A Framework for Set Similarity Join on Multi-Attribute Data

Leonardo Andrade Ribeiro, Felipe Ferreira Borges, Diego Junior do Carmo Oliveira

[1]Instituto de Informática – Universidade Federal de Goiás (UFG) – Goiânia – GO – Brazil

{laribeiro,felipeferreiraborges,diegooliveira}@inf.ufg.br

Abstract. *Set similarity join, which finds all pairs of similar sets in a collection, plays an important role in data cleaning and integration. Many algorithms have been proposed to efficiently answer set similarity join on single-attribute data. However, real-world data often contain multiple attributes. In this paper, we propose a framework to enhance existing algorithms with additional filters for dealing with multi-attribute data. We then present a simple, yet effective filter based on lightweight indexes, for which exact and probabilistic implementation alternatives are evaluated. Finally, we devise a cost model to identify the best attribute ordering to reduce processing time. Our experimental results show that our approach is effective and significantly outperforms previous work.*

1. Introduction

Modern enterprises increasingly acquire and store large amounts of data. Massive repositories built from numerous sources, often referred to as data lakes, are becoming popular in industry. Analytic tasks tap into such repositories to enable better decision making. Data quality is a major concern in this scenario because dirty data can jeopardize analysis results [Chu et al. 2016]. A recent survey of data scientists has confirmed that dirty data still is the main problem faced at work [Kaggle 2017]. Moreover, data cleaning is a laborious process, frequently requiring more time than the analysis itself. Indeed, another study has shown that cleaning and organizing data is the most time-consuming task of a data scientist workflow [CrowdFlower 2016]. Thus, speeding up data cleaning tasks is crucial for delivering analysis results in a timely fashion.

Set similarity join is a core operation for string data cleaning [Chaudhuri et al. 2006, Xiao et al. 2011, Ribeiro and Härder 2011, Mann et al. 2016, Wang et al. 2017], which pairs strings represented as sets whose similarity is not less than a specified threshold. A set similarity function is employed in the join predicate to mathematically approximate some notion of similarity. Set similarity join is attractive owing to its efficiency in dealing with large datasets and versatility in supporting a variety of similarity functions. Duplicate detection is a major example of the use of set similarity join in data cleaning [Chu et al. 2016]. Duplicates are multiple and non-identical representations of a real-world entity. Such kind of redundant information inevitably appears in data lakes that integrate independent data sources containing overlapping information. Under the premise that duplicates are similar in some aspect to one another, set similarity joins can be used to find pairs of potential duplicates.

Traditional set similarity join algorithms assume string data represented by a single set over which a simple similarity predicate is defined. However, real-world data is often multi-attribute. While we can still use traditional algorithms by representing multi-attribute data as a single set — either by selecting a single attribute for similarity matching

Table 1. Records containing people's personal information.

ID	Name	Street	City	State
1	Tom Allen	Texas Ave.	Augusta	Maine
2	Tom Alen	Texas Ave.	August	Maine
3	Tom Alen	Clancys St.	New York	New York
4	T. Augusta	Main St.	Allen	Texas

or concatenating string values from multiple attributes, such approach may produce unsatisfactory results. For example, consider the sample database shown in Table 1. The four records represent three distinct individuals, because records `1` and `2` actually refer to the same person, i.e., they are duplicates. If only the attribute `Name` is considered for similarity matching, record `3` could be deemed as a duplicate of records `1` and `2`. Instead, if all attributes are concatenated into a single string, then is record `4` that could be considered as a duplicate of records `1` and `2`, because values of `Name` and `City`, as well as `Street` and `State`, are similar.

The above problems are avoided by representing multi-attribute data as multiple sets. Accordingly, multiple similarity predicates can now be defined to compose the join condition. To the best of our knowledge, Li et al. [Li et al. 2015] and Oliveira et al. [Oliveira et al. 2018, do Carmo Oliveira et al. 2017] are the only previous works that have addressed set similarity joins on multi-attribute data. In a centralized setting, Li et al. proposed a prefix tree index to enable pruning of candidate pairs over multiple similarity predicates. In a distributed setting, Oliveira et al. proposed a data partitioning strategy based on a cost model to reduce both communication and computation costs.

In this paper, we present a filter-based approach to speed up set similarity join on multi-attribute data in a centralized setting. We propose an algorithmic framework that allows incorporating additional filters into traditional algorithms. We then present a simple, yet effective filter that can be implemented using simple data structures. In this context, we evaluate exact as well as approximate implementation alternatives. Finally, we devise a cost model to identify the best attribute ordering for similarity join processing. We conduct an empirical evaluation on publicly available datasets. Our results show that our proposal outperforms the algorithm of Li et al. by orders of magnitude.

