Efficient Similarity Join and Search on Multi-Attribute Data

Guoliang Li† Jian He† Dong Deng† Jian Li‡
†Department of Computer Science, Tsinghua University, Beijing, China
‡Institute for Interdisciplinary Information Sciences, Tsinghua University, Beijing, China
{liguoliang, lijian83}@tsinghua.edu.cn; {hejian13, dd11}@mails.tsinghua.edu.cn

ABSTRACT

In this paper we study similarity join and search on multi-attribute data. Traditional methods on single-attribute data have pruning power only on single attributes and cannot efficiently support multi-attribute data. To address this problem, we propose a prefix tree index which has holistic pruning ability on multiple attributes. We propose a cost model to quantify the prefix tree which can guide the prefix tree construction. Based on the prefix tree, we devise a filter-verification framework to support similarity search and join on multi-attribute data. The filter step prunes a large number of dissimilar results and identifies some candidates using the prefix tree and the verification step verifies the candidates to generate the final answer. For similarity join, we prove that constructing an optimal prefix tree is NP-complete and develop a greedy algorithm to achieve high performance. For similarity search, since one prefix tree cannot support all possible search queries, we extend the cost model to support similarity search and devise a budget-based algorithm to construct multiple high-quality prefix trees. We also devise a hybrid verification algorithm to improve the verification step. Experimental results show our method significantly outperforms baseline approaches.

Categories and Subject Descriptors
H.2 [Database Management]: Database applications;
H.3.3 [Information Search and Retrieval]: Search process

Keywords
Similarity Search; Similarity Join; Multi-Attribute Data

1. INTRODUCTION

Real-world data is rather dirty due to typographical errors and different representations of the same entity [11]. Traditional exact-matching join and search operations cannot tolerate the dirty data and thus similarity join and search are recently proposed to tolerate the dirty data. Given two multi-attribute tables (e.g., products and movies), similarity join finds all *similar* pairs from the two tables, where the similarity can be quantified by similarity functions, e.g., edit distance and Jaccard. Given a table with multiple attributes and a query, similarity search finds the records from the table that are similar to the query. Similarity join and search are two important operations in data cleaning and integration and have many real-world applications, e.g., duplicate detection, data fusion, and spell checking [11]. For example, in the event of plane crash like Malaysia airlines flight MH17, there are many data sources about passengers on MH17, e.g., tables from Malaysia airlines, departure country, destination county, and passengers' countries. We require to integrate them to generate high-quality data. As another example, a user wants to search movies directed by "`James Cameron`" and acted by "`Arnold Schwarzenegger`" from a movie database with actors, directors, and movie names. However the user does not know the exact spelling of "`Arnold Schwarzenegger`" and inputs a query with typos. Obviously exact-matching search cannot find any results while similarity search can alleviate this problem and help the user to find the relevant answers.

Existing algorithms on string similarity join [20,26,28,29] and search [9,14,15,22,30] have pruning power only on single attributes. For similarity join and search with constraints on multiple attributes, they have to first use a single attribute to identify candidates and then check whether the candidates satisfy the constraints on other attributes. Obviously these algorithms are rather expensive because they have no pruning power on other attributes and generate many intermediate results. It calls for effective methods to support holistic similarity join and search on multi-attribute data.

To address this problem, we propose a prefix tree index which has holistic pruning power on multiple attributes. Based on the prefix tree, we devise a filter-verification framework. The filter step prunes a large number of dissimilar results and identifies some candidates using the prefix tree and the verification step verifies the candidates to generate the final answer. For similarity join, we propose a cost model to quantify the prefix tree. We prove that constructing an optimal prefix tree is NP-complete and we develop a greedy algorithm to achieve high performance. Different from similarity join, one prefix tree cannot support all possible search queries with different similarity functions and thresholds. To address this issue, we extend the cost model to support similarity search. We develop a budge-based algorithm to build high-quality prefix trees to achieve high search performance. Since the similarity join and search queries contain multiple attributes, the verification order on attributes has a significant effect on the verification performance. In addition, many filtering algorithms can be used

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

http://dx.doi.org/10.1145/2723372.2723733.

to verify the candidates which also have a great effect on the performance. To this end, we devise a hybrid algorithm to improve the verification performance. To summarize, we make the following contributions. (1) We propose a prefix tree index which has holistic pruning power on multiple attributes and can be utilized to support both similarity join and search queries. (2) For similarity join, we develop a cost model to quantify the prefix tree. We prove that constructing an optimal prefix tree is NP-complete and we develop a greedy algorithm to construct a high-quality prefix tree (see Section 3). (3) For similarity search, we devise a budget-based method to construct multiple high-quality prefix trees to support similarity search queries (see Section 4). (4) We propose a hybrid verification algorithm to improve the verification performance (see Section 5). (5) Experimental results on real-world datasets show our method significantly outperforms baseline approaches (see Section 6).

2. PRELIMINARIES

We first formally define the problem in Section 2.1 and then briefly introduce a well-known technique – prefix filter in Section 2.2. Lastly, we review related works in Section 2.3.

2.1 Problem Definition

Consider two multi-attribute tables $\mathcal{R}$ and $\mathcal{S}$. Table $\mathcal{R}$ has x attributes $\mathcal{R}^1, \mathcal{R}^2, \cdots, \mathcal{R}^x$ and m rows $r_1, r_2, \cdots, r_m$. Table $\mathcal{S}$ has y attributes $\mathcal{S}^1, \mathcal{S}^2, \cdots, \mathcal{S}^y$ and n rows $s_1, s_2, \cdots, s_n$. Let $r_p = [r_p^1, r_p^2, \cdots, r_p^x]$ where r_p^i denotes the value of r_p on attribute $\mathcal{R}^i$, and $s_q = [s_q^1, s_q^2, \cdots, s_q^y]$ where s_q^j denotes the value of s_q on attribute $\mathcal{S}^j$.

An atomic similarity join operation $\mathcal{R}^i \sim \mathcal{S}^j$ returns all similar pairs $\{(r_p \in \mathcal{R}, s_q \in \mathcal{S})\}$ such that $r_p^i \sim s_q^j$, where $\sim$ is a similarity operator which will be defined later.

A complex similarity join operation is composed by more than one atomic similarity join operation, e.g., $\Phi = \mathcal{R}^{i_1} \sim \mathcal{S}^{j_1} \wedge \mathcal{R}^{i_2} \sim \mathcal{S}^{j_2} \wedge \cdots \wedge \mathcal{R}^{i_k} \sim \mathcal{S}^{j_k}$, which finds all similar pairs $\{(r_p \in \mathcal{R}, s_q \in \mathcal{S})\}$ such that $r_p^{i_1} \sim s_q^{j_1} \wedge r_p^{i_2} \sim s_q^{j_2} \wedge \cdots \wedge r_p^{i_k} \sim s_q^{j_k}$ where $\forall t \in [1,k]$, $i_t \in [1,x]$ and $j_t \in [1,y]$. We call $\Phi_t = \mathcal{R}^{i_t} \sim \mathcal{S}^{j_t}$ a predicate or attribute if the context is clear.

We utilize similarity measures to define the similarity operator $\sim$, which can be broadly classified into two categories, character-based similarity and token-based similarity.

Character-based Similarity. It defines the similarity between two strings based on character transformations. The well-known character-based similarity is edit distance. Consider two string values r_p^i and s_q^j. The edit distance $\mathsf{ED}(r_p^i, s_q^j)$ is the minimum number of single-character edit operations (including insertion, deletion and substitution) needed to transform r_p^i to s_q^j. The edit similarity extends edit distance by taking into account the string length, which is defined as $\mathsf{ES}(r_p^i, s_q^j) = 1 - \frac{\mathsf{ED}(r_p^i, s_q^j)}{\max(|r_p^i|, |s_q^j|)}$, where $|r_p^i|$ is the length of r_p^i. Two strings r_p^i and s_q^j are similar with respect to edit similarity if their similarity exceeds a given threshold τ, i.e.,

$$r_p^i \overset{\mathsf{ES},\tau}{\sim} s_q^j \text{ iff. } \mathsf{ES}(r_p^i, s_q^j) \geq \tau.$$

For example, given the Name attribute of table $\mathcal{R}$ in Figure 1, r_1^1 = '`Jeffery Ullman`', r_4^1='`Jeffer Ullman`'. We have $\mathsf{ED}(r_1^1, r_4^1) = 1$ and $\mathsf{ES}(r_1^1, r_4^1) = 1 - \frac{1}{14} = \frac{13}{14}$.

Token-based Similarity. It first splits each string into a set of tokens by white spaces and then utilizes the set-based similarity functions to quantify the similarity. The well-known similarity functions include Overlap and Jaccard, defined as below (see Appendix B for more functions).

- Overlap similarity: $\mathsf{OLP}(r_p^i, s_q^j) = |r_p^i \cap s_q^j|$
- Jaccard similarity: $\mathsf{JAC}(r_p^i, s_q^j) = \frac{|r_p^i \cap s_q^j|}{|r_p^i \cup s_q^j|}$

where $|r_p^i|$ denotes the size of set r_p^i.

r_p^i and s_q^j are similar with respect to the token-based similarity if their similarity exceeds a threshold τ, e.g.,

$$r_p^i \overset{\mathsf{JAC},\tau}{\sim} s_q^j \text{ iff. } \mathsf{JAC}(r_p^i, s_q^j) \geq \tau.$$

For example, given the Address attribute of table $\mathcal{R}$ in Figure 1, r_3^2 ='`CS Berkeley`' and r_4^2 ='`CS Stanford CA`'. $\mathsf{OLP}(r_3^2, r_4^2) = 1$ and $\mathsf{JAC}(r_3^2, r_4^2) = \frac{1}{4}$.

Traditional similarity join and search methods focus on atomic operations, while in this paper we emphasize on complex operations. Next, we formalize the problems of string similarly join and search on multi-attribute data.

Definition 1. (Similarity Join on Multi-Attribute Data) *Given two multi-attribute tables $\mathcal{R}$ and $\mathcal{S}$ and a complex similarity operation $\Phi = \mathcal{R}^{i_1} \sim \mathcal{S}^{j_1} \wedge \mathcal{R}^{i_2} \sim \mathcal{S}^{j_2} \wedge \cdots \wedge \mathcal{R}^{i_k} \sim \mathcal{S}^{j_k}$, it finds all similar pairs $\{(r_p \in \mathcal{R}, s_q \in \mathcal{S})\}$ such that $r_p^{i_1} \sim s_q^{j_1} \wedge r_p^{i_2} \sim s_q^{j_2} \wedge \cdots \wedge r_p^{i_k} \sim s_q^{j_k}$.*

Definition 2. (Similarity Search on Multi-Attribute Data) *Given a multi-attribute table $\mathcal{R}$ and a query $\mathcal{Q} = (\mathcal{R}^{i_1} \sim \mathcal{Q}^{i_1}, \mathcal{R}^{i_2} \sim \mathcal{Q}^{i_2}, \cdots, \mathcal{R}^{i_k} \sim \mathcal{Q}^{i_k})$, where $\mathcal{Q}^{i_t}$ denotes the query value on attribute $\mathcal{R}^{i_t}$ for $t \in [1,k]$, it finds similar strings $r_p \in \mathcal{R}$ such that $r_p^{i_1} \sim \mathcal{Q}^{i_1} \wedge r_p^{i_2} \sim \mathcal{Q}^{i_2} \wedge \cdots \wedge r_p^{i_k} \sim \mathcal{Q}^{i_k}$.*

For example, given two multi-attribute tables $\mathcal{R}$ and $\mathcal{S}$ in Figure 1 and a complex operation $\mathcal{R}^1 \overset{\mathsf{ES},0.8}{\sim} \mathcal{S}^1 \wedge \mathcal{R}^2 \overset{\mathsf{JAC},0.5}{\sim} \mathcal{S}^2$. The first predicate is $\mathsf{ES}(r_p^1, s_q^1) \geq 0.8$ and the second is $\mathsf{JAC}(r_p^2, s_q^2) \geq 0.5$. Similarity join on multi-attribute data returns $\{(r_4, s_1)\}$ as $\mathsf{ES}(r_4^1, s_1^1) = 0.93 \geq 0.8$ and $\mathsf{JAC}(r_4^2, s_1^2) = 0.67 \geq 0.5$. Although $r_1^1 = s_1^1$, (r_1, s_1) is not a similar pair, as their Address values are not similar ($\mathsf{JAC}(r_1^2, s_1^2) = 0 \leq 0.5$).

Given a multi-attribute table $\mathcal{R}$ in Figure 1 and a query $\mathcal{Q} = (\mathcal{R}^1 \overset{\mathsf{ES},0.8}{\sim}$ '`Jenifer Widom`', $\mathcal{R}^2 \overset{\mathsf{OLP},2}{\sim}$ '`CS Stanford`'). The first predicate is $\mathsf{ES}(r_p^1, \mathcal{Q}^1) \geq 0.8$ and the second is $\mathsf{OLP}(r_p^2, \mathcal{Q}^2) \geq 2$. Similarity search on multi-attribute data returns $\{r_2\}$ as $\mathsf{ES}(r_2^1, \mathcal{Q}^1) = 0.93 \geq 0.8$ and $\mathsf{OLP}(r_2^2, \mathcal{Q}^2) = 2 \geq 2$.

$\mathcal{R}$: **Customer records of BOA**

Id	Name ($\mathcal{R}^1$)	Address ($\mathcal{R}^2$)	Country ($\mathcal{R}^3$)
r_1	Jeffery Ullman	EE UCLA	USA
r_2	Jennifer Widom	CS Stanford CA	USA
r_3	Jerry Wang	CS Berkeley	ENG
r_4	Jeffer Ullman	CS Stanford CA	USA
r_5	Don Antenio	CS Stanford CA	USA

$\mathcal{S}$: **Customer records of Wells Fargo**

Id	Name ($\mathcal{S}^1$)	Address ($\mathcal{S}^2$)	Country ($\mathcal{S}^3$)
s_1	Jeffery Ullman	Stanford CA	USA
s_2	Herry Wang	Berkeley CA	ENG
s_3	Don Ertenio	CS Berkeley	ENG

Figure 1: Two example tables: $\mathcal{R}$ and $\mathcal{S}$.

2.2 Prefix Filter

We briefly introduce the prefix filter [1,2]. It first transforms each string to a set. For edit distance, the set contains all q-grams of the string where a q-gram is a substring with length q. For token-based similarity, the set contains all tokens in the string. Then it converts character-based and token-based similarities to the overlap similarity: if string s is similar to string r, the overlap of their corresponding sets must exceed o, where o can be deduced as below.

