}_2(r_1)|$.

Example 5. *Consider r_1 in Example 4. To estimate $|\mathcal{C}_1(r_1)|$, we merge the lists in $\Phi_1(r_1) = \{\triangle\mathcal{I}_1(e_1), \triangle\mathcal{I}_1(e_5)\}$, then we can obtain $\mathcal{C}_1(r_1)$, $|\mathcal{C}_1^{>}(r_1)|$ and $\mathcal{H}$ as shown in Figure 7(a). To estimate $|\mathcal{C}_2(r_1)|$ based on Equation 10, we also need to compute $\big|\biguplus \triangle\mathcal{I}(e)\big|$ and $\frac{1}{K}\sum_{i=1}^{K} \mathbf{1}_{\mathcal{C}_1^{=}(r)}(s^i)$. Figure 7(b) illustrates this process. Since $\triangle\Phi_2(r_1) = \{\triangle\mathcal{I}_2(e_5), \triangle\mathcal{I}_1(e_6), \triangle\mathcal{I}_2(e_6)\}$, $\big|\biguplus \triangle\mathcal{I}(e)\big| = |\triangle\mathcal{I}_2(e_5)| + |\triangle\mathcal{I}_1(e_6)| + |\triangle\mathcal{I}_2(e_6)| = 4$. Suppose $K = 3$ objects, $\{s_5, s_4, s_2\}$, are randomly selected with replacement from $\biguplus \triangle\mathcal{I}(e)$. For the object s_5, as $\mathcal{H}[s_5] \neq 1$, we have $s_5 \notin \mathcal{C}_1^{=}(r)$, thus $\mathbf{1}_{\mathcal{C}_1^{=}(r)}(s_5) = 0$. For the object s_4, as $\mathcal{H}[s_4] = 1$, we have $s_4 \in \mathcal{C}_1^{=}(r)$, thus $\mathbf{1}_{\mathcal{C}_1^{=}(r)}(s_4) = 1$. For the object s_2, as $\mathcal{H}[s_2] \neq 1$, we have $s_2 \notin \mathcal{C}_1^{=}(r)$, thus $\mathbf{1}_{\mathcal{C}_1^{=}(r)}(s_2) = 0$. Therefore, $\frac{1}{K}\sum_{i=1}^{K} \mathbf{1}_{\mathcal{C}_1^{=}(r)}(s^i) = \frac{1}{3}(0+1+0) = \frac{1}{3}$. Based on Equation 10, we obtain $|\widehat{\mathcal{C}_2(r_1)}| = 1 + \frac{1}{3}\cdot 1 \cdot 4 = \frac{7}{3}$.*

4.4.3 *Estimating candidate-set size w.r.t ℓ-prefix scheme ($\ell > 2$)*

We extend the estimation method of candidate-set size w.r.t 2-prefix scheme to support ℓ-prefix scheme, $|\mathcal{C}_\ell(r)|$ ($\ell > 2$), which uses $\mathcal{H}$ and $|\mathcal{C}_1^{=}(r)|$ to estimate $|\mathcal{C}_2(r)|$, where $\mathcal{H}$ is obtained by merging the lists in $\Phi_1(r)$. Next we show that the corresponding $\mathcal{H}$ and $|\mathcal{C}_{\ell-1}^{=}(r)|$ can also be computed before the estimation of $|\mathcal{C}_\ell(r)|$ ($\ell > 2$). We use $|\mathcal{C}_3(r)|$ as an example to introduce our idea. $|\mathcal{C}_3(r)|$ needs to be estimated only when 2-prefix scheme is better than 1-prefix scheme. In this case, 1-prefix scheme will not be selected as r's prefix scheme. We estimate $|\mathcal{C}_3(r)|$ in order to decide either 2-prefix scheme or 3-prefix scheme is better. Using a similar analysis as Section 4.4.1, we can merge the lists in $\Phi_2(r)$ in advance, and obtain $\mathcal{H}$ and $|\mathcal{C}_2^{=}(r)|$ before the estimation of $|\mathcal{C}_3(r)|$. Since the lists in $\Phi_1(r)$ have been merged when estimating $|\mathcal{C}_2(r)|$, we only need to merge the lists in $\triangle\Phi_2(r)$. Similarly, we can also deduce that the corresponding $\mathcal{H}$ and $|\mathcal{C}_{\ell-1}^{=}(r)|$ can be computed before the estimation of $|\mathcal{C}_\ell(r)|$ ($\ell > 2$). Thus an unbiased estimator of $|\mathcal{C}_\ell(r)|$ is