The rest of this paper is organized as follows. Section 2 provides background material. Section 3 formally describes the problem and overviews existing techniques. Section 4 presents our proposed solution. Experimental results are reported in Section 5 and related work discussed in Section 6. Finally, Section 7 wraps up with the conclusions.

2. Background

In this section, we review traditional set similarity join concepts, definitions, and optimization techniques for single-attribute data. Finally, we describe a general algorithm based on a filtering-verification framework.

2.1. Basic Concepts

We focus on set-overlap-based similarity, in which the similarity between two strings is derived from the overlap of their set representations. To this end, strings are first mapped

to sets of representation units; such units are referred to as *tokens*. Then, set overlap can be measured in various ways to obtain different notions of similarity.

There are several methods for mapping strings to sets of tokens. A well-known method is based on the concept of *q-grams*, i.e., sub-strings of length *q* obtained by "sliding" a window over the characters of a given string. To this end, the string is (conceptually) extended by prefixing and suffixing it with $q-1$ occurrences of a special character *"$"*, so all its characters participate in exact *q q*-grams. For example, the string *"Tom Allen"* can be mapped to the set of *3*-grams tokens {*'$$T'*, *'$To'*, *'Tom'*, *'om '*, *'m A'*, *' Al'*, *'All'*, *'lle'*, *'len'*, *'en$'*, *'n$$'*}. Note that the result of this mapping method can be a multiset. Thus, we append the symbol of a sequential ordinal number to each occurrence of a token to convert multisets into sets, e.g, the multiset {a, b, b} is converted to {*a∘1*, *b∘1*, *b∘2*}. In the following, we assume that all strings in the database have already been mapped to sets; the resulting set collection is denoted by $\mathcal{C}$.

Given two sets r and s, a set similarity function $sim\,(r,s)$ returns a value in $[0,1]$ to represent their similarity; larger value indicates that r and s have higher similarity. Popular set similarity functions are defined as follows [Xiao et al. 2011].

Definition 1 (Set Similarity Functions) *Let r and s be two sets. We have:*

- *Jaccard similarity:* $J\,(x,y) = \frac{|r \cap s|}{|r \cup s|}$.
- *Dice similarity:* $D\,(r,s) = \frac{2 \times |r \cap s|}{|r|+|s|}$.
- *Cosine similarity:* $C\,(r,s) = \frac{|r \cap s|}{\sqrt{|r| \times |s|}}$.

Example 1 *Consider the sets* $r = \{$***A***, ***B***, ***C***, ***D***, ***E***, ***F***, ***G***, ***H***$\}$ *and* $s = \{$***A***, ***B***, ***D***, ***E***, ***G***, ***H***$\}$. *We have* $|r| = 8$, $|s| = 6$, *and* $|r \cap s| = 6$. *Therefore,* $J\,(r,s) = \frac{6}{8+6-6} = 0.75$, $D\,(r,s) = \frac{2 \times 6}{8+6} \approx 0.86$, *and* $C\,(r,s) = \frac{6}{\sqrt{8 \times 6}} \approx 0.87$.

Definition 2 (Set Similarity Join) *Given the set collection $\mathcal{C}$, a set similarity function sim, and a similarity threshold τ in the interval $[0,1]$, the Set Similarity Join on C returns all set pairs $(r,s) \in \mathcal{C} \times \mathcal{C}$ s.t. $sim\,(r,s) \geq \tau$.*

We focus in rest of this paper on the Jaccard similarity. Thus, $sim\,(r,s)$ by default denotes $J\,(r,s)$, unless stated otherwise. Nevertheless, all concepts and techniques presented in the following can be extended to Dice and Cosine [Xiao et al. 2011]. Finally, we henceforth use the term similarity function (join) to mean set similarity function (join).

2.2. Optimization Techniques

Similarity functions measure the overlap between two input sets to derive a similarity value. Thus, predicates involving such functions can be equivalently rewritten in terms of an *overlap bound* [Chaudhuri et al. 2006]. Formally, given two sets r and s, then $sim\,(r,s) \geq \tau$ iff $|r \cap s| \geq \frac{\tau \times (|r|+|s|)}{1+\tau}$ [Ribeiro and Härder 2011]. Now, the similarity join can be reduced to the problem of identifying all set pairs r and s with enough overlap.