- If $\mathsf{OLP}(r, s) \geq \tau$ then $o = \tau$.

Name							Country	
e_1^1	e_2^1	e_3^1	e_4^1	e_5^1	e_6^1	...	e_1^3	e_2^3
an	do	ef	en	er	fe	...	ENG	USA

Address					
e_1^2	e_2^2	e_3^2	e_4^2	e_5^2	e_6^2
Berkeley	CA	CS	EE	Stanford	UCLA

Prefix Table for $\mathcal{R}$ and $\mathcal{S}$

	Name	Address	Country
r_1	$\{e_1^1, e_3^1, e_5^1\}$	$\{e_4^2, e_6^2\}$	$\{e_2^3\}$
r_2	$\{e_2^1, e_4^1, e_5^1\}$	$\{e_2^2, e_3^2, e_5^2\}$	$\{e_2^3\}$
r_3	$\{e_1^1, e_5^1, e_{11}^1\}$	$\{e_1^2, e_3^2\}$	$\{e_1^3\}$
r_4	$\{e_1^1, e_3^1, e_5^1\}$	$\{e_2^2, e_3^2, e_5^2\}$	$\{e_2^3\}$
r_5	$\{e_1^1, e_2^1, e_4^1\}$	$\{e_2^2, e_3^2, e_5^2\}$	$\{e_2^3\}$
s_1	$\{e_1^1, e_3^1, e_5^1\}$	$\{e_5^2, e_2^2\}$	$\{e_2^3\}$
s_2	$\{e_1^1, e_5^1, e_{10}^1\}$	$\{e_1^2, e_2^2\}$	$\{e_1^3\}$
s_3	$\{e_2^1, e_4^1, e_5^1\}$	$\{e_1^2, e_3^2\}$	$\{e_1^3\}$

Figure 2: Prefix Tokens for $\mathcal{R}$ and $\mathcal{S}$ in Figure 1 (Use Dictionary Order).

- If $\mathsf{JAC}(r,s) \geq \tau$ then $o = \lceil \tau |r| \rceil$.
- If $\mathsf{ED}(r,s) \leq \tau$ then $o = |r| - q\tau$.
- If $\mathsf{ES}(r,s) \geq \tau$, then $o = \lceil |r| - (|r| + q - 1)\frac{(1-\tau)}{\tau}|r| \rceil$.

where $|r|$ is the size of r(for ED/ES, r is the q-gram set).

For simplicity, we use string and set interchangeably if the context is clear and each element in the set is called a token. The prefix filter fixes a global order for the tokens and sorts tokens of each set by this global order. Finally for each set r, it selects the first $|r| - o + 1$ tokens as its prefix, denoted as $\mathtt{pre}(r)$. It is easy to prove that if two strings r and s are similar, $\mathtt{pre}(r) \cap \mathtt{pre}(s) \neq \phi$ [29].

For example, given two multi-attribute tables $\mathcal{R}$ and $\mathcal{S}$ in Figure 1 and a complex similarity operation $\mathcal{R}^1 \overset{\mathsf{ED},1}{\sim} \mathcal{S}^1 \wedge \mathcal{R}^2 \overset{\mathsf{JAC},0.3}{\sim} \mathcal{S}^2 \wedge \mathcal{R}^3 \overset{\mathsf{OLP},1}{\sim} \mathcal{S}^3$. For attributes $\mathcal{R}^1$ and $\mathcal{S}^1$, we generate 2-gram set. For simplicity, we use the dictionary order as a global order. For each attribute of each record, we compute its prefix and generate a table of prefixes as shown in Figure 2. We use $\mathcal{R}^2 \overset{\mathsf{JAC},0.3}{\sim} \mathcal{S}^2$ to demonstrate how to compute the prefix. The prefix length for $\mathsf{JAC}(r_p^2, s_q^2) \geq 0.3$ is $|\mathtt{pre}(r_p^2)| = |r_p^2| - \lceil |r_p^2|\tau \rceil + 1$. So for r_2, $|\mathtt{pre}(r_1^2)| = 2$ and $\mathtt{pre}(r_1^2) = \{\mathtt{EE}, \mathtt{UCLA}\} = \{\mathsf{e}_4^2, \mathsf{e}_6^2\}$.

2.3 Related Works

Similarity Join on Single Attribute. Since the prefix filter is effective, many methods are proposed to optimize it for different similarity operators [6,16,20,24,26,28,29]. ED-join [12,28] proposed a location-based mismatch filter to reduce prefix length and a content-based mismatch filter to reduce the number of candidates for ED. Pivotal prefix filter [4] reduced the prefix length for ED. `PassJoin` [17] proposed segment filter to improve pruning power. `PPJoin` [29] used the positions of prefix and suffix to improve pruning power for token-based similarities. Length filter was proposed to prune dissimilar answers based on length difference [8]. `TrieJoin` [23] used a different framework that directly computed real similarity using the trie structure.

Similarity Search on Single Attribute. Li et al. [14] proposed a list-merge framework that converts other similarities to overlap, builds inverted index, and merges inverted lists of the query tokens to identify answers. Li et al. [15] proposed variable length q-grams (VGram) to improve the pruning power. Zhang et al. [30] proposed B^{ed}-tree which uses q-grams as signatures and indexes the q-grams using the B-tree. Deng et al. [5] addressed the top-k similarity search problem with the trie structure. Hadjieleftheriou et al. [9] devised a technique to support data updates in similarity search. Wang et al. [25] proposed a dynamic prefix length scheme for effective filtering in similarity search.

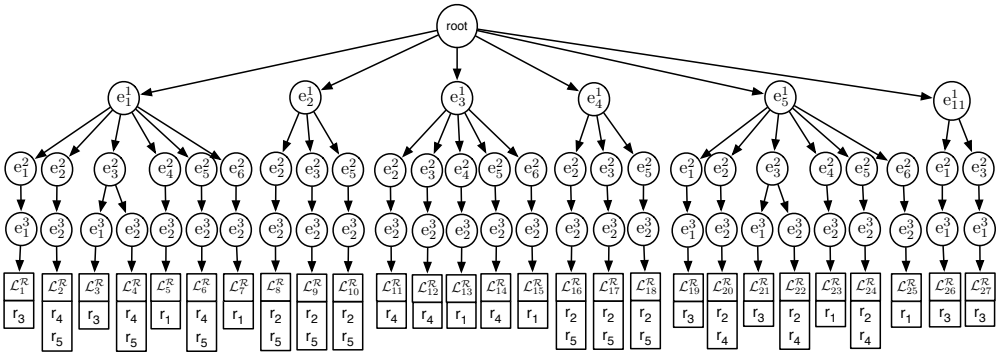


Figure 3: Complete Prefix Tree for $\mathcal{R}$ in Figure 2.

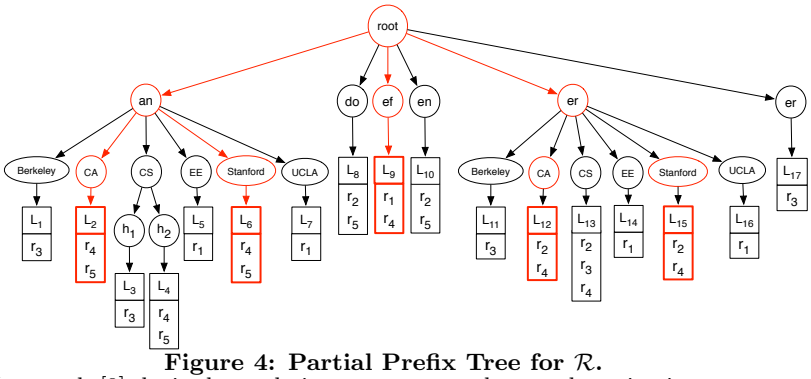


Figure 4: Partial Prefix Tree for $\mathcal{R}$.

Blocking-based Join Algorithm. There are many blocking-based join algorithms on entity resolution [11,13,19,27]. To link the records, they defined some rules (e.g., the same zip code leads to the same entity), utilized the rules to generate blocks (e.g., records with the same zip code will be in the same block), and took records in the same block as candidates. Different from the blocking-based algorithms [3,18] that focused on learning blocking schemes to improve the recall, our paper emphasizes on devising effective indexes and efficient algorithms to improve the performance for given fuzzy-matching rules. Sarma et al. [21] proposed a blocking-tree structure and devised a bottom-up algorithm to improve the recall. Our method on similarity joins has the following differences. First, the objectives are different. They merged blocks by enumerating every pair of leaf nodes to improve the recall while we merged leaf nodes with the same parent to reduce the candidate sizes to improve the efficiency. The high enumeration overhead is not acceptable in our problem. Second, the strategies of building tree structures are different. The inverted lists on our tree have overlaps while their blocks are disjoint. Third, the bottom-up algorithm is still expensive for our problem, because it checks many tree nodes to determine whether to combine. To further improve the performance, we devise a greedy top-down algorithm.

3. SIMILARITY JOIN WITH PREFIX TREE

Given a complex similarity operation $\Phi = \mathcal{R}^{i_1} \sim \mathcal{S}^{j_1} \wedge \mathcal{R}^{i_2} \sim \mathcal{S}^{j_2} \wedge \cdots \wedge \mathcal{R}^{i_k} \sim \mathcal{S}^{j_k}$, we devise a filter-verification framework to answer the join query. The filter step identifies the candidate pairs $\langle r, s \rangle$ such that $\mathtt{pre}(r^{i_t}) \cap \mathtt{pre}(s^{j_t}) \neq \phi$ for every $t \in [1, k]$ and the verification step verifies the candidate paris by computing the real similarity on each attribute $\mathcal{R}^{i_t}$ and $\mathcal{S}^{j_t}$. We study the filter step in this section and the verification algorithm will be discussed in Section 5.

Algorithm 1: PREFIXTREE-JOIN ($\mathcal{R}$, $\mathcal{S}$, Φ)

Input: $\mathcal{R}$, $\mathcal{S}$: Two multi-attribute tables
Φ: A complex similarity operation
Output: $\mathcal{A}$: Answer

```
1 Build one prefix tree with two tables R and S;
2 for each leaf node do
3     if there are two inverted lists L^R and L^S then
4         for each (r, s) ∈ L^R × L^S do
5             VERIFY(r, s);
6             if r is similar to s then
7                 Add ⟨r, s⟩ to A;
```

3.1 Prefix Tree

To efficiently identify the candidates, we build a *complete prefix tree*. Given a similarity operation Φ, we first sort the predicates in Φ. Suppose the sorted predicates are $\mathcal{R}^{i_1} \sim \mathcal{S}^{j_1} \wedge \mathcal{R}^{i_2} \sim \mathcal{S}^{j_2} \wedge \cdots \wedge \mathcal{R}^{i_k} \sim \mathcal{S}^{j_k}$ (the details on sorting the attributes will be discussed in Section 3.2.3). For simplicity, we first discuss how to construct a complete prefix tree for $\mathcal{R}$ with the attribute order $\mathcal{R}^{i_1}, \mathcal{R}^{i_2}, \cdots, \mathcal{R}^{i_k}$ as follows. For each record $r \in \mathcal{R}$, each prefix token combination $\langle e^1, e^2, \cdots, e^k \rangle$ where $e^t \in \texttt{pre}(r^{i_t})$ corresponds to a path from the root to a leaf node, and each token e^t corresponds to a tree node. At the leaf node, we keep an inverted list of records that contain this token combination. For example, Figure 3 shows the complete prefix tree for $\mathcal{R}$ in Figure 2.

Next we discuss how to utilize the complete prefix tree to support similarity joins. We extend the complete prefix tree to support two tables, where two inverted lists on each leaf node l are maintained, $\mathcal{L}_l^{\mathcal{R}}$ for $\mathcal{R}$ and $\mathcal{L}_l^{\mathcal{S}}$ for $\mathcal{S}$. For each record $r \in \mathcal{R}$, we append it to the inverted list $\mathcal{L}^{\mathcal{R}}$ of the corresponding leaf node, and for each record $s \in \mathcal{S}$, we append it to inverted list $\mathcal{L}^{\mathcal{S}}$. Obviously, for each leaf node l, the pair $(r, s) \in \mathcal{L}_l^{\mathcal{R}} \times \mathcal{L}_l^{\mathcal{S}}$ must be a candidate because r and s share a prefix token on every predicate. On the other hand, if (r, s) is a candidate, they must appear on the inverted lists of the same leaf node, because the complete prefix tree contains all prefix token combinations.

Algorithm 1 shows the pseudo-code. PREFIXTREE-JOIN first constructs one prefix tree for tables $\mathcal{R}$ and $\mathcal{S}$, and a given predicate order of the complex similarity operation Φ (line 1). On each leaf node, it maintains two inverted lists: $\mathcal{L}^{\mathcal{R}}$ for $\mathcal{R}$ and $\mathcal{L}^{\mathcal{S}}$ for $\mathcal{S}$. Then it identifies all leaf nodes, and for each leaf node, if there are two inverted lists (line 3), it enumerates the candidate pairs in these two lists (line 4). Next it verifies them (line 5), and if they are actually similar, it adds this pair to the result set (line 7).

For example, given two tables $\mathcal{R}$ and $\mathcal{S}$ with a complex similarity operation $\mathcal{R}^3 \overset{\mathsf{OLP},1}{\sim} \mathcal{S}^3 \wedge \mathcal{R}^2 \overset{\mathsf{JAC},0.3}{\sim} \mathcal{S}^2$. We first construct a complete prefix tree as shown in Figure 5. Then we enumerate all leaf nodes to generate candidate pairs. Take the leaf node e_1^2 as an example. We add the pairs from its two inverted lists $\mathcal{L}_1^{\mathcal{R}} \times \mathcal{L}_1^{\mathcal{S}}$, i.e., $\{(r_3, s_2), (r_3, s_3)\}$, to the candidate set. Finally, there are 9 candidate pairs. If we use the brute-force enumeration, there are $3 \cdot 5 = 15$ candidate pairs. After verification, the results are $\{(r_4, s_1), (r_3, s_2)\}$.

However the complete prefix tree has a rather large index size (see space complexity in Appendix D), because it requires to enumerate every token combination for each record, especially for records with long prefixes. To address this issue, we propose a *partial prefix tree* which is a shrunk complete prefix tree. Different from complete prefix tree, we do not maintain every path. Instead, we select some subtrees and for each subtree, we shrink it as a leaf node, and merge the inverted lists of its leaf descendants as the inverted list of the new leaf node.