$$|\widehat{\mathcal{C}_\ell(r)}| = |\mathcal{C}_{\ell-1}^{>}(r)| + \frac{1}{K}\sum_{i=1}^{K} \mathbf{1}_{\mathcal{C}_{\ell-1}^{=}(r)}(s^i) \cdot \Big| \biguplus_{\triangle\mathcal{I}(e)\in\triangle\Phi_\ell(r)} \triangle\mathcal{I}(e)\ \Big|, \tag{11}$$

where $(s^1, s^2, \cdots, s^K)$ are K sampled objects randomly selected with replacement from $\biguplus_{\triangle\mathcal{I}(e)\in\triangle\Phi_\ell(r)} \triangle\mathcal{I}(e)$, and $\mathbf{1}_{\mathcal{C}_{\ell-1}^{=}(r)}(s^i) = 1$ if $s^i \in \mathcal{C}_{\ell-1}^{=}(r)$ holds and 0 otherwise.

Our estimation algorithm can obtain an unbiased estimator of $|\mathcal{C}_\ell(r)|$ and the estimator will become more accurate with the increase of the number of sampled objects. The correctness is proved in Theorem 1.

Theorem 1. *Let* $0 < \delta < 1$, $\epsilon > 0$, $K \geq \frac{2}{\epsilon^2} \cdot \log \frac{2}{\delta} \cdot \frac{1}{\mathcal{C}^*}$. *Then we have (1)* $\mathbb{E}(|\widehat{\mathcal{C}_\ell(r)}|) = |\mathcal{C}_\ell(r)|$; *(2)* $\mathbb{P}(\frac{\left||\mathcal{C}_\ell(r)| - |\widehat{\mathcal{C}_\ell(r)}|\right|}{|\mathcal{C}_\ell(r)|} \geq \epsilon) \leq \delta$, *where* $\mathbb{E}(\cdot)$ *denotes the expected value and* $\mathcal{C}^* = |\mathcal{C}_\ell(r)| - |\mathcal{C}^{>}_{\ell-1}(r)|$.

Next we analyze the cost of estimating $\Theta_\ell(r)$, i.e., $\mathrm{E}_\ell(r)$. $\Theta_\ell(r)$ consists of filter cost and verification cost. The filter cost can be estimated with $|\triangle\Phi_\ell(r)|$ cost as shown at the beginning of Section 4.4. To estimate the verification cost, we need to estimate candidate-set size w.r.t ℓ-prefix scheme. Recall our estimation algorithm, selecting K random objects needs $|\triangle\Phi_\ell(r)| + K \cdot \log|\triangle\Phi_\ell(r)|$ cost, and checking $\mathcal{H}[s^i] = \ell - 1$ for all random objects needs $|K|$ cost. Therefore, the total cost of estimating $\Theta_\ell(r)$ is $\mathrm{E}_\ell(r) = 2 \cdot |\triangle\Phi_\ell(r)| + K \cdot \log|\triangle\Phi_\ell(r)| + K$. Based on the definition of $\Phi_\ell(r)$, we have $|\Phi_\ell(r)| = |\mathcal{P}_\ell(r)| \cdot \ell = (|r| - t + \ell) \cdot \ell$. Thus $|\triangle\Phi_\ell(r)| = |\Phi_\ell(r)| - |\Phi_{\ell-1}(r)| = |r| - t + 2\ell - 1$ which is quite small and increases linearly with ℓ.

5. ADAPTIVE FRAMEWORK FOR SimSearch

In this section, we study how to extend our adaptive framework to support a SimSearch query. Recall SimJoin, given $\mathcal{R}$ and $\mathcal{S}$, our framework first builds delta inverted indexes on $\mathcal{S}$ based on a specified similarity threshold, and then utilizes the index to find similar objects for each $r \in \mathcal{R}$ w.r.t the same specified similarity threshold. Different from a SimJoin query, before answering a SimSearch query, we have no idea about which threshold will be specified, so the index built on $\mathcal{S}$ should be able to deal with any threshold.