We can significantly reduce the comparison space by exploiting the *prefix filtering principle* [Chaudhuri et al. 2006]. Prefixes allow discarding candidate pairs by examining only a fraction of the original sets. To this end, we fix a global order $\mathcal{O}$ on the universe $\mathcal{U}$ from which all tokens from the sets in $\mathcal{C}$ are drawn. We formally define the concept of prefix and the prefix filtering principle as follows.

Algorithm 1: Similarity join algorithm.

```
   Input: A sorted set collection C, a threshold τ
   Output: All pairs (r, s) s.t. sim(r, s) ≥ τ
1  I_1, ... I_|U| ← ∅
2  foreach r ∈ C do
3      M ← an empty map from set to a similarity score
4      foreach t ∈ pref(r, τ) do
5          foreach s ∈ I_t do
6              if Filter(r, s, τ) then
7                  M[s] ← −∞
8              else
9                  M[s] ← M[s] + 1
10         I_t ← I_t ∪ {r}
11     Emit(Verify(x, M, τ))
```

Definition 3 (Prefix) *A set $r' \subseteq r$ is a prefix of r if r' contains the first $|r'|$ tokens of r. Further, we denote by $pref\,(r, \tau)$ the prefix of r of size $\lfloor (1 - \tau) \times |r| \rfloor + 1$.*

Lemma 1 (Prefix Filtering Principle [Chaudhuri et al. 2006]) *Let r and s be two sets. If $sim\,(r, s) \geq \tau$, then $pref\,(r, \tau) \cap pref\,(s, \tau) \neq \varnothing$.*

Example 2 *Consider again the sets r and s in Example 1; note that both sets are already lexicographically sorted. For $\tau = 0.8$, we have $pref\,(r, 0.8) = \{A, B\}$ and $pref\,(s, 0.8) = \{A\}$.*

Note in the example above that $sim\,(r, s) < 0.8$ even though r and s share a token in their prefixes. The prefix filtering principle defines a condition necessary, but not sufficient to satisfy the original overlap constraint: an additional verification must be performed on the remaining tokens of both sets. Further, the number of candidates can be reduced by using *document frequency ordering*, $\mathcal{O}_{df}$, as global token order to obtain sets ordered by increasing token frequency in the collection $\mathcal{C}$. The motivation is to minimize the number of sets agreeing on prefix elements and, in turn, candidate pairs by moving lower frequency tokens to the prefix positions.

Other popular optimizations include *size-based filtering* [Ribeiro and Härder 2011] and *positional filtering* [Xiao et al. 2011]. Size-based filtering exploits the fact that a set r can only be similar to sets whose size is within $[|r| \times \tau, |r| \times \tau^{-1}]$. Positional filtering exploits the position of tokens in common between two sets to derive tighter overlap bounds.

2.3. Similarity Join Algorithm

Most current similarity join algorithms follow a filtering-and-verification approach supported by an inverted index [Mann et al. 2016]. Algorithm 1 provides a high-level description of this approach. An inverted list I_t stores all sets containing a token t in their prefix (Line 1). The input collection C is scanned and, for each set r, its prefix tokens are used to find candidate sets in the corresponding inverted lists (Lines 4–5). This is the filtering phase, where a variety of filters are applied for pruning candidates (Lines 6–7). If a candidate set passes through, its similarity score is accumulated in a map (Line 9). A reference to r is appended to the inverted lists associated to its prefix tokens (Line 10). Note that by indexing only prefix tokens, sets with no overlap in their prefixes are never

Table 2. Prefixes of the records in $\mathcal{C}$.

Set	$pref_0$	$pref_1$	$pref_2$
r	A, B	D	C
s	A	B, D	C
u	B	A, B	C
v	B	B	A, C

considered as candidate pairs. After the filtering phase, the similarity between r and each of its candidates is fully calculated in the verification phase and similar pairs are sent to the output (Line 11).

3. Similarity Join on Multi-Attribute Data

In this section, we begin by defining the problem of answering similarity joins on multi-attribute data. Then, we describe the algorithm of Li et al. and discuss its shortcomings.