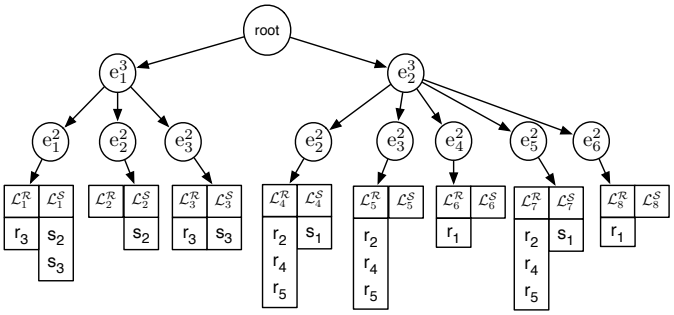


Figure 5: Prefix Tree for Tables $\mathcal{R}$ and $\mathcal{S}$ in Figure 2 ($\mathcal{L}^{\mathcal{R}}$($\mathcal{L}^{\mathcal{S}}$): inverted list for $\mathcal{R}$ ($\mathcal{S}$)).

For example, Figure 4 shows a partial prefix tree for $\mathcal{R}$ in Figure 2. Compared to the complete prefix tree in Figure 3, the partial prefix tree shrinks the subtrees rooted at $\{e_2^1, e_3^1, e_4^1, e_{11}^1\}$ and removes many unnecessary branches. Obviously the partial prefix tree is much smaller than the complete prefix tree since the partial prefix tree does not maintain every token combination. However there may be many possible partial prefix trees and next we discuss how to construct an optimal partial tree.

3.2 Optimal Prefix Tree

For simplicity, we use prefix tree and partial prefix interchangeably if the context is clear. To construct an optimal prefix tree, we first define a cost model to evaluate a prefix tree in Section 3.2.1. Then we discuss how to construct an optimal prefix tree with a specific predicate order in Section 3.2.2. Next, we prove that the problem of selecting the optimal predicate order is NP complete and propose a greedy algorithm in Section 3.2.3.

3.2.1 Join Cost Model with Prefix Tree

Since the join algorithm enumerates all pairs in the two inverted lists, our goal is to minimize the Cartesian product of the inverted lists. Next we define the cost model.

DEFINITION 3 (PREFIX TREE JOIN COST). *Given a prefix tree $\mathcal{T}$ built with two tables $\mathcal{R}$ and $\mathcal{S}$, the join cost is*

$$\Theta(\mathcal{T}) = \sum_{l \in \text{LEAF}(\mathcal{T})} |\mathcal{L}_l^{\mathcal{R}}||\mathcal{L}_l^{\mathcal{S}}| cost_v \quad (1)$$

where LEAF($\mathcal{T}$) *is the leaf node set of the prefix tree, $\mathcal{L}_l^{\mathcal{R}}$ and $\mathcal{L}_l^{\mathcal{S}}$ are respectively the inverted lists of $\mathcal{R}$ and $\mathcal{S}$ on the leaf node l and $cost_v$ is the average cost of verifying a candidate.*

For example, consider the prefix tree built with tables $\mathcal{R}$ and $\mathcal{S}$ in Figure 5. The join cost of prefix tree $\Theta(\mathcal{T}) = \sum_{i=1}^{8} |\mathcal{L}_i^{\mathcal{R}}||\mathcal{L}_i^{\mathcal{S}}| cost_v = 9 * cost_v$. [1]

Then we define the optimal prefix tree.

DEFINITION 4. (OPTIMAL PREFIX TREE WITH A PREDICATE ORDER) *Given a similarity operation with a predicate order and two tables $\mathcal{R}$ and $\mathcal{S}$, a partial prefix tree is an optimal prefix tree with the predicate order if it has the minimum join cost among all prefix trees with this predicate order.*

For example, Figure 6 shows an optimal prefix tree of the complete prefix tree in Figure 5.

[1] Our cost model does not consider the duplicate candidate pairs in different leaf nodes. For example, (r_4, s_1) will be counted 2 times under different leaves in Figure 5. In practice, we only verify a candidate pair once by utilizing a hash table to avoid the duplicate verification.

Algorithm 2: OPTIMAL-BOTTOMUP ($\mathcal{R}$, $\mathcal{S}$, Φ, π)

Input: $\mathcal{R}$, $\mathcal{S}$: Two multi-attribute data tables
Φ: A sorted complex similarity operation
Output: $\mathcal{T}$: Optimal prefix tree

1 Construct a complete prefix tree $\mathcal{T}$ with $\mathcal{R}$,$\mathcal{S}$ and Φ;
2 **for** *each node* $v \in \mathcal{T}$ *from bottom to top* **do**
3 **if** v *is a leaf* **then** $\Theta(v) = |\mathcal{L}_v^{\mathcal{R}}||\mathcal{L}_v^{\mathcal{S}}|$ **else**
4 $\mathcal{L}_v^{\mathcal{R}} = \cup_{c \in \text{CHILD}(v)} \mathcal{L}_c^{\mathcal{R}}$;
5 $\mathcal{L}_v^{\mathcal{S}} = \cup_{c \in \text{CHILD}(v)} \mathcal{L}_c^{\mathcal{S}}$;
6 **if** $\sum_{c \in \text{CHILD}(v)} \Theta(c) < |\mathcal{L}_v^{\mathcal{R}}||\mathcal{L}_v^{\mathcal{S}}|$ **then**
7 $\Theta(v) = \sum_{c \in \text{CHILD}(v)} \Theta(c)$;
8 **else**
9 $\Theta(v) = |\mathcal{L}_v^{\mathcal{R}}||\mathcal{L}_v^{\mathcal{S}}|$;
10 Mark v as leaf and remove its children;
11 **return** $\mathcal{T}$;

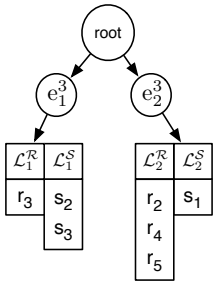


Figure 6: Optimal Prefix Tree for Tables $\mathcal{R}$ and $\mathcal{S}$.

3.2.2 Optimal Prefix Tree with A Predicate Order

Given a complex similarity operation with a specified order of predicates, we show how to construct an optimal prefix tree with the minimum join cost. The basic idea is that we first build a complete prefix tree and then determine if we combine some paths in the complete prefix tree in a bottom-up manner based on the join cost model. We devise a two-step algorithm to construct the optimal prefix tree as shown in Algorithm 2. In the first step, it constructs a complete prefix tree given a specific predicate order (line 1). In the second step, it revisits nodes from bottom to top to prune subtrees (line 3). When checking each node, it compares the cost of the subtree ($|\mathcal{L}_v^{\mathcal{R}}||\mathcal{L}_v^{\mathcal{S}}|$) and the cost of merging its children ($\sum_{c \in \text{CHILD}(v)} \Theta(c)$)[2]. If the latter is smaller, we keep its children (line 7); otherwise we merge its children(lines 9-10). As it visits nodes in a bottom-up manner, it guarantees the subtree rooted at each node is optimal.

For example, given two tables $\mathcal{R}$ and $\mathcal{S}$ in Figure 1, and a complex similarity operation with an order $\mathcal{R}^3 \overset{\text{OLP},1}{\sim} \mathcal{S}^3 \wedge \mathcal{R}^2 \overset{\text{JAC},0.3}{\sim} \mathcal{S}^2$. We first construct the complete tree as shown in Figure 5. Then we enumerate all the nodes from bottom to top to prune branches. Taking node e_1^3 as an example, the cost of its subtree is $|\mathcal{L}_1^{\mathcal{R}}||\mathcal{L}_1^{\mathcal{S}}| + |\mathcal{L}_2^{\mathcal{R}}||\mathcal{L}_2^{\mathcal{S}}| + |\mathcal{L}_3^{\mathcal{R}}||\mathcal{L}_3^{\mathcal{S}}| = 3$, which is larger than the merging cost $|\mathcal{L}_1^{\mathcal{R}} \cup \mathcal{L}_2^{\mathcal{R}} \cup \mathcal{L}_3^{\mathcal{R}}||\mathcal{L}_1^{\mathcal{S}} \cup \mathcal{L}_2^{\mathcal{S}} \cup \mathcal{L}_3^{\mathcal{S}}| = 2$. Thus we remove the children of e_1^3 and mark e_1^3 as a leaf node. Similarly, we prune the subtree of node e_2^3 and all nodes with an empty list and construct an optimal prefix tree as shown in Figure 6. The join cost with the optimal prefix tree is 5 while that of the complete tree is 9.

The OPTIMAL-BOTTOMUP algorithm can find the optimal tree correctly as shown in Lemma 1.

LEMMA 1. *The partial tree built by* OPTIMAL-BOTTOMUP *is an optimal prefix tree with the given predicate order.*

[2]For ease of presentation we omit the same coefficient $cost_v$

Algorithm 3: GREEDY-TOPDOWN ($\mathcal{R}$, $\mathcal{S}$, Φ)

Input: $\mathcal{R}$, $\mathcal{S}$: Two multi-attribute tables
Φ: A sorted complex similarity operation
Output: $\mathcal{T}$: A prefix tree

1 Initialize $\mathcal{F}$ with (root, Φ_1); /* Φ_1: the 1st operation*/
2 **for** *each* $(v, \Phi_i) \in \mathcal{F}$ **do**
3 Split v to generate children CHILD(v) with Φ_{i+1};
4 **if** $\sum_{c \in \text{CHILD}(v)} |\mathcal{L}_c^{\mathcal{R}}||\mathcal{L}_c^{\mathcal{S}}| < |\mathcal{L}_v^{\mathcal{R}}||\mathcal{L}_v^{\mathcal{S}}|$ **then**
5 Add all (c, Φ_{i+1}) to $\mathcal{F}$ for each $c \in$ CHILD(v);
6 **else**
7 Mark v as a leaf;
8 **return** $\mathcal{T}$;

Though Algorithm 2 can build the optimal prefix tree with a specified ordered complex similarity operation, its construction cost is expensive (see the complexity in Appendix D). To address this issue we propose a greedy algorithm to construct a partial prefix tree. Different from the optimal solution which directly builds the complete prefix tree, we construct a prefix tree in a top-down manner. For each node, we compare the cost of splitting the subtree rooted at it and that of not splitting the subtree. If the former is better, we split the node; otherwise we take it as a leaf node. The pseudo-code is shown in Algorithm 3. Given a complex operation Φ, it first initializes a queue $\mathcal{F}$ with the root and the first similarity predicate Φ_1 (line 1). Then for each record in the queue $\mathcal{F}$, it splits the node and generates its children (line 3). If $\sum_{c \in \text{CHILD}(v)} |\mathcal{L}_c^{\mathcal{R}}||\mathcal{L}_c^{\mathcal{S}}| < |\mathcal{L}_v^{\mathcal{R}}||\mathcal{L}_v^{\mathcal{S}}|$, it splits the node and adds its children into $\mathcal{F}$ (line 5); otherwise, it takes this node as a leaf node (line 7).

For example, consider the prefix tree in Figure 5. Given node e_3^1, the cost of splitting node e_3^1 is $|\mathcal{L}_1^{\mathcal{R}}||\mathcal{L}_1^{\mathcal{S}}| + |\mathcal{L}_2^{\mathcal{R}}||\mathcal{L}_2^{\mathcal{S}}| = 3$, and the cost of not splitting node e_3^1 is $|\mathcal{L}_1^{\mathcal{R}} \cup \mathcal{L}_2^{\mathcal{R}}||\mathcal{L}_1^{\mathcal{S}} \cup \mathcal{L}_2^{\mathcal{S}}| = 2$. Thus we do not split e_3^1.

3.2.3 Optimal Prefix Tree without A Predicate Order

Given a similarity operation with $|\Phi|$ atomic predicates, there are $|\Phi|!$ predicate order, and for each order, there is an optimal prefix tree. We would like to find the optimal prefix tree among these prefix trees.

DEFINITION 5. (OPTIMAL PREFIX TREE) *Given a similarity operation and two tables $\mathcal{R}$ and $\mathcal{S}$, a partial prefix tree is an optimal prefix tree if it has the minimum join cost among all partial prefix trees.*

However, the problem of finding the optimal prefix tree is NP-Hard as formalized in Lemma 2.

LEMMA 2. *Given two tables $\mathcal{R}$ and $\mathcal{S}$, a complex similarity operation Φ, finding the optimal prefix tree is NP-Hard.*

Since find the optimal prefix tree is NP-Hard, we propose an approximation algorithm. We first evaluate the cost of each predicate on the whole table as follows

$$\text{COST}(\Phi_i) = \sum_{c \in \text{CHILD}(root)} |\mathcal{L}_c^{\mathcal{R}}||\mathcal{L}_c^{\mathcal{S}}| \quad (2)$$

where Φ_i is the i^{th} operation of Φ. Then it sorts the predicates by COST(Φ_i) and constructs a prefix tree based on the sorted order. For example, given tables in Figure 1 and a complex operation $\mathcal{R}^1 \overset{\text{ED},1}{\sim} \mathcal{S}^1 \wedge \mathcal{R}^2 \overset{\text{JAC},0.3}{\sim} \mathcal{S}^2 \wedge \mathcal{R}^3 \overset{\text{OLP},1}{\sim} \mathcal{S}^3$. We first evaluate the cost of each predicate. The size of the inverted list built with Φ_3 is $|\mathcal{L}_1^{\mathcal{R}}| = 1, |\mathcal{L}_1^{\mathcal{S}}| = 2, |\mathcal{L}_2^{\mathcal{R}}| = 4, |\mathcal{L}_2^{\mathcal{S}}| = 1$. COST($\Phi_3$) $= \sum_{c=1}^{2} |\mathcal{L}_c^{\mathcal{R}}||\mathcal{L}_c^{\mathcal{S}}| = 6$. Similarly, we compute COST(Φ_2) $= 15$ and COST(Φ_1) $= 26$. As COST(Φ_3) $<$ COST(Φ_2) $<$ COST(Φ_1), the predicate order is $\langle \Phi_3, \Phi_2, \Phi_1 \rangle$.

4. SIMILARITY SEARCH WITH PREFIX TREE

We extend the prefix tree to support similarity search on multi-attribute data. We first give an overview in Section 4.1. Then we discuss how to utilize the prefix tree to answer a search query in Section 4.2. Finally we propose to build the prefix trees for similarity search in Section 4.3.

4.1 Overview

For a similarity join query, the cost of constructing a prefix tree is less than the join cost. However for a similarity search query, the construction cost is much more expensive than the cost of answering a search query. Thus for a similarity search query, we want to construct the prefix tree offline so as to utilize it to answer online search queries. To utilize prefix trees to support similarity search queries, we need to answer the following questions.