A straightforward method is to build delta inverted indexes for all possible thresholds. However, the number of possible thresholds may be large, e.g. there are $\max_{s\in\mathcal{S}}|s|$ possible thresholds for a SimSearch query w.r.t overlap similarity, so the method will incur a huge index size. In the following, we design an index structure that has the same size as the inverted index built on $\mathcal{S}$ but can support a SimSearch query with any threshold.

We have an observation that the objects with the same length will have the same number of elements in their ℓ-prefix set (i.e., $|s| - t + \ell$). In this way we can group objects in $\mathcal{S}$ according to their lengths. Let $\mathcal{S}^{|s|}$ denote the group of objects with length $|s|$. The maximal threshold of a SimSearch query for $\mathcal{S}^{|s|}$ is $|s|$. Instead of building delta inverted indexes on $\mathcal{S}^{|s|}$ for each threshold in $[1, |s|]$, we build delta inverted indexes only for the maximal threshold $|s|$, denoted by $\triangle\mathcal{I}^{|s|}_1$, $\triangle\mathcal{I}^{|s|}_2$, $\cdots$, $\triangle\mathcal{I}^{|s|}_{|s|}$. We can easily see the total index size is the same as the inverted index built on $\mathcal{S}$, i.e., $\mathcal{O}(\sum_{s\in\mathcal{S}}|s|)$. For example, consider $\mathcal{S}$ in Figure 2. We show its index structure in Figure 8. Since all objects in $\mathcal{S}$ has the same length, there is only one group, i.e. $\mathcal{S}^5$. For this group, we use the same method as SimJoin to build delta inverted indexes for threshold 5, i.e., $\triangle\mathcal{I}^5_1$, $\cdots$, $\triangle\mathcal{I}^5_5$.

Consider a query object r, a threshold θ and a deduced overlap threshold t (Section 2.2.1). To use our adaptive framework to find candidates $s \in \mathcal{S}$ such that $|r \cap s| \geq t$, we can use the above index structure to generate an inverted list $\mathcal{I}_\ell(e)$ which consists of objects whose ℓ-prefix set contains e. Since $\triangle\mathcal{I}^{|s|}_i(e)$ consists of the objects with length $|s|$ whose i-th element is e, and ℓ-prefix set contains $|s| - t + \ell$ elements, the objects with length $|s|$ whose ℓ-

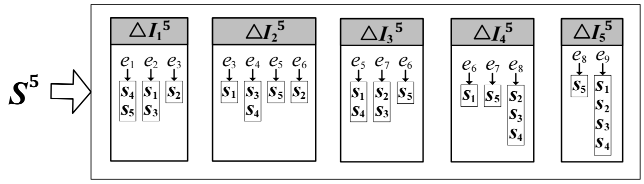


Figure 8: An SimSearch index structure built on $\mathcal{S}$ in Figure 2.

prefix set contains e can be represented by $\cup_{i=1}^{|s|-t+\ell}\triangle\mathcal{I}^{|s|}_i(e)$. Notice in Table 2, we have deduced the upper-bound ($|s|_u$) and the lower-bound ($|s|_l$) of the length of r's candidates. Therefore, the inverted list $\mathcal{I}_\ell(e)$ can be generated by $\cup_{|s|=|s|_l}^{|s|_u}\cup_{i=1}^{|s|-t+\ell}\triangle\mathcal{I}^{|s|}_i(e)$. For example, consider the index structure in Figure 8. Given $r = \{e_5, e_6, e_7, e_8, e_9\}$, $\theta = 4$ and a deduced overlap threshold $t = \lceil\theta\rceil = 4$, we compute $|s|_l = \lceil\theta\rceil = 4$, $|s|_u = \max_{s\in\mathcal{S}}|s| = 5$. Suppose we want to generate $\mathcal{I}_2(e_5)$. Based on our index structure, we have $\cup_{|s|=4}^{5}\cup_{i=1}^{|s|-4+2}\triangle\mathcal{I}^{|s|}_i(e_5)$=$\triangle\mathcal{I}^5_2(e_5) \cup \triangle\mathcal{I}^5_3(e_5)$.

