Leveraging Set Correlations in Exact Set Similarity Join

Xubo Wang[1] Lu Qin[2] Xuemin Lin[1] Ying Zhang[2]
Lijun Chang[1]

[1] University of New South Wales, Australia
{xwang,lxue,ljchang}@cse.unsw.edu.au
[2] University of Technology, Sydney, Australia
{lu.qin,ying.zhang}@uts.edu.au

Technical Report
UNSW-CSE-TR-201616
October 2016

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

Exact set similarity join, which finds all the similar set pairs from two collections of sets, is a fundamental problem with a wide range of applications. The existing solutions for set similarity join follow a filtering-verification framework, which generates a list of candidate pairs through scanning indexes in the filtering phase, and reports those similar pairs in the verification phase. Though much research has been conducted on this problem, the set correlations, which we find out is quite effective on improving the algorithm efficiency through computational cost sharing, has never been studied. Therefore, in this paper, instead of considering each set individually, we explore the set correlations in different levels to reduce the overall computational costs. First, it has been shown that most of the computational time is spent on the filtering phase, which can be quadratic to the number of sets in the worst case for the existing solutions. Thus we explore index-level correlations to reduce the filtering cost to be linear to the size of the input while keeping the same filtering power. We achieve this by grouping related sets into blocks in the index and skipping useless index probes in joins. Second, we explore answer-level correlations to further improve the algorithm based on the intuition that if two sets are similar, their answers may have a large overlap. We derive an algorithm to incrementally generate the answer of one set from an already computed answer of another similar set rather than compute the answer from scratch to reduce the computational cost. Finally, we conduct extensive performance studies using 20 real datasets with various data properties from a wide range of domains. The experimental results demonstrate that our algorithm outperforms all the existing algorithms across all datasets and can achieve more than an order of magnitude speed-up against the state-of-the-art algorithms.

1 Introduction

Set similarity join is a fundamental problem. Given two collections of sets, each of which contains a set of elements, set similarity join finds all similar set pairs from the collections. Here, two sets are similar if their similarity value is above a user-defined threshold regarding a given similarity function. Two commonly used similarity functions are Jaccard similarity and Cosine similarity. Set similarity join is adopted in a variety of application domains including data cleaning [19, 25], infomation extraction [24], data integration [26], personalized recommendation [27], community mining [37], entity resolution [40], near duplicate detection [38, 44] and machine learning [46]. In the literature, two categories of set similarity join problems are widely studied, namely, exact set similarity join [19, 25, 45, 37, 44] and approximate set similarity join [36, 30]. In this paper, we focus on the exact set similarity join problem.

State-of-the-art. The existing solutions for exact set similarity join follow a filtering-verification framework [20, 43, 41, 31, 28, 32]. In the filtering phase, a set of candidate pairs is generated and in the verification phase, all candidate pairs are verified and those with similarity above the threshold are reported as answers. The most widely used filter is the prefix filter [20, 43, 41, 31, 32], which is based on the property that if two sets are similar, they must share at least one common element in their prefixes. For a certain set, to compute its answer (i.e., its similar sets) using this property, one only needs to probe the inverted lists for the elements in its prefix, where the inverted list for an element consists of the sets containing the element. In [32], a comprehensive empirical evaluation of set similarity join techniques was conducted on ten real-world datasets from various domains. The experimental results demonstrate that for prefix-filter based algorithms, the verification can be performed very efficiently in practice and the majority of computational costs are spent on the filtering phase. Therefore, the authors conclude that the key to improve the set similarity join algorithms is to reduce the filtering cost.

Motivation. The main cost of the filtering phase for prefix-filter based algorithms is spent on probing inverted lists for elements in the prefix of each set. Existing solutions mainly focus on how to reduce the cost when computing the answer for each individual set. Nevertheless, none of the existing solutions consider the correlations among sets. Intuitively, if we consider some correlated sets together, rather than individually, we can leverage their correlations to share the computational costs and thus speed up the algorithm. Motivated by this, in this paper, we aim to investigate different correlations among sets to explore possible computational cost sharing in exact set similarity join.

Contributions. In order to leverage set correlations in exact set similarity join, the following issues need to be addressed: (1) What correlations can be used to speed up exact set similarity join? and (2) How can we use the set correlations to effectively reduce the overall computational cost? In this paper, we answer these questions and make the following contributions.

(1) Theoretical analysis of existing algorithms. We conduct a comprehensive theoretical analysis of the time and space complexities of existing algorithms for exact set similarity join. We discover that although only linear space is required for existing prefix-filter based algorithms, the time complexity can be $O(n^2 \cdot l_{avg} + V)$, where n is the number of sets in the input, l_{avg} is the average set size, and V is the verification cost, which is small in practice according to [32]. We show that the main reason for the high cost in the filtering phase is that cost sharing is not considered when computing

the answers for different sets.

(2) Skipping technique based on index-level correlations. We first explore the correlations of sets in the inverted list of each element and propose a light-weight inverted index where the sets indexed in the same inverted list are partitioned into different skipping blocks, and the sets in the same skipping block are sorted according to a predefined order. Using these skipping blocks, we show that once a certain set is filtered, when probing a skipping block in the filtering phase, the set along with all the remaining sets in the same skipping block can be skipped thereafter. This property enables us to design an algorithm with a space complexity linear to the input size and a time complexity of $O(n \cdot l_{avg} + V)$, which improves the time complexity of $O(n^2 \cdot l_{avg} + V)$ for the existing algorithms. The improvement is significant because the verification cost V is much smaller than the filtering cost in practice.

(3) Skipping technique based on answer-level correlations. We then investigate the correlations of the answers for different sets. Intuitively, if two sets are similar, their answers should have a large overlap. Given two similar sets where the answer for one set has been computed, we can generate the answer for the other set by adding and removing a small number of sets incrementally without computing the answer from scratch. Based on this intuition, we design a method that can compute the answer for one set based on the answer for another set incrementally. We also derive a cost model to determine whether to apply the answer-level skipping technique when computing the answer of each set to reduce the overall computational cost.

(4) Extensive performance studies. Finally, we conduct extensive experiments using 20 real-world datasets to compare our algorithm with the state-of-the-art algorithms. The datasets are selected from different domains with various data properties. According to the experimental results, our algorithm outperforms all the other algorithms in all tests; and with comparable memory usage, our algorithm can achieve a speedup of more than an order of magnitude compared to the state-of-the-art algorithms.

Outline. The remainder of this paper is organized as follows: Section 2 presents the preliminaries and formally defines the problem. Section 3 shows the existing solutions and provides a thorough theoretical analysis of the time and space complexities for the existing solutions. Section 4 analyzes the drawbacks of existing solutions and discusses possible improvements by leveraging set correlations. Section 5 and Section 6 detail our skipping techniques based on index-level correlations and answer-level correlations respectively. Section 7 evaluates our algorithm through extensive performance studies. Section 8 reviews related work and Section 9 concludes the paper. The proofs of all lemmas and theorems are shown in the appendix.

2 Problem Definition

Suppose $\mathcal{E} = \{e_1, e_2, \ldots, e_m\}$ is a finite universe of elements and a record r is a subset of elements in $\mathcal{E}$, i.e., $r \subseteq \mathcal{E}$. Given two records r_i and r_j, we use a similarity function $\mathsf{Sim}(r_i, r_j) \in [0, 1]$ to measure the similarity between r_i and r_j. Two commonly used similarity functions, namely Jaccard Similarity and Cosine Similarity, are respectively defined as $\mathsf{Jac}(r_i, r_j) = \frac{|r_i \cap r_j|}{|r_i \cup r_j|}$ and $\mathsf{Cos}(r_i, r_j) = \frac{|r_i \cap r_j|}{\sqrt{|r_i| \times |r_j|}}$.

Given two collections of records $\mathcal{R}$ and $\mathcal{S}$, a similarity function Sim, and a threshold τ, the R-S join of $\mathcal{R}$ and $\mathcal{S}$ finds all record pairs $r \in \mathcal{R}$ and $s \in \mathcal{S}$ such that $\mathsf{Sim}(r, s) \geq \tau$. In this paper, for simplicity and without loss of generality, we focus on the self-join,

i.e., $\mathcal{R} = \mathcal{S}$ and we use the Jaccard similarity to describe our techniques. We will show how to support R-S join with $\mathcal{R} \neq \mathcal{S}$ in Section B.1 and how to handle Cosine Similarity in Section B.2. The problem studied in this paper is formally defined as follows:

Problem Statement. Given a database of records $\mathcal{R} = \{r_1, r_2, \ldots, r_n\}$ and a threshold τ, set similarity join computes the set of pairs $\mathcal{A} = \{(r_i, r_j) | r_i \in \mathcal{R}, r_j \in \mathcal{R}, j > i, \mathsf{Jac}(r_i, r_j) \geq \tau\}$.

Definition 2.1: (Answer Set). Given a database of records $\mathcal{R} = \{r_1, r_2, \ldots, r_n\}$ and a threshold τ, for any $1 \leq i \leq n$, the answer set of r_i is defined as:

$$\mathcal{A}(r_i) = \{(r_i, r_j) | r_j \in \mathcal{R}, j > i, \mathsf{Jac}(r_i, r_j) \geq \tau\}. \tag{2.1}$$

Obviously, $\mathcal{A} = \bigcup_{r_i \in \mathcal{R}} \mathcal{A}(r_i)$. □

We use n to denote the number of records in the record database $\mathcal{R}$, and use m to denote the number of distinct elements in $\mathcal{R}$. For each record $r_i \in \mathcal{R}$, the length of r_i is the number of elements in r_i, which is denoted as $|r_i|$. For each record r and a position $k \in [1, |r|]$, we use $r[k]$ to denote the k-th element of r. For each element $e_i \in \mathcal{E}$, we use $I(e_i)$ to denote the set of records in $\mathcal{R}$ that contains e_i, i.e., $I(e_i) = \{r \in \mathcal{R} | e_i \in r\}$. We call $I(e_i)$ the inverted list for element e_i. The number of records in $I(e_i)$ is denoted as $|I(e_i)|$, i.e., $|I(e_i)| = |\{r \in \mathcal{R} | e_i \in r\}|$. The average record length is denoted as l_{avg}, i.e., $l_{avg} = \frac{\sum_{r_i \in \mathcal{R}} |r_i|}{n}$, and the maximum record length is denoted as l_{max}, i.e., $l_{max} = \max_{r_i \in \mathcal{R}} |r_i|$. Following [20, 43, 41, 31, 32], we assume that the records $r \in \mathcal{R}$ are sorted according to the non-decreasing order of $|r|$, i.e., $|r_i| \leq |r_j|$ for any $1 \leq i < j \leq n$, and we assume that the elements $e \in \mathcal{E}$ are sorted according to the non-decreasing order of $|I(e)|$, i.e., $|I(e_i)| \leq |I(e_j)|$ for any $1 \leq i < j \leq m$.

r_1	$\{e_1, e_6, e_7, e_8, e_9, e_{10}, e_{11}, e_{13}, e_{14}, e_{15}, e_{17}, e_{18}\}$
r_2	$\{e_3, e_7, e_8, e_9, e_{10}, e_{12}, e_{13}, e_{14}, e_{15}, e_{16}, e_{17}, e_{18}\}$
r_3	$\{e_2, e_7, e_8, e_9, e_{10}, e_{12}, e_{13}, e_{14}, e_{15}, e_{16}, e_{17}, e_{18}\}$
r_4	$\{e_6, e_7, e_9, e_{10}, e_{11}, e_{12}, e_{13}, e_{14}, e_{15}, e_{16}, e_{17}, e_{18}\}$
r_5	$\{e_5, e_6, e_8, e_9, e_{10}, e_{11}, e_{12}, e_{13}, e_{15}, e_{16}, e_{17}, e_{18}\}$
r_6	$\{e_2, e_3, e_4, e_5, e_8, e_{11}, e_{13}, e_{14}, e_{15}, e_{16}, e_{17}, e_{18}\}$
r_7	$\{e_2, e_3, e_4, e_5, e_9, e_{11}, e_{13}, e_{14}, e_{15}, e_{16}, e_{17}, e_{18}\}$
r_8	$\{e_3, e_4, e_5, e_6, e_7, e_8, e_{10}, e_{11}, e_{12}, e_{14}, e_{16}, e_{17}, e_{18}\}$
r_9	$\{e_3, e_4, e_5, e_6, e_7, e_8, e_{10}, e_{12}, e_{14}, e_{15}, e_{16}, e_{17}, e_{18}\}$
r_{10}	$\{e_2, e_4, e_5, e_6, e_7, e_9, e_{11}, e_{12}, e_{13}, e_{14}, e_{15}, e_{16}, e_{17}, e_{18}\}$

Table 2.1: A Collection of Records

Example 2.1: Consider the dataset in Table 2.1. Assume the given threshold is $\tau = 0.7$. The answer set of r_1 is $\mathcal{A}(r_1) = \{(r_1, r_4)\}$ as only $\mathsf{Jac}(r_1, r_4) = \frac{10}{14} \geq \tau$. For the element e_3, we have $I(e_3) = \{r_2, r_6, r_7, r_8, r_9\}$. □

Frequently used notations are summarized in Table 2.2.

3 Existing Solutions

A naive solution for set similarity join is to use the brute-force method that enumerates $O(n^2)$ pairs of records and calculates the similarity for each of the enumerated pairs. Obviously, this solution is too expensive. To improve the efficiency, the state-of-the-art

Notation	Description
$\mathcal{R} = \{r_1, \ldots, r_n\}$	The collection of records
$\mathcal{E} = \{e_1, \ldots, e_m\}$	The set of elements in $\mathcal{R}$
τ	The similarity threshold
$\mathcal{A}, \mathcal{A}(r_i)$	The join answer and the answer set for record r_i
$I(e_i)$	The inverted list for element e_i
l_{avg}, l_{max}	The average and maximum record lengths in $\mathcal{R}$
$\mathsf{pre}_{t_i}(r_i)$	The prefix of length t_i for record r_i
$\mathsf{pos}_{e_i}(r_j)$	The position of element e_i in r_j
$B_{e_i}(l)$	The skipping block for element e_i and length l
$I^*(e_i)$	The inverted list with skipping blocks for element e_i

Table 2.2: Frequently Used Notations

solutions are all based on a *filtering-verification* framework. In the filtering phase, a set of candidate pairs is generated based on some filtering criteria. In the verification phase, the similarity of each candidate pair is calculated and those pairs with similarity no smaller than the threshold are output. Based on different filtering schemes, existing solutions can be divided into two categories: prefix-filter based methods and partition-filter based methods.

3.1 Prefix-Filter based Methods

Given a record $r \in \mathcal{R}$, a prefix of r, denoted as $\mathsf{pre}_t(r)$, is the first t elements in r. To compute answer set $\mathcal{A}(r_i)$, instead of exploring all records in $\cup_{e \in r_i} I(e)$, prefix-filter based algorithms only need to probe the inverted lists of elements in $\mathsf{pre}_t(r)$ based on the following lemma:

Lemma 3.1: *Given two records r_i and r_j, if $\mathsf{Jac}(r_i, r_j) \geq \tau$, we have $\mathsf{pre}_{t_i}(r_i) \cap \mathsf{pre}_{t_j}(r_j) \neq \emptyset$, where $t_i = \lfloor (1-\tau) \cdot |r_i| \rfloor + 1$ and $t_j = \lfloor (1-\tau) \cdot |r_j| \rfloor + 1$.* □

Based on Lemma 3.1, given a record r_i, in order to compute $\mathcal{A}(r_i)$, we only need to consider those records that are in the inverted lists of the first t_i elements in r_i, i.e.,

$$\mathcal{A}(r_i) \subseteq \{(r_i, r_j) | r_j \in \cup_{e \in \mathsf{pre}_{t_i}(r_i)} I(e), j > i\}. \tag{3.1}$$

Based on Equation (3.1), existing prefix-filter based methods aim to reduce the size of candidates according to the following three categories of filtering techniques [20, 43, 31, 41]:

- **Prefix Filter:** Instead of exploring $I(e)$ for all $e \in r_i$, we only need to probe $I(e)$ for the elements $e \in \mathsf{pre}_{t_i}(r_i)$.
- **Length Filter:** For each $e \in \mathsf{pre}_{t_i}(r_i)$, we do not need to visit all records in $I(e)$. Instead, we only need to visit those r_j ($j > i$) with length no larger than a length threshold $u_i(e)$.
- **Position Filter:** When visiting a certain $r_j \in I(e)$ for element $e \in \mathsf{pre}_{t_i}(r_i)$, the position of e in r_j can be used to filter r_j.

