Efficient Set Similarity Join on Multi-Attribute Data Using Lightweight Filters

Leonardo Andrade Ribeiro, Felipe Ferreira Borges, Diego Oliveira

Universidade Federal de Goiás, Brazil
{laribeiro,felipeferreiraborges,diegooliveira}@inf.ufg.br

Abstract. We consider the problem of efficiently answering set similarity joins on multi-attribute data. Traditional set similarity join algorithms assume string data represented by a single set and, thus, miss the opportunity to exploit predicates over multiple attributes to reduce the number of similarity computations. In this article, we present a framework to enhance existing algorithms with additional filters for dealing with multi-attribute data. We then instantiate this framework with a lightweight filtering technique based on a simple, yet effective data structure, for which exact and probabilistic implementations are evaluated. In this context, we devise a cost model to identify the best attribute ordering to reduce processing time. Moreover, alternative approaches are also investigated and a new algorithm combining key ideas from previous work is introduced. Finally, we present a thorough experimental evaluation, which demonstrates that our main proposal is efficient and significantly outperforms competing algorithms.

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

Keywords: Advanced Query Processing, Data Cleaning, Data Integration, Multi-Attribute Data, Similarity Join

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 the 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 [Sarawagi and Kirpal 2004; Chaudhuri et al. 2006; Xiao et al. 2011; Ribeiro and Härder 2011; Mann et al. 2016; Wang et al. 2017; Deng et al. 2018], 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.

Copyright©2021 Permission to copy without fee all or part of the material printed in JIDM is granted provided that the copies are not made or distributed for commercial advantage, and that notice is given that copying is by permission of the Sociedade Brasileira de Computação.

Table I. 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

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 or concatenating string values from multiple attributes, such approach may produce unsatisfactory results. For example, consider the sample database shown in Table I. 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. 2017; Oliveira et al. 2018] 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 article, 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 lightweight filtering technique 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.

This article is an extended and revised version of a previous conference paper [Ribeiro et al. 2020]. As part of the new material, we present a novel algorithm based on the central concept behind the algorithm of Li et al. while avoiding some of its shortcomings by adopting the processing paradigm of existing solutions. By comparing our main proposal with such competing approaches based on the same fundamental concept, we can better identify key aspects with a major impact on runtime performance. Accordingly, we provide a more detailed and comprehensive set of experiments.

The rest of this article is organized as follows. Section 2 provides background material. Section 3 formally describes the problem, overviews existing techniques, and present a new algorithm derived from these techniques. Section 4 presents our filter-based approach. 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-and-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., substrings 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 2.1 *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 2.1. *Consider the sets* $r = \{\boldsymbol{A}, \boldsymbol{B}, C, \boldsymbol{D}, \boldsymbol{E}, F, \boldsymbol{G}, \boldsymbol{H}\}$ *and* $s = \{\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{D}, \boldsymbol{E}, \boldsymbol{G}, \boldsymbol{H}\}$. *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 2.2 *Set Similarity Join.* Given two set collections $\mathcal{C}_1$ and $\mathcal{C}_2$, a set similarity function *sim*, and a similarity threshold τ in the interval $[0, 1]$, the Set Similarity Join between $\mathcal{C}_1$ and $\mathcal{C}_2$ returns all set pairs $(r, s) \in \mathcal{C}_1 \times \mathcal{C}_2$ 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*. Formally, given two sets r and s, then $sim(r, s) \geq \tau$ iff $|r \cap s| \geq \frac{\tau \times (|r| + |s|)}{1 + \tau}$ [Chaudhuri et al. 2006]. As a result, 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* [Sarawagi and Kirpal 2004; 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