Position-aware Pruning. As discussed above, we can merge some delta inverted lists in our index structure to generate an inverted list $\mathcal{I}_\ell(e)$. Next we propose a technique to prune delta inverted lists in order to further improve the performance. Consider a delta inverted list $\triangle\mathcal{I}^{|s|}_i(e)$. For any object $s \in \triangle\mathcal{I}^{|s|}_i(e)$, we have $s[i] = e$. Let e be the j-th element of a query object r, i.e. $r[j] = e$. We show the first punning condition on the left part of Figure 9. Since $s[i] = r[j]$ and the elements in s and r are sorted based on the same global ordering, the elements before $s[i]$ at most share j common elements with those before $r[j]$ and the elements after $s[i]$ at most share $|s| - i$ common elements with those after $r[j]$, the overlap between s and r is at most $j + (|s| - i)$. If $j + (|s| - i) < t$ holds, then the overlap between s and r must smaller than t, thus we can prune $\triangle\mathcal{I}^{|s|}_i(e)$. Similarly, we obtain another pruning condition as shown on the right part of Figure 9. That is if $i + (|r| - j) < t$ holds, we can prune $\triangle\mathcal{I}^{|s|}_i(e)$. Therefore, for $\cup_{|s|=|s|_l}^{|s|_u}\cup_{i=1}^{|s|-t+\ell}\triangle\mathcal{I}^{|s|}_i(e)$, we prune $\triangle\mathcal{I}^{|s|}_i(e)$ if $i > j + |s| - t$ or $i < j - |r| + t$. Recall the above example. We prune $\triangle\mathcal{I}^5_3(e_5)$ as $i > j + |s| - t$ (i.e., $3 > 1 + 5 - 4$).

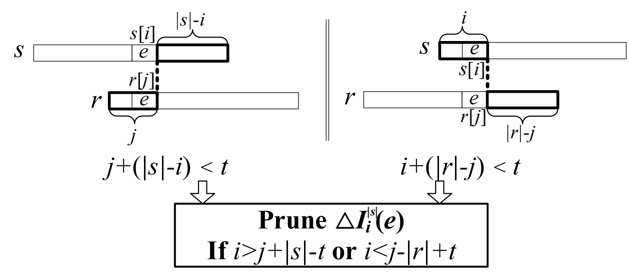


Figure 9: An illustration of position-aware pruning.

6. EXPERIMENT

We have implemented our techniques to support SimSearch and SimJoin queries, and compared with the following state-of-the-art methods. ppjoin and ppjoin+ [27] are prefix-filtering based algorithms that can answer SimJoin queries for Jaccard and Cosine similarities. They both utilize position filtering to optimize their algorithms. However ppjoin+ also employs suffix filtering to further prune candidates. Ed-Join [25] is a prefix-filtering based algorithm that can han-

Table 3: Dataset statistics

Data Sets	Sizes	*avg_len*	*max_len*	*min_len*
QueryLog-String	1,208,844	20.94	500	1
DBLP-String	1,385,925	105.294	1626	1
DBLP-Set	1,385,925	15.74	290	1
ENRON-Set	517,431	133.57	3162	1

dle SimJoin queries for Edit distance. Trie-Join [23] is a trie based algorithm that can support SimJoin queries for Edit distance. ChunkGram [19] is a prefix-filtering based algorithm that can answer SimJoin and SimSearch queries for Edit distance. Flamingo[4] is a data cleaning package that includes DivideSkip [13] algorithm to answer SimSearch queries for Jaccard similarity, Cosine similarity, and Edit distance. We downloaded these algorithms from their respective websites. Although there are some other methods, such as Part-Enum [1], B^{ed}-Tree [29], All-Pairs [2], prior work [19,27] has shown that they cannot outperform the above selected algorithms.

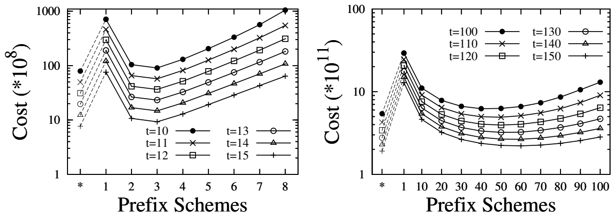
We used four real data sets to evaluate our methods. 1) DBLP-String was obtained from the DBLP Bibliography[5]. Each string is a concatenation of author names and the title of a publication. 2) QueryLog-String is a collection of query strings that were randomly chosen from the AOL Query Log[6]. 3) DBLP-Set was derived from DBLP-String by splitting each string into a token set based on non-alphanumeric characters. 4) ENRON-Set was obtained from the Enron email collection[7]. We split the email title and body into a token set based on non-alphanumeric characters. We assume the elements in each data set have no weight, which is the same as many prior work [5,13,14,19,25,27,29]. Table 3 shows more details about the data sets.