For each record in $\mathcal{R}$, a prefix-filter based algorithm first computes the prefix length by prefix filter. Then it scans the corresponding inverted lists of prefix elements with length filter and position filter to generate candidates in C. All candidate pairs will be verified and output to $\mathcal{A}$ if similarity value is no smaller than the given threshold. A self-explanatory algorithm framework for the prefix-filter based approaches that incorporates the above filters is shown in Algorithm 1. We assume that the inverted list $I(e)$ for each element e has been built, and records in $I(e)$ are sorted in non-decreasing order

Algorithm 1: Prefix-Filter based Framework

Input: A set of records $\mathcal{R}$; a similarity threshold τ
Output: All similar record pairs (r_i, r_j) with $\mathsf{Jac}(r_i, r_j) \geq \tau$

```
1  for each record r_i ∈ R do
       /* Filtering Phase */
2      C(r_i) ← ∅;
3      Calculate the prefix length t_i (prefix filter);
4      for each element e ∈ pre_{t_i}(r_i) do
5          Calculate the length threshold u_i(e) (length filter);
6          for each record r_j ∈ I(e) with j > i do
7              if |r_j| > u_i(e) then
8                  break;
9              if (r_i, r_j) ∉ C(r_i) and r_j is not filtered by the position of e in r_j (position
               filter) then
10                 C(r_i) ← C(r_i) ∪ {(r_i, r_j)};
       /* Verification Phase */
11     A(r_i) ← ∅;
12     for each (r_i, r_j) ∈ C(r_i) do
13         if Jac(r_i, r_j) ≥ τ then
14             A(r_i) ← A(r_i) ∪ {(r_i, r_j)};
15     output A(r_i);
```

of record length. Below, we briefly introduce each of the state-of-the-art prefix-filter based algorithms.

Algorithm AllPairs. We do not need to probe all records in the inverted list $I(e)$ for each element $e \in \mathsf{pre}_{t_i}(r_i)$. In AllPairs [20], a *length filter* is proposed according to the lemma shown below:

Lemma 3.2: *Given two records r_i and r_j, if $\mathsf{Jac}(r_i, r_j) \geq \tau$, we have $|r_j| \leq u_i(e) = \lfloor \frac{|r_i|}{\tau} \rfloor$.* □

Algorithm PPJ/PPJ+. In PPJ [43], a *position filter* is proposed. Given a record r and an element $e_i \in r$, the position of e_i in r, denoted as $\mathsf{pos}_{e_i}(r)$ is defined as:

$$\mathsf{pos}_{e_i}(r) = |\{e_j | e_j \in r, j \leq i\}|. \tag{3.2}$$

Note that $\mathsf{pos}_{e_i}(r)$ can be precomputed together with the inverted list $I(e_i)$ for each $r \in I(e_i)$. With the position information, the *position filter* introduced in PPJ [43] is based on the following lemma:

Lemma 3.3: *Given two records r_i and r_j, if $\mathsf{Jac}(r_i, r_j) \geq \tau$, for any $e \in r_i \cap r_j$, we have $|r_j| - \mathsf{pos}_e(r_j) + |\mathsf{pre}_{\mathsf{pos}_e(r_i)}(r_i) \cap \mathsf{pre}_{\mathsf{pos}_e(r_j)}(r_j)| \geq \lceil \frac{\tau}{1+\tau}(|r_i| + |r_j|) \rceil$.* □

Specifically, if e is the first element in $r_i \cap r_j$, the *position filter* in Lemma 3.3 can be formulated as: $|r_j| - \mathsf{pos}_e(r_j) + 1 \geq \lceil \frac{\tau}{1+\tau}(|r_i| + |r_j|) \rceil$.

In addition, since all records in r are sorted in non-decreasing order of $|r|$, in PPJ [43], when computing $\mathcal{A}(r_i)$, an improved *prefix filter* is proposed according to the lemma shown below:

Lemma 3.4: *Given two records r_i and r_j with $|r_j| \geq |r_i|$, if $\mathsf{Jac}(r_i, r_j) \geq \tau$, we have $\mathsf{pre}_{t_i}(r_i) \cap \mathsf{pre}_{t_j}(r_j) \neq \emptyset$, where $t_i = \lfloor \frac{1-\tau}{1+\tau} \cdot |r_i| \rfloor + 1$ and $t_j = \lfloor (1-\tau) \cdot |r_j| \rfloor + 1$.* □

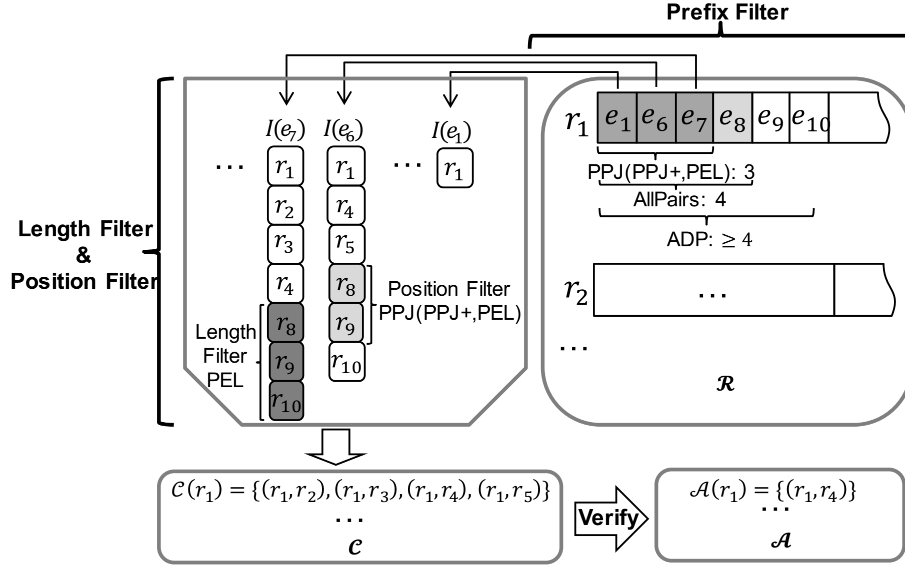


Figure 3.1: Illustration of Prefix-Filter based Methods

Compared to Lemma 3.1, Lemma 3.4 improves the prefix length of r_i from $\lfloor(1-\tau)\cdot|r_i|\rfloor+1$ to $\lfloor\frac{1-\tau}{1+\tau}\cdot|r_i|\rfloor+1$.

In [43], the authors further proposed an algorithm PPJ+ by making use of the suffix information to filter more candidates after applying the basic filters.

Algorithm PEL. In PEL [31], the authors improved the *length filter* proposed in AllPairs [20] using the position information according to the following lemma:

Lemma 3.5: *Given two records r_i and r_j with $\mathsf{Jac}(r_i, r_j) \geq \tau$, for the first element e in $r_i \cap r_j$, i.e., the element $e \in r_i \cap r_j$ with the smallest $\mathsf{pos}_e(r_i)$, we have $|r_j| \leq u_i(e) = \lfloor\frac{|r_i|-(\mathsf{pos}_e(r_i)-1)\cdot(\tau+1)}{\tau}\rfloor$.* □

Based on Lemma 3.5, when exploring the inverted list of the k-th element of r_i, i.e., $I(e)$ for $e = r_i[k]$, we only need to consider those records r_j with length no larger than $\lfloor\frac{|r_i|-(k-1)\cdot(\tau+1)}{\tau}\rfloor$, which improves the bound of $\lfloor\frac{|r_i|}{\tau}\rfloor$ shown in Lemma 3.2.

Algorithm ADP. In [32] the authors proposed an algorithm ADP using an extended prefix filter. The main idea is based on a generalization of Lemma 3.1, which is shown below:

Lemma 3.6: *Given two records r_i and r_j, if $\mathsf{Jac}(r_i, r_j) \geq \tau$, we have $|\mathsf{pre}_{t_{i,l}}(r_i) \cap \mathsf{pre}_{t_{j,l}}(r_j)| \geq l$, where $t_{i,l} = \lfloor(1-\tau)\cdot|r_i|\rfloor + l$ and $t_{j,l} = \lfloor(1-\tau)\cdot|r_j|\rfloor + l$.* □

According to Lemma 3.6, if one probes a longer prefix of r_i, more prefix information is gained and therefore more dissimilar pairs can be pruned. The length of the prefix to be probed is computed adaptively according to a cost function.

Some other heuristics are introduced in [34] and [21]. However, as compared in [32], these heuristics cannot significantly improve the efficiency of the prefix-filter based algorithms.

Result Verification. In [32], Mann et al. introduced a new verification algorithm to efficiently verify whether the similarity of two records reaches a certain threshold τ. The verification algorithm uses a merge-like method and adopts an early termination condi-

tion based on Lemma 3.3. As stated in [32], based on the early termination condition, verification is surprisingly fast in practice.

Summary and Comparison. To summarize, in order to compute $\mathcal{A}(r_i)$, the most effective filters are shown below:

- **Prefix Filter:** We only need to probe $I(e)$ for $e \in \mathsf{pre}_{t_i}(r_i)$, where

$$t_i = \lfloor \frac{1-\tau}{1+\tau} \cdot |r_i| \rfloor + 1 \tag{3.3}$$

- **Length Filter:** For each $e \in \mathsf{pre}_{t_i}(r_i)$, we only need to visit those r_j $(j > i)$ with $|r_j| \leq u_i(e)$, where

$$u_i(e) = \lfloor \frac{|r_i| - (\mathsf{pos}_e(r_i) - 1) \cdot (\tau + 1)}{\tau} \rfloor \tag{3.4}$$

- **Position Filter:** When visiting a new record $r_j \in I(e)$ for $e \in \mathsf{pre}_{t_i}(r_i)$, (r_i, r_j) is a candidate pair if

$$|r_j| - \mathsf{pos}_e(r_j) + 1 \geq \lceil \frac{\tau}{1+\tau}(|r_i| + |r_j|) \rceil \tag{3.5}$$

In [32], a comprehensive empirical evaluation was conducted to compare the prefix-filter based methods on ten different real-world datasets from various domains. The experimental results demonstrated that after applying the above-mentioned filters in the filtering phase and the fast merge-like verification algorithm in the verification phase, the dominant cost is spent on the filtering phase. Note that the algorithms PPJ+ and ADP increase the cost of the filtering phase using sophisticated filters to further reduce the number of candidate pairs. However, as shown in [32], the sophisticated filters used in the algorithms PPJ+ and ADP do not pay off because of the fast verification. Therefore, the authors in [32] conclude that *the key to improving the set similarity join algorithm is to improve the efficiency of the filtering phase.*

Time/Space Complexity. To analyze the time complexity of the prefix-filter based algorithms that follow the framework of Algorithm 1, we divide the time cost into the following two parts:

- **Filtering Cost:** The total time spent on the filtering phase to generate all candidate pairs, i.e., the total time spent on line 2-10 of Algorithm 1. We denote the filtering cost as F.
- **Verification Cost:** The total time spent on the verification phase to compute the answer sets, i.e., the total time spent on line 11-14 of Algorithm 1. We denote the verification cost as V.

The total time complexity of Algorithm 1 is $O(F + V)$. Here, the verification cost V is the time to verify the pairs after applying the prefix filter (Equation (3.3)), length filter (Equation (3.4)), and position filter (Equation (3.5)). As shown in [32], the verification cost is usually much smaller than the filtering cost. The following theorem shows that the filtering cost of Algorithm 1 can be quadratic to the number of records in $\mathcal{R}$ in the worst case.

Theorem 3.1: *The time complexity of Algorithm 1 is $O(n^2 \cdot l_{avg} + V)$.* □

Regarding the space complexity of Algorithm 1, the following lemma can be easily obtained, which shows that the memory usage of Algorithm 1 is linear to the size of the input:

Theorem 3.2: *The space complexity of Algorithm 1 is $O(n \cdot l_{avg})$.* □

Example 3.1: We use an example (Figure 3.1) to illustrate the prefix-filter based methods. Assume the given threshold is $\tau = 0.7$. Consider the first record r_1 from Table 2.1.

(1) For the prefix filter, the prefix length of AllPairs is $\lfloor (1-\tau)\cdot|r_1|\rfloor + 1 = 4$. PPJ (PPJ+, PEL) reduces the prefix length to $\lfloor \frac{1-\tau}{1+\tau}\cdot|r_1|\rfloor + 1 = 3$. The prefix length of ADP is no smaller than 4.

(2) For the length filter, the length threshold of AllPairs (PPJ, PPJ+, ADP) for r_1 is $u_1(e) = \lfloor\frac{12}{0.7}\rfloor = 17$. No record will be filtered. While for PEL, the length threshold decreases as $u_1(e) = 17, 14, 12$ for $e = e_1, e_6, e_7$ respectively. When probing $I(e_7)$, r_8, r_9 and r_{10} will be filtered since their record length is larger than $u_1(e_7) = 12$.

(3) For the position filter, PPJ (PPJ+, PEL) utilizes the position of each element to prune records. For example, when probing $I(e_6)$, r_8 will be removed as $|r_8| - \mathsf{pos}_{e_6}(r_8) + 1 = 10 < \lceil\frac{\tau}{1+\tau}(|r_1| + |r_8|)\rceil = 11$. However, r_{10} could not be filtered since $|r_{10}| - \mathsf{pos}_{e_6}(r_{10}) + 1 = 11 = \lceil\frac{\tau}{1+\tau}(|r_1| + |r_{10}|)\rceil = 11$. The position information is not used by AllPairs and ADP. □

3.2 Partition-Filter based Methods

In [28], a partition-filter based algorithm PTJ was proposed. In PTJ, each record in $\mathcal{R}$ is partitioned into several disjoint sub-records such that two records are similar only if they share a common sub-record. Specifically, given an integer p, a partition scheme $\mathsf{scheme}(\mathcal{E}, p)$ is a partition of $\mathcal{E}$ which divides the element universe $\mathcal{E}$ into p subsets $\mathcal{E}^1$, $\mathcal{E}^2$, ..., $\mathcal{E}^p$, where $\cup_{i=1}^{p}\mathcal{E}^i = \mathcal{E}$ and $\mathcal{E}^i \cap \mathcal{E}^j = \emptyset$ for any $1 \le i < j \le p$. Under a partition scheme $\mathsf{scheme}(\mathcal{E}, p)$, a record r can be partitioned into p sub-records r^1, r^2, ..., r^p where $r^i = r \cap \mathcal{E}^i$ for any $1 \le i \le p$. The partition-filter based methods are designed based on the following lemma:

Lemma 3.7: *For any two records r_i and r_j, suppose we use the same partition scheme $\mathsf{scheme}(\mathcal{E}, p_i)$ to partition r_i and r_j into $p_i = \lfloor\frac{1-\tau}{\tau}|r_i|\rfloor + 1$ sub-records, if $\mathsf{Jac}(r_i, r_j) \ge \tau$, then there exist at least one integer $k \in [1, p_i]$ with $r_i^k = r_j^k$.* □

The PTJ algorithm is designed following a filtering-verification framework. The filtering phase consists of two steps:

- In the first step, for each record $r_i \in \mathcal{R}$, we generate all its sub-records under the partition scheme $\mathsf{scheme}(\mathcal{E}, p_i)$ and index them using the inverted list.
- In the second step, for each record $r_j \in \mathcal{R}$, we generate all the sub-records of r_j under the partition schemes $\mathsf{scheme}(\mathcal{E}, \lfloor\frac{1-\tau}{\tau}l\rfloor + 1)$ for any $\tau|r_j| \le l \le |r_j|$. For each such sub-record r_j^k in each of the generated partition schemes, we probe all records r_i in the inverted list of r_j^k and consider (r_i, r_j) as a candidate pair.

In the verification phase, all candidate pairs are verified and those pairs (r_i, r_j) with $\mathsf{Jac}(r_i, r_j) \ge \tau$ are output.