3.1. Problem Statement

Let's first redefine our terminology and notation to deal with multi-attribute data. We assume that each record in the input database follows the same schema and has been mapped to a list of sets representing its attribute values. For simplicity, the term record refers henceforth to a record representation as a list of sets. Thus, we now denote a record by $r = r_0, ..., r_n$, where r_i represents the set derived from the *i*th attribute value; we call r_i a *set attribute*. Accordingly, $\mathcal{C}$ now denotes a collection of records and τ a list of similarity thresholds $\tau_0, ..., \tau_n$. Finally, $sim\,(r, s)$ is redefined as a conjunctive *similarity expression* over the input records r and s, where each conjunct is a similarity predicate:

$$sim\,(r, s) = \bigwedge_{i=0}^{n} sim_i\,(r_i, s_i) \geq \tau_i.$$

Definition 4 (Similarity Join on Multi-Attribute Data) *Given a record collection $\mathcal{C}$ and a similarity expression sim, the Similarity Join on $\mathcal{C}$ returns all record pairs $(r, s) \in \mathcal{C} \times \mathcal{C}$ s.t. $sim\,(r, s) = true$.*

3.2. Existing Solution

Prefix filtering is prevalently adopted by state-of-the-art algorithms on single-attribute data [Mann et al. 2016]. An intuitive way of using prefix filtering on multi-attribute data is to concatenate the prefix tokens of all set attributes. We call the result of such concatenation a *record token* and the set of all possible record tokens for some ordering of the set attributes a *record prefix*. Clearly, given two records r and s, if $sim\,(r, s) \geq \tau$, then r and s must share a record token. For example, consider the prefixes of the records composed by three set attributes in Table 2; for simplicity, only the prefixes are shown. The only candidate pairs are (r, s), which share the record token $A \circ D \circ C$, and (u, v), which share the record token $B \circ B \circ C$.

To quickly identify pairs with record tokens in common, Li et al. [Li et al. 2015] builds a *prefix tree*, where each original prefix token corresponds to a node and a root-to-leaf path forms a record token. Leaf nodes are associated with an inverted list of records

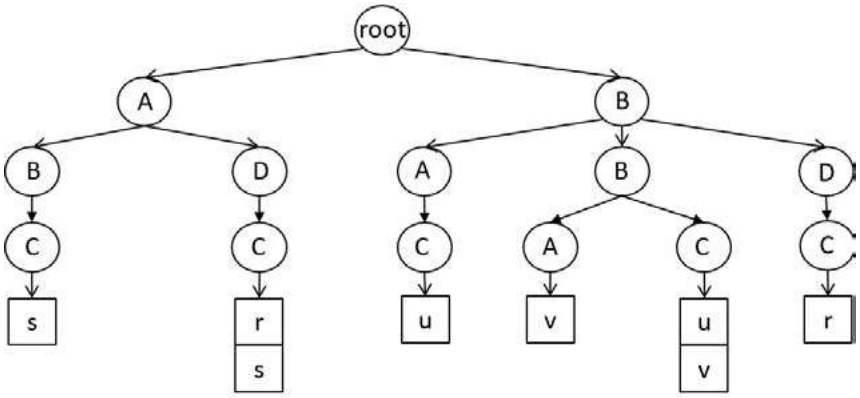


Figure 1. Prefix tree index.

containing the corresponding record token in their prefix. Figure 1 shows the prefix tree for the records in Table 2. The proposed algorithm, called henceforth *PrefTreeJoin*, first builds the prefix tree before comparing all record pairs appearing in the inverted lists. The complete prefix tree can be very large since its size grows exponentially with the number of set attributes involved in similarity predicates. Thus, a *partial prefix tree* is derived from the complete prefix tree in a bottom-up manner by eliminating unnecessary branches and merging the corresponding inverted lists. To avoid building the complete prefix beforehand, a greedy algorithm is proposed that directly constructs a partial prefix tree in a top-down manner. However, this algorithm requires knowledge of the sizes of inverted lists under all tree nodes. Therefore, it is unclear how to obtain such information without constructing the complete prefix tree first.

Besides the above issue, PrefTreeJoin has two major drawbacks. First, the construction of the partial prefix tree, whether in a bottom-up or top-down manner, is computationally expensive. Indeed, it can even take more time than the filtering and verification phases in some datasets (see [Li et al. 2015], Figure 8). Second, the algorithm is *blocking*, i.e., it cannot output any result without reading all its input. Similarity join is typically used in concert with other operations in a data analysis process and such blocking behavior prevents pipelined execution.

4. Our Solution

In this section, we present our solution for efficiently computing similarity join over a record collection. We first introduce an algorithmic framework to enable additional filters in similarity join algorithms. Then, we instantiate this framework with a filtering technique based on lightweight indexes. Finally, we present a cost model to identify the best order for set attributes.