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))
```

tokens from the sets in $\mathcal{C}$ are drawn. We formally define the concept of prefix and the prefix filtering principle as follows.

Definition 2.3 *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 2.4 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 2.2. *Consider again the sets r and s in Example 2.1; 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 set collections[1]. 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* [Sarawagi and Kirpal 2004] 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 $\left[|r| \times \tau, |r| \times \tau^{-1}\right]$. 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

[1] A secondary ordering, e.g., lexicographic ordering, is used to break ties consistently.

Table II. 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

(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 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). Verification can be highly optimized by exploiting the token ordering in a merge-like fashion. Furthermore, the overlap bound can be used to define early stopping conditions: at each iteration, it is checked whether the overlap bound has already been met or cannot be reached anymore from the current matching position [Ribeiro and Härder 2011]. In fact, it has been shown that a fast verification procedure can overshadow the performance gains of complex filters [Mann et al. 2016].

Although not shown in the algorithm, the set collection can be sorted according to set size to enable further index reduction either at indexing time [Bayardo et al. 2007] or dynamically during the filtering phase [Ribeiro and Härder 2011]. It has been shown that the reduction in runtime achieved by such optimizations largely compensates for the additional sorting cost. Finally, note that Algorithm 1 is actually a self-join on a single set collection. Its extension to binary joins is trivial: we first index the smaller collection and then go through the larger collection to identify matching pairs. For simplicity and without loss of generality, we will assume self-joins in the rest of this article.

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. Finally, we present a new algorithm incorporating the central concept of the algorithm of Li et al. into the filtering-and-verification approach described in the previous section.

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 ith 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 3.1 *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. $sim\,(r, s) = true$.

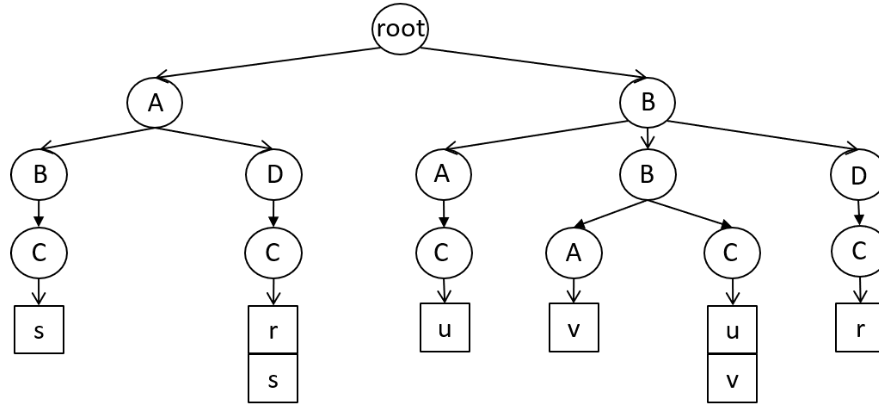


Fig. 1. Prefix tree index for the record collection in Table II.

Algorithm 2: The PrefixTreeJoin algorithm.

Input: A record collection $\mathcal{C}$, a similarity expression *sim*

Output: A set $\mathcal{A}$ containing all pairs (r, s) s.t. $sim\,(r, s) \geq \tau$

1 Build a prefix tree $\mathcal{P}$ with $\mathcal{C}$;
2 **foreach** *inverted list* $\mathcal{L}$ *under each leaf node of* P **do**
3   **foreach** $(r, s) \in \mathcal{L}$ **do**
4   **if** $sim\,(r, s) \geq \tau$ **then**
5             $\mathcal{A} \leftarrow \mathcal{A} \cup (r, s)$
6 **return** $\mathcal{A}$

3.2 The PrefTreeJoin Algorithm