First, different search queries have various similarity functions and thresholds. For example, given the table in Figure 1, a user is familiar with the address but is not sure how to spell the name, and issues a query (NAME $\overset{\text{ES},0.6}{\sim}$ '`Jenifer Ullman`', ADDRESS $\overset{\text{JAC},0.8}{\sim}$ '`CS Stanford`'). Another user knows the name but is not familiar with the address, and issues a query (NAME $\overset{\text{ES},0.9}{\sim}$ '`Jeffery Ullman`', ADDRESS $\overset{\text{JAC},0.7}{\sim}$ '`EE Stanford`'). Obviously the two queries involve different functions and thresholds. Since the prefixes depend on the similarity functions and thresholds, can we still use prefix trees to support similarity search? The answer is yes. A common technique is to set a smallest possible similarity threshold that a system can tolerate, e.g., 0.6, and we utilize this threshold to construct prefix trees. An alternative is to build delta prefix trees. That is we build a prefix tree for each threshold range, e.g., $[1, 0.9]$, $(0.9, 0.8]$, $(0.8, 0.7]$, $(0.7, 0.6]$. Given a query threshold 0.8, we use the first two prefix trees to answer the query. For simplicity, we take the first method as an example. To support various functions, we transform them to the overlap similarity and use the smallest threshold o to generate the prefix (see Section 2.2).

Second, since there are many search queries and different queries involve different attribute combinations, a single prefix tree cannot efficiently answer all search queries and we have to construct multiple prefix trees to answer search queries. There are two issues we need to address. The first is how to construct multiple high-quality prefix trees to answer search queries? Since search queries are not given and different queries have different similarity functions and thresholds, the cost model for similarity joins cannot support search queries. To address this issue, we propose a new model to support similarity search in Section 4.3. The second is that given multiple prefix trees, how to efficiently utilize them to answer a search query? We propose efficient algorithms to address this issue in Section 4.2.

Based on these questions, we devise a framework to support similarity search queries on multi-attribute data. First we construct multiple prefix trees offline (see Section 4.3). Then given an online query, we devise efficient algorithms to answer the query using these prefix trees (see Section 4.2).

4.2 Prefix-Tree-Based Search Algorithms

Using A Single Prefix Tree $\mathcal{T}$ to Answer A Query $\mathcal{Q}$. If the first-level attribute of $\mathcal{T}$ appears in $\mathcal{Q}$, we can use $\mathcal{T}$ to answer $\mathcal{Q}$. The basic idea is to find the leaf nodes of $\mathcal{T}$ such that the tokens on the path from the root to each leaf also appear in the prefix tokens of $\mathcal{Q}$. The records on the inverted lists of leaf nodes are candidates. To this end, we first find the longest sequence of attributes from $\mathcal{T}$, i.e., $\mathcal{R}^{i_1}, \mathcal{R}^{i_2}, \cdots, \mathcal{R}^{i_t}$, where each attribute also appears in the query attributes. Then from the root, we identify its children with labels of prefix tokens in $\mathcal{Q}^{i_1}$. For each identified child, if it is a leaf, the records on inverted lists are candidates; otherwise we identify its children with labels of prefix tokens in $\mathcal{Q}^{i_2}$. Iteratively, we can find all leaf nodes.

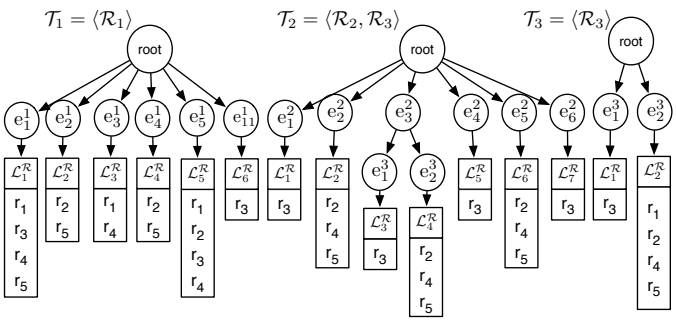


Figure 7: Mutiple Prefix Trees for Table $\mathcal{R}$.

Algorithm 4 shows the pseudo-code for using a prefix tree to answer $\mathcal{Q}$. Given a query $\mathcal{Q}$, SINGLEPREFIXTREESEARCH first recursively searches the prefix tree to get a set of leaves where each token on the path is also a prefix token of $\mathcal{Q}$ (line 1-9): it first identifies the longest sequence (line 1) and adds the root to the candidate node list (line 2); Then for each node, it checks whether its child contains a prefix token in $\mathcal{Q}$; If yes, this child will be added into the node list (line 4); Iteratively it finds all leaf nodes (line 9). Next it retrieves the inverted list of each identified leaf (line 11) and verifies each candidate based on the other similarity predicates (line 13). If a record is similar to the query, it is added into the result set (line 14).

For example, consider a prefix tree of $\mathcal{R}$ in Figure 4 and taking $s_1 \in \mathcal{S}$ as a search query $\mathcal{Q}$ with $\texttt{pre}(\mathcal{Q}^1) = \{e_1^1, e_3^1, e_5^1\}$, $\texttt{pre}(\mathcal{Q}^2) = \{e_2^2, e_5^2\}$, $\texttt{pre}(\mathcal{Q}^3) = \{e_2^3\}$. From the root, we find its children $\{e_1^1, e_3^1, e_5^1\}$ which contain a prefix token in $\texttt{pre}(\mathcal{Q}^1)$. e_3^1 is a leaf node and we get an inverted list $\mathcal{L}_9^{\mathcal{R}}$. Then we check whether e_1^1 and e_5^1 have children with labels e_5^2, e_2^2, and we find 4 leaf nodes whose inverted lists are $\{\mathcal{L}_2^{\mathcal{R}}, \mathcal{L}_6^{\mathcal{R}}, \mathcal{L}_{11}^{\mathcal{R}}, \mathcal{L}_{15}^{\mathcal{R}}\}$ (surrounded by red rectangles in Figure 3). From these five lists, we get 4 candidates $\{r_1, r_2, r_4, r_5\}$. Finally we verify candidates and get a result $\{r_4\}$.

Thus given a prefix tree $\mathcal{T}_i$, the time cost of using $\mathcal{T}_i$ to answer $\mathcal{Q}$, denoted by $\text{COST}(\mathcal{T}_i, \mathcal{Q})$, includes two parts. The first part is to identify the leaves and the second part is to verify the candidates, which can be computed as below.

$$\text{COST}(\mathcal{T}_i, \mathcal{Q}) = \Big(\prod_{\mathcal{R}^{j_k} \in \Psi(\mathcal{T}_i \cap \mathcal{Q})} |\texttt{pre}(\mathcal{Q}^{j_k})| \Big) + |\mathcal{C}_i| cost_v \quad (3)$$

where $\Psi(\mathcal{T}_i \cap \mathcal{Q})$ is the longest sequence of attributes in $\mathcal{T}_i$ and $\mathcal{Q}$, and $\mathcal{C}_i$ is the candidate set by searching $\mathcal{Q}$ in $\mathcal{T}_i$. $|\mathcal{C}_i|$ can be estimated by adding the size of all identified inverted lists which is very efficient.

Using Multiple Prefix Trees to Answer A Query. Next we study how to utilize multiple prefix trees to answer a search query. First we identify the candidate prefix trees that can answer the query $\mathcal{Q}$, by checking whether the attribute in the first level appears in $\mathcal{Q}$. Second, we rank these candidate prefix trees based on the cost of using the prefix tree to answer $\mathcal{Q}$, i.e., $\mathcal{T}_1, \mathcal{T}_2, \cdots$, where $\text{COST}(\mathcal{T}_1, \mathcal{Q}) \leq \text{COST}(\mathcal{T}_2, \mathcal{Q}) \leq \cdots$.

Algorithm 4: SinglePrefixTree-Search ($\mathcal{T}$, $\mathcal{Q}$, Φ)

Input: $\mathcal{T}$: Prefix tree
$\mathcal{Q}$: Search query
Φ: A complex similarity operation
Output: $\mathcal{A}$: Answer set

```
1  Identify the longest sequence of attributes
   R^{i_1}, R^{i_2}, ..., R^{i_t} of T that also appear in Q;
2  NodeSet = {root};
3  for j = 1 to t do
4      for node in NodeSet do
5          NodeSet' = φ;
6          for e^j in prefix of Q^{i_j} do
7              if node has a child n_j with label e^j then
8                  if n_j is a leaf then LeafSet ← n_j else
                       NodeSet' ← n_j
9          NodeSet = NodeSet';
10 for each leaf l ∈ LeafSet do
11     Retrive its inverted list L(l);
12     for each candidate c ∈ L(l) do
13         Verify (c, Q);
14         if c is simlar to Q then Add c to A;
```

Then we have two strategies to answer the query. The first one is that we use the prefix tree $\mathcal{T}_1$ with the minimum cost to generate a candidate set $\mathcal{C}_1$ and verify the candidates using other attributes. The second one is that we utilize another tree $\mathcal{T}_i$ to identify another candidate set $\mathcal{C}_i$ and intersect $\mathcal{C}_1$ and $\mathcal{C}_i$. Based on the new candidate set $\mathcal{C}_1 \cap \mathcal{C}_i$, we can still utilize these two strategies to compute the answer. Iteratively, we can find all the results.

Next we define a cost model to select a better strategy. The cost of the first strategy is

$$\textsc{Cost}_1 = \textsc{Cost}(\mathcal{T}_1, \mathcal{Q}). \tag{4}$$

The cost of the second strategy is

$$\textsc{Cost}_2 = \prod_{\mathcal{R}^{j_k} \in \Psi(\mathcal{T}_1 \cap \mathcal{Q})} |\texttt{pre}(\mathcal{Q}^{j_k})| + \prod_{\mathcal{R}^{j_k} \in \Psi(\mathcal{T}_i \cap \mathcal{Q})} |\texttt{pre}(\mathcal{Q}^{j_k})| + |\mathcal{C}_1| + |\mathcal{C}_i| + |\mathcal{C}_1 \cap \mathcal{C}_i| cost_v. \tag{5}$$

where $|\mathcal{C}_1 \cap \mathcal{C}_i|$ can be obtained through sampling.

If $\textsc{Cost}_1 \leq \textsc{Cost}_2$, we select the first strategy; otherwise we select the second strategy.

The pseudo code is shown in Algorithm 5. It first retrieves prefix trees that can answer $\mathcal{Q}$ from $\mathcal{W}$ (line 1). Then it estimates $\textsc{Cost}(\mathcal{T}_i, \mathcal{Q})$ and sorts $\mathcal{T}_i$ (line 2). It estimates the cost of the two methods (line 4-7). If the intersection-based method is better, it intersects the two candidates and iteratively decides to verify or intersect two candidate sets (line 9-11). Otherwise, it verifies the candidates (line 12).

4.3 Multiple Prefix-Trees Construction

The cost of searching prefix trees depends on two factors. The first is the query distribution, e.g. which attributes or attribute combinations are frequently searched. The second is data distribution, e.g. some attributes have more unique values than other attributes. Since we do not know the distribution of search queries, in this paper we only consider the data distribution.

As there are at least x prefix trees, we first build x trees using each attribute at the first level. We can use these trees to answer any query. If there is more space, we optimize

Algorithm 5: MultiTreeSearch ($\mathcal{R}$, $\mathcal{Q}$, Φ, $\mathcal{W}$)

Input: $\mathcal{R}$: Multi-attribute data table
$\mathcal{Q}$: Search query
$\mathcal{W}$: Multiple prefix trees $\mathcal{T}_1, \mathcal{T}_2..\mathcal{T}_m$
Output: $\mathcal{A}$: Answer set

```
1  W_Q ← {T_i | T_i ∈ W, R^{i_1} ∈ Q} ;
2  Sort T_i ∈ W_Q by Cost(T_i, Q);
3  A ← Candidates using T_1;
4  Cost_1 = Cost(T_1, Q) + |C_1| cost_v;
5  for each T_i ∈ W_Q do
6      Estimate Cost_2(T_i) by Equation(5) ;
7  Choose T_p with smallest Cost_2(T_p) ;
8  if Cost_2(T_p) < Cost_1 then
9      C_p ← Candidates using T_p;
10     A = A ∩ C_p;
11     Decide to verify A or further intersect;
12 else Verify all records in A;
13 return A;
```

these prefix trees and propose a cost-based method. Next we define the cost and benefit of adding an attribute $\mathcal{R}^j$ to current tree $\mathcal{T}_i$. The cost is the space increase by adding $\mathcal{R}^j$ to $\mathcal{T}_i$ and the benefit is query performance improvement.

As the new prefix tree shares many tree nodes with $\mathcal{T}_i$, we only consider the cost of new inverted lists by adding $\mathcal{R}^j$ to $\mathcal{T}_i$. As each record in $\mathcal{T}$ appears $|\texttt{pre}(r^{\mathcal{R}_j})|$ times in the new tree and thus the space cost can be computed as below.

$$\textsc{Cost}_{\textsc{Space}}(\mathcal{T}_i, \mathcal{R}^j) = \sum_{l \in \textsc{Leaf}(\mathcal{T}_i)} \sum_{r \in \mathcal{L}_l^{\mathcal{R}}} (|\texttt{pre}(r^{\mathcal{R}_j})|) \tag{6}$$

The benefit is hard to measure and different queries have different benefits. Here our objective is to avoid slow queries, i.e., there are no rather long inverted lists on the prefix tree. Thus we want to make inverted lists evenly distributed after adding $\mathcal{R}^j$. To achieve this goal, we use a modified Shannon entropy to capture this information.

Definition 6. *The entropy of prefix tree $\mathcal{T}$ in search is*

$$\mathcal{H}(\mathcal{T}) = \sum_{l \in \textsc{Leaf}(\mathcal{T})} -\mathcal{P}(l) \log \mathcal{P}(l) \tag{7}$$

where $\textsc{Leaf}(\mathcal{T})$ *is the leaf set of* $\mathcal{T}$ *and* $\mathcal{P}(l)$ *is the ratio of inverted-list size of node* l *to the sum of inverted-list sizes,*

$$\mathcal{P}(l) = \frac{|\mathcal{L}_l^{\mathcal{R}}|}{\sum_{l' \in \textsc{Leaf}(\mathcal{T}_i)} |\mathcal{L}_{l'}^{\mathcal{R}}|}. \tag{8}$$

Obviously, the larger entropy, the smaller average search cost of using prefix tree to answer a query. With this entropy cost model, we can define the benefit of adding $\mathcal{R}^j$,

$$\textsc{Benefit}(\mathcal{T}_i, \mathcal{R}^j) = \mathcal{H}(\mathcal{T}_i, \mathcal{R}^j) - \mathcal{H}(\mathcal{T}_i) \tag{9}$$

where $\mathcal{H}(\mathcal{T}_i, \mathcal{R}^j)$ denotes the entropy of the new tree by adding $\mathcal{R}^j$ to $\mathcal{T}_i$.