4.1. Overview

We can straightforwardly adapt existing similarity join algorithms to multi-attribute data. To this end, we first select a set attribute, on which a regular filtering phase is carried out; we call this selected set attribute *primary set* and the remaining ones *secondary sets*. Then, we only need to adapt the verification phase for evaluating not only the primary set against the corresponding sets of the matching candidates but also evaluate the similarity predicates on the secondary sets.

(a) Overview of our approach. (b) Secondary index.

Figure 2. Our proposed solution.

Note that the above approach already avoids all drawbacks of PrefTreeJoin: the inverted index is more compact, dynamically built as the record collection is processed, and result pairs can be produced at each iteration. However, we miss the opportunity to exploit multiple similarity predicates to reduce the number of similarity computations. Indeed, building an inverted index over the prefixes of a single set has less pruning power than building a prefix tree over all sets. For example, in Table 2, selecting the first attribute as primary set generates 4 candidate pairs, namely (r, s), (r, u), (r, v), and (u, v), whereas a prefix tree generates only 2.

In this context, the main idea behind our approach is to enhance the filtering phase by using additional, lightweight indexes for filtering on the secondary sets. Figure 2 (a) illustrates our proposal. Again, a set attribute is defined as primary set, on which the *primary index* is built and regular filters are applied. But now further indexes are built on (part of) the secondary sets; we refer to those indexes as *secondary indexes*. We assume that set attributes are ordered: for each record $r = r_0, ..., r_n$, r_0 is the primary set and r_i, $1 \leq i \leq n$ are the secondary sets. We defer the discussion on determining the set attribute ordering to Section 4.4.

4.2. Algorithmic Framework

We now present our framework for incorporating secondary indexes into existing similarity join algorithms. The algorithmic framework is described in Algorithm 2. The underlying data structures are created for each secondary index prior to scanning the record collection (Line 2). The filtering phase starts processing the primary set of the current probing record r: prefix tokens of r_0 are used to find matching candidates and filters are applied on r_0 and s_0 to prune record pairs (Lines 5–8). Then, additional filtering is performed on the surviving pairs using the secondary indexes (Lines 10–13). These filtering checks are applied on r_i and $s.id$, the corresponding secondary attribute of r and the record identifier of s, respectively. Finally, the secondary sets of r are indexed after the verification phase (Lines 18–19).

Note that only the identifier of the candidate records is needed to probe the secondary indexes. The performance benefits of using record identifiers are twofold: it avoids scanning the prefixes of the secondary sets for each candidate and allows fast searching in the underlying data structures. Thus, the overhead introduced by probing the secondary indexes in the filtering phase is minimized.

Algorithm 2: Algorithmic framework.

```
Input: A record collection C; the number of secondary indexes l; a list of similarity thresholds τ
Output: All pairs (r, s) s.t. sim(r, s) ≥ τ
I_1, I_2, ... I_|U| ← ∅
S^1 ... S^l ← BuildIndex
foreach r ∈ C do
    M ← an empty map from record to a similarity score
    foreach t ∈ pref(r_0, τ_0) do
        foreach s ∈ I_t do
            if Filter(r_0, s_0, τ_0) then
                M[s] ← −∞
            else
                for 1 ≤ i ≤ l do
                    if Filter(r_i, s.id, τ_i, S^i) then
                        M[s] ← −∞
                        break
                if M[s] ≠ −∞ then
                    M[s] ← M[s] + 1
        I_t ← I_t ∪ {r}
    Emit(Verify(x, M, τ))
    for 1 ≤ i ≤ l do
        Index(r_i, r.id, τ_i, S^i)
```

4.3. Secondary Indexes

Secondary indexes must enable prefix filtering using the secondary sets of the probing record and identifiers of candidate records. In addition, they must lend themselves to an efficient implementation. Figure 2 (b) depicts our proposed secondary index, which maps prefix tokens of secondary sets to inverted lists of record identifiers. The indexing of secondary sets is shown in Algorithm 3. In the filtering phase, we check whether the identifier of s appears in any inverted list associated with the prefix tokens of r; these steps are formalized in Algorithm 4.

Note that we can enable more filters by storing more information on the secondary indexes, such as set sizes and token positions. However, besides increasing index space, simply adopting all available filtering techniques may not improve performance. For example, a key observation in a recent experimental evaluation of several similarity join algorithms is that overly complex filters can instead increase execution runtime [Mann et al. 2016]. This observation matches our own experience and has motivated our design of lightweight indexes on secondary sets.