All the algorithms were implemented in C++ and compiled using GCC 4.2.3 with -O3 flag. We used inverse document frequency (IDF) to sort the elements. All the experiments were run on a Ubuntu machine with an Intel Core 2 Quad X5450 3.00GHz processor and 4 GB memory.

6.1 Variable-Length Prefix Scheme

In this section, we compare variable-length prefix scheme with fixed-length prefix scheme by computing their total cost in the filter and verification step w.r.t overlap similarity. For the variable-length prefix scheme, we specified the prefix scheme for each object with the minimum cost. For the fixed-length prefix scheme, we specified the same prefix scheme for all objects. Figure 10 reports the results on DBLP-Set and ENRON-Set data sets. In the X axis, "$*$" refers to the variable-length prefix scheme, and an integer refers to the fixed-length prefix scheme and the integer value refers to the specified prefix scheme. We see that the variable-length prefix scheme always took less cost than the fixed-length prefix scheme. For example, on the DBLP-Set data set, when the threshold is $t = 15$, even if the best prefix scheme was specified for fixed-length prefix scheme, i.e. 3-prefix scheme, its cost $(9.25 * 10^8)$ was still 21% larger than that of the variable-length prefix scheme $(7.66*10^8)$. The reason is that different objects may have different optimal prefix schemes. Therefore, we need to study how to adaptively selecting a prefix scheme for an object instead of using a fixed one.

[4]http://flamingo.ics.uci.edu
[5]http://www.informatik.uni-trier.de/~ley/db
[6]http://www.gregsadetsky.com/aol-data/
[7]http://www.cs.cmu.edu/~enron/

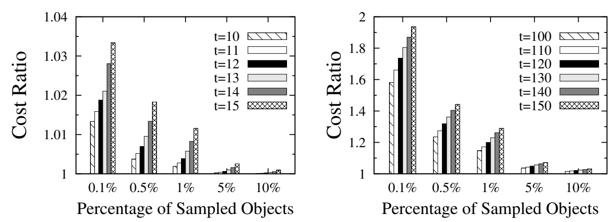


(a) DBLP-Set (b) ENRON-Set

Figure 10: Comparison of variable-length prefix scheme and fixed-length prefix scheme

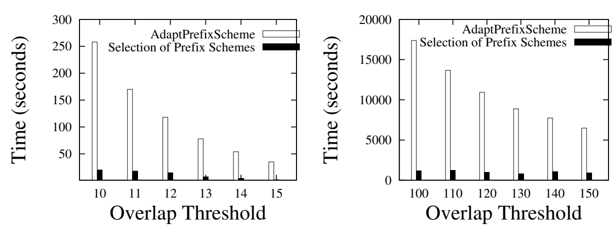
6.2 Adaptive Selection of Prefix Schemes

In this section, we evaluate the quality of our adaptive selection method. If our method could not estimate the cost effectively, it would select a bad prefix scheme. We computed the cost of performing a SimJoin query using our method and that of the optimal method which used the prefix scheme with the minimal cost, and reported the ratio of the cost of our method to that of the optimal method, by varying percentages of sampled objects. Figure 11 shows the result. We can see with the increase of percentage of sampled objects, the cost ratio became smaller. On the DBLP-Set data set, when the percentage is larger than 1%, the cost ratio was smaller than 1.015. That is, our method at most needed 1.5% more cost than the optimal method. On the ENRON-Set data set, we found the optimal prefix scheme was typically longer than 30 (see Figure 10(b)), thus our method needed to perform cost estimation more than 30 times for an object. However even for such data set, when the percentage is larger than 1%, our method at most needed 30% more cost than the optimal method. These results indicated that our estimation method was very effective. Note that increasing the percentage of sampled objects would make the estimation process more expensive, and we sampled 1% objects for our estimation method in the following experiments.