Using the cost and benefit, our goal is to achieve the maximum benefit with a given space budget $\mathcal{B}$. However, this generalizes the classical knapsack problem, which is NP-Hard. To this end, we devise a greedy algorithm to build the prefix trees. We select the attribute $\mathcal{R}^j$ and $\mathcal{T}_i$ with the largest value $\frac{\textsc{Benefit}(\mathcal{T}_i, \mathcal{R}^j)}{\textsc{Cost}_{\textsc{Space}}(\mathcal{T}_i, \mathcal{R}^j)}$, and insert the new tree into the tree set. The algorithm terminates until there is no space to accommodate any prefix tree. Algorithm 6 shows pseudo code. It first creates one-level prefix trees for each attribute (lines 1-2). Then it greedily adds a new prefix tree (lines 6-9). It terminates when meeting the space budget (line 6).

Algorithm 6: BUILDMULTIPLETREES ($\mathcal{R}$, $\mathcal{B}$)

Input: $\mathcal{R}$: Multi-attribute table
$\mathcal{B}$: Space budget
Output: $\mathcal{W}$: A set of prefix trees

```
for each attribute R^i do
    Construct T_i with R^i and add T_i to W;
while B > 0 do
    for each T_i ∈ W do
        for each R^j do
            Compute Cost_Space(T_i, R^j) with Equation 6;
            Compute Benefit(T_i, R^j) with Equation 9;
    Expand T_i using R^j with largest Benefit(T_i,R^j) / Cost_Space(T_i,R^j);
    W ← T(T_i, R^j);
    B ← B − Cost_Space(T_i, R^j) ;
return W ;
```

For example, consider adding an attribute to $\mathcal{T}_1$ in Figure 7. If we add attribute $\mathcal{R}^3$, the benifit $\text{BENEFIT}(\mathcal{T}_1, \mathcal{R}^3) = 0.298$ and the cost $\text{COST}_{\text{SPACE}}(\mathcal{T}_1, \mathcal{R}^3) = 4$. Similarly we can estimate $\text{BENEFIT}(\mathcal{T}_1, \mathcal{R}^2) = 1.057$ and $\text{COST}_{\text{SPACE}}(\mathcal{T}_1, \mathcal{R}^2) = 39$. $\frac{\text{BENEFIT}(\mathcal{T}_1,\mathcal{R}^3)}{\text{COST}_{\text{SPACE}}(\mathcal{T}_1,\mathcal{R}^3)} = 0.0745$ and $\frac{\text{BENEFIT}(\mathcal{T}_1,\mathcal{R}^2)}{\text{COST}_{\text{SPACE}}(\mathcal{T}_1,\mathcal{R}^2)} = 0.0271$, As $\frac{\text{BENEFIT}(\mathcal{T}_1,\mathcal{R}^3)}{\text{COST}_{\text{SPACE}}(\mathcal{T}_1,\mathcal{R}^3)} > \frac{\text{BENEFIT}(\mathcal{T}_1,\mathcal{R}^2)}{\text{COST}_{\text{SPACE}}(\mathcal{T}_1,\mathcal{R}^2)}$, we expend $\mathcal{T}_1$ with $\mathcal{R}^3$.

5. HYBRID VERIFICATION

In this section, we study how to efficiently verify a candidate pair. There are two challenges in verifying a pair. The first is to determine the order of verifying different predicates. The second is to decide the order of using different filtering algorithms. To address these challenges, we propose a hybrid verification framework and prove its optimality.

5.1 Hybrid-based Verification Framework

Given a complex similarity operation Φ and a candidate pair $\langle r, s\rangle$, we check whether $\langle r, s\rangle$ satisfies each predicate in Φ. A straightforward way is to examine each predicate one by one through computing the real similarity. However this method has two weaknesses. First, directly computing the similarity is expensive. For example, the cost to verify $\langle r, s\rangle$ on the token-based function is $|r| + |s|$ while the cost on the character-based function is $(2\tau + 1) \cdot \min(|r|, |s|)$. Second, the pruning power of each predicate is also different and it is important to select an appropriate order of predicates to verify candidates. For example, given a candidate pair $\langle r_1, s_1\rangle$ and a complex similarity operation $\mathcal{R}^1 \overset{\mathsf{ED},1}{\sim} \mathcal{S}^1 \wedge \mathcal{R}^2 \overset{\mathsf{JAC},0.3}{\sim} \mathcal{S}^2 \wedge \mathcal{R}^3 \overset{\mathsf{OLP},1}{\sim} \mathcal{S}^3$, if we verify them with the order $\langle \Phi_1, \Phi_3, \Phi_2\rangle$. The cost is $(3 \cdot 14) + (1 + 1) + (2 + 2) = 48$. However, if we use the order $\langle \Phi_2, \Phi_1, \Phi_3\rangle$, then the pair fails in predicate Φ_2, and thus the cost is only $2 + 2 = 4$.

To reduce the verification cost, we can use lightweight filters instead of computing the actual similarity. If the pair cannot pass the filter, we prune it; otherwise we use other filters or compute the actual similarity. For example, the length filter can prune those pairs with length difference larger than τ for edit distance. However different filters have different cost and pruning power and it is also important to select an appropriate order for using filtering algorithms.

Formally, suppose a similarity operation Φ has k similarity predicates $\Phi_1, \Phi_2, \cdots, \Phi_k$, v filters $\Gamma_1, \Gamma_2, \cdots, \Gamma_v$, and each predicate Φ_i also has a best verification algorithm Λ_i. There are many possible ways to verify the pair. A naive way that does not use any filter is $\langle \Phi_1, \Lambda_1\rangle, \langle \Phi_2, \Lambda_2\rangle, \cdots, \langle \Phi_k, \Lambda_k\rangle$.

Algorithm 7: HYBRIDVERIFICATION ($\mathcal{C}$, $\mathcal{Q}$, Φ, $\mathcal{F}$)

Input: $\mathcal{C}$: Candidate after prefix tree; $\mathcal{Q}$: Search query
Φ: A complex similarity operation,
$\mathcal{F}$: A set of filters and verification algorithms
Output: $\mathcal{A}$: Answer

```
1 Compute and sort verification methods by Θ(⟨Φ,Γ⟩) / P(⟨Φ,Γ⟩);
2 for each ⟨r, s⟩ ∈ C do
3     for each ⟨Φ, Γ⟩ do
4         if (r, s) does not pass ⟨Φ, Γ⟩ then
5             break;
6     Add ⟨r, s⟩ to A;
7     Update Θ(⟨Φ,Γ⟩) / P(⟨Φ,Γ⟩);
8 return A;
```

Another solution that uses all possible filters is $\langle \Phi_1, \Gamma_{11}\rangle$, $\langle \Phi_1, \Gamma_{12}\rangle, \cdots, \langle \Phi_1, \Lambda_1\rangle, \langle \Phi_2, \Gamma_{21}\rangle, \langle \Phi_2, \Gamma_{22}\rangle, \cdots, \langle \Phi_2, \Lambda_2\rangle$, $\langle \Phi_k, \Gamma_{k1}\rangle, \langle \Phi_k, \Gamma_{k2}\rangle, \cdots, \langle \Phi_k, \Lambda_k\rangle$. We require to find an order with the minimum cost to verify the pair. Suppose each filter $\langle \Phi_i, \Gamma_j\rangle$ (or each verification algorithm $\langle \Phi_i, \Lambda_i\rangle$) has a time cost Θ to preform the filter (or the algorithm) and a pruning probability $\mathcal{P}$ to prune the pair. We discuss how to compute the time cost and pruning ability in Section 5.2.

Given a verification sequence $\Psi = \{\langle \Phi_1, \Gamma_1\rangle, \langle \Phi_2, \Gamma_2\rangle, \cdots\}$, we compute its expected verification cost as follows.

$$\text{COST}(\Psi) = \sum_{\langle \Phi,\Gamma\rangle \in \Psi} \Theta(\langle \Phi, \Gamma\rangle) \cdot \mathcal{P}_\Psi(\langle \Phi, \Gamma\rangle), \qquad (10)$$

where $\Theta(\langle \Phi, \Gamma\rangle)$ is the cost to use the filter Γ to verify predicate Φ and $\mathcal{P}_\Psi(\langle \Phi, \Gamma\rangle)$ is the probability that $\langle \Phi, \Gamma\rangle$ is used to verify the pair given the sequence Ψ, that is the probability the verification methods before $\langle \Phi, \Gamma\rangle$ in Ψ cannot prune the pair, which can be computed as

$$\mathcal{P}_\Psi(\langle \Phi, \Gamma\rangle) = \prod_{\langle \Phi',\Gamma'\rangle \text{ is before } \langle \Phi,\Gamma\rangle} 1 - \mathcal{P}(\langle \Phi', \Gamma'\rangle), \qquad (11)$$

where $\mathcal{P}(\langle \Phi', \Gamma'\rangle)$ is the probability that the filter Γ' on Φ' can prune a pair.

We design a hybrid verification algorithm to address this problem. Algorithm 7 gives the pseudo code of the algorithm. It first computes $\frac{\Theta(\langle \Phi,\Gamma\rangle)}{\mathcal{P}(\langle \Phi,\Gamma\rangle)}$ for each pair of filter and attribute, and sorts them in an ascending order (line 1). Then it utilizes this order to verify each candidate pair. If the candidate passes the filter, it selects the next filter or verification algorithm; otherwise terminates (lines 2 - 5). Finally it adds the pair which passes all verification algorithms to the answer set (line 6). Then it updates the expected cost for other verification methods based on Equation 11 (line 7). Iteratively, the algorithm can find a verification order.

Note that the above greedy algorithm is known to be optimal (see e.g., [10]), and for the sake of completeness, we provide a simple proof of this fact in Appendix C.

For example, given a filter Γ_1, a verification algorithm Λ_2 and two similarity predicates Φ_1, Φ_2 with the following expected cost $\frac{\Theta(\langle \Phi,\Gamma\rangle)}{\mathcal{P}(\langle \Phi,\Gamma\rangle)}$. $\langle \Phi_1, \Gamma_1\rangle$: $\frac{1.0}{0.4} = 2.5$, $\langle \Phi_1, \Lambda_2\rangle$:$\frac{8.3}{0.9} = 9.2$, $\langle \Phi_2, \Gamma_1\rangle$: $\frac{1.0}{0.25} = 4.0$, $\langle \Phi_2, \Lambda_2\rangle$:$\frac{2.2}{0.6} = 3.7$. We sort verification methods by $\frac{\Theta(\langle \Phi,\Gamma\rangle)}{\mathcal{P}(\langle \Phi,\Gamma\rangle)}$ and get an order $\langle \Phi_1, \Gamma_1\rangle, \langle \Phi_2, \Lambda_2\rangle$, $\langle \Phi_2, \Gamma_1\rangle, \langle \Phi_1, \Lambda_2\rangle$. Then we compute $\mathcal{P}(\langle \Phi, \Gamma\rangle|\Psi)$ iteratively by Equation (11) and get a sequence $\mathcal{P}(\langle \Phi_1, \Gamma_1\rangle|\Psi) = 1$, $\mathcal{P}(\langle \Phi_2, \Lambda_2\rangle|\Psi) = 0.6, \mathcal{P}(\langle \Phi_2, \Gamma_1\rangle|\Psi) = 0.24, \mathcal{P}(\langle \Phi_1, \Lambda_2\rangle|\Psi) = 0.18$. Thus $\text{COST}(\Psi) = \sum_{\langle \Phi_i,\Gamma_j\rangle \in \Psi} \Theta(\langle \Phi_i, \Gamma_j\rangle) \cdot \mathcal{P}(\langle \Phi_i, \Gamma_j\rangle|\Psi) = 2.5 \cdot 1 + 2.2 \cdot 0.6 + 4.0 \cdot 0.24 + 8.3 \cdot 0.18 = 6.274$. If we choose order $\langle \Phi_1, \Lambda_2\rangle, \langle \Phi_2, \Gamma_1\rangle, \langle \Phi_2, \Lambda_2\rangle, \langle \Phi_1, \Gamma_1\rangle$, $\text{COST}(\Psi) = 8.64$.

5.2 Verification Cost and Pruning Probability

We first discuss the verification cost of various filters. Here we only take important filters as examples. It is worth noting that our method is orthogonal to the filtering techniques and our method can be easily extended to support any filter.

Length Filter. It prunes $\langle r, s\rangle$ if their length difference is larger than τ for ED and $|r|\frac{1-\tau}{\tau}$ for JAC. The time cost Θ is $\mathcal{O}(1)$ for the search problem, since the length/size can be materialized; but for the join problem, the cost is $\mathcal{O}(1)$ for the attribute in the similarity operation as we can get the length when computing the prefixes; $\mathcal{O}(|r|+|s|)$ otherwise.

Prefix Filter. It prunes $\langle r, s\rangle$ if their prefixes have no common token. The time cost Θ is $\mathcal{O}(\mathtt{pre}(r)+\mathtt{pre}(s))$. (If the prefix filter is used in the filtering step for this attribute, we will not use it again and set its pruning probability as 0.)

Count Filter. It is useful only for character-based functions. If two strings are similar, they must share $\max(|r|,|s|)-q+1-\tau q$ common q grams. The time cost Θ is $\mathcal{O}(|r|+|s|)$.

Content Filter. It is useful only for character-based similarity functions. For example, for ED, it computes the sum of the number of character difference and if the difference is larger than 2τ, it prunes the pair. The cost is $\mathcal{O}(|r|+|s|)$.

The well-known verification algorithm for ED is the dynamic-programming algorithm with time cost of $\mathcal{O}(2\tau\min(|r|,|s|))$. The verification algorithm for JAC is $\mathcal{O}(|r|+|s|)$.

Next we analyze the filtering probability. Initially, we can utilize a sampling-based method to compute the pruning probability of each filter. Then, after verifying each record, we update the pruning probability of each filter. However it is not free to get the pruning probability and it is expensive to compute an appropriate order for every pair. Instead, we periodically update the order. For similarity join, we group the candidates based on a record in $\mathcal{R}$ and use the same order for candidates in the same group; for similarity search, we can use the same order for candidates.

6. EXPERIMENT

We have implemented our techniques to support similarity join and search on multi-attribute data and our goal is to evaluate the efficiency. We used two real datasets DBLP and IMDB in the experiments, as shown in Table 1.