We implement the inverted lists of secondary indexes using set data structures. Thus, filtering on secondary sets is performed based on fast set membership checking. A potential issue is that the size of the inverted list can grow very large and consume significant memory resources. We can mitigate this problem using Bloom filter [Bloom 1970], a space-efficient, probabilistic data structure. On one hand, it produces no false negatives and, thus, no true matching pair is erroneously pruned, i.e., correctness is preserved. On the other hand, false positives are possible, which results in unnecessary comparisons in the verification phase. We compare Bloom filter against an exact set implementation in Section 5.

Algorithm 3: $Index\ (r_i, id, \tau_i, \mathcal{S}^i)$

```
1 foreach t ∈ pref (r_i, τ_i) do
2     S_l^i ← S_l^i ∪ {id}
```

Algorithm 4: $Filter\ (r_i, id, \tau_i, \mathcal{S}^i)$

```
1 return id ∉ ⋃_{t∈pref(r_i,τ_i)} S_t^i
```

4.4. Set Attribute Ordering

Identifying a suitable set attribute ordering is crucial to our approach since it determines the primary and secondary sets. Moreover, we can also apply this ordering to similarity computations in the verification phase. A natural choice is to sort the attributes in increasing order of processing cost. We can estimate the processing cost of a set attribute from the number of candidates generated from its prefix. Our cost model is defined as follows.

Definition 5 (Set Attribute Cost) *Let $pref_i$ be the set of all tokens appearing in the prefixes of the ith set attribute and $pf_i\,(t)$ be the frequency of token t in those prefixes. The cost of the ith set attribute, denoted by $\mathcal{P}_i$, is given by:*

$$\mathcal{P}_i = \sum_{t \in pref_i} \binom{pf_i\,(t)}{2}.$$

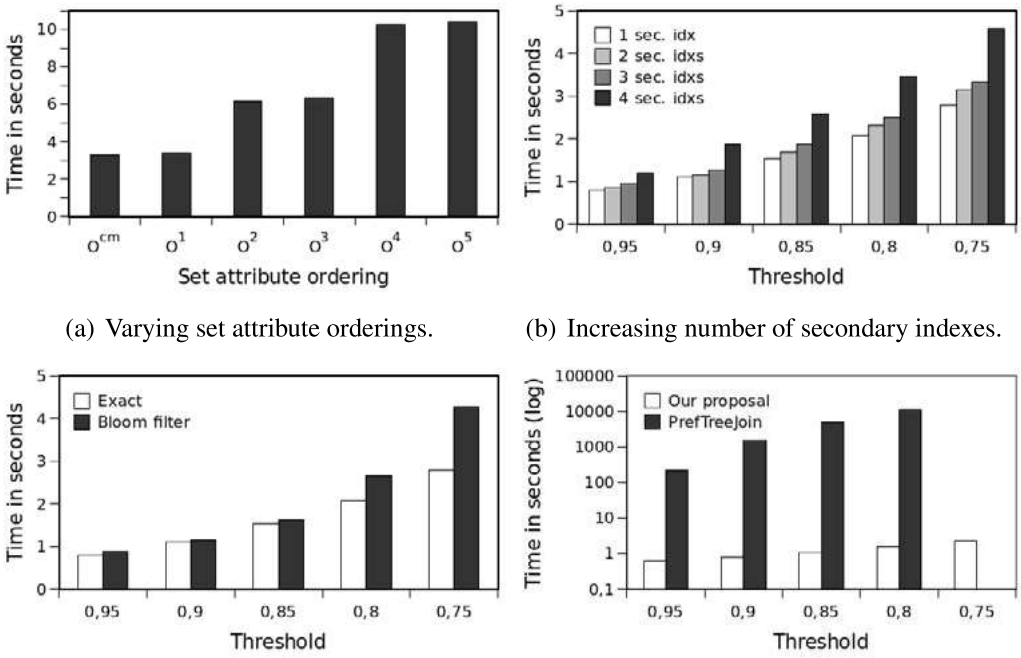
For example, in Table 2, we have the following processing costs: $\mathcal{P}_0 = 1 + 3 = 4$, $\mathcal{P}_1 = 0 + 3 + 1 = 4$, and $\mathcal{P}_2 = 0 + 6 = 6$.

5. Experiments

We now report the results of our experimental study. The goals of the empirical experiments are to evaluate the performance impact of 1) set attribute ordering based on our cost model, 2) number of secondary indexes, and 3) inverted list implementation, and 4) compare our proposal against PrefTreeJoin.