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.* Prefix tokens have holistic pruning power on multiple similarity predicates: 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 II; for simplicity, only the prefixes are shown. There are two candidate pairs: (r, s), which shares the record token $A \circ D \circ C$, and (u, v), which shares the record token $B \circ B \circ C$.

Li et al. proposed the PrefTreeJoin algorithm [Li et al. 2015], which builds a builds a *prefix tree* to quickly identify pairs with record tokens in common. In the prefix tree, 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 containing the corresponding record token in their prefix. Figure 1 shows the prefix tree for the records in Table II. PrefTreeJoin first builds the prefix tree before comparing all record pairs appearing in the inverted lists as described in Algorithm 2. Because the same record pair can appear in more than one inverted list, a hash table is used to avoid unnecessary comparisons and duplicate pairs in the result.

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

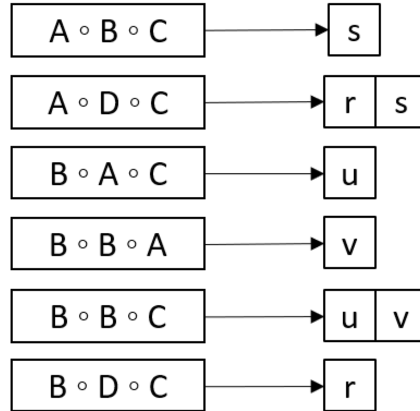


Fig. 2. Inverted index for the record collection in Table II.

Algorithm 3: The PrefInvJoin Algorithm.

Input: A record collection $\mathcal{C}$, a list of similarity thresholds τ
Output: All pairs (r, s) s.t. $sim\,(r, s) \geq \tau$

1 $I_1, \ldots I_{|pref_0| \times \ldots \times |pref_n|} \leftarrow \varnothing$
2 **foreach** $r \in \mathcal{C}$ **do**
3 $\quad M \leftarrow \varnothing$
4 $\quad$ **foreach** $t_0 \circ \ldots \circ t_n \in pref\,(r_0, \tau_1) \times \ldots \times pref\,(r_n, \tau_n)$ **do**
5 $\quad\quad M \leftarrow M \cup I_{t_0 \circ \ldots \circ t_n}$
6 $\quad\quad I_{t_0 \circ \ldots \circ t_n} \leftarrow I_{t_0 \circ \ldots \circ t_n} \cup \{r\}$
7 $\quad$ *Emit* (*Verify* (x, M, τ))

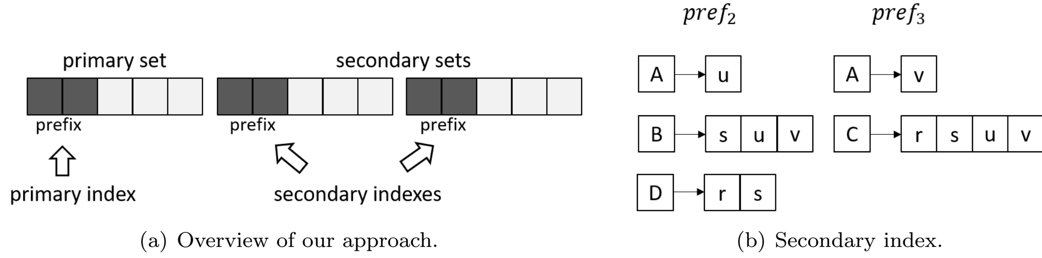
---

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, an additional pass over the data in a preprocessing phase is needed to collect these statistics.

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.

3.3 The PrefInvJoin Algorithm

We now present a new algorithm called *PrefInvJoin*, which integrates record tokens into a filtering-and-verification approach. To this end, PrefInfJoin uses an inverted index to find all records with overlapping record tokens; Figure 3.3 illustrates this idea. The steps of PrefInvJoin are formalized in Algorithm 3. For each record r in the input collection, PrefInvJoin generates its record prefix (Line 4) by computing the Cartesian product of the prefixes of all set attributes. For each record token in the prefix of r, the records in the associated inverted list are added to the set of matching candidates (Line 5) before appending a reference to r to this list (Line 6). Finally, record r is then compared against each of its candidates and those record pairs satisfying the similarity threshold are sent to the output (Line 7).



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

Fig. 3. Our proposed solution.