(a) DBLP-Set (b) ENRON-Set

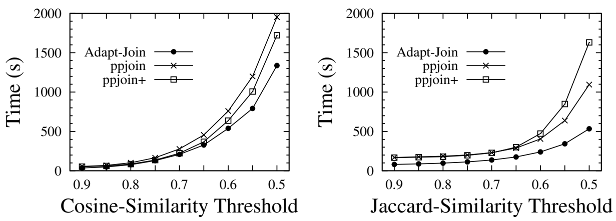
Figure 11: Evaluating effectiveness of adaptive selection of prefix schemes.



(a) DBLP-Set (b) ENRON-Set

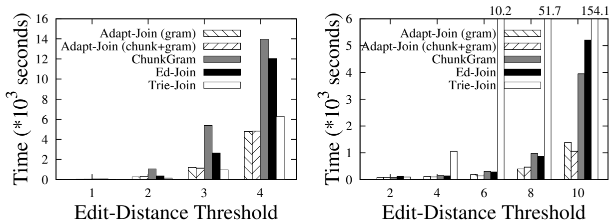
Figure 12: Evaluating efficiency of adaptive selection of prefix schemes.

Next we evaluate the efficiency of our adaptive selection method. We varied the overlap thresholds, and computed the running time of AdaptPrefixScheme. AdaptPrefixScheme needed to include the selection time of prefix



(a) DBLP-Set (b) ENRON-Set

Figure 13: Comparison of adaptive similarity-join algorithms and existing methods w.r.t set similarity.



(a) QueryLog-String (b) DBLP-String

Figure 14: Comparison of adaptive similarity-join algorithms and existing methods w.r.t edit distance.

schemes. In Figure 12, we see that the selection of prefix schemes took a little time comparing to the total running time of AdaptPrefixScheme. For example, on the ENRON-Set data set, when the threshold is $t = 130$, AdaptPrefixScheme took 8887s while only 807s was used for the selection of prefix schemes.

6.3 SimJoin Query

In this section, we evaluate our adaptive framework for SimJoin query by comparing with state-of-the-art methods.

For set objects, we implemented an algorithm, namely Adapt-Join, by replacing the prefix-filtering framework of ppjoin with our adaptive framework. We compared Adapt-Join with ppjoin and ppjoin+ on answering SimJoin query for Jaccard and Cosine similarities. In Figure 13, with thresholds decreasing, the running time of Adapt-Join increased slower than that of the other algorithms. This is because Adapt-Join could adaptively select prefix schemes while ppjoin and ppjoin+ simply used 1-prefix scheme. For small thresholds, 1-prefix scheme would generate large numbers of candidates for verification. Our framework adaptively selected longer prefix schemes to reduce the candidate number.

For string objects, we implemented two algorithms based on our adaptive framework, namely Adapt-Join (gram) and Adapt-Join (chunk+gram). They differed in the methods of mapping a string object to a set object. The first one used a gram-based method [5]. The second one used a gram-chunk-based method [19]. We compared them with ChunkGram, Ed-Join, and Trie-Join on answering a SimJoin query for Edit Distance. In Figure 14, we can see our algorithms outperformed ChunkGram and Ed-Join on both data sets. Especially for large edit-distance thresholds (e.g. 10), in Figure 14(b), our algorithms were 4-5 times faster. This is because ChunkGram and Ed-Join were based on prefix-filtering framework which generated large numbers of candidates for large thresholds. However our algorithms used an adaptive framework which can reduce the number of candidates by adaptively selecting prefix schemes.

On QueryLog-String data set, we see that Trie-Join consumed the least time when the edit-distance threshold is smaller than 3. This is because the QueryLog-String data set contained a lot of short strings and Trie-Join used a trie-based framework which was especially efficient for short strings. However, when the threshold became larger, our algorithms outperformed Trie-Join. In addition, on DBLP-Set data set, our algorithms were several orders of magnitude faster than Trie-Join since the data set mainly consisted of long strings and the trie-based framework was not suitable for such data set. Therefore, our algorithms were very robust for different data sets with different string lengths.