To reduce the number of probed pairs in the second step of the filtering phase, PTJ further indexes all the 1-deletion neighborhoods (the subset by eliminating any element from a sub-record) of sub-records generated in the first step of the filtering phase, and uses a greedy algorithm to allocate a mixture of the sub-records and their 1-deletion neighborhoods and probe their inverted lists. The time and space complexities of PTJ are shown below:

Theorem 3.3: *The time complexity of PTJ is $O(n \cdot l_{max}^2 + V')$, where V' is the time cost to verify all candidate pairs.* □

Theorem 3.4: *The space complexity of* PTJ *is* $O(n \cdot l_{max})$. □

In [28], it is shown that PTJ can effectively reduce the number of candidates. Nevertheless, PTJ requires a cost of $O(n \cdot l_{max}^2)$ to partition the records based on different partition schemes and build the inverted lists of all sub-records and their 1-deletion neighborhoods, which is expensive. As shown in the experiments (Section 7), with the fast verification algorithm introduced in [32], PTJ can hardly outperform the prefix-filter based methods. Regarding the space complexity of the partition based approach PTJ, comparing Theorem 3.3 to Theorem 3.1, PTJ increases the space cost from $O(n \cdot l_{avg})$ to $O(n \cdot l_{max})$ to store the inverted lists of all sub-records and their 1-deletion neighborhoods. As shown in the experiments (Section 7), the space used by PTJ is much larger than that used by the prefix-filter based algorithms.

Example 3.2: We utilize the dataset in Table 2.1 to simply explain the basic idea of PTJ. Suppose $\tau = 0.7$. The join is processed in the following two steps:

In the first step, we build the index. Consider the records with size 12. Partition scheme $\mathsf{scheme}(\mathcal{E}, p_i)$ where $p_i = \lfloor \frac{1-\tau}{\tau}|r_i| \rfloor + 1 = 6$ is adopted. Here, the element universe $\mathcal{E}$ is evenly partitioned into 6 sub-sets $\mathcal{E}^1 = \{e_1, e_2, e_3\}$, $\mathcal{E}^2 = \{e_4, e_5, e_6\}$, $\mathcal{E}^3 = \{e_7, e_8, e_9\}$, $\mathcal{E}^4 = \{e_{10}, e_{11}, e_{12}\}$, $\mathcal{E}^5 = \{e_{13}, e_{14}, e_{15}\}$, and $\mathcal{E}^6 = \{e_{16}, e_{17}, e_{18}\}$. Each record with size 12 is partitioned into sub-records based on the partition scheme $\mathsf{scheme}(\mathcal{E}, 6)$. For example, for r_1, we have $r_1^1 = \{e_1\}$, $r_1^2 = \{e_6\}$, $r_1^3 = \{e_7, e_8, e_9\}$, $r_1^4 = \{e_{10}, e_{11}\}$, $r_1^5 = \{e_{13}, e_{14}, e_{15}\}$, and $r_1^6 = \{e_{17}, e_{18}\}$, and thus we add r_1 to the corresponding inverted lists of these sub-records. Other records are partitioned and processed similarly to build the inverted index.

In the second step, we find similar pairs. Consider the record r_8. Since $|r_8| = 13$ and $\tau|r_8| = 9.1$, r_8 should be partitioned using scheme $\mathsf{scheme}(\mathcal{E}, \lfloor \frac{1-\tau}{\tau} l \rfloor + 1)$ where $l = 10, 11, 12, 13$. Take $l = 12$ as an example, we partition r_8 into 6 sub-records $r_8^1 = \{e_3\}$, $r_8^2 = \{e_4, e_5, e_6\}$, $r_8^3 = \{e_7, e_8\}$, $r_8^4 = \{e_{10}, e_{11}, e_{12}\}$, $r_8^5 = \{e_{14}\}$, and $r_8^6 = \{e_{16}, e_{17}, e_{18}\}$ and probe the corresponding inverted index. In this way, we get 6 candidate pairs $(r_2, r_8), (r_3, r_8), (r_4, r_8), (r_5, r_8), (r_6, r_8)$, and (r_7, r_8). When calculating the real similarity of the pairs, none are reported as answers. □

4 Algorithm Analysis

As shown in Section 3, the time complexities of the existing prefix-filter based algorithms and partition-filter based algorithms are $O(n^2 \cdot l_{avg} + V)$ and $O(n \cdot l_{max}^2 + V')$ respectively. Here, the dominant costs are spent on the filtering phase, which are $O(n^2 \cdot l_{avg})$ and $O(n \cdot l_{max}^2)$ for prefix-filter and partition-filter based algorithms respectively. As indicated in [32], it does not pay off to use more sophisticated filters in the filtering phase to further reduce the number of candidates to be verified in the verification phase. Therefore, in this paper, we do not intend to use more filters to reduce the number of candidates. Instead *we aim to improve the efficiency of the filtering phase without increasing the verification cost.*

Next, we investigate possible improvement spaces for both the prefix-filter based methods and partition-filter based methods.

- For the prefix-filter based methods, a light-weight linear-sized inverted index is used to index all elements in the records. The major cost is spent on probing the inverted list of each element $e \in \mathsf{pre}_{t_i}(r_i)$ when computing the candidate set $C(r_i)$ for each record $r_i \in \mathcal{R}$ in the filtering phase. We observe that not all probed records will be

added into the candidate set $C(r_i)$ because the probed records need to satisfy all the conditions in the three filters. Therefore, there are improvement spaces to further reduce the number of probed records to improve the filtering efficiency for prefix-filter based methods.

- For the partition-filter based methods, a complex inverted index has to be built on the sub-records and their 1-deletion neighborhoods. The major cost of the algorithm is spent on partitioning the records according to different partition schemes and building the complex inverted index. When probing the inverted index, all the probed records are added to the candidate set and verified in the verification phase. Therefore, the number of probed records can hardly be reduced to improve the filtering efficiency of the partition-filter based methods.

Based on the above analysis, in this paper, we follow the *prefix-filter based framework* and design algorithms to improve the filtering cost by reducing the number of probed records.

We observe that in the existing prefix-filter based algorithms, the records are processed independently, and thus cost sharing is never considered when computing the answer sets for different records. Specifically, first, when computing the candidate set $C(r_i)$ for a certain record r_i, the records indexed in the inverted lists of elements in the prefix of r_i are probed independently; and second, the answer sets $\mathcal{A}(r_i)$ for all records r_i are computed independently. To address this problem, in this paper, we will explore the correlations among records and seek for possible computational cost sharing when processing correlated records. Specifically, we aim to develop effective skipping techniques to reduce the number of index probes by considering the following two levels of record correlations:

- **Index-Level Skipping:** For each element e, we partition all records in the inverted list $I(e)$ into different skipping blocks. When processing the index $I(e)$ for element e in record r_i, we first probe the skipping blocks in $I(e)$. For each skipping block, we probe the records in the skipping block according to a predefined order. We can guarantee that once a record fails a certain condition, the record along with all the remaining records in the block can be skipped thereafter when probing $I(e)$. We will show that with the index-level skipping technique, we can improve the time complexity of the algorithm from $O(n^2 \cdot l_{avg} + V)$ to $O(n \cdot l_{avg} + V)$.
- **Answer-Level Skipping:** The answer-level skipping technique is based on the intuition that if two records r_i and r_j are similar, their result sets $\mathcal{A}(r_i)$ and $\mathcal{A}(r_j)$ also tend to be similar. Based on such an observation, suppose $\mathcal{A}(r_i)$ is computed, we do not need to compute $\mathcal{A}(r_j)$ from scratch. Instead, we can compute $\mathcal{A}(r_j)$ based on $\mathcal{A}(r_i)$ by adding a small number of new similar pairs and removing a small number of existing dissimilar pairs. In this way, the computational cost can be significantly reduced. We will show our answer-level skipping algorithm with correctness guarantee, and a cost model to determine when to apply the answer-level skipping technique.

In the following, we will introduce the index-level and answer-level skipping techniques in Section 5 and Section 6 respectively.

5 Index-Level Skipping

Motivation. Before introducing the details of the index-level skipping technique, we first look at a motivating example below:

Example 5.1: Suppose we compute $\mathcal{A}(r_1)$ for the dataset shown in Table 2.1. The prefix $\mathsf{pre}_{t_1}(r_1)$ contains e_1, e_6, and e_7. Assume that we are now probing the inverted list $I(e_6) = \{r_1, r_4, r_5, r_8, r_9, r_{10}\}$. The number of probed entries is 5 according to Algorithm 1. However, it is worth noticing that, for any record $r_j \in \{r_8, r_9\}$, the record length and element position are the same with $|r_j| = 13$ and $\mathsf{pos}_{e_6}(r_j) = 4$. They all fail the position condition as $|r_j| - \mathsf{pos}_{e_6}(r_j) + 1 = 10 < \lceil \frac{\tau}{1+\tau}(|r_1| + |r_j|) \rceil = 11$. In fact, whenever we probe the inverted list $I(e_6)$, once $r_j = r_8$ fails the position condition, we can skip the position condition checking for $r_j = r_9$. □

This example demonstrates that by considering the relationships of the records in the inverted list of a certain element, the cost of probing the inverted list can potentially be reduced. Note that we have three types of filters: prefix filter, length filter, and position filter. The key to reduce the cost of probing the inverted lists is to reduce the number of failed attempts (the record probes that fail the filter condition) for each filter. Below, we analyze each of the three types of the filters individually:

- For the prefix filter (line 3-4 of Algorithm 1), when probing the inverted list $I(e)$ for $e \in r_i$, we guarantee that e is selected from the prefix $\mathsf{pre}_{t_i}(r_i)$ of r_i. In other words, the prefix condition in Equation (3.3) will never fail.
- For the length filter (line 5-8 of Algorithm 1), when probing the inverted list $I(e)$ for $e \in r_i$, we need to prune those records r_j with $|r_j| > u_i(e)$ (Equation (3.4)). By sorting the records in each inverted list $I(e)$ according to the non-decreasing order of length, such a condition can be satisfied without failed attempts.
- For the position filter (line 9 of Algorithm 1), in order to avoid failed attempts, we need to find an appropriate order of records in the inverted list $I(e)$ for $e \in r_i$. However, this is difficult since when probing $I(e)$, we need to consider three factors $|r_i|$, $|r_j|$ and $\mathsf{pos}_e(r_j)$ at the same time as shown in Equation (3.5).

Based on the above analysis, the key to avoid failed attempts is to avoid the failure of the position condition in Equation (3.5). In the remainder of the paper, we use *failed attempt* to denote the attempt that fails the position condition in Equation (3.5).

The Rationale. In the position condition in Equation (3.5) used in the position filter, there are three factors, namely, $|r_i|$, $|r_j|$, and $\mathsf{pos}_e(r_j)$. For ease of analysis, we rewrite Equation (3.5) as:

$$(1 + \tau) \cdot \mathsf{pos}_e(r_j) + \tau \cdot |r_i| \leq |r_j| + \tau + 1 \quad (5.1)$$

It is easy to see that if a certain pair (r_i, r_j) fails the position condition in Equation (5.1) when probing the inverted list $I(e)$, all the pairs $(r_{i'}, r_{j'})$ with $|r_{j'}| = |r_j|$, $|r_{i'}| \geq |r_i|$, and $\mathsf{pos}_e(r_{j'}) \geq \mathsf{pos}_e(r_j)$ will also fail the position condition in Equation (5.1) when probing the inverted list $I(e)$. Making use of such a domination relationship can possibly reduce the number of failed attempts when probing the inverted list $I(e)$. Recall that the records r_j in the inverted list $I(e)$ are sorted by non-decreasing order of $|r_j|$. We need to keep this order since the length filter relies on it. To use the domination relationship, we further group all the records r_j with the same $|r_j|$ in $I(e)$, and construct a new inverted list $I^*(e)$ which is defined as:

Definition 5.1: (Inverted list $I^*(e)$). For each element e, the inverted list $I^*(e)$ consists of all entries $(r_j, \mathsf{pos}_e(r_j))$ such that $e \in r_j$. The entries in $I^*(e)$ are grouped into *skipping blocks*. Each skipping block $B_e(l)$ consists of all entries $(r_j, \mathsf{pos}_e(r_j))$ in $I^*(e)$ with $|r_j| = l$. Skipping blocks $B_e(l)$ in $I^*(e)$ are sorted in non-decreasing order of l, and entries $((r_j), \mathsf{pos}_e(r_j))$ in each skipping block are sorted in non-decreasing order of $\mathsf{pos}_e(r_j)$. □

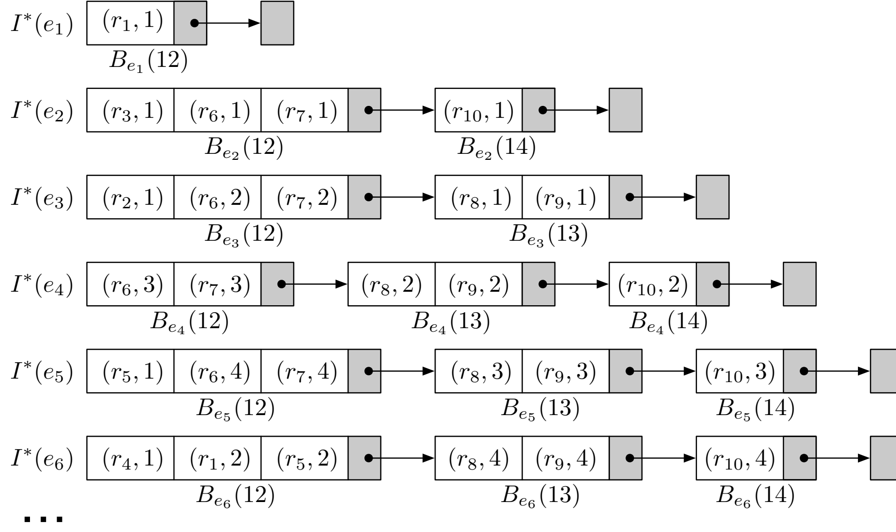


Figure 5.1: Illustration of inverted list $I^*(e)$

Index Construction. From the above definition, we can see that the main difference between $I^*(e)$ and $I(e)$ is the sorting order. That is, each record is added to $I^*(e)$ based on record size and element position. Therefore, the time complexity of constructing $I^*(e)$ and $I(e)$ are the same. So are the space complexity. Note that the index can be built offline since it is independent of any user given parameter. Below, we give an example to illustrate the inverted index $I^*(e)$ with skipping blocks.

Example 5.2: The inverted lists $I^*(e)$ of elements e in the dataset of Table 2.1 are illustrated in Figure 5.1. Take element e_3 as an example. The records that contain e_3 are r_2, r_6, r_7, r_8 and r_9. Records r_2, r_6, and r_7 are grouped into skipping block $B_{e_3}(12)$ since their lengths are all 12. Records r_8 and r_9 are grouped into $B_{e_3}(13)$. Inside each skipping block, the records are sorted in non-decreasing order of the position of e_3. For example in $B_{e_3}(12)$, the entries are $(r_2, 1)$, $(r_6, 2)$, and $(r_7, 2)$, with the non-decreasing order of positions 1, 2, and 2 respectively. □

Next, we show how the skipping blocks are used to reduce the number of failed attempts by skipping useless index probes. The key is to make use of the non-decreasing order of $\mathsf{pos}_e(r_j)$ for all entries $(r_j, \mathsf{pos}_e(r_j))$ in the same skipping block. Based on such an order, we can derive the following lemma:

Lemma 5.1: *When computing $C(r_i)$ by probing $B_e(l)$, once an entry $(r_j, \mathsf{pos}_e(r_j))$ fails the position condition in Equation (5.1), we have:*

(1) All un-probed entries in $B_e(l)$ will fail Equation (5.1) when computing $C(r_i)$; and

(2) All un-probed entries in $B_e(l)$ along with $(r_j, \mathsf{pos}_e(r_j))$ will fail Equation (5.1) when computing $C(r_{i'})$ for any $i' > i$. □

Based on Lemma 5.1, we know that once an entry fails the position condition in Equation (5.1), we have: (1) the un-probed entries in the same skipping block $B_e(l)$ can be skipped for the current iteration; and (2) the entry along with all the un-probed entries in the same skipping block $B_e(l)$ can be skipped for all future iterations when probing the same inverted list $I^*(e)$. Below, we show our algorithm for index-level

Algorithm 2: Index-Level Skipping

Input: A set of records $\mathcal{R}$; a similarity threshold τ
Output: All similar record pairs (r_i, r_j) with $\mathsf{Jac}(r_i, r_j) \geq \tau$

```
1  for each record r_i ∈ R do
2      C(r_i) ← ∅;
3      Calculate the prefix length t_i using Equation (3.3);
4      for each element e ∈ pre_{t_i}(r_i) do
5          Calculate the length threshold u_i(e) using Equation (3.4);
6          for each skipping block B_e(l) in I*(e) do
7              if l > u_i(e) then
8                  break;
9              for each (r_j, pos_e(r_j)) ∈ B_e(l) before B_e(l).pos do
10                 if (r_j, pos_e(r_j)) fails Equation (5.1) then
11                     B_e(l).pos ← position of (r_j, pos_e(r_j)) in B_e(l);
12                     break;
13                 if (r_i, r_j) ∉ C(r_i) then
14                     C(r_i) ← C(r_i) ∪ {(r_i, r_j)};
15             if B_e(l).pos = 1 then
16                 Remove B_e(l) from I*(e);
17     Line 11-15 of Algorithm 1;
```

skipping.