Despite its simplicity, the PrefInvJoin algorithm avoids all the drawbacks of PrefTreeJoin. The inverted index is more compact than the prefix tree and is dynamically constructed as the record collection is processed. Since now we only perform equality checks on the whole record tokens to find candidate pairs, we can simply hash them into 32 bit integers to both reduce the memory footprint and speed up access time. Note that hash collisions do not affect the correctness but could introduce additional false positives. In practice, the effect of such collisions is nevertheless negligible. Finally, in a similar fashion to Algorithm 1, PrefInv-Join can produce results at each iteration.

4. OUR SOLUTION

In this section, we present our main 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 lightweight filtering technique based on simple 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.

Similar to PrefInvJoin, the above approach also avoids the drawbacks of PrefTreeJoin. However, we miss the opportunity to exploit multiple similarity predicates to reduce the number of similarity computations. Indeed, using the prefixes of a single set in the filtering phase has less pruning power than using record prefixes. For example, in Table II, selecting the first attribute as primary set generates 4 candidate pairs, namely (r, s), (r, u), (r, v), and (u, v), whereas using record prefixes generates only 2.

In this context, the main idea behind our proposal is to enhance the filtering phase by using additional, simple index structures for filtering on the secondary sets. Figure 3 (a) illustrates our approach. 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.

Algorithm 4: Algorithmic framework.

Input: A record collection $\mathcal{C}$; number of secondary indexes l; a list of similarity thresholds τ
Output: All pairs (r, s) s.t. $sim\,(r, s) \geq \tau$

1 $I_1, I_2, \ldots I_{|\mathcal{U}|} \leftarrow \varnothing$
2 $\mathcal{S}^1 \ldots \mathcal{S}^l \leftarrow BuildIndexes$
3 **foreach** $r \in \mathcal{C}$ **do**
4 $\quad M \leftarrow$ an empty map from record to a similarity score
5 $\quad$ **foreach** $t \in pref\,(r_0, \tau_0)$ **do**
6 $\quad\quad$ **foreach** $s \in I_t$ **do**
7 $\quad\quad\quad$ **if** $Filter\,(r_0, s_0, \tau_0)$ **then**
8 $\quad\quad\quad\quad M[s] \leftarrow -\infty$
9 $\quad\quad\quad$ **else**
10 $\quad\quad\quad\quad$ **for** $1 \leq i \leq l$ **do**
11 $\quad\quad\quad\quad\quad$ **if** $Filter\left(r_i, s.id, \tau_i, \mathcal{S}^i\right)$ **then**
12 $\quad\quad\quad\quad\quad\quad M[s] \leftarrow -\infty$
13 $\quad\quad\quad\quad\quad\quad$ **break**
14 $\quad\quad\quad\quad$ **if** $M[s] \neq -\infty$ **then**
15 $\quad\quad\quad\quad\quad M[s] \leftarrow M[s] + 1$
16 $\quad\quad I_t \leftarrow I_t \cup \{r\}$
17 $\quad Emit\,(Verify\,(x, M, \tau))$
18 $\quad$ **for** $1 \leq i \leq l$ **do**
19 $\quad\quad Index\left(r_i, r.id, \tau_i, \mathcal{S}^i\right)$

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 4. 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.

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 3 (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 5. 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 6.

Algorithm 5: *Index* $\left(r_i, id, \tau_i, \mathcal{S}^i\right)$

1 **foreach** $t \in \mathit{pref}\left(r_i, \tau_i\right)$ **do**
2 $\quad \mathcal{S}^i_t \leftarrow \mathcal{S}^i_t \cup \{id\}$

Algorithm 6: *Filter* $\left(r_i, id, \tau_i, \mathcal{S}^i\right)$

1 **return** $id \notin \bigcup_{t \in \mathit{pref}(r_i, \tau_i)} \mathcal{S}^i_t$