6.4 SimSearch Query

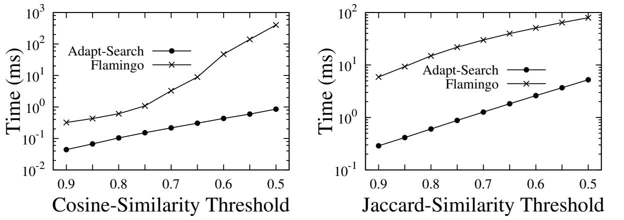
In this section, we compare our adaptive framework with state-of-the-art methods for SimSearch queries.

For set objects, we implemented an algorithm based on the adaptive framework in Section 5, namely Adapt-Search. We compared Adapt-Search with Flamingo using Jaccard and Cosine similarities. We randomly generated 10,000 queries from each data set and compared the average running time. In Figure 15, we can see that Adapt-Search was faster than Flamingo by 1-2 orders of magnitude.

For string objects, we respectively used a gram-based method and a chunk+gram based method to map string objects to set objects, and implemented two algorithms, namely Adapt-Search (gram) and Adapt-Search (chunk+gram). We compared them with ChunkGram and Flamingo using Edit Distance. Figure 16 shows the average time of 10,000 queries. We have four observations from the figure. First, Adapt-Search (gram) performed the best among all algorithms. Second, Adapt-Search (chunk+gram) outperformed ChunkGram on both data sets since Adapt-Search (chunk+gram) used an adaptive framework while ChunkGram adopted the prefix-filtering framework. Third, ChunkGram cannot perform well on the QueryLog-String data set. Based on prefix filtering, ChunkGram can remove some frequent grams (chunks) from a gram (chunk) set to obtain a prefix set, however for the QueryLog-String data set which consisted of short string objects, it can only remove a few grams (chunks) for each string object. Therefore, many frequent grams (chunks) will be left in the prefix set and ChunkGram generated large numbers of candidates for verification. Fourth, Flamingo performed well on QueryLog-String data set since it used an effective filtering method to reduce a large number of candidates, but performed worse on DBLP-String data set since this filtering method became much expensive on the data set with long strings.

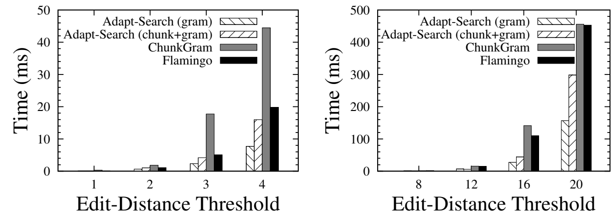
7. RELATED WORK

Similarity joins have been widely studied in [1,2,4,5,8, 16,19–29]. Existing methods usually adopted the prefix-filtering framework. Chaudhuri et al. [4] proposed a primitive operator based on prefix filtering to address the similarity-join problem. Bayardo et al. [2] utilized the prefix-filtering framework and the ordering of vectors to find similar vector pairs from a collection of vector data. Xiao et al. [27] improved [2] by using positional filtering and suffix filtering. Xiao et al. [25] extended the prefix-filtering framework to support edit distance by using a gram-based method. Xiao et al. [26] proposed an approach to deal with top-k similarity joins, which can directly find the top-k results without a given threshold. Vernica et al. [22] proposed to use MapReduce to support similarity joins. Qin et al. [19] proposed a novel asymmetric method to map string objects to set objects, and then employed the prefix-filtering framework to address similarity-join and similarity-search problems. We used our adaptive framework to extend these methods, and



(a) DBLP-Set (b) ENRON-Set

Figure 15: Comparison of adaptive similarity-search algorithms and existing methods w.r.t set similarity.



(a) QueryLog-String (b) DBLP-String

Figure 16: Comparison of adaptive similarity-search algorithms and existing methods w.r.t edit distance.

showed the superiority of our framework over the prefix-filtering framework in the experiment.

There are many studies on similarity search [3,6,10,13,14, 19,29]. Our method differs from theirs as we use an adaptive method. Li et al. [14] developed a technique that can choose high-quality grams of variable lengths, called VGRAM, from a collection of strings. Our problem is orthogonal to theirs since they focused on how to map string objects to sets (i.e., VGRAM sets) while we focus on how to select elements from each obtained set.

The other related studies are selectivity estimation of SimSearch and SimJoin queries [7,9,11,12,17]. Existing methods typically require an expensive initial process to support efficient estimations, such as computing min-wise signatures [17] or creating the summary structures of objects [9]. They are not applicable to address our problem. There are also some studies on approximate string matching [18] and approximate entity extraction [15].