The Algorithm. To make use of Lemma 5.1, in each skipping block $B_e(l)$, we define $B_e(l).\mathsf{pos}$ to be the position of the first entry that fails the position condition in Equation (5.1). Initially $B_e(l).\mathsf{pos}$ is set to be $|B_e(l)| + 1$. Our algorithm for index-level skipping is shown in Algorithm 2. To compute the answer set for each $r_i \in \mathcal{R}$, we use the prefix filter and length filter the same as those used in Algorithm 1 (line 3-5). For each element $e \in \mathsf{pre}_{t_i}(r_i)$, we probe all the skipping blocks $B_e(l)$ in the inverted list $I^*(e)$ according to the non-decreasing order of l (line 6). Once l exceeds the length threshold $u_i(e)$, we skip all the remaining skipping blocks (line 7-8). Otherwise, we probe the first $B_e(l).\mathsf{pos} - 1$ entries in the current skipping block $B_e(l)$. For each entry $(r_j, \mathsf{pos}_e(r_j))$ (line 9), if the position condition Equation (5.1) fails, we set $B_e(l).\mathsf{pos}$ as the position of $(r_j, \mathsf{pos}_e(r_j))$ in $B_e(l)$ and skip all the remaining entries in the skipping block (line 10-12); Otherwise, we add (r_i, r_j) as a candidate pair (line 13-14). Once all entries in the skipping block $B_e(l)$ can be skipped, i.e., $B_e(l).\mathsf{pos} = 1$, we remove $B_e(l)$ from $I^*(e)$ (line 15-16). The verification phase is the same as that used in Algorithm 1. Note that for simplicity, we do not add the constraint $i < j$ for each candidate pair (r_i, r_j) generated in Algorithm 2. To ensure such a constraint, when probing an entry $(r_j, \mathsf{pos}_e(r_j)) \in B_e(l)$, if $j \leq i$, we just need to remove the entry from $B_e(l)$ since it will never be used afterwards. The correctness of Algorithm 2 is guaranteed by Lemma 5.1.

Example 5.3: Consider the dataset in Table 2.1. Suppose the threshold $\tau = 0.7$ and we have built the inverted index with skipping blocks for all elements in Table 2.1. We use record r_1 as an example to illustrate Algorithm 2. The prefix of r_1 includes e_1, e_6 and e_7. Their inverted lists are $I^*(e_1) = \{B_{e_1}(12) : \{(r_1, 1)\}\}$, $I^*(e_6) = \{B_{e_6}(12) : \{(r_4, 1), (r_1, 2), (r_5, 2)\}, B_{e_6}(13) : \{(r_8, 4), (r_9, 4)\}, B_{e_6}(14) : \{(r_{10}, 4)\}\}$, and $I^*(e_7) = \{B_{e_7}(12) : \{(r_2, 2), (r_3, 2), (r_4, 2), (r_1, 3)\}, B_{e_7}(13) : \{(r_8, 5), (r_9, 5)\}, B_{e_7}(14) : \{(r_{10}, 5)\}\}$ respectively. For element e_1, since $I^*(e_1)$ only contains record r_1, we probe the inverted list $I^*(e_6)$

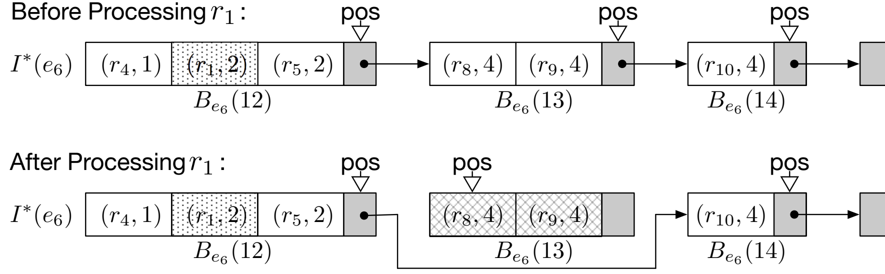


Figure 5.2: Illustration of Algorithm 2

for element e_6. The length threshold $u_1(e_6) = 14$ where all records in the dataset will pass the length filter. Then we access each skipping block in inverted list $I^*(e_6)$. For skipping block $B_{e_6}(12)$, all entries pass the position filter. So we add $\{(r_1, r_4), (r_1, r_5)\}$ into $C(r_1)$. For skipping block $B_{e_6}(13)$, $(r_8, 4)$ fails the position filter. This results in $B_{e_6}(13).pos = 1$ (as shown in Figure 5.2). Thus, the whole block $B_{e_6}(13)$ is removed from $I^*(e_6)$. For skipping block $B_{e_6}(14)$, $(r_{10}, 4)$ passes the position filter and thus $\{(r_1, r_{10})\}$ is added into $C(r_1)$. After similarly processing skipping blocks in $I^*(e_7)$, we get the candidate set $C(r_1) = \{(r_1, r_2), (r_1, r_3), (r_1, r_4), (r_1, r_5), (r_1, r_{10})\}$. □

Cost Saving. It is easy to see that Algorithm 2 and Algorithm 1 have the same verification cost since they generate the same set of candidates. Therefore, the major cost saving of Algorithm 2 is the reduction of the number of failed attempts when probing the inverted lists in the filtering phase. To illustrate the cost saving of our algorithm, we define the failure space as follows:

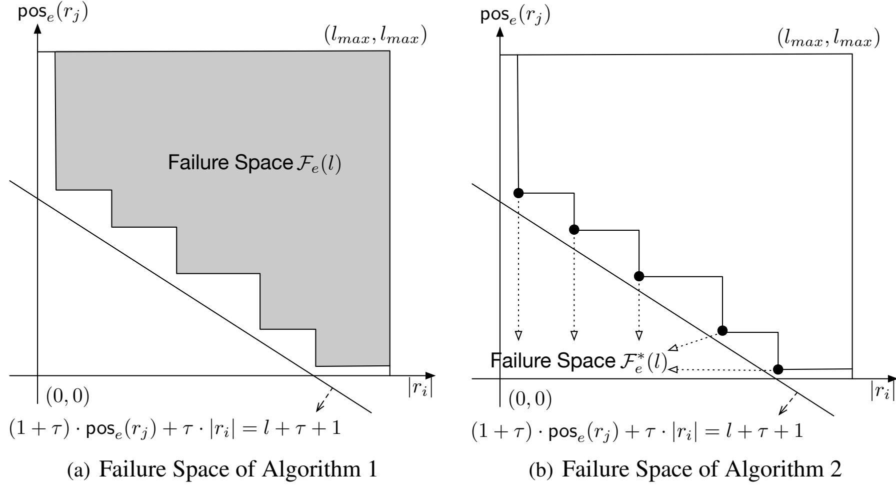
Definition 5.2: (Failure Space). For each element e and $1 \leq l \leq l_{max}$, the failure space w.r.t. e and l is the set of points $(|r_i|, \mathsf{pos}_e(r_j))$ with $|r_j| = l$ that fails the position condition in Equation (5.1) when applying a prefix-filter based algorithm. For each element e and $1 \leq l \leq l_{max}$, we denote the failure space of Algorithm 1 and Algorithm 2 w.r.t. e and l as $\mathcal{F}_e(l)$ and $\mathcal{F}^*_e(l)$ respectively. □

For any two points $(|r_i|, \mathsf{pos}_e(r_j))$ and $(|r_{i'}|, \mathsf{pos}_e(r_{j'}))$, $(|r_{i'}|, \mathsf{pos}_e(r_{j'}))$ is dominated by $(|r_i|, \mathsf{pos}_e(r_j))$ if $|r_{i'}| \geq |r_i|$, $\mathsf{pos}_e(r_{j'}) \geq \mathsf{pos}_e(r_j)$, and $(|r_{i'}|, \mathsf{pos}_e(r_{j'})) \neq (|r_i|, \mathsf{pos}_e(r_j))$. Given a set of points $\mathcal{F}$, the skyline of $\mathcal{F}$, denoted as $\mathsf{skyline}(\mathcal{F})$, is the set of points that are not dominated by any other points in $\mathcal{F}$. We have:

Lemma 5.2: *For any element e and $1 \leq l \leq l_{max}$, we have $\mathcal{F}^*_e(l) = \mathsf{skyline}(\mathcal{F}_e(l))$.* □

Figure 5.3 (a) and Figure 5.3 (b) illustrate the failure spaces of Algorithm 1 and Algorithm 2 respectively for each element e and $1 \leq l \leq l_{max}$. In each figure, the x axis is $|r_i|$, and the y axis is $\mathsf{pos}_e(r_j)$. The total space is $[0, l_{max}] \times [0, l_{max}]$. The failure space $\mathcal{F}_e(l)$ is the grey area in the upper right part of the line $(1+\tau)\cdot\mathsf{pos}_e(r_j)+\tau\cdot|r_i| = l+\tau+1$ as shown in Figure 5.3 (a); and the failure space $\mathcal{F}^*_e(l)$ only consists of the skyline points of $\mathcal{F}_e(l)$ as shown in Figure 5.3 (b). Compared to Algorithm 1, Algorithm 2 reduces the two-dimensional failure space $\mathcal{F}_e(l)$ to the one-dimensional skyline space $\mathcal{F}^*_e(l)$. We use the following example to further illustrate the cost saving.

Example 5.4: Consider the dataset in Table 2.1 and the threshold $\tau = 0.7$. The number of failed attempts for Algorithm 1 is 22 while Algorithm 2 only produces 5 failed attempts. This represents a 77% reduction in failed attempts comparing to Algorithm 1 and shows that our index-level skipping technique is effective on reducing the number



(a) Failure Space of Algorithm 1

(b) Failure Space of Algorithm 2

Figure 5.3: Illustration of Failure Spaces $\mathcal{F}_e(l)$ and $\mathcal{F}_e^*(l)$

of failed attempts. □

Complexity Analysis. The cost saving on the failed attempts results in the reduction of the time complexity of the algorithm. We can derive the following two theorems:

Theorem 5.1: *The time complexity of Algorithm 2 is* $O(n \cdot l_{avg} + V)$. □

Theorem 5.2: *The space complexity of Algorithm 2 is* $O(n \cdot l_{avg})$. □

Comparing theorems 5.1 and 5.2 to theorems 3.1 and 3.2, our Algorithm 2 improves the time complexity of the filtering phase from $O(n^2 \cdot l_{avg})$ to $O(n \cdot l_{avg})$ with the same space complexity which is linear to the size of the input. Comparing theorems 5.1 and 5.2 to theorems 3.3 and 3.4, our Algorithm 2 improves the time complexity of the filtering phase from $O(n \cdot l_{max}^2)$ to $O(n \cdot l_{avg})$, and improves the space complexity from $O(n \cdot l_{max})$ to $O(n \cdot l_{avg})$.

6 Answer-Level Skipping

Motivation. In this section, we discuss the technique for answer-level skipping. Recall that for each record $r_i \in \mathcal{R}$, Algorithm 1 calculates the answer set $\mathcal{A}(r_i)$ for r_i from scratch without considering the relationships among records. Note that after computing $\mathcal{A}(r_i)$ for record r_i, each record $r_{i'}$ in $\mathcal{A}(r_i)$ has a high similarity with record r_i. Therefore, for each record r_j, if r_j is similar to r_i, it is highly possible that r_j is also similar to $r_{i'}$. In other words, it is highly possible that the answer set $\mathcal{A}(r_i)$ is similar to the answer set $\mathcal{A}(r_{i'})$. Based on this observation, if we can compute $\mathcal{A}(r_{i'})$ incrementally based on $\mathcal{A}(r_i)$, we can avoid computing $\mathcal{A}(r_{i'})$ from scratch and therefore reduce the computational cost. Next, we show how to incrementally generate $\mathcal{A}(r_{i'})$ based on $\mathcal{A}(r_i)$.

The Rationale. Suppose $\mathcal{A}(r_i)$ is computed for a certain record r_i. For a record $r_{i'}$ that is similar to r_i, to compute $\mathcal{A}(r_{i'})$, we need to consider the difference between r_i and $r_{i'}$. Specifically, we show that it is sufficient to only consider the elements in $r_{i'} \setminus r_i$ to find all new candidates that need to be added to the current answer set based on the following lemma:

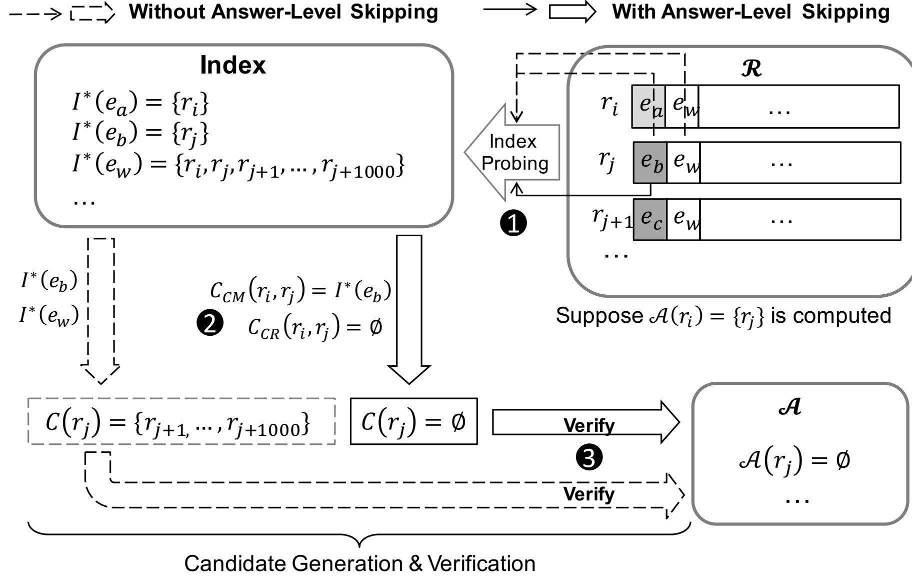


Figure 6.1: Illustration of Cost Saving by Answer-Level Skipping

Lemma 6.1: *Given a record r_i and the result set $\mathcal{A}(r_i)$, for any record $r_{i'}$ with $i' > i$, we have:*

$$\mathcal{A}(r_{i'}) \subseteq C_{CM}(r_i, r_{i'}) \cup C_{CR}(r_i, r_{i'}), \tag{6.1}$$

where
$$C_{CM}(r_i, r_{i'}) = \{(r_{i'}, r_j) | r_j \in \cup_{e \in r_{i'} \setminus r_i} I(e)\} \tag{6.2}$$

and
$$C_{CR}(r_i, r_{i'}) = \{(r_{i'}, r_j) | (r_i, r_j) \in \mathcal{A}(r_i)\}. \tag{6.3}$$

Here $C_{CM}(r_i, r_{i'})$ and $C_{CR}(r_i, r_{i'})$ are called the complementary candidates and current candidates of $r_{i'}$ w.r.t. r_i respectively. □

According to Lemma 6.1, to compute $\mathcal{A}(r_{i'})$ based on $\mathcal{A}(r_i)$, we can simply compute $C_{CM}(r_i, r_{i'})$ and $C_{CR}(r_i, r_{i'})$, and then verify all the candidates in $C_{CM}(r_i, r_{i'}) \cup C_{CR}(r_i, r_{i'})$. Figure 6.1 shows a simple example to illustrate the cost saving by using answer-level skipping. Suppose $\mathcal{A}(r_i) = \{r_j\}$ has been computed, to compute $\mathcal{A}(r_j)$, we need to verify 1000 candidate pairs without answer-level skipping, and no candidate pairs need to be verified if we compute $\mathcal{A}(r_j)$ using Lemma 6.1. Next, we show how to reduce the number of probed complementary candidates in $C_{CM}(r_i, r_{i'})$ and improve the computation of the current candidates $C_{CR}(r_i, r_{i'})$ using two steps, namely, complementary candidates probe and current candidates update as follows:

- In the first step, we probe the inverted lists of elements in $r_{i'} \setminus r_i$ and try to reduce the number of probed candidates.
- In the second step, we try to avoid computing the similarity between $r_{i'}$ and each record in $\mathcal{A}(r_i)$ from scratch. Instead, we aim to incrementally compute the similarity by exploring the differences between r_i and $r_{i'}$.