We used two, publicly available, real-world datasets: DBLP[1], containing Computer Science publications and IMDB[2], containing movie information. We generated two instances from each source dataset by randomly selecting 20k records with 3 and 5 string attributes: title and author names for DBLP and title and actor names for IMDB. Further, 4 duplicates were generated from each record, obtained by performing transformations on string attributes such as characters insertions, deletions, and substitutions (totaling 100k records). We converted strings to upper-case letters, eliminated repeated white spaces, and generated the corresponding token sets using q-grams of size 3. A single similarity predicate based on Jaccard was specified for each attribute, all predicates with a same threshold value within [0.75, 0.95]. We used the MPJoin algorithm [Ribeiro and Härder 2011] in our framework. Similarity computations in the verification phase were performed following the set attribute ordering for both MPJoin and PrefTreeJoin. All algorithms were implemented using Java JDK 11 (Oracle). Overall performance was measured in average

[1]http://dblp.uni-trier.de
[2]http://www.imdb.com



(a) Varying set attribute orderings.

(b) Increasing number of secondary indexes.

(c) Exact vs. Bloom filter.

(d) Our proposal vs. PrefTreeJoin.

Figure 3. Experimental results.

wall-clock time over repeated runs. We ran our experiments on an Intel Xeon E5-26200 six-core, 2 GHz, 15MB CPU cache, and 16 GB of main memory.

Figure 3 shows the results on the DBLP dataset; due to space constraints, we skip the results on IMDB as we observed similar trends. First, we evaluated the effectiveness of our cost model in identifying a suitable set attribute ordering. Figure 3(a) shows the timings of all set attribute permutations on the dataset instance with 3 attributes (threshold value fixed at 0.75 and 2 secondary indexes). The ordering derived from our cost model, denoted by O^{cm} in the figure, provides the best result, more than 3x times faster than the worst-performing ordering. In the following experiments, the set attributes were ordered according to O^{cm}.

Figure 3(b) shows the results for an increasing number of secondary indexes on the dataset with 5 attributes. The best configuration on this dataset used a single index and performance drops as more indexes are added. The reason for this behavior is that candidate pairs are filtered mostly due to the first index. Additional indexes marginally increase the number of pruned pairs and, thus, do not pay off the overhead of checking and maintaining them. For example, more than 2.6M pairs are filtered by the first index and only about 32K additional pairs are filtered with the inclusion of the second index. The following experiments used a single index.

Figure 3(c) shows the results of the comparison between an exact index implementation based on hash table and an approximate alternative based on Bloom filter (with an expected false positive probability of 3%). While performance is comparable at high thresholds, the Bloom filter variant is noticeably slower at low threshold values as the number of false positives increases leading to many unnecessary similarity comparisons.

Finally, we compared our proposal with PrefTreeJoin. Figure 3(d) shows the results on the dataset with 3 attributes. Our proposal is orders of magnitude faster than PrefTreeJoin. As already mentioned, the construction of the partial prefix tree is very costly and most of execution time was spent in this phase. Moreover, the prefix tree is too large for low thresholds. Indeed, PrefTreeJoin did not finish at the threshold value of 0.75 because it ran out of memory.

6. Related Work

There is a wealth of literature on efficiently answering set similarity joins [Chaudhuri et al. 2006, Xiao et al. 2011, Ribeiro and Härder 2011, Mann et al. 2016, Wang et al. 2017]. The vast majority of existing algorithms assume single-attribute data, in which a filtering-verification framework supported by an inverted index is prevalently adopted. In contrast to prior work on multi-attribute data [Li et al. 2015], our techniques can be readily integrated into such algorithms for dealing with multi-attribute data. Most proposals are geared towards the filtering phase, in which a variety of filters were developed to reduce the workload of the verification phase (see Section 2). Optimization techniques proposed for the verification phase include: accounting for previous matches in the filtering phase to skip initial set positions [Xiao et al. 2011]; leveraging token ordering to enable merge-like routines [Ribeiro and Härder 2011]; applying early termination conditions [Ribeiro and Härder 2011]; and exploiting overlap among the matches of different sets [Wang et al. 2017]. In [Li et al. 2015], an algorithm is proposed to determine the verification order of different similarity predicates. All these optimizations in the verification phase are orthogonal to our work here, which focuses on the filtering phase.