DBLP: It is a publication dataset (dblp.uni-trier.de/xml/). We ran self-join on DBLP with 4 attributes {title, journal, author, year}. The similarity operation was $\mathtt{title}\overset{\mathsf{COS},\tau}{\sim}\mathtt{title}\wedge\mathtt{journal}\overset{\mathsf{JAC},\tau}{\sim}\mathtt{journal}\wedge\mathtt{author}\overset{\mathsf{ES},\tau}{\sim}\mathtt{author}\wedge\mathtt{year}\overset{\mathsf{ES},\tau}{\sim}\mathtt{year}$ by varying τ in [0.75, 0.95].

IMDB: It is a movie database (imdb.com). We ran self-join on IMDB with 4 attributes {title, genres, director, writer}. The similarity operation was $\mathtt{title}\overset{\mathsf{ES},\tau}{\sim}\mathtt{title}\wedge\mathtt{genres}\overset{\mathsf{JAC},\tau}{\sim}\mathtt{genres}\wedge\mathtt{director}\overset{\mathsf{ES},\tau}{\sim}\mathtt{director}\wedge\mathtt{writer}\overset{\mathsf{ES},\tau}{\sim}\mathtt{writer}$ by varying τ in [0.75, 0.95].

All the algorithms were implemented in C++ and compiled using GCC 4.7 with the -O3 flag. All the experiments were conducted on a machine running 64bit Ubuntu 14.04 with Intel Xeon E2650 2.0GHz processor and 16GB RAM.

6.1 Evaluation on Prefix Trees for Join

In this section, we evaluated our prefix trees for similarity join. We first compared our optimal prefix tree and the greedy prefix tree with CBlockTree [21], which was used to merge blocks to improve the recall of the blocking-based scheme (see Appendix C for details). Figure 8 shows the candidate numbers and join time, where the bars from bottom to top respectively denote the time of building trees, filtering time and verification time. We had the following observations. First, CBlockTree reduced the number of candidates, because it enumerated every two leaves and merged the leaves with the join cost on the merged node smaller than the cost on two individual leaves, while our method only checked whether to combine the leaf nodes with the same parent. However, enumerating every pair of leaf nodes had expensive overhead, and thus CBlockTree had worse performance than our method. Second, our greedy prefix tree nearly had the same number of candidates with the optimal prefix tree, because if merging a subtree had a large benefit, our greedy prefix tree would not split the subtree and our cost model can effectively decide whether to split a node or not. Third, our greedy prefix tree had much better join performance than the optimal prefix tree. The reason is obvious that it took less time to construct the greedy prefix tree, and the optimal tree generated a large complete prefix tree and then merged many subtrees, which was rather time consuming, especially for large datasets. Another reason was that the greedy tree had similar pruning power with the optimal tree and thus it did not loss much performance.

Table 1: Datasets.

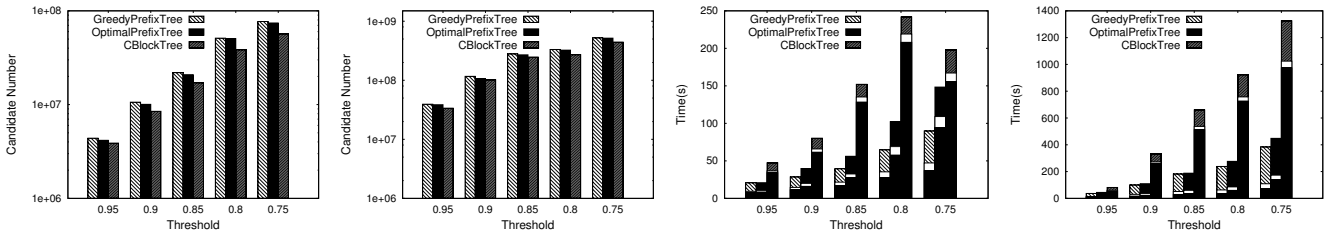
Datasets	# Records	# Attributes	Size
DBLP	1M	4	174MB
IMDB	400K	4	159MB

Then, we compared different orders of predicates in constructing the prefix trees. We compared the random order with our greedy order. Figure 9 shows the results. We can see that our greedy order significantly outperformed the random order, even by 1-2 orders of magnitude. This is because if we used predicates with large cost in top levels, then both indexing and filtering took much time. Our greedy algorithm selected an appropriate order of predicates and made inverted lists on leaf nodes much shorter. For example, on DBLP, for $\tau = 0.8$, the random order took nearly 1000 seconds while our greedy order took less than 80 seconds.

6.2 Evaluation on Prefix Trees on Search

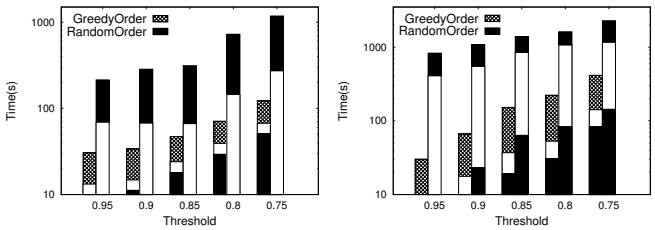
We generated 100K search queries based on the same similarity operations, varied the thresholds, and reported the average search time. To build prefix index, we set a threshold bound (0.7) for each predicate and built our multiple prefix trees with the bound. We first evaluated our techniques by using multiple prefix trees to answer a query (proposed in Section 4.2). Suppose the size of the multiple prefix tree with the first level was 1. We set $\mathcal{B} = 2$ in this experiment. We implemented three methods. (1) Intersection-based Method: It computed the candidates using each predicate and then intersected them. (2) Pipeline-based Method: It selected the best predicate and utilized this predicate to generate candidates. (3) Cost-based Method: It used our cost model to generate the candidates. Figure 10 shows the results (bottom/top bars denote filtering/verification time).

We had the following observations. First, on DBLP, the pipeline-based method was better than the intersection-based method, because the pipeline-based method first used a highly selective attribute to identify less candidates and then verified the candidates, and thus achieved rather high performance. However, the intersection-based method used each predicate to identify many candidates and thus was expensive. For example, consider a query (AUTHOR $\overset{\mathsf{ES},0.8}{\sim}$ 'Dieter Baum, Vladimir V. Kalashnikov', JOURNAL $\overset{\mathsf{JAC},0.8}{\sim}$



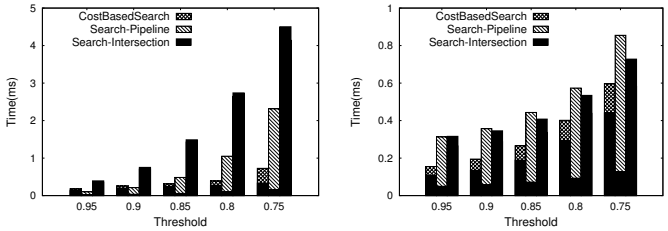
(a) Candidate Number (DBLP) (b) Candidate Number (IMDB) (c) Efficiency (DBLP) (d) Efficiency (IMDB)

Figure 8: Evaluation on Prefix Trees for Similarity Join: Optimal vs Greedy vs CBlockTree



(a) Efficiency (DBLP) (b) Efficiency (IMDB)

Figure 9: Evaluation on Predicate Order for Join

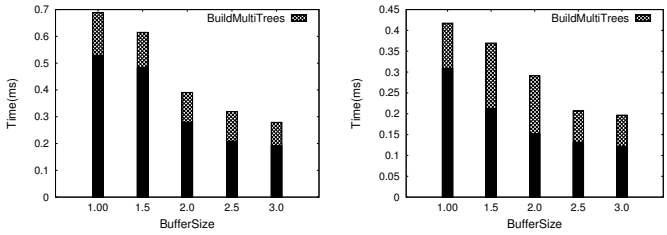


(a) Efficiency (DBLP) (b) Efficiency (IMDB)

Figure 10: Evaluation on Prefix Trees for Search.

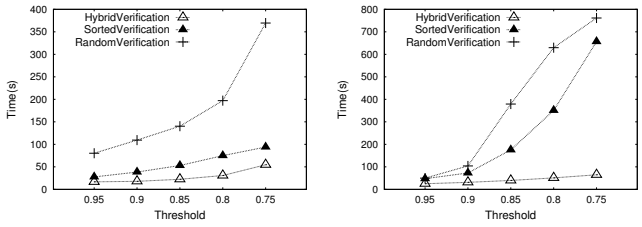
'`University Trier`, `Mathematik/Informatik`'). Attribute JOURNAL only generated 300 candidates (by prefix filtering), while attribute AUTHOR generated over 10K candidates. The intersection-based method identified 10K candidates from attribute AUTHOR and 300 candidates from JOURNAL while the pipeline-based method only identified 300 candidates and then verified them. Obviously, the latter was faster. Second, on `IMDB`, the intersection-based method was better than the pipeline-based method because every predicate generated nearly the same number of answers and the intersection-based method can further reduce the candidate number by intersecting them, which had lower cost than directly verifying each candidate. For example, a query generated 1000 candidates on attribute TITLE and 1000 candidates on attribute DIRECTOR, and there were 100 results. The pipeline-based method identified 1000 candidates and verified 1000 candidates while the intersection-based method identified 2000 candidates and verified 150 candidates (there were 150 candidates after the intersection). As the verification cost was much more expensive than the filtering cost, the intersection-based method was better. Third, our cost-based method always achieved the best performance because it judiciously selected appropriate predicates to generate the candidates. If the best predicate generated a small number of candidates, it directly verified the candidates; otherwise it selected another predicate to further reduce the number of candidates. For example, on `DBLP`, for $\tau = 0.75$, the intersection-based method took 4.6 milliseconds for each query, and the pipeline-based method took 2.3 milliseconds, while our cost-based method improved it to 0.4 milliseconds.

Then we evaluated the performance by varying the given budget $\mathcal{B}$. Suppose the size of all the 1-level prefix trees was 1. We varied $\mathcal{B}$ in 1.0, 1.5, 2.0, 2.5, and 3.0, and evaluated



(a) Efficiency (DBLP) (b) Efficiency (IMDB)

Figure 11: Evaluation on Search by Varying Budgets



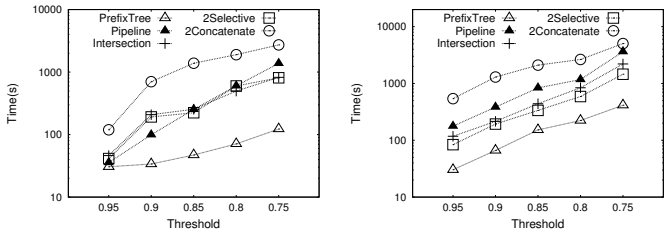
(a) Efficiency (DBLP) (b) Efficiency (IMDB)

Figure 12: Evaluation on Verification Algorithms.

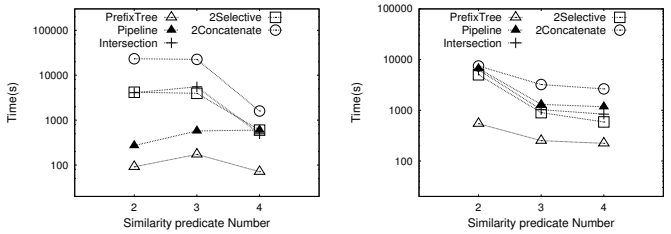
the average search performance. Figure 11 shows the results. We can see with the increase of $\mathcal{B}$, our method achieved higher performance, this is because we can use more prefix trees to answer search queries, which obviously can improve the performance. Especially, if each query attribute was contained by a prefix tree, then the prefix tree can significantly improve the performance than the 1-level prefix tree.

6.3 Evaluation on Verification Techniques

We evaluated our verification techniques. We compared three methods. (1) Random Verification: we randomly verified a candidate. (2) Sorted Verification: we sorted the predicates based on our techniques in Section 5.1. (3) Hybrid Verification: we used our cost model in Section 5.1 to verify a candidate. We used the length filter, prefix filter, count filter, and content filter in the experiment. Figure 12 shows the results. We can see that the hybrid verification outperformed the other two methods. This is because our hybrid verification method always judiciously selected appropriate orders of predicates to verify a candidate and used the appropriate filtering algorithms to prune dissimilar pairs. Our hybrid verification algorithm was better than the sorted verification method, especially for a small threshold. This is because selecting a predicate order was not enough to achieve high performance and our hybrid verification method can also utilize lightweight filters to prune many candidates. For example, our method can utilize the length filter and count filter on some attributes to prune many candidate with little cost. The improvement of our hybrid method over the sorted-based method on `IMDB` was significant because there were many candidates in `IMDB` and our hybrid can utilize different filters to effectively prune these candidates. However, on `DBLP`, there were few candidates using an appropriate predicate and thus the improvement was not significant.



(a) Varying τ (DBLP) (b) Varying τ (IMDB)



(c) Varying # Predicates(DBLP) (d) Varying # Predicates(IMDB)

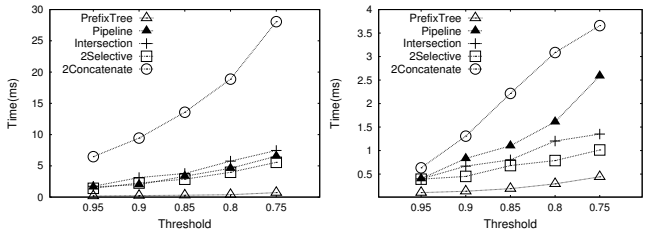
Figure 13: Comparison on Similarity Joins.

6.4 Comparison with Baselines

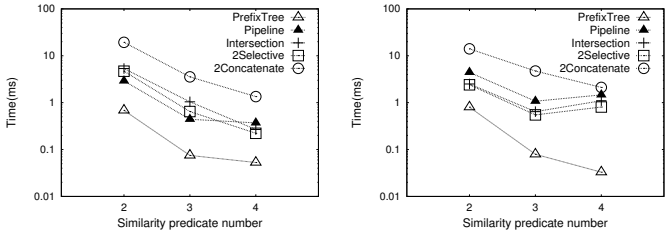
We compared our method with state-of-the-art q-gram-based methods, PPJoin [29] for token-based similarity and ED-join [28] for character-based similarity. For fair comparison, we used the same filters with these two algorithms, including length filter, count filter, content filter, and prefix filter. We extended them and took them as baseline approaches (see Appendix C for details): (1) INTERSECTION: It used PPJoin or ED-join to identify the candidates on every predicate, intersected the candidates, and verified the intersected candidates. (2) PIPELINE: It selected the best predicate using our model and used PPJoin or ED-join to identify the candidates, and then verified the candidates. (3) 2SELECTIVE: It first identified candidates for two most selective attributes and then intersected them. (4) 2CONCATENATE: It concatenated two selective attributes as a single attribute and then used PPJoin or ED-join to identify the candidates. We also extended them to support search by using the threshold 0.7 (which was the same as our method).