In addition, the answer-level skipping technique may not always be of benefit. For each $r_{i'} \in \mathcal{A}(r_i)$, we need a cost function to determine whether using the answer-level skipping technique to compute $\mathcal{A}(r_{i'})$ is better than computing $\mathcal{A}(r_{i'})$ from scratch. Below, we will introduce the two steps in detail, followed by a discussion on the skipping condition.

Step 1: Complementary Candidates Probe. In this step, we handle $C_{CM}(r_i, r_{i'}) = \{(r_{i'}, r_j) | r_j \in \cup_{e \in r_{i'} \setminus r_i} I(e)\}$. For each element $e \in r_{i'} \setminus r_i$, we probe the inverted list $I(e)$,

and consider the pairs $(r_{i'}, r_j)$ for $r_j \in I(e)$ as candidates. It is not necessary to consider all the records in $I(e)$. Therefore, we try to reduce the number of probed records by considering the three conditions:

- For the prefix filter, the element e may not be in the prefix $\mathsf{pre}_{t_{i'}}(r_{i'})$ of $r_{i'}$. Therefore, the prefix filter cannot be used.
- For the length filter, since e is not necessarily the first element in $r_i \cap r_{i'}$, the length filter in Lemma 3.5 cannot be used. However, the length filter in Lemma 3.2 can still be used since it only requires $|r_j| \leq \lfloor \frac{|r_{i'}|}{\tau} \rfloor$.
- For the position filter, Equation (3.5) also requires e to be the first element in $r_i \cap r_{i'}$. Therefore, the position filter cannot be used.

Based on the above analysis, for each element $e \in r_{i'} \setminus r_i$, we probe the records $r_j \in I(e)$ with $j > i'$ in non-decreasing order of $|r_j|$ and add $(r_{i'}, r_j)$ into the candidate set $C(r_{i'})$. Then we report $(r_{i'}, r_j)$ to be an answer in $\mathcal{A}(r_{i'})$ if it passes the verification. Once the length condition $|r_j| \leq \lfloor \frac{|r_{i'}|}{\tau} \rfloor$ fails, we break the current iteration and visit the next element $e \in r_{i'} \setminus r_i$. Here, before adding $(r_{i'}, r_j)$ into $C(r_{i'})$, we need to make sure that (1) $(r_{i'}, r_j) \notin C(r_{i'})$, which is used to avoid duplicated answers; and (2) $(r_i, r_j) \notin \mathcal{A}(r_i)$, since for those $(r_i, r_j) \in \mathcal{A}(r_i)$, the possible answer $(r_{i'}, r_j)$ will be handled in the next step - current candidates update.

Step 2: Current Candidates Update. In this step, we handle $C_{CR}(r_i, r_{i'}) = \{(r_{i'}, r_j) | (r_i, r_j) \in \mathcal{A}(r_i)\}$. For each $(r_i, r_j) \in \mathcal{A}(r_i)$, we aim to verify whether $\mathsf{Jac}(r_{i'}, r_j) \geq \tau$ incrementally without computing $\mathsf{Jac}(r_{i'}, r_j)$ from scratch. Note that for any (r_i, r_j), $\mathsf{Jac}(r_i, r_j) \geq \tau$ is equivalent to $|r_i \cap r_j| \geq \lceil \frac{\tau}{\tau+1}(|r_i| + |r_j|) \rceil$. After computing $\mathcal{A}(r_i)$, for each $(r_i, r_j) \in \mathcal{A}(r_i)$, the value of $|r_i \cap r_j|$ is already computed. Therefore, the key to incrementally verify $\mathsf{Jac}(r_{i'}, r_j) \geq \tau$ is to compute $|r_{i'} \cap r_j|$ based on $|r_i \cap r_j|$ incrementally. In order to do so, we can derive the following lemma:

Lemma 6.2: *For any three records r_i, $r_{i'}$, and r_j, we have:*

$$|r_{i'} \cap r_j| = |r_i \cap r_j| + \delta(r_i, r_{i'}, r_j) \tag{6.4}$$

where $\delta(r_i, r_{i'}, r_j) = |(r_{i'} \setminus r_i) \cap r_j| - |(r_i \setminus r_{i'}) \cap r_j|$. □

In order to make use of Lemma 6.2, we can initialize $|r_{i'} \cap r_j|$ as $|r_i \cap r_j|$ for all $(r_i, r_j) \in \mathcal{A}(r_i)$. Then we go through the inverted index of each element $e \in r_{i'} \setminus r_i$, and for each $r_j \in I(e)$, if $(r_i, r_j) \in \mathcal{A}(r_i)$, we increase $|r_{i'} \cap r_j|$ by one. Next, we go through the inverted index of each element $e \in r_i \setminus r_{i'}$, and for each $r_j \in I(e)$, if $(r_i, r_j) \in \mathcal{A}(r_i)$, we decrease $|r_{i'} \cap r_j|$ by one. In this way, we incrementally update $|r_{i'} \cap r_j|$ from $|r_i \cap r_j|$ by only considering the differences between $r_{i'}$ and r_i. Finally, for each (r_i, r_j), we check whether $|r_{i'} \cap r_j| \geq \lceil \frac{\tau}{\tau+1}(|r_{i'}| + |r_j|) \rceil$ holds. If so, we report $(r_{i'}, r_j)$ as an answer in $\mathcal{A}(r_{i'})$ without further verification.

Example 6.1: Consider the dataset in Table 2.1 with threshold $\tau = 0.7$. Take r_2 and r_3 as an example. Assume that we have already computed answer set of r_2 which is $\mathcal{A}(r_2) = \{(r_2, r_3), (r_2, r_4)\}$. Now we want to compute the answer set of record r_3 based on $\mathcal{A}(r_2)$. Here, $r_3 \setminus r_2 = \{e_2\}$. We probe the inverted list $I(e_2) = \{r_3, r_6, r_7, r_{10}\}$. Thus, $C_{CM}(r_2, r_3) = \{(r_3, r_6), (r_3, r_7), (r_3, r_{10})\}$. After result verification for candidates in $C_{CM}(r_2, r_3)$, none of them are added to $\mathcal{A}(r_3)$. We also have $C_{CR}(r_2, r_3) = \{(r_3, r_4)\}$. Instead of computing the similarity of r_3 and r_4 from scratch, we use Lemma 6.2. That is, $|r_3 \cap r_4| = |r_2 \cap r_4| + \delta(r_2, r_3, r_4)$. $|r_2 \cap r_4|$ is already known as 10 when computing $\mathsf{Jac}(r_2, r_4)$. Now we need to compute $\delta(r_2, r_3, r_4) = |(r_3 \setminus r_2) \cap r_4| - |(r_2 \setminus r_3) \cap r_4|$. Since $|(r_3 \setminus r_2) \cap r_4| = |\emptyset| = 0$ and $|(r_2 \setminus r_3) \cap r_4| = |\emptyset| = 0$, $\delta(r_2, r_3, r_4) = 0$. Thus,

$|r_3 \cap r_4| = 10 \geq \lceil \frac{\tau}{\tau+1}(|r_3| + |r_4|) \rceil$. Therefore, we have $\mathcal{A}(r_3) = \{(r_3, r_4)\}$. Using the answer-level skipping technique, when computing $\mathcal{A}(r_3)$, we reduce the number of probed records from 11 to 7 and reduce the number of verifications from 5 to 3. □

When to Skip. After computing $\mathcal{A}(r_i)$ for a record r_i, for each $r_{i'} \in \mathcal{A}(r_i)$, we need to determine whether to compute $r_{i'}$ from scratch or to use the above answer-level skipping technique. In order to do so, we need to estimate the cost to compute $\mathcal{A}(r_{i'})$ from scratch and the cost to compute $\mathcal{A}(r_{i'})$ using the answer-level skipping technique. The number of probed records is employed to estimate the cost since the exact cost is hard to compute. We use $\mathsf{cost}_{scratch}$ to denote the estimated cost to compute $\mathcal{A}(r_{i'})$ from scratch, and use cost_{skip} to denote the estimated cost to compute $\mathcal{A}(r_{i'})$ using the answer-level skipping technique. Below, we discuss how to compute $\mathsf{cost}_{scratch}$ and cost_{skip}:

- It is hard to compute $\mathsf{cost}_{scratch}$ without actually probing the records in the inverted lists of the elements in the prefix of $r_{i'}$. Therefore, we use the length of the inverted lists for the elements in the prefix of $r_{i'}$ to estimate the cost. Intuitively, the larger the length of the inverted lists for the elements in the prefix of $r_{i'}$, the more records will be probed when computing $\mathcal{A}(r_{i'})$ from scratch. We have:

$$\mathsf{cost}_{scratch} = c \times \sum_{e \in \mathsf{pre}_{i'}} |I(e)|. \tag{6.5}$$

  Here, c is a constant with $0 < c \leq 1$.

- To compute cost_{skip}, we need to investigate the two steps to compute $\mathcal{A}(r_{i'})$ incrementally. In step 1, we probe the records in the inverted lists of elements in $r_{i'} \setminus r_i$. Therefore, the number of probed records in step 1 is bounded by $\sum_{e \in r_{i'} \setminus r_i} |I(e)|$. In step 2, we probe the records in the inverted lists of elements in $r_{i'} \setminus r_i$ and $r_i \setminus r_{i'}$ to compute $\delta(r_i, r_{i'}, r_j)$ for all $(r_i, r_j) \in \mathcal{A}(r_i)$, and then we go through all records in $\mathcal{A}(r_i)$ to update the new answer set $\mathcal{A}(r_{i'})$. Therefore, the number of probed records in step 2 is bounded by $\sum_{e \in r_{i'} \setminus r_i} |I(e)| + \sum_{e \in r_i \setminus r_{i'}} |I(e)| + |\mathcal{A}(r_i)|$. To summarize, the estimated cost to compute $\mathcal{A}(r_{i'})$ using the skipping technique can be computed as:

$$\mathsf{cost}_{skip} = 2 \times \sum_{e \in r_{i'} \setminus r_i} |I(e)| + \sum_{e \in r_i \setminus r_{i'}} |I(e)| + |\mathcal{A}(r_i)|. \tag{6.6}$$

After computing $\mathsf{cost}_{scratch}$ and cost_{skip}, if $\mathsf{cost}_{skip} \geq \mathsf{cost}_{scratch}$, we compute $\mathcal{A}(r_{i'})$ from scratch. Otherwise, we compute $\mathcal{A}(r_{i'})$ using the answer-level skipping technique.

The Algorithm. Based on the above discussion, the algorithm for answer-level skipping is shown in Algorithm 3. We use $\mathcal{R}_{skip}$ to maintain the set of records whose answer sets have been computed using the answer-level skipping technique (line 1). For each record $r_i \in \mathcal{R}$ (line 2), if $r_i \in \mathcal{R}_{skip}$, we simply skip it and continue to handle the next r_i (line 3-4). Otherwise, we compute $\mathcal{A}(r_i)$ using Algorithm 2 (line 5). After computing $\mathcal{A}(r_i)$, we go through all records $(r_i, r_{i'})$ in $\mathcal{A}(r_i)$ to find possible opportunities for answer-level skipping (line 6). For each $(r_i, r_{i'}) \in \mathcal{A}(r_i)$, we compute $\mathsf{cost}_{scratch}$ (line 7) and cost_{skip} (line 8) as discussed above, and only apply the answer-level skipping technique if $\mathsf{cost}_{skip} < \mathsf{cost}_{scratch}$ (line 9-10). When applying the answer-level skipping technique for record $r_{i'}$, we first add $r_{i'}$ into $\mathcal{R}_{skip}$ (line 11) and initialize $C(r_{i'})$ and $\mathcal{A}(r_{i'})$ (line 12). Next, we apply the first step - complementary candidates probe - by probing the inverted lists $I(e)$ for $e \in r_{i'} \setminus r_i$ as discussed above (line 13-20). Then, we apply the second step - current candidates update - by probing the inverted lists $I(e)$ for $e \in r_{i'} \setminus r_i$ (line 21-23) and $e \in r_i \setminus r_{i'}$ (line 24-26), and updating $\mathcal{A}(r_{i'})$ based on

Algorithm 3: Answer-Level Skipping

Input: A set of records $\mathcal{R}$; a similarity threshold τ
Output: All similar record pairs (r_i, r_j) with $\mathsf{Jac}(r_i, r_j) \geq \tau$

```
1  R_skip ← ∅;
2  for each record r_i ∈ R do
3      if r_i ∈ R_skip then
4          continue;
5      Line 2-16 of Algorithm 2;
6      for each (r_i, r_i') ∈ A(r_i) do
7          cost_scratch ← c × Σ_{e∈pre_{t_i'}} |I(e)|;
8          cost_skip ← 2 × Σ_{e∈r_i'\r_i} |I(e)| + Σ_{e∈r_i\r_i'} |I(e)| + |A(r_i)|;
9          if cost_skip ≥ cost_scratch then
10             continue;
11         R_skip ← R_skip ∪ {r_i'};
12         C(r_i') ← ∅; A(r_i') ← ∅;
           /* Step 1: Complementary Candidates Probe */
13         for each element e ∈ r_i' \ r_i do
14             for each record r_j ∈ I(e) with j > i' do
15                 if |r_j| > ⌊|r_i'|/τ⌋ then
16                     break;
17                 if (r_i', r_j) ∉ C(r_i') and (r_i, r_j) ∉ A(r_i) then
18                     C(r_i') ← C(r_i') ∪ {(r_i', r_j)};
19                     if Jac(r_i', r_j) ≥ τ then
20                         A(r_i') ← A(r_i') ∪ {(r_i', r_j)};
           /* Step 2: Current Candidates Update */
21         for each element e ∈ r_i' \ r_i do
22             for each record r_j ∈ I(e) with (r_i, r_j) ∈ A(r_i) do
23                 δ(r_i, r_i', r_j) ← δ(r_i, r_i', r_j) + 1;
24         for each element e ∈ r_i \ r_i' do
25             for each record r_j ∈ I(e) with (r_i, r_j) ∈ A(r_i) do
26                 δ(r_i, r_i', r_j) ← δ(r_i, r_i', r_j) − 1;
27         for each (r_i, r_j) ∈ A(r_i) do
28             if |r_i ∩ r_j| + δ(r_i, r_i', r_j) ≥ ⌈τ/(1+τ) (|r_i'| + |r_j|)⌉ then
29                 A(r_i') ← A(r_i') ∪ {(r_i', r_j)};
30         output A(r_i');
```

$\mathcal{A}(r_i)$ (line 27-29) as discussed above. Finally, we output $\mathcal{A}(r_{i'})$ as the answer set for $r_{i'}$.

Example 6.2: We use the dataset from Table 2.1 to illustrate Algorithm 3. Suppose the threshold $\tau = 0.7$ and $c = 1$. Initially, $\mathcal{R}_{skip}$ is $\emptyset$. We consider the first record r_1. Since $r_1 \notin \mathcal{R}_{skip}$, we use Algorithm 2 to compute its answer set $\mathcal{A}(r_1) = \{(r_1, r_4)\}$. Then we consider $(r_1, r_4) \in \mathcal{A}(r_1)$. Since $r_4 \setminus r_1 = \{e_{12}, e_{16}\}$ and $r_1 \setminus r_4 = \{e_1, e_8\}$, we have $\mathsf{cost}_{skip} = 2 \times \sum_{e \in r_4 \setminus r_1} |I(e)| + \sum_{e \in r_1 \setminus r_4} |I(e)| + |\mathcal{A}(r_1)| = 2 * 16 + 8 + 1 = 41 > \mathsf{cost}_{scratch} = c \times \sum_{e \in \mathsf{pre}_{t_4}} |I(e)| = 20$. Therefore, r_4 is not added into $\mathcal{R}_{skip}$. Next, we process record $r_2 \notin \mathcal{R}_{skip}$. We can get $\mathcal{A}(r_2) = \{(r_2, r_3), (r_2, r_4)\}$ using Algorithm 2. For (r_2, r_3), since $r_3 \setminus r_2 = \{e_2\}$ and $r_2 \setminus r_3 = \{e_3\}$, we have $\mathsf{cost}_{skip} = 2 \times \sum_{e \in r_3 \setminus r_2} |I(e)| + \sum_{e \in r_2 \setminus r_3} |I(e)| +$

$|\mathcal{A}(r_2)| = 2*4+5+2 = 15 < \mathsf{cost}_{scratch} = c \times \sum_{e \in \mathsf{pre}_{r_3}} |I(e)| = 18$. Therefore, r_3 is added to $\mathcal{R}_{skip}$ and the answer set of r_3 is computed incrementally using Algorithm 3 as explained in Example 6.1. The remaining records are processed similarly. □

Cost Analysis. We first analyze the time cost for Algorithm 3. We divide the cost into two parts, namely, C_1: the cost to compute the answer sets for the non-skipped records $\mathcal{R} \setminus \mathcal{R}_{skip}$, and C_2: the cost to compute the answer sets for the skipped records $\mathcal{R}_{skip}$. Below, we analyze the two costs individually:

- For C_1, the failed attempts when computing the answer set for each individual record in $\mathcal{R} \setminus \mathcal{R}_{skip}$ using Algorithm 3 may be different from those when computing the answer set for the same record using Algorithm 2. However, it is easy to see that with the index level skipping technique, the total failed attempts in Algorithm 3 is subsumed by the total failed attempts in Algorithm 2.
- For C_2, with the skipping condition, for each record $r_i \in \mathcal{R}_{skip}$, the cost to compute $\mathcal{A}(r_i)$ from scratch is expected to be more expensive than the cost to compute $\mathcal{A}(r_i)$ using the answer-level skipping technique.