Recent work exploits massive parallelism available in modern graphics processing units to speed up similarity join processing [Ribeiro-Júnior et al. 2017]. Besides stand-alone algorithms, set similarity joins can be realized using relational database technology. Previous work proposed expressing set similarity joins declaratively in SQL [Ribeiro et al. 2016] or implementing it within the query engine [Chaudhuri et al. 2006].

In another widely used approach, strings are numerically represented by high dimensional vectors, where each dimension is a word (or token) extracted from the dataset. A weighting scheme is typically employed to produce weighted vectors. The similarity between two vectors is then determined by the cosine of the angle between them, which reduces to the dot-product for l^2 normalized vectors. Similarity join on vectors is often referred to as *All Pairs Similarity Search* [Bayardo et al. 2007]. Several optimization techniques for sets can be adapted to vectors, including size-based filter, index reduction based on data ordering, and most importantly to our context, prefix filter [Bayardo et al. 2007]. Therefore, our filters can be adapted as well to optimize similarity join on vectors. We leave the evaluation of this approach for future work.

7. Conclusions

In this paper, we proposed a framework to enhance set similarity join algorithms for dealing with multi-attribute data. Our framework allows easy integration of additional filters into existing algorithms for single-attribute data. We further instantiate the framework with a simple, yet effective filter based on lightweight indexes. Implementation alternatives were evaluated for this index using exact and probabilistic data structures. We

proposed a cost model to identify the best ordering of set attributes to reduce processing time. Our performance study demonstrates that our approach is effective and significantly outperforms the existing algorithm for multi-attribute data.

Acknowledgment This work was partially supported by the Brazilian agency CAPES.

References

Bayardo, R. J., Ma, Y., and Srikant, R. (2007). Scaling up All Pairs Similarity Search. In *Proceedings of the WWW Conference*, pages 131–140.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426.

Chaudhuri, S., Ganti, V., and Kaushik, R. (2006). A Primitive Operator for Similarity Joins in Data Cleaning. In *Proceedings of the ICDE Conference*, page 5.

Chu, X., Ilyas, I. F., Krishnan, S., and Wang, J. (2016). Data Cleaning: Overview and Emerging Challenges. In *Proceedings of the SIGMOD Conference*, pages 2201–2206.

CrowdFlower (2016). 2016 Data Science Report. https://visit.figure-eight.com/data-science-report.html.

do Carmo Oliveira, D. J., Borges, F. F., and Ribeiro, L. A. (2017). Uma abordagem para processamento distribuído de junção por similaridade sobre múltiplos atributos. In *Proceedings of the Brazilian Symposium on Databases*, pages 300–305.

Kaggle (2017). The State of Data Science & Machine Learning. https://www.kaggle.com/kaggle/kaggle-survey-2017.

Li, G., He, J., Deng, D., and Li, J. (2015). Efficient Similarity Join and Search on Multi-Attribute Data. In *Proceedings of the SIGMOD Conference*, pages 1137–1151.

Mann, W., Augsten, N., and Bouros, P. (2016). An Empirical Evaluation of Set Similarity Join Techniques. *PVLDB*, 9(9):636–647.

Oliveira, D. J. C., Borges, F. F., Ribeiro, L. A., and Cuzzocrea, A. (2018). Set Similarity Joins with Complex Expressions on Distributed Platforms. In *Proceedings of the Symposium on Advances in Databases and Information Systems*, pages 216–230.

Ribeiro, L. A. and Härder, T. (2011). Generalizing Prefix Filtering to Improve Set Similarity Joins. *Information Systems*, 36(1):62–78.

Ribeiro, L. A., Schneider, N. C., de Souza Inácio, A., Wagner, H. M., and von Wangenheim, A. (2016). Bridging Database Applications and Declarative Similarity Matching. *Journal of Information and Data Management*, 7(3):217–232.

Ribeiro-Júnior, S., Quirino, R. D., Ribeiro, L. A., and Martins, W. S. (2017). Fast Parallel Set Similarity Joins on Many-core Architectures. *Journal of Information and Data Management*, 8(3):255–270.

Wang, X., Qin, L., Lin, X., Zhang, Y., and Chang, L. (2017). Leveraging Set Relations in Exact Set Similarity Join. *Proceedings of the VLDB Endowment*, 10(9):925–936.

Xiao, C., Wang, W., Lin, X., Yu, J. X., and Wang, G. (2011). Efficient Similarity Joins for Near-Duplicate Detection. *ACM Transactions on Database Systems*, 36(3):15:1–15:41.