6.4.1 Comparison on Similarity Joins

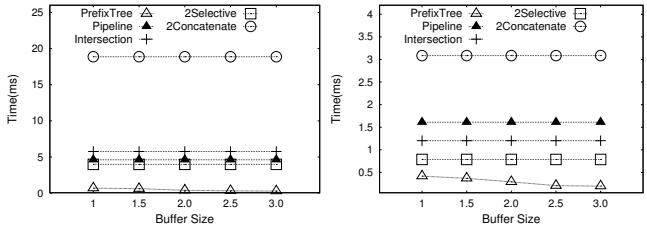
We compared the performance by varying the thresholds and the number of similarity predicates, and the result is shown in Figure 13. We had the following observations. First, with the decrease of τ, the performance of all the methods become worse because a small threshold will generate more results and thus all the algorithms generated more candidates and involved more verification time. Second, the query with more predicate numbers may not have worse performance, because a "lightweight" attribute will improve the performance. For example, on DBLP, answering the query with attributes `title`, `journal`, `author` was easier than answering query with attributes `title`, `author`, because the attribute `journal` can prune many dissimilar pairs. Third, the intersection-based method was always worse because it was very expensive to identify candidates for every predicate. Fourth, 2CONCATENATE achieved the lowest performance because it generated many more candidates than INTERSECTION. Actually, the candidate set obtained by concatenating two attributes is a super set of the candidate set obtained by intersecting two attributes, because if r is a candidate of intersecting two attributes, then the concatenated attribute of r must satisfy the combined constraint and thus r is a candidate of concatenating two attributes (see Appendix C for more details). Fifth, alghouth 2SELECTIVE was better than INTERSECTION, it was still worse than our method, because it only used two attributes and cannot adaptively select a better strategy for queries with more than two attributes. Sixth, our method always outperformed the baseline approaches. This was attributed to the pruning power of our prefix tree and the effectiveness of our hybrid verification algorithm. The first technique can prune a large number of dissimilar results and the second technique can efficiently verify each candidate pair.



(a) Varying τ (DBLP) (b) Varying τ (IMDB)



(c) Varying # Predicates(DBLP) (d) Varying # Predicates(IMDB)



(e) DBLP (f) IMDB

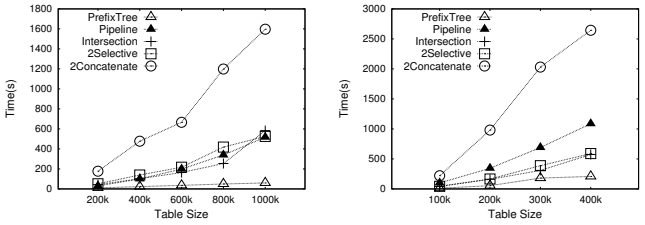
Figure 14: Comparison on Similarity Search with Baselines (Including filtering&verification)

6.4.2 Comparison on Similarity Search

We compared the search performance by varying the thresholds, the number of similarity predicates, and the budge $\mathcal{B}$. The result is shown in Figure 14. We had the following observations. First, our method was still much better than the four baselines for different buffer sizes, even the same buffer size with existing methods. This is because our algorithm selected a better search strategy to reduce the number of candidates and utilized a cost-based verification algorithm to improve the verification performance. Second, similar to similarity join, with the decrease of thresholds, the performance was also decreased because there will be more candidates and answers. Third, with the increase of the buffer $\mathcal{B}$, the baseline kept the same performance and our method became better, because our method can utilize the budget to generate more prefix trees. Fourth, the search time actually decreased when we used more similarity predicates. This is because the last two similarity predicates actually had very high pruning power and cheap filtering cost.

6.5 Scalability

We evaluated the scalability of different methods by varying the dataset sizes. Figure 15 shows the scalability on join and Figure 16 shows the scalability on search. We can see that with the increase of the dataset sizes, our method



(a) Scalability (DBLP) (b) Scalability (IMDB)

Figure 15: Scalability on Joins.

Table 2: Scalability of index size

# of records (DBLP)	400k	600k	800k	1M
Index Size	39MB	52MB	81MB	95MB
# of records (IMDB)	100k	200k	300k	400k
Index Size	100MB	150MB	230MB	309MB

scaled well and always outperformed the baselines. On similarity joins, with the increase of dataset sizes, our method increased linearly. For example, on DBLP, our method took 10 seconds to join 200K records, and 50 seconds for 1M records. On similarity search, our method also increased linearly with the increase of dataset sizes on both of the two datasets. This is attributed to the high pruning power of our prefix trees which can prune a large number of dissimilar results and the efficient verification algorithms which can efficiently verify each candidate pair.

We also evaluated the index size, which included two parts: prefix tree nodes and inverted lists. Since the index size can be controlled by $\mathcal{B}$ for search, here we reported the index size of similarity joins on the DBLP dataset in Table 2, where we indexed all the four attributes with $\tau = 0.8$. We can see the index size scaled well as the dataset sizes increased.

7. CONCLUSION

We have studied similarity search and join on multi-attribute data. We devised a prefix tree index structure and utilized it to support similarity search and join. For similarity join, we proposed a cost model to quantify the prefix tree and proved that finding the optimal prefix tree is NP-complete and we devised a greedy algorithm to construct a high-quality prefix tree. We extended the prefix tree to support similarity search and constructed multiple prefix trees to answer search queries. We devised a hybrid verification algorithm by finding an appropriate order of predicates and filtering algorithms to improve the verification performance. Experimental results on two real datasets show that our method significantly outperformed baseline approaches.

Acknowledgement. This work was partly supported by the 973 Program of China (2015CB358700 and 2011CB302206), and the NSFC project (61373024, 61422205, 61472198), Tencent, Huawei, SAP, YETP0105, the "NExT Research Center" (WBS: R-252-300-001-490), and the FDCT/106/2012/A3.

8. REFERENCES

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

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

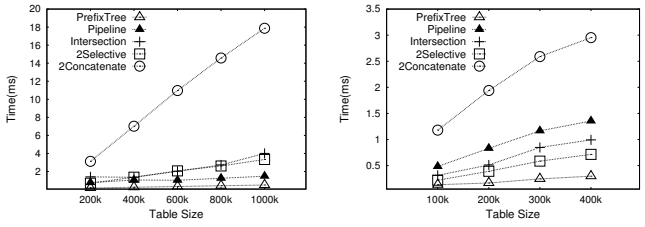
[3] N. N. Dalvi, V. Rastogi, A. Dasgupta, A. D. Sarma, and T. Sarlós. Optimal hashing schemes for entity matching. In *WWW*, pages 295–306, 2013.

[4] D. Deng, G. Li, and J. Feng. A pivotal prefix based filtering algorithm for string similarity search. In *SIGMOD Conference*, pages 673–684, 2014.

[5] D. Deng, G. Li, J. Feng, and W.-S. Li. Top-k string similarity search with edit-distance constraints. In *ICDE*, pages 925–936, 2013.

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

[7] M. Garey and D. Johnson. *A guide to the theory of NP-completeness*. WH Freeman and Company, 1979.



(a) Scalability (DBLP) (b) Scalability (IMDB)

Figure 16: Scalability on Search.

[8] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.

[9] M. Hadjieleftheriou, N. Koudas, and D. Srivastava. Incremental maintenance of length normalized indexes for approximate string matching. In *SIGMOD Conference*, pages 429–440, 2009.

[10] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *SIGMOD Conference*, pages 267–276, 1993.

[11] M. A. Hernández and S. J. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Min. Knowl. Discov.*, 2(1):9–37, 1998.

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

[13] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: similarity measures and algorithms. In *SIGMOD Conference*, pages 802–803, 2006.

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

[15] C. Li, B. Wang, and X. Yang. Vgram: Improving performance of approximate queries on string collections using variable-length grams. In *VLDB*, pages 303–314, 2007.

[16] G. Li, D. Deng, and J. Feng. A partition-based method for string similarity joins with edit-distance constraints. *ACM Trans. Database Syst.*, 38(2):9, 2013.

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

[18] A. McCallum, K. Nigam, and L. H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *SIGKDD*, pages 169–178, 2000.

[19] M. Michelson and C. A. Knoblock. Learning blocking schemes for record linkage. In *AAAI*, pages 440–445, 2006.

[20] J. Qin, W. Wang, Y. Lu, C. Xiao, and X. Lin. Efficient exact edit similarity query processing with the asymmetric signature scheme. In *SIGMOD Conference*, pages 1033–1044, 2011.

[21] A. D. Sarma, A. Jain, A. Machanavajjhala, and P. Bohannon. An automatic blocking mechanism for large-scale de-duplication tasks. In *CIKM*, pages 1055–1064, 2012.

[22] J. Wang, G. Li, D. Deng, Y. Zhang, and J. Feng. Two birds with one stone: An efficient hierarchical framework for top-k and threshold-based string similarity search. In *ICDE*, 2015.

[23] J. Wang, G. Li, and J. Feng. Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. *PVLDB*, 3(1):1219–1230, 2010.

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

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

[26] W. Wang, J. Qin, C. Xiao, X. Lin, and H. T. Shen. Vchunkjoin: An efficient algorithm for edit similarity joins. *IEEE Trans. Knowl. Data Eng.*, 25(8):1916–1929, 2013.

[27] S. E. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina. Entity resolution with iterative blocking. In *SIGMOD Conference*, pages 219–232, 2009.

[28] C. Xiao, W. Wang, and X. Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB*, 1(1):933–944, 2008.

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

[30] Z. Zhang, M. Hadjieleftheriou, B. C. Ooi, and D. Srivastava. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In *SIGMOD*, pages 915–926, 2010.

APPENDIX

A. PROOF

A.1 Proof of Lemma 1

LEMMA 1. *The partial tree built by the* OPTIMAL-BOTTOMUP *is optimal.*

PROOF. Given any two tables $\mathcal{R}$ and $\mathcal{S}$, a complex similarity operation Φ with i atomic operations and a specified attribute order, we want to prove that the partial tree built by the OPTIMAL-BOTTOMUP is optimal. We prove it by reduction. For $i = 1$, there is only one atomic operation in Φ. There are only two prefix trees for this situation, one contains only one root node and the other one is a complete prefix tree. The OPTIMAL-BOTTOMUP selects the optimal one from these two prefix trees by comparing their costs. Thus the partial tree built by OPTIMAL-BOTTOMUP is optimal. Suppose for $i = n - 1$ the OPTIMAL-BOTTOMUP can find the optimal prefix tree. For $i = n$, for any prefix tree with depth larger than 0, the optimal one must be the one that all the subtrees rooted at the first level nodes are gotten by the OPTIMAL-BOTTOMUP method (when $i = n - 1$). Then the OPTIMAL-BOTTOMUP will compare the cost of this prefix tree with that of the prefix tree with only one root node and select the optimal one. As there is only one prefix tree with depth equal to 0 that is the one only contains a root node, the OPTIMAL-BOTTOMUP can find the optimal prefix tree when $i = n$. □

A.2 Proof of Lemma 2

To prove Lemma 2, we first introduce the problem of exact set cover as follows.

DEFINITION 1 (EXACT-COVER). *We are given a set $\mathcal{U}$ and a collection $\mathcal{C}$ of subsets of $\mathcal{U}$. Each set $S_i \in \mathcal{C}$ contains exactly 3 elements of $\mathcal{U}$. The* EXACT-COVER *problem asks if there exist $\frac{|\mathcal{U}|}{3}$ subsets in $\mathcal{C}$ which exactly covers the set $\mathcal{U}$ (i.e., each element of $\mathcal{U}$ is covered exactly once).*

The EXACT-COVER problem is a well known NP-Complete problem [7].

DEFINITION 2. (OPTIMAL PARTIAL PREFIX TREE PROBLEM (PARTIAL)). *Given two tables $\mathcal{R}$ and $\mathcal{S}$ and a complex similarity operation Φ, the* PARTIAL *problem is to find the optimal prefix tree with the minimum join cost.*

LEMMA 2. *The* PARTIAL *problem is NP-hard.*

PROOF. We present a reduction from the EXACT-COVER problem, which is NP-Complete. In our reduction, the two tables are the same, i.e., $\mathcal{R} = \mathcal{S}$. Given an instance ($\mathcal{U} = \{1, \ldots, n\}, \mathcal{C} = \{S_1, \ldots, S_m\}$) of EXACT-COVER, we construct a PARTIAL instance $(\Phi, \mathcal{R})$ as follows. The complex similarity operation Φ contains m atomic operations $\Phi_1, \Phi_2, \ldots, \Phi_m$ and the table R contains $n+1$ records $r_0, r_1, \ldots, r_n$. Each Φ_i corresponds to subset $S_i \in \mathcal{C}$ and record r_i corresponds to element $i \in \mathcal{U}$ for $1 \le i \le n$. r_0 is a *special record*. For each 3-size subset $S_i = \{x, y, z\} (1 \le i \le m, 1 \le x, y, z \le n)$, let $\texttt{pre}(r_x^i) = \{e_x\}$, $\texttt{pre}(r_y^i) = \{e_y\}$, $\texttt{pre}(r_z^i) = \{e_z\}$ and $\texttt{pre}(r_w^i) = \{e\}$ where $w \notin S_i$. Note e_x, e_y, e_z and e are all different tokens. For the special record r_0, let $\texttt{pre}(r_0^i) = \{e, e_0^i\}$ where e_0^i is another different token. See Table 1 for an example.

R	$\texttt{pre}(R^1)$	$\texttt{pre}(R^2)$	$\texttt{pre}(R^3)$	$\texttt{pre}(R^4)$
r_0	$\{e_0^1, e\}$	$\{e_0^2, e\}$	$\{e_0^3, e\}$	$\{e_0^4, e\}$
r_1	$\{e_1\}$	$\{e\}$	$\{e\}$	$\{e\}$
r_2	$\{e_2\}$	$\{e_2\}$	$\{e\}$	$\{e\}$
r_3	$\{e_3\}$	$\{e_3\}$	$\{e_3\}$	$\{e\}$
r_4	$\{e\}$	$\{e_4\}$	$\{e_4\}$	$\{e_4\}$
r_5	$\{e\}$	$\{e\}$	$\{e_5\}$	$\{e_5\}$
r_6	$\{e\}$	$\{e\}$	$\{e\}$	$\{e_6\}$

Table 1: The table $\mathcal{R}$ ($\mathcal{R} = \mathcal{S}$) constructed from Exact-Cover instance ($\mathcal{U} = \{1, \ldots, 6\}$), $\mathcal{C} = \{S_1, \ldots, S_4\}$, where $S_1 = \{1, 2, 3\}$, $S_2 = \{2, 3, 4\}$, $S_3 = \{3, 4, 5\}$, $S_4 = \{4, 5, 6\}$.