Based on the above analysis, the time cost consumed by Algorithm 3 is expected to be smaller than that consumed by Algorithm 2. Regarding the space cost, it is easy to see that Algorithm 3 consumes linear space $O(n \cdot l_{avg})$ w.r.t. the size of the input by keeping the inverted lists for all elements in memory, which is the same as that consumed by Algorithm 2.

7 Performance Studies

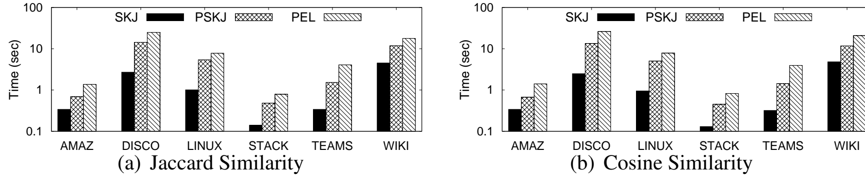
In this section, we present our experimental results. All of our experiments are conducted on a machine with an Intel Xeon E5-2690 (8 Cores) 2.9GHz CPU and 32GB main memory running Linux (Red Hat Enterprise Linux 6.4, 64 bit).

Datasets. We used 20 real datasets selected from different domains with various data properties. The datasets include those datasets used in existing works. The detailed characteristics of the 20 datasets are shown in Table 7.1. For each dataset, we show the type of the dataset, what each record and each element represents, the number of records in the dataset, the maximum and average record length, the number of different elements in the dataset, and the time to build the index with skipping blocks. As shown in Table 7.1, the index can be built very efficiently since the longest index time for the 20 datasets is only 4.52 seconds.

Algorithms. We compared our algorithm (Algorithm 3), denoted as SKJ, with all the state-of-the-art algorithms for exact set similarity join, including PPJ and PPJ+ [43], ADP [41], PEL [31] and PTJ [28]. For PPJ, PPJ+, ADP and PEL, we obtained the source codes from the authors of [32], since the authors claimed in [32] that their implementation is faster than the implementation provided by the original authors. The source code of PTJ was obtained from the authors of [28]. In all algorithms, we used the same candidate verification algorithm provided by the authors of [32], which was claimed to be much faster than other implementations. All algorithms were implemented in C++ and compiled with GCC with the -O3 flag. The preprocessing time is not included in all tests. We use 0.9 as the default threshold value τ, and we vary τ from 0.5 to 0.95 in Section 7.3.

Dataset	Abbreviation	Type	Record	Elements	# Record	Max Length l_{max}	Avg Length l_{avg}	# Elements	Index Time(s)
Amazon [1]	AMAZ	Rating	Product	Rating	1,230,915	3,096	4.67	2,146,057	0.12
AOL [2]	AOL	Text	Query	Keyword	10,054,183	245	3.01	3,873,246	1.30
Bookcrossing [3]	BOOKC	Rating	Book	User	340,523	2,502	3.38	105,278	0.02
Citeulike [4]	CITEU	Folksonomy	Tag	User	153,277	8,814	3.51	22,715	0.01
DBLP [28]	DBLP	Text	Biboliography	3-gram	873,524	1,538	94.06	44,798	4.49
Delicious [5]	DELIC	Folksonomy	User	Tag	833,081	16,996	98.42	4,512,099	2.50
Discogs [6]	DISCO	Affiliation	Artist	Label	1,754,823	47,522	3.02	270,771	0.14
Enron [7]	ENRON	Text	Email	Word	517,431	3,162	133.57	1,113,219	2.51
Flickr [21]	FLICK	Folksonomy	Photo	Word/Tag	1,235,799	102	10.05	810,659	0.42
Kosarak [8]	KOSA	Interaction	User	Link	990,001	2,497	8.10	41,269	0.27
Lastfm [9]	LAST	Interaction	User	Song	1,084,620	773	4.07	992	0.16
Linux [10]	LINUX	Interaction	Thread	User	337,509	6,627	1.78	42,045	0.01
Livejournal [11]	LIVEJ	Affiliation	User	Group	3,201,203	300	35.08	7,489,073	4.52
Reuters [12]	RUTRS	Text	Story	Word	781,265	1585	77.53	283,911	2.72
Spotify [13]	SPOT	Interaction	User	Track	437,836	11,608	12.75	759,041	0.09
Stack [14]	STACK	Rating	User	Post	545,196	4,917	2.39	96,680	0.02
Sualize [15]	SUALZ	Folksonomy	Picture	Tag	495,402	433	3.63	82,035	0.06
Teams [16]	TEAMS	Affiliation	Athlete	Team	901,166	17	1.52	34,461	0.03
Tweet [28]	TWEET	Text	Tweet	Word	2,000,000	70	21.57	1,713,437	2.44
Wikipedia [17]	WIKI	Authorship	User	Article	3,819,691	1,916,898	31.96	21,504,191	1.65

Table 7.1: Characteristics of datasets



(a) Jaccard Similarity (b) Cosine Similarity

Figure 7.1: Evaluating the Skipping Techniques

7.1 Evaluation of the Skipping Techniques

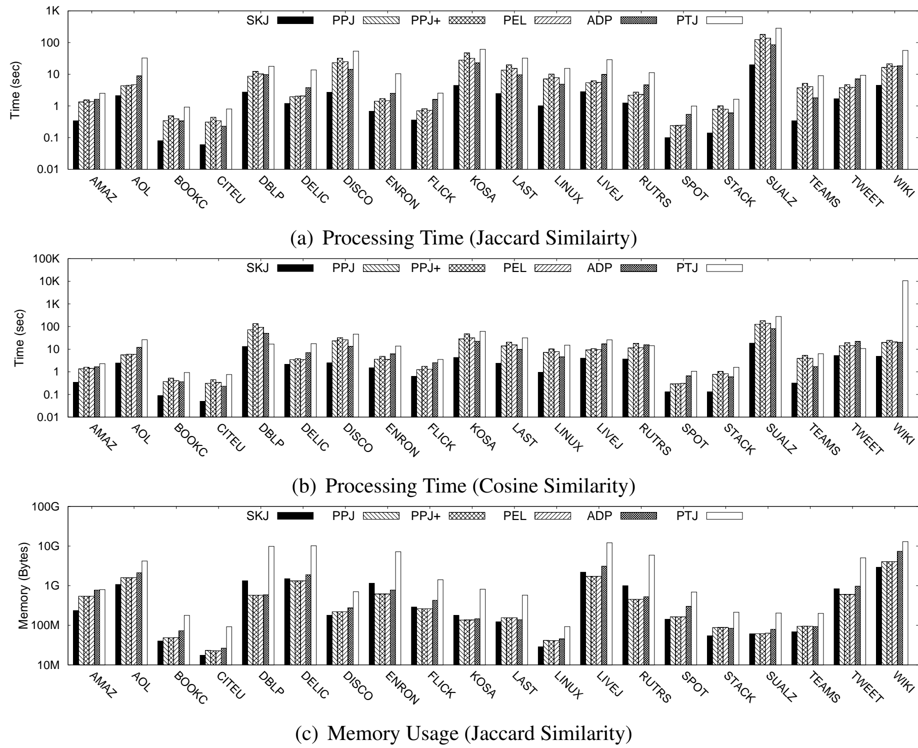
We first evaluate the efficiency of our index-level skipping and answer-level skipping techniques. We use PSKJ to denote the algorithm with the index-level skipping technique, i.e., Algorithm 2, and use SKJ to denote the algorithm with both the index-level skipping technique and answer-level skipping technique, i.e., Algorithm 3. We compare the two skipping based algorithms with the state-of-the-art prefix-filter based algorithm PEL with all the three filtering techniques introduced in Section 3.1. We test the three algorithms over six representative datasets. The experimental results are shown in Figure 7.1(a) and Figure 7.1(b) for the Jaccard similarity function and Cosine similarity function respectively.

The experimental results demonstrate the power of our index-level and answer-level skipping techniques for the two similarity functions. First, we can see that PSKJ consistently outperforms PEL and is faster than PEL by 2-3 times. This is because PSKJ saves the index probing cost by skipping useless entry visits and filtering condition checks. The experimental results are consistent with our theoretical analysis in Section 5. Second, we find that SKJ further improves PSKJ and performs best over all datasets. This is because the answer-level skipping technique effectively reduces the computational cost by incrementally computing the answer sets for records. Remarkably, SKJ can achieve more than an order of magnitude faster than PEL. For example, on the DISCO dataset for the Cosine similarity function, the running times of algorithms SKJ, PSKJ and PEL are 2.49 seconds, 13.38 seconds, and 26.16 seconds, respectively. For the two similarity functions, the relative performances for the three algorithms are similar. In the following we use SKJ to compare with other algorithms.

7.2 Comparison with Existing Solutions

This experiment compares our algorithm SKJ with the state-of-the-art algorithms PPJ, PPJ+, ADP, PEL, and PTJ for exact set similarity join. We conduct the experiment over all 20 datasets for Jaccard and Cosine similarities. The experimental results are shown in Figure 7.2.

Figure 7.2(a) and Figure 7.2(b) show the processing time of all algorithms for the Jaccard similarity and Cosine similarity respectively. We can see that our algorithm SKJ consistently outperforms all the state-of-the-art algorithms in all datasets. For example, on the TEAMS dataset for the Jaccard similarity, SKJ is 11, 15, 12, 5, and 26 times faster than PPJ, PPJ+, PEL, ADP, and PTJ, respectively. The results demonstrate the large computational cost saving achieved by using our index-level and answer-level skipping techniques. The performances of the existing prefix-filter based algorithms PPJ, PPJ+, PEL, and ADP are quite similar, which is consistent with the result obtained by [32]. PPJ+ performs the worst among the four algorithms PPJ, PPJ+, PEL, and ADP, because with the fast candidate verification algorithm [32], the sophisticated filters used in PPJ+ do not pay off. The performance of ADP is depen-



(a) Processing Time (Jaccard Similairty)

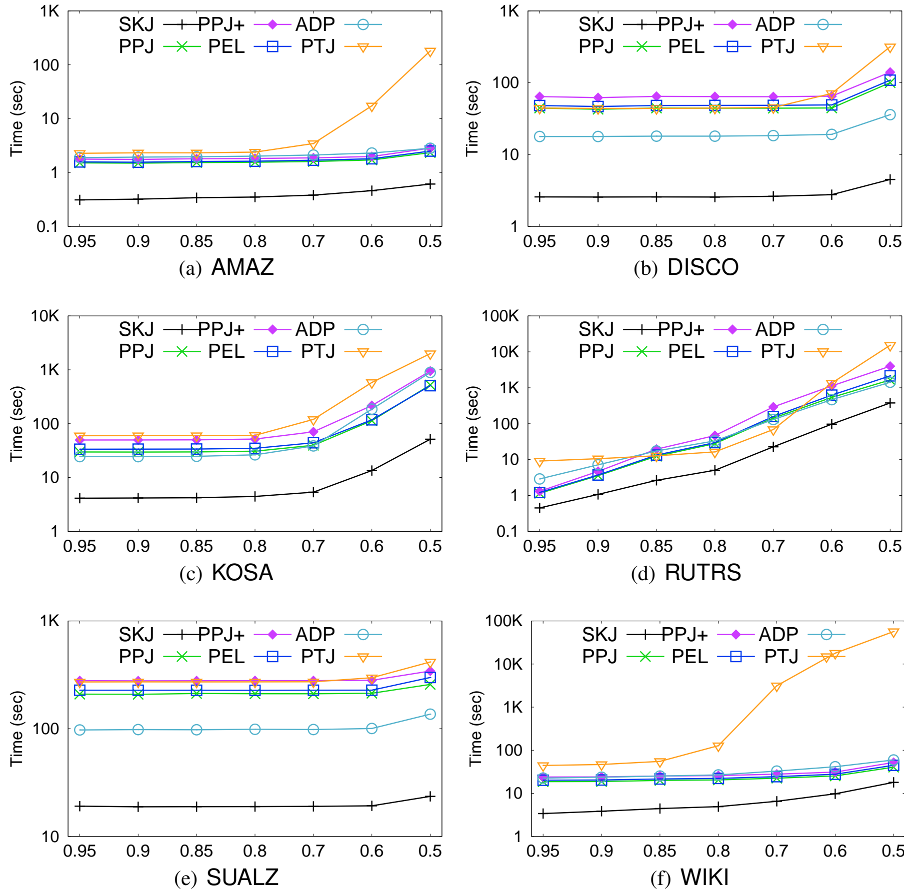
(b) Processing Time (Cosine Similarity)

(c) Memory Usage (Jaccard Similarity)

Figure 7.2: Comparison with Existing Algorithms

dent on the balance between the overhead of extended prefix searching and the cost saved by reducing candidate size. For the partition-filter based algorithm PTJ, it cannot outperform the prefix-filter based algorithms PPJ, PPJ+, PEL, and ADP in most cases. It is worth noting that, we do reproduce the results in [28] to compare PTJ with the original implementations of PPJ+ and ADP over the three datasets used in [28]. However, as stated in [32], their implementations of prefix-filter based algorithms is much faster than the original implementations in all cases. On the other hand, PTJ uses complex index structures such as hash table to reduce the number of candidates, and thus spends too much time on the filtering phase. As a result, PTJ can hardly outperform the prefix-filter based algorithms, which use a light-weight index structure and spend much less time on the filtering phase. Nevertheless, PTJ wins the existing prefix-filter based algorithms on some datasets such as DBLP and TWEET for the Cosine similarity. This is because the candidate number generated by PTJ is much closer to the number of results than that of PPJ, PPJ+, PEL and ADP. However, our algorithm SKJ still outperforms PTJ in these cases.

We also compare the memory cost of the six algorithms and show the experimental results for Jaccard similarity in Figure 7.2(c). We can see that the memory usage for the prefix-filter based algorithms PPJ, PPJ+, PEL, ADP, as well as our algorithm SKJ are similar since they all consume a memory size linear to the size of the input dataset. The small difference is due to the different auxiliary data structures used for different algorithms. The partition-filter based algorithm PTJ consumes much larger memory than the prefix-filter based algorithms in all datasets. For example, for the DELIC dataset, the memory usages for PPJ, PPJ+, PEL, ADP, SKJ, and PTJ are 1.3 GB, 1.3 GB, 1.3 GB, 1.9 GB, 1.5 GB, and 10.1 GB respectively. This is because PTJ needs to build the inverted lists for all sub-records and their 1-deletion neighborhoods



(a) AMAZ (b) DISCO (c) KOSA (d) RUTRS (e) SUALZ (f) WIKI

Figure 7.3: Vary Threshold τ (Jaccard Similarity)

which consumes $O(n \cdot l_{max})$ space as shown in Section 3.1. The memory cost of the six algorithms for the Cosine similarity are similar to that for the Jaccard similarity, and therefore we omit the result here. In the following, when varying different parameters, we only show our experimental results for Jaccard similarity since the performance of the algorithms for Cosine similarity are similar to that for Jaccard similarity.

7.3 Variation of the Threshold

In this experiment, we test the performance of our algorithm SKJ for different threshold τ and compare it with the existing algorithms PPJ, PPJ+, PEL, ADP, and PTJ. We vary τ from 0.95 to 0.5. Figure 7.3 shows the experimental results on six representative datasets for the Jaccard similarity.