8. CONCLUSION

In this paper, we have studied the problem of similarity join and similarity search. We proposed an adaptive framework to support the two types of queries. We theoretically and experimentally proved that the prefix filtering did not always achieve high performance. We also found that different objects should use different prefix lengths. We developed a cost model to judiciously select an appropriate prefix for each object. We devised delta inverted indexes to efficiently select an appropriate prefix. We extended our method to support SimSearch queries. We have implemented our method and compared with state-of-the-art methods. Experimental results show that our adaptive outperforms the prefix-filtering framework and achieves high performance for both similarity join and similarity search.

Acknowledgement. This work was partly supported by the National Natural Science Foundation of China under Grant No. 61003004, the National Grand Fundamental Research 973 Program of China under Grant No. 2011CB302206, National S&T Major Project of China under Grant No. 2011ZX01042-001-002, a project of Tsinghua University under Grant No. 20111081073, and the "NExT Research Center" funded by MDA, Singapore, under the Grant No. WBS:R-252-300-001-490.

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] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD Conference*, pages 313–324, 2003.
[4] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, pages 5–16, 2006.
[5] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.
[6] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava. Fast indexes and algorithms for set similarity selection queries. In *ICDE*, pages 267–276, 2008.
[7] M. Hadjieleftheriou, X. Yu, N. Koudas, and D. Srivastava. Hashed samples: selectivity estimators for set similarity selection queries. *PVLDB*, 1(1):201–212, 2008.
[8] E. H. Jacox and H. Samet. Metric space similarity joins. *ACM Trans. Database Syst.*, 33(2), 2008.
[9] L. Jin and C. Li. Selectivity estimation for fuzzy string predicates in large data sets. In *VLDB*, pages 397–408, 2005.
[10] M.-S. Kim, K.-Y. Whang, J.-G. Lee, and M.-J. Lee. n-gram/2l: A space and time efficient two-level n-gram inverted index structure. In *VLDB*, pages 325–336, 2005.
[11] H. Lee, R. T. Ng, and K. Shim. Power-law based estimation of set similarity join size. *PVLDB*, 2(1):658–669, 2009.
[12] H. Lee, R. T. Ng, and K. Shim. Similarity join size estimation using locality sensitive hashing. *PVLDB*, 4(6):338–349, 2011.
[13] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, 2008.
[14] C. Li, B. Wang, and X. Yang. Vgram: Improving performance of approximate queries on string collections using variable-length grams. In *VLDB*, pages 303–314, 2007.
[15] G. Li, D. Deng, and J. Feng. Faerie: efficient filtering algorithms for approximate dictionary-based entity extraction. In *SIGMOD Conference*, pages 529–540, 2011.
[16] G. Li, D. Deng, J. Wang, and J. Feng. Pass-join: A partition-based method for similarity joins. *PVLDB*, 5(3):253–264, 2011.
[17] A. Mazeika, M. H. Böhlen, N. Koudas, and D. Srivastava. Estimating the selectivity of approximate string queries. *ACM Trans. Database Syst.*, 32(2):12, 2007.
[18] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.
[19] J. Qin, W. Wang, Y. Lu, C. Xiao, and X. Lin. Efficient exact edit similarity query processing with the asymmetric signature scheme. In *SIGMOD Conference*, pages 1033–1044, 2011.
[20] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD Conference*, pages 743–754, 2004.
[21] Y. N. Silva, W. G. Aref, and M. H. Ali. The similarity join database operator. In *ICDE*, pages 892–903, 2010.
[22] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD Conference*, pages 495–506, 2010.
[23] J. Wang, G. Li, and J. Feng. Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. *PVLDB*, 3(1):1219–1230, 2010.
[24] 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.
[25] C. Xiao, W. Wang, and X. Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB*, 1(1):933–944, 2008.
[26] C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k set similarity joins. In *ICDE*, pages 916–927, 2009.
[27] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, 2008.
[28] J. Zhai, Y. Lou, and J. Gehrke. Atlas: a probabilistic algorithm for high dimensional similarity search. In *SIGMOD Conference*, pages 997–1008, 2011.
[29] Z. Zhang, M. Hadjieleftheriou, B. C. Ooi, and D. Srivastava. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In *SIGMOD*, 2010.