Next we show that the EXACT-COVER instance has an exact cover if and only if there is a partial prefix tree for the corresponding PARTIAL instance with cost $n + \frac{n}{3} + 1$. Suppose we can find $\frac{n}{3}$ subsets $T_1, T_2, \ldots, T_{\frac{n}{3}}$ from $\mathcal{C}$ which can exactly cover U, we build a prefix tree with $\frac{n}{3} + 1$ levels. Each internal node (including the root) corresponding to a chosen subset T_i and each leaf corresponds to an element. Level 1 is the root node, which corresponds to T_1. The internal node at level i corresponds to T_i. For each $i \in [1, \frac{n}{3}]$, suppose $T_i = \{x, y, z\}$. Then, we can see T_i has five child nodes e, e_x, e_y, e_z, e_0^i. For ease of exposition, we always arrange e as the leftmost child of any internal node. Hence, all nodes on leftmost path (except the root) are labeled with token e and the rests are leaves. Moreover, each leaf node corresponds to a record in R (namely, e_x corresponds to r_x, ..., e_0^i corresponds to r_0). We say a leaf corresponding to r_0 a *special leaf*. Therefore each leaf is associated with two same inverted lists (since $\mathcal{R} = \mathcal{S}$) with exactly one (the same) record. Each r_i for $1 \le i \le n$ appears exactly once and the special record r_0 appears $\frac{n}{3} + 1$ times as special leaves (each internal node has a special leaf, except that the last level has two special leaves, see Figure 1). This tree has a cost of $n + \frac{n}{3} + 1$.

Now we show the other direction: If there is a prefix tree with cost $n + \frac{n}{3} + 1$, the corresponding EXACT-COVER instance has an exact cover. Consider an arbitrary prefix tree. Suppose the set of internal nodes corresponds to a collection Q of subsets of $\mathcal{C}$. Suppose k elements are covered by Q. We can see the cost of the tree is $|Q|$ (each internal node has a special leaf r_0) $+k$ (each non-special leaf)$+ (n-k+1)^2$ (all uncovered elements and r_0 are all in the leftmost leaf). We know that $|Q| \ge k/3$. There are three cases:

1. If $k \le n-1$, the above quantity is larger than $4n/3+1$, which does not satisfy the assumption.
2. If $k = n$ and $|Q| > n/3$, the above quantity is larger than $4n/3 + 1$, which does not satisfy the assumption.
3. If $k = n$ and $|Q| = n/3$, the above quantity is equal to $4n/3 + 1$ and the $n/3$ subsets in Q is an exact cover of $\mathcal{U}$.

This concludes that PARTIAL is NP-hard. □

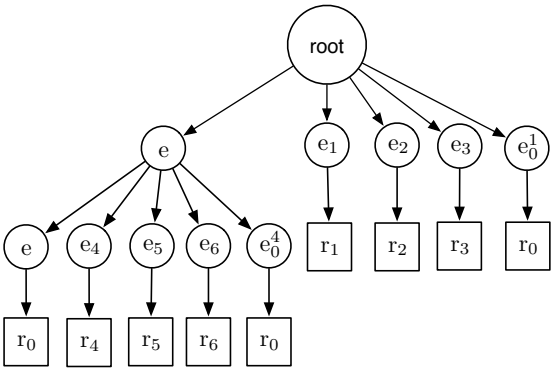


Figure 1: The optimal prefix tree built based on an exact cover $T_1 = S_1$ and $T_2 = S_4$ of the Exact-Cover instance in Table 1.

B. OTHER SIMILARITY FUNCTIONS

We give the definitions of COS and DICE.

- Cosine similarity: $\mathsf{COS}(r_p^i, s_q^j) = \frac{|r_p^i \cap s_q^j|}{\sqrt{|r_p^i| \cdot |s_q^j|}}$.
- Dice similarity: $\mathsf{DICE}(r_p^i, s_q^j) = \frac{2|r_p^i \cap s_q^j|}{|r_p^i| + |s_q^j|}$.

We compute the threshold o for COS and DICE.

- If $\mathsf{COS}(r, s) \geq \tau$ then $o = \lceil \tau^2 |r| \rceil$.
- If $\mathsf{DICE}(r, s) \geq \tau$ then $o = \lceil \frac{\tau}{2-\tau} |r| \rceil$.

C. BASELINE APPROACHES

Given an atomic similarity operation $\mathcal{R}^i \sim \mathcal{S}^j$, the prefix filter identifies the pairs of records whose prefixes on the two attributes overlap as the candidates. To efficiently compute the candidate, we can build an inverted index on top of the tokens. Each token is associated with an inverted list of records whose prefixes on the attribute contain the token. Then we can utilize the inverted lists to identify candidates.

Baseline Approaches. We extend prefix filtering to support complex similarity operations and give the following baselines.

(1) Intersection-based Method. For each predicate, we use existing algorithms to identify the similar answers on this predicate, and then intersect them to compute the final answers. This method has two weaknesses. First, it is rather expensive to compute the similar results for every predicate, especially there are many predicates. Second, there may be large numbers of candidates for each predicate and it is time- and space-consuming to maintain and intersect the intermediate results.

(2) Pipeline-based Method. It first computes the results for a predicate and then verifies these candidates using other similarity predicates. Since the candidates are record pairs, we cannot utilize existing indexes to do further pruning and have to verify them by computing their similarities on other predicates. This method has the following disadvantages. First, a single attribute has low pruning power and it will generate a large number of candidates. Second, it is important to determine an appropriate order of predicates as different orders have different pruning power. It is challenging to select the best order.

(3) 2Concatenate Method. It concatenates two selective attributes as a single attribute. However it is nontrivial to concatenate two attributes, because different attributes may use different similarity functions, e.g., edit distance and Jaccard. To address this issue, we first transform them to the overlap similarity. Then given the two most selective attributes, we calculate the overlap thresholds o^1 and o^2 for the two attributes. Next we concatenate tokens of the two attributes, use $o = o^1 + o^2$ as the new overlap threshold, and generate the signatures for the concatenated attribute. This method would generate large numbers of candidates. Actually the candidate set obtained by the concatenate-based method is a super set of the candidate set obtained by the intersection-based method.[3]

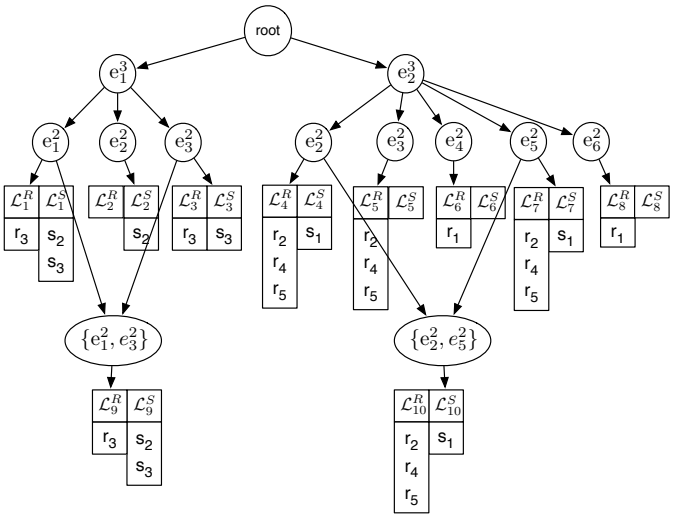


Figure 2: Merge leaves in Prefix Tree to further reduce cost. (Tables $\mathcal{R}$ and $\mathcal{S}$ from Figure 2)

(4) 2Selective Method. Given a simlarity operation, it first selectes two most selective attributes, identifies the candidates of the two attributes, and then intersects them.

(5) Extending The Roll-up Algorithm [21] to Generate CBlockTree. We extended the roll up algorithm [21] to construct a CBlockTree so as to reduce the join cost of our prefix tree. Figure 2 illustrates a CBlockTree. It first builds a complete prefix tree, then enumerates every pair of leaf nodes and merges any two leaves with the join cost on the merged node smaller than the join cost on the two individual leaves. Although this method can reduce the candidate size, it has a large overhead to enumerate every pair of leaves. Since a prefix has many leaves, checking every pair of leaves is rather expensive. Moreover, our problem focuses on improving the performance, and thus this method is not efficient to address our problem.

[3]Consider a record with two attributes, $\{a, b, c, d\}$ and $\{\alpha, \beta, \gamma, \delta\}$, where the token orders are $a < b < c < d$ and $\alpha < \beta < \gamma < \delta$ respectively. Suppose $o_1 = 2$ and $o_2 = 1$. The prefixes of the two attributes are $\{a, b\}$ and $\{\alpha\}$. The intersection-based method takes $(\mathcal{L}(a) \cup \mathcal{L}(b)) \cap \mathcal{L}(\alpha)$ as candidates, where $\mathcal{L}(a)$ is the inverted list of a. The pipeline-based method takes $\mathcal{L}(a) \cup \mathcal{L}(b)$ or $\mathcal{L}(\alpha)$ as candidates. The concatenated attribute is $\{a, b, c, d, \alpha, \beta, \gamma, \delta\}$. Then the prefix filtering selects 3 signatures from $\{a, b, c, d, \alpha, \beta, \gamma, \delta\}$ in order. Based on the order, there are 4 possible cases: $\{a, b, c\}$, $\{a, b, \alpha\}$, $\{a, \alpha, \beta\}$, $\{\alpha, \beta, \gamma\}$. The candidates are respectively $\mathcal{L}(a) \cup \mathcal{L}(b) \cup \mathcal{L}(c)$, $\mathcal{L}(a) \cup \mathcal{L}(b) \cup \mathcal{L}(\alpha)$, $\mathcal{L}(a) \cup \mathcal{L}(\alpha) \cup \mathcal{L}(\beta)$, $\mathcal{L}(\alpha) \cup \mathcal{L}(\beta) \cup \mathcal{L}(\gamma)$. In any case, the result is a super set of the candidates obtained by the intersection-based method. In addition, in most cases, the concatenate-based method is even worse than the pipeline-based method.

D. COMPLEXITY

D.1 Time Complexity of Constructing The Optimal Prefix Tree

Time Complexity: Given two tables $\mathcal{R}$ and $\mathcal{S}$ and a complex similarity operation $\Phi = \mathcal{R}^{i_1} \sim \mathcal{S}^{j_1} \wedge \mathcal{R}^{i_2} \sim \mathcal{S}^{j_2} \wedge \cdots \wedge \mathcal{R}^{i_k} \sim \mathcal{S}^{j_k}$. The OPTIMAL-BOTTOMUP builds the optimal prefix tree by two steps. The first step of building the complete prefix tree has a time complexity of $\mathcal{O}\Big((|\mathcal{R}| + |\mathcal{S}|)|\hat{\texttt{pre}}|^k\Big)$, where $|\hat{\texttt{pre}}|$ is the maximal prefix length of an attribute. (It is worth noting that $|\hat{\texttt{pre}}|$ is not large, which is not larger than the size of the corresponding token set). The second step of generating the optimal tree has a time complexity of $\mathcal{O}\Big(k(|\mathcal{R}| + |\mathcal{S}|)|\hat{\texttt{pre}}|^k\Big)$, where k is the number of predicates in the complex similarity operation. This is because there are at most $(|\mathcal{R}| + |\mathcal{S}|)|\hat{\texttt{pre}}|^k$ records in inverted lists of all leaves, and each record will be checked at most k times to build the optimal tree.

D.2 Time Complexity of Constructing The Greedy Prefix Tree

Time Complexity: In the worst case, the greedy prefix tree is exactly the same as the complete prefix tree, and thus the time complexity of building the greedy prefix tree is $\mathcal{O}\Big((|\mathcal{R}| + |\mathcal{S}|)|\hat{\texttt{pre}}|^k\Big)$. In practice, many internal nodes will not split and the greedy algorithm can be very efficient.

D.3 Space Complexity of Prefix Tree

Space Complexity: In the worst case, the optimal prefix tree is exactly the same as the complete prefix tree. The size of the prefix tree consists two parts. The first part is the number of records on inverted lists of leaf nodes, i.e., $\mathcal{O}\Big((|\mathcal{R}| + |\mathcal{S}|)|\hat{\texttt{pre}}|^k\Big)$, where $|\hat{\texttt{pre}}|$ is the maximal prefix length of an attribute. The second part is the number of tree nodes, which is $\mathcal{O}\Big(|\texttt{pre}(\mathcal{R})|^k + |\texttt{pre}(\mathcal{S})|^k\Big)$, where $|\texttt{pre}(\mathcal{R})|$ is the number of unique prefix tokens in an attribute. Thus the total space cost is $\mathcal{O}\Big((|\mathcal{R}| + |\mathcal{S}|)|\hat{\texttt{pre}}|^k + |\texttt{pre}(\mathcal{R})|^k + |\texttt{pre}(\mathcal{S})|^k\Big)$. In practice, the size of the optimal prefix tree is much smaller than the size of the complete prefix tree, because many internal nodes will not split.

E. OPTIMALITY OF THE GREEDY VERIFICATION ALGORITHM

For simplicity, we use c_i to denote $\Theta(\langle\Phi_i, \Gamma_i\rangle)$ and p_i to denote $\mathcal{P}(\langle\Phi_i, \Gamma_i\rangle)$. Our goal is to find an order π such that $\textsc{Cost}(\pi) = \sum_i c_{\pi(i)} \cdot \prod_{j=1}^{i-1}(1 - p_{\pi(j)})$ is minimized. We need to prove the optimal order is the increasing order of c_i/p_i. We prove by contradiction. Suppose π is an optimal order but $c_{\pi(k)}/p_{\pi(k)} > c_{\pi(k+1)}/p_{\pi(k+1)}$ for some k. Consider the order π' obtained by swapping $\pi(k)$ and $\pi(k+1)$. We can see that

$$\textsc{Cost}(\pi) - \textsc{Cost}(\pi') = \prod_{j=1}^{k-1}(1 - p_{\pi(j)}) \times$$
$$\big(c_{\pi(k)} + c_{\pi(k+1)}(1 - p_k) - (c_{\pi(k+1)} + c_{\pi(k)} \cdot (1 - p_{\pi(k+1)})\big)$$
$$= \prod_{j=1}^{k-1}(1 - p_{\pi(j)}) \left(c_{\pi(k)} p_{\pi(k+1)} - c_{\pi(k+1)} p_{\pi(k)}\right) > 0.$$

This contradicts that π is the optimal order. Thus our greedy verification algorithm is optimal.