From the experimental results, we can see that when the threshold τ decreases, the running time of all algorithms increases. The reasons are twofold. First, for prefix-filter based algorithms, a smaller τ leads to a longer prefix, and for partition based algorithms, a smaller τ leads to more partitions. Therefore, a smaller τ leads to a longer filtering time. Second, a smaller τ will result in more similar pairs which leads to more candidates and thus requires longer verification time. In all datasets, we can see that our algorithm SKJ consistently outperforms all other algorithms for different threshold values. For example, on the DISCO dataset, when $\tau = 0.95$, SKJ is 17, 25, 19, 7, and

17 times faster than PPJ, PPJ+, PEL, ADP, and PTJ respectively; When τ = 0.5, SKJ is 22, 31, 24, 8, and 70 times faster than PPJ, PPJ+, PEL, ADP, and PTJ respectively. In AMAZ and WIKI, when τ decreases to be smaller than 0.8, the processing time for PTJ increases sharply. This is because when τ is small, a large number of partitions are generated by PTJ. It is worth noticing that even when τ is small, our algorithm SKJ can still achieve a high speedup comparing to existing algorithms. This is because for a smaller threshold, SKJ will produce a larger answer set which results in more computational cost sharing.

The experimental results for scalability testing are shown in Section C.1 in the Appendix.

8 Related Work

Exact Set Similarity Join. Exact set similarity join has been extensively studied in the literature [18, 19, 20, 21, 28, 31, 32, 33, 34, 35, 39, 42, 43]. As we have introduced in Section 3, existing solutions all follow the filtering-verification framework and can be divided into two categories based on the filtering mechanism, namely, prefix-filter based algorithms and partition-filter based algorithms.

For prefix-filter based algorithms, Bayardo et al. [20] first proposed prefix-filter based framework. Xiao et al. [43] introduced positional filter and suffix filter to the prefix-filter based framework. An optimized length filter was proposed by Mann et al. in [31]. Wang et al. [41] devised adaptive length prefix to strengthen filtering power. Other prefix-filter based algorithms include [34] to remove the useless entries in the inverted lists of elements, and [21] to group the records with the same prefix. However, they cannot significantly improve the algorithm. The details of the state-of-the-art prefix-filter based algorithms are introduced in Section 3.1. Mann et al. [32] introduced an efficient candidate verification algorithm that improves the efficiency of all existing prefix-filter based algorithms. They conducted a comprehensive study on existing prefix-filter based techniques and demonstrated that with some basic filtering techniques, using sophisticated filters to further reduce the number of candidates does not pay off. Our method falls into this category. Since none of the existing solutions have ever considered the correlations of records in join processing, in this paper, we aim to improve the algorithm by considering such information.

For partition-filter based algorithms, Arasu et al. [19] developed a two-level algorithm with partitioning and enumerating to find exact similar sets. Deng et al. [28] designed partition-based method for exact set similarity join, which is introduced in details in Section 3.2. The partition-filter based algorithm [28] can effectively reduce the number of candidates. However, it usually results in an expensive filtering cost and high memory overhead to maintain the complex inverted lists for partitioned sub-records and their 1-deletion neighborhoods.

Other work focus on processing exact set similarity join in a distributed environment using MapReduce [33, 35, 39], which is not the focus of this paper.

Approximate Set Similarity Join. Approximate set similarity join is also widely studied [22, 23, 29, 36, 30, 45], which can only produce approximate join results. Existing works on the approximate set similarity join problem are mostly based on the Locality Sensitive Hashing (LSH) technique [30], which is a probabilistic scheme by hashing similar records into the same cluster with high probability. Among them, MinHash

[22] is a quick estimation method for Jaccard similarity. Zhai et al. proposed a probabilistic algorithm ATLAS for similarity search with a low threshold on records with a high dimension [45]. Satuluri et al. proposed a Bayesian algorithm BayesLSH to extend LSH to be used in candidate pruning and similarity estimation [36]. Chakrabarti et al. [23] adopted sequential hypothesis testing on LSH to adaptively prune candidates aggressively and provide tighter qualitative guarantees over BayesLSH. In this paper, we focus on exact similarity join with different problem settings from the approximate set similarity join.

9 Conclusion

In this paper, we study the exact set similarity join problem, which is a fundamental problem with a wide range of applications. Existing solutions compute the answer set for each record individually, which may result in a large number of redundant computations. In this paper, we aim to leverage the correlations among records to seek for possible cost sharing in the join process. We first explore index-level correlations to group records in the inverted index of elements and design effective skipping techniques for records in each group. We then investigate the answer-level correlations to compute the answer set of a record incrementally based on the answer set of a similar record. Our algorithm improves the state-of-the-art algorithms both theoretically and in practice. We conducted extensive experiments on 20 real datasets with various data properties. The experimental results demonstrate that our algorithm outperforms all the other algorithms in all datasets and can achieve a speedup of more than one order of magnitude against the state-of-the-art algorithms.

Bibliography

[1] http://liu.cs.uic.edu/download/data/.

[2] http://www.cim.mcgill.ca/ dudek/206/Logs/AOL-user-ct-collection.

[3] http://www.informatik.uni-freiburg.de/ cziegler/BX/.

[4] http://www.citeulike.org/faq/data.adp.

[5] http://dai-labor.de/IRML/datasets.

[6] http://www.discogs.com/.

[7] http://www.cs.cmu.edu/ enron.

[8] http://fimi.ua.ac.be/data/.

[9] http://www.dtic.upf.edu/ ocelma/MusicRecommendationDataset/lastfm-1K.html.

[10] http://konect.uni-koblenz.de/networks/lkml_person-thread.

[11] http://socialnetworks.mpi-sws.org/data-imc2007.html.

[12] http://trec.nist.gov/data/reuters/reuters.html.

[13] http://dbis-twitterdata.uibk.ac.at/spotifyDataset/.

[14] http://www.clearbits.net/torrents/1881-dec-2011.

[15] http://vi.sualize.us/.

[16] http://wiki.dbpedia.org/Downloads.

[17] http://dumps.wikimedia.org/.

[18] D. C. Anastasiu and G. Karypis. L2AP: fast cosine similarity search with prefix L-2 norm bounds. In *Proc. of ICDE'14*.

[19] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *Proc. of VLDB'06*.

[20] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *Proc. of WWW'07*.

[21] P. Bouros, S. Ge, and N. Mamoulis. Spatio-textual similarity joins. *PVLDB*, 6(1).

[22] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations (extended abstract). In *Proc. of STOC'98*.

[23] A. Chakrabarti and S. Parthasarathy. Sequential hypothesis tests for adaptive locality sensitive hashing. In *Proc. of WWW'15*.

[24] K. Chakrabarti, S. Chaudhuri, V. Ganti, and D. Xin. An efficient filter for approximate membership checking. In *Proc. of SIGMOD'08*.

[25] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *Proc. of ICDE'06*.

[26] W. W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *Proc. of SIGMOD'98*.

[27] A. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: scalable online collaborative filtering. In *Proc. of WWW'07*.

[28] D. Deng, G. Li, H. Wen, and J. Feng. An efficient partition based method for exact set similarity joins. *PVLDB*, 9(4).

[29] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proc. of VLDB'99*.

[30] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proc. of STOC'98*.

[31] W. Mann and N. Augsten. PEL: position-enhanced length filter for set similarity joins. In *Proc. of GVD'14*.

[32] W. Mann, N. Augsten, and P. Bouros. An empirical evaluation of set similarity join techniques. *PVLDB*, 9(9).

[33] A. Metwally and C. Faloutsos. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *PVLDB*, 5(8).

[34] L. A. Ribeiro and T. Härder. Generalizing prefix filtering to improve set similarity joins. *Inf. Syst.*, 36(1).

[35] A. D. Sarma, Y. He, and S. Chaudhuri. Clusterjoin: A similarity joins framework using map-reduce. *PVLDB*, 7(12).

[36] V. Satuluri and S. Parthasarathy. Bayesian locality sensitive hashing for fast similarity search. *PVLDB*, 5(5).

[37] E. Spertus, M. Sahami, and O. Buyukkokten. Evaluating similarity measures: a large-scale study in the orkut social network. In *Proc. of SIGKDD'05*.

[38] M. Theobald, J. Siddharth, and A. Paepcke. Spotsigs: robust and efficient near duplicate detection in large web collections. In *Proc. of SIGIR'08*.

[39] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *Proc. of SIGMOD'10*.

[40] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. Crowder: Crowdsourcing entity resolution. *PVLDB*, 5(11).

[41] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *Proc. of SIGMOD'12*.

[42] C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k set similarity joins. In *Proc. of ICDE'09*.

[43] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *Proc. of WWW'08*.

[44] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near-duplicate detection. *ACM Trans. Database Syst.*, 36(3).

[45] J. Zhai, Y. Lou, and J. Gehrke. ATLAS: a probabilistic algorithm for high dimensional similarity search. In *Proc. of SIGMOD'11*.

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

A Proofs of Lemmas/Theorems

Proof Sketch of Theorem 3.1. The time complexity of the filtering phase (line 2-10) of Algorithm 1 to compute $C(r_i)$ for each record r_i depends on the two nested loops in line 4 and line 6. In the worst case, the loop in line 4 explores a constant proportion c of elements $e \in r_i$ and the loop in line 6 probes all records in $I(e)$. Therefore, the total time of the filtering phase is $O(\sum_{r\in\mathcal{R}}(c \cdot \sum_{e\in r} |I(e)|)) = O(\sum_{r\in\mathcal{R}} \sum_{e\in r} n) = O(n^2 \cdot l_{avg})$. □

Proof Sketch of Theorem 3.2. We only need to store all records in $\mathcal{R}$ and the inverted list $I(e)$ for all elements e in $\mathcal{R}$. Therefore, the space complexity of Algorithm 1 is linear to the size of the database, which is $O(\sum_{r\in\mathcal{R}} |r|) = O(n \cdot l_{avg})$. □

Proof Sketch of Theorem 3.3. The time complexity of filtering phase is dependent on two operations: set partition and index list accessing. As stated in [28], the time complexity of set partition is $O(n \cdot (1-\tau) \cdot l_{max}^2)$. In the worst case, the time complexity of accessing index lists is $O(c \cdot n \cdot l_{avg})$. Therefore, the total time complexity of filtering phase is $O(n \cdot (1-\tau) \cdot l_{max}^2 + c \cdot n \cdot l_{avg}) = O(n \cdot l_{max}^2)$.

Proof of Theorem 3.4 can be found in [28].

Proof Sketch of Lemma 5.1. We first prove (1). Since $(r_j, \mathsf{pos}_e(r_j))$ fails Equation (5.1), we have $(1+\tau) \cdot \mathsf{pos}_e(r_j) + \tau \cdot |r_i| > |r_j| + \tau + 1$. For any un-probed entry $(r_{j'}, \mathsf{pos}_e(r_{j'}))$ in $B_e(l)$, we know that $|r_{j'}| = |r_j|$ and $\mathsf{pos}_e(r_{j'}) \geq \mathsf{pos}_e(r_j)$. Therefore, $(1+\tau) \cdot \mathsf{pos}_e(r_{j'}) + \tau \cdot |r_i| > |r_{j'}| + \tau + 1$ holds. Consequently, $(r_{j'}, \mathsf{pos}_e(r_{j'}))$ fails Equation (5.1).

We then prove (2). First, since $(r_j, \mathsf{pos}_e(r_j))$ fails Equation (5.1), we have $(1+\tau) \cdot \mathsf{pos}_e(r_j) + \tau \cdot |r_i| > |r_j| + \tau + 1$. For any $r_{i'}$ with $i' > i$, we have $|r_{i'}| \geq |r_i|$. Therefore, $(1+\tau) \cdot \mathsf{pos}_e(r_j) + \tau \cdot |r_{i'}| > |r_j| + \tau + 1$. Consequently, $(r_j, \mathsf{pos}_e(r_j))$ will fail Equation (5.1) when computing $C(r_{i'})$ for any $i' > i$. Second, from (1), we know that for any un-probed entry $(r_{j'}, \mathsf{pos}_e(r_{j'}))$ in $B_e(l)$, $(1+\tau) \cdot \mathsf{pos}_e(r_{j'}) + \tau \cdot |r_i| > |r_{j'}| + \tau + 1$. For any $r_{i'}$ with $i' > i$, we have $|r_{i'}| \geq |r_i|$. Therefore, $(1+\tau) \cdot \mathsf{pos}_e(r_{j'}) + \tau \cdot |r_{i'}| > |r_{j'}| + \tau + 1$. Consequently, $(r_{j'}, \mathsf{pos}_e(r_{j'}))$ will fail Equation (5.1) when computing $C(r_{i'})$ for any $i' > i$. □

Proof Sketch of Lemma 5.2. We prove the lemma using three steps:

(a) It is obvious that $\mathcal{F}_e^*(l) \subseteq \mathcal{F}_e(l)$.

(b) We prove that for any non-skyline point $(|r_i|, \mathsf{pos}_e(r_j)) \in \mathcal{F}_e(l)$*, we have* $(|r_i|, \mathsf{pos}_e(r_j)) \notin \mathcal{F}_e^*(l)$*.* Since $(|r_i|, \mathsf{pos}_e(r_j))$ is a non-skyline point, there exists another point $(|r_{i'}|, \mathsf{pos}_e(r_{j'})) \in \mathcal{F}_e(l)$ with $|r_{i'}| \leq |r_i|$ and $\mathsf{pos}_e(r_{j'}) \leq \mathsf{pos}_e(r_j)$. If $i' < i$, $(|r_i|, \mathsf{pos}_e(r_j))$ is probed after $(|r_{i'}|, \mathsf{pos}_e(r_{j'}))$. Since $\mathsf{pos}_e(r_{j'}) \leq \mathsf{pos}_e(r_j)$, according to Lemma 5.1 (2), $(|r_i|, \mathsf{pos}_e(r_j))$ is skipped by Algorithm 2. Therefore, $(|r_i|, \mathsf{pos}_e(r_j)) \notin \mathcal{F}_e^*(l)$. Otherwise, we have $i' = i$. In this situation, according to Lemma 5.1 (1), $(|r_i|, \mathsf{pos}_e(r_j))$ is skipped by Algorithm 2. Therefore, $(|r_i|, \mathsf{pos}_e(r_j)) \notin \mathcal{F}_e^*(l)$.

(c) We prove that for any skyline point $(|r_i|, \mathsf{pos}_e(r_j)) \in \mathcal{F}_e(l)$*, we have* $(|r_i|, \mathsf{pos}_e(r_j)) \in \mathcal{F}_e^*(l)$*.* Without loss of generality, we suppose that when there are multiple (r_i, r_j) with the same $(|r_i|, \mathsf{pos}_e(r_j))$, we choose the r_i with the smallest i. We prove (c) by contradiction. Suppose $(|r_i|, \mathsf{pos}_e(r_j)) \notin \mathcal{F}_e^*(l)$, $(|r_i|, \mathsf{pos}_e(r_j))$ should be skipped by another point $(|r_{i'}|, \mathsf{pos}_e(r_{j'})) \in \mathcal{F}_e^*(l)$ with $i' \leq i$ according to Algorithm 2. If $i' < i$, we have $\mathsf{pos}_e(r_{j'}) \leq \mathsf{pos}_e(r_j)$ according to Lemma 5.1 (2), and we have $(|r_{i'}|, \mathsf{pos}_e(r_{j'})) \neq (|r_i|, \mathsf{pos}_e(r_j))$. Therefore, $(|r_i|, \mathsf{pos}_e(r_j))$ is dominated by $(|r_{i'}|, \mathsf{pos}_e(r_{j'}))$. Otherwise, $i' = i$. In this case, we have $\mathsf{pos}_e(r_{j'}) < \mathsf{pos}_e(r_j)$ according to Lemma 5.1 (1). Therefore, $(|r_i|, \mathsf{pos}_e(r_j))$ is dominated by $(|r_{i'}|, \mathsf{pos}_e(r_{j'}))$. By combing the two cases, we can prove that $(|r_i|, \mathsf{pos}_e(r_j))$ is a non-skyline point in $\mathcal{F}_e(l)$, which contradicts the assumption that $(|r_i|, \mathsf{pos}_e(r_j))$ is a skyline point in $\mathcal{F}_e(l)$.

According to (a), (b), and (c), the lemma is proved. □

Proof Sketch of Theorem 5.1. We divide the time cost of Algorithm 2 into the following three parts:

(1) The time spent on the failed attempts, i.e., the attempts $(r_i, r_j) \notin C(r_i)$*, in the filtering phase.* According to Lemma 5.2, for each skipping block $B_e(l)$, only the skyline points in $\mathcal{F}_e(l)$ are attempted in Algorithm 2. Obviously, the total number of skyline points in $\mathcal{F}_e(l)$ is limited by the number of different $\mathsf{pos}_e(r_j)$ values in $B_e(l)$, which is bounded by $|B_e(l)|$. Therefore, the total number of failed attempts is bounded by $\sum_{e \in \mathcal{E}} \sum_{B_e(l) \in I^*(e)} |B_e(l)| = \sum_{e \in \mathcal{E}} |I(e)|$ which is bounded by the input size $n \cdot l_{avg}$. Each failed attempt consumes constant time. Therefore, the total time spent on the failed attempts is bounded by $O(n \cdot l_{avg})$.

(2) The time spent on the attempts $(r_i, r_j) \in C(r_i)$ *in the filtering phase.* We show that this cost is dominated by the verification cost V. Note that the pair (r_i, r_j) may be visited multiple times when probing the inverted lists for different $e \in \mathsf{pre}_{t_i}(r_i)$. Obviously, for each such visit, we have $e \in r_i \cap r_j$. Since $(r_i, r_j) \in C(r_i)$, the position condition in Equation (3.5) is satisfied, i.e., $|r_j| - \mathsf{pos}_e(r_j) + 1 \geq \lceil \frac{\tau}{1+\tau}(|r_i| + |r_j|) \rceil$. Consequently, we have $|r_j| - \mathsf{pos}_e(r_j) + |\mathsf{pre}_{\mathsf{pos}_e(r_i)}(r_i) \cap \mathsf{pre}_{\mathsf{pos}_e(r_j)}(r_j)| \geq \lceil \frac{\tau}{1+\tau}(|r_i| + |r_j|) \rceil$. In other words, the condition in Lemma 3.3 is not violated when visiting the element $e \in r_i \cap r_j$. Recall that the stop condition used in the result verification is based on the condition in Lemma 3.3. Therefore, the verification of the pair (r_i, r_j) does not stop when visiting element $e \in r_i \cap r_j$. As a result, the cost spent on probing each $(r_i, r_j) \in C(r_i)$ is dominated by the cost of verifying (r_i, r_j). According to the above discussion, the time spent on the attempts $(r_i, r_j) \in C(r_i)$ in the filtering phase is bounded by $O(V)$.

(3) The time spent on the verification phase. This cost is $O(V)$.

Combining (1), (2), and (3), the overall time complexity of Algorithm 2 is $O(n \cdot l_{avg} + V)$. □

Proof Sketch of Theorem 5.2. The main space cost for Algorithm 2 is spent on the

inverted lists $I^*(e)$ for $e \in \mathcal{E}$. Note that the number of entries in $I^*(e)$ is exactly the same as that in $I(e)$. Therefore, the total space cost is bounded by $O(\sum_{e \in \mathcal{E}} |I(e)|) = O(n \cdot l_{avg})$ □

Proof Sketch of Lemma 6.1. For any $(r_{i'}, r_j) \in \mathcal{A}(r_{i'})$, we have $\mathsf{Jac}(r_{i'}, r_j) \geq \tau$, which is equivalent to $|r_{i'} \cap r_j| \geq \frac{\tau}{1+\tau}(|r_{i'}| + |r_j|)$. We consider the following two cases:

- Case 1: $r_{i'} \cap r_j \subseteq r_i$. In this case, we have $|r_i \cap r_j| \geq |r_{i'} \cap r_i \cap r_j| = |r_{i'} \cap r_j \cap r_i| = |r_{i'} \cap r_j| \geq \frac{\tau}{1+\tau}(|r_{i'}| + |r_j|) \geq \frac{\tau}{1+\tau}(|r_i| + |r_j|)$ since $|r_{i'}| \geq |r_i|$. Therefore, $(r_i, r_j) \in \mathcal{A}(r_i)$. In other words, in this case, we have $(r_{i'}, r_j) \in C_{CR}(r_i, r_{i'}) = \{(r_{i'}, r_j) | (r_i, r_j) \in \mathcal{A}(r_i)\}$.
- Case 2: $r_{i'} \cap r_j \nsubseteq r_i$. In this case, there exists an element $e \in r_{i'} \cap r_j$ and $e \notin r_i$. We have $e \in r_{i'} \setminus r_i$ and thus $r_j \in \cup_{e \in r_{i'} \setminus r_i} I(e)$. In other words, in this case, we have $(r_{i'}, r_j) \in C_{CM}(r_i, r_{i'}) = \{(r_{i'}, r_j) | r_j \in \cup_{e \in r_{i'} \setminus r_i} I(e)\}$.

According to Case 1 and Case 2, we conclude that $\mathcal{A}(r_{i'}) \subseteq C_{CM}(r_i, r_{i'}) \cup C_{CR}(r_i, r_{i'})$. □

Proof Sketch of Lemma 6.2. For any two records $r_{i'}$ and r_i, we have $r_{i'} = (r_{i'} \setminus r_i) \cup (r_{i'} \cap r_i)$ and $r_{i'} \cap r_i = r_i \setminus (r_i \setminus r_{i'})$. Therefore, $r_{i'} = (r_{i'} \setminus r_i) \cup r_i \setminus (r_i \setminus r_{i'})$. Consequently, for any three records r_i, $r_{i'}$, and r_j, we have $|r_{i'} \cap r_j| = |((r_{i'} \setminus r_i) \cup r_i \setminus (r_i \setminus r_{i'})) \cap r_j| = |((r_{i'} \setminus r_i) \cap r_j) \cup (r_i \cap r_j) \setminus ((r_i \setminus r_{i'}) \cap r_j)|$. We also have $(r_{i'} \setminus r_i) \cap r_i = \emptyset$ and $r_i \setminus r_{i'} \subseteq r_i$. Therefore, we can derive that $|r_{i'} \cap r_j| = |(r_{i'} \setminus r_i) \cap r_j| + |r_i \cap r_j| - |(r_i \setminus r_{i'}) \cap r_j| = |r_i \cap r_j| + \delta(r_i, r_{i'}, r_j)$. □

B Discussion

B.1 Handling R-S Join

Our algorithm can be easily extended to handle R-S join. Given two collections of records $\mathcal{R}$ and $\mathcal{S}$, for each record $r_i \in \mathcal{R}$, we aim to compute the answer set $\mathcal{A}(r_i) = \{(r_i, s_j) | r_i \in \mathcal{R}, s_j \in \mathcal{S}, \mathsf{Jac}(r_i, s_j) \geq \tau\}$. The two skipping techniques can be adapted to handle R-S join as follows:

- For the index-level skipping technique, we only build the inverted index with skipping blocks for the collection $\mathcal{S}$. In join processing, for each record $r_i \in \mathcal{R}$, we find the candidate pairs (r_i, s_j) with $s_j \in \mathcal{S}$ using the same skipping techniques to probe the inverted index for collection $\mathcal{S}$.
- For the answer-level skipping technique, we precompute the similar pairs $(r_i, r_{i'})$ in collection $\mathcal{R}$. In join processing, for each record $r_i \in \mathcal{R}$, after computing $\mathcal{A}(r_i)$, for each $r_{i'}$ that is similar to r_i, we can use the same answer-level skipping technique and cost estimation function to compute $\mathcal{A}(r_{i'})$ incrementally.

B.2 Handling Cosine Similarity

Our algorithm can be easily extended to handle the cosine similarity function. Specifically, the three filters in Equation (3.3), Equation (3.4), and Equation (3.5) are replaced by the following three filters respectively:

- **Prefix Filter:** We only need to probe $I(e)$ for $e \in \mathsf{pre}_{t_i}(r_i)$, where

$$t_i = \lfloor (1 - \tau) \cdot |r_i| \rfloor + 1 \tag{B.1}$$

- **Length Filter:** For each $e \in \mathsf{pre}_{t_i}(r_i)$, we only need to visit those r_j ($j > i$) with $|r_j| \leq u_i(e)$, where

$$u_i(e) = \lfloor \frac{(|r_i| - \mathsf{pos}_e(r_i) + 1)^2}{|r_i| \cdot \tau^2} \rfloor \tag{B.2}$$

- **Position Filter:** When visiting $r_j \in I(e)$ for $e \in \mathsf{pre}_{t_i}(r_i)$ and r_j has not been added to $C(r_i)$, (r_i, r_j) is a candidate pair if

$$|r_j| - \mathsf{pos}_e(r_j) + 1 \geq \lceil \sqrt{|r_i| \cdot |r_j|} \cdot \tau \rceil \tag{B.3}$$

In addition, the condition in line 15 of Algorithm 3 is replaced by $|r_j| > \lfloor \frac{|r_{i'}|}{\tau^2} \rfloor$, and the condition in line 28 of Algorithm 3 is replaced by $|r_i \cap r_j| + \delta(r_i, r_{i'}, r_j) \geq \lceil \sqrt{|r_{i'}| \cdot |r_j|} \cdot \tau \rceil$.

C Additional Experiments

C.1 Scalability Testing

In this subsection, we test the scalability of the algorithms PPJ, PPJ+, PEL, ADP, PTJ, and SKJ by varying the number of records and number of elements in six representative and large datasets. When varying the number of records, for each dataset, we randomly select 20%, 40%, 60%, 80%, and 100% of records in the original dataset, and conduct the experiments on the sampled datasets. When varying the number of elements, for each dataset, we randomly select 20%, 40%, 60%, 80%, and 100% of elements in $\mathcal{E}$, and for each record, we only keep those selected elements in the record, and then we conduct the experiments on the newly generated datasets.

Varying Number of Records. In this experiment, we vary the ratio of the number of selected records from 20% to 100% for each dataset, and test the efficiency of the six algorithms. Figure C.1 shows the experimental results for the Jaccard similarity. We can see that the running time of all algorithms increases stably as the number of record increases for all datasets. This is because when the number of records increases, the cost of both filtering and verification phase increases for all algorithms. Our algorithm SKJ consistently outperforms other algorithms in all cases.

Varying Number of Elements. In this experiment, we vary the ratio of the number of elements from 20% to 100% for each dataset, and test the efficiency of the six algorithms. The experimental results for the Jaccard similarity are shown in Figure C.2. We observe that when the number of elements increases, the processing time of all algorithms tend to increase because a larger number of elements may lead to a higher computational cost to process the elements. However, in some cases, when the number of elements increases, the processing time decreases for all algorithms. For example, on dataset LIVEJ, when we increase the ration of the number of elements from 40% to 60%, the running time of PPJ, PPJ+, PEL, ADP, PTJ, and SKJ decreases from 29.6 seconds, 39.1 seconds, 30.7 seconds, 22.4 seconds, 45.6 seconds, and 3.5 seconds to 7.4 seconds, 9.4 seconds, 7.9 seconds, 11.1 seconds, 20.5 seconds, and 2.1 seconds respectively. This is because when the number of elements decreases, it is possible that more similar record pairs are generated, which may increase the processing time of each algorithm. In all tests, our algorithm SKJ consistently outperforms other algorithms.

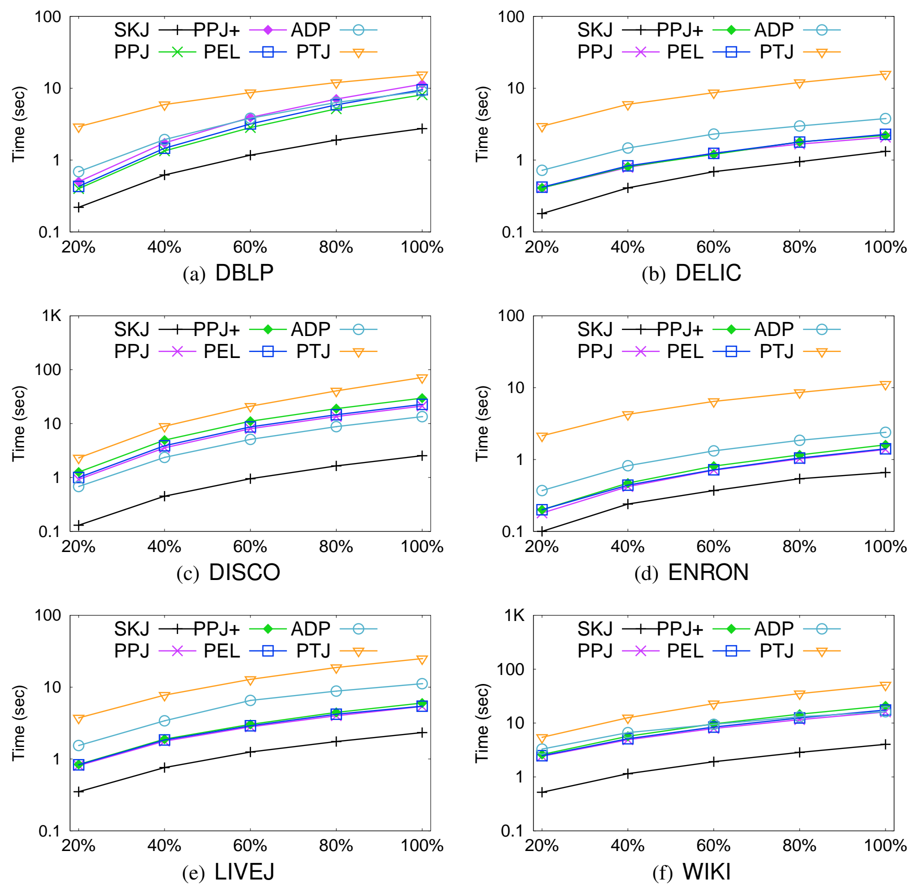


Figure C.1: Vary # Records (Jaccard Similarity)

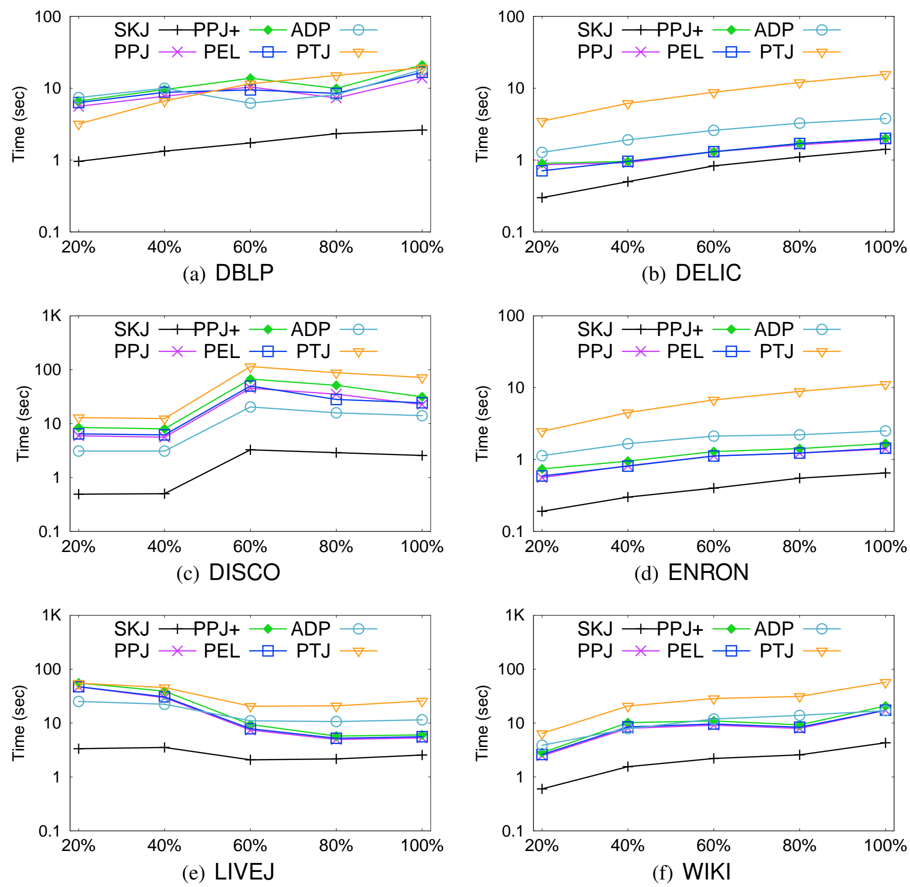


Figure C.2: Vary # Elements (Jaccard Similarity)