We call the algorithm obtained from instantiating our framework with secondary indexes FSSJoin (filtered set similarity join). 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 filters based 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. In other words, Bloom filter essentially trades runtime-efficiency for space-efficiency. We compare Bloom filter against an exact set implementation in Section 5.

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

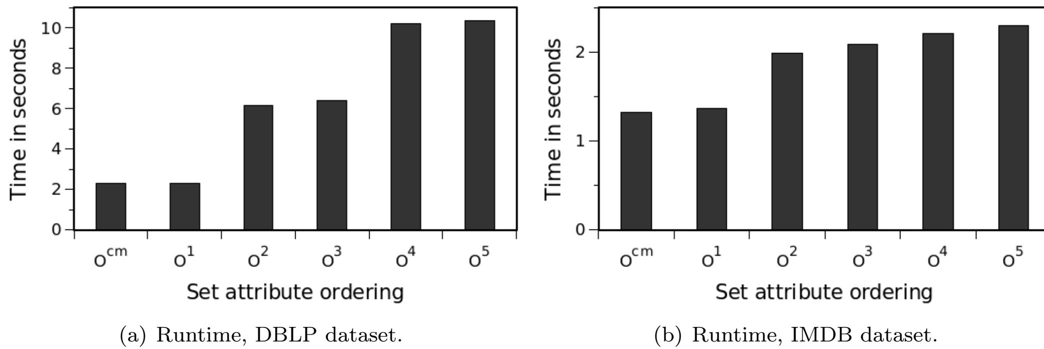
$$\mathcal{P}_i = \sum_{t \in \mathit{pref}_i} \binom{\mathit{pf}_i\left(t\right)}{2}.$$

Example 4.1. *Consider the record prefixes in Table II. 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, 4) compare FSSJoin against competing algorithms, namely PrefTreeJoin and PrefInvJoin, and 5) test the scalability of FSSJoin.

Dataset	Atribute	Avg. set size	Max. set size	# of distinct tokens
DBLP	title	73,23	250	32513
	author	15,754	40	23302
IMDB	title	19,72	122	37623
	actor	15,946	40	25037



(a) Runtime, DBLP dataset.

(b) Runtime, IMDB dataset.

Fig. 4. Varying set attribute ordering.

5.1 Experimental Setup

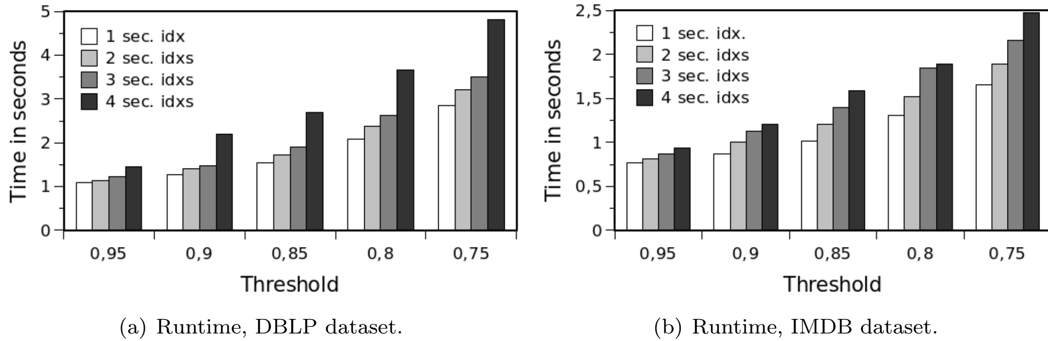
We used two, publicly available, real-world datasets: DBLP[2], containing Computer Science publications and IMDB[3], containing movie information. We generated instances from each source dataset by randomly selecting records with multiple string attributes: title and author names for DBLP and title and 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. We converted strings to upper-case letters and eliminated repeated white spaces. Each string was then mapped to a set of tokens by using q-grams of size 3, hashing each token into an integer value, and ordering the tokens within a set according to their frequency in the collection. We generated several datasets from each source with different numbers of records and attributes in the experiments; the default values are 100k records and three attributes. Table 5.1 shows some important statistics about the datasets (with 100k records).

A single similarity predicate based on Jaccard was specified for each attribute, all predicates with the same threshold value within [0.5, 0.95]; the default threshold value is 0.75. The implementation of FSSJoin was based on the MPJoin algorithm [Ribeiro and Härder 2011]. Similarity computations in the verification phase were performed following the set attribute ordering for all algorithms; we evaluated the hybrid verification method proposed by Li et al. [Li et al. 2015] but did not observe any performance gain. We implemented all algorithms using Java JDK 11 (Oracle). Overall performance was measured in average 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.

5.2 Evaluation of FSSJoin

We begin with an analysis of the main components of our FSSJoin. First, we evaluated the effectiveness of our cost model in identifying a suitable set attribute ordering. Figure 4 show the timings of all set attribute permutations on dataset instances with 3 attributes (threshold value fixed at 0.75 and 1 secondary index). The ordering derived from our cost model is denoted by O^{cm} in the experimental charts. O^{cm} provides the best result on both datasets. As compared to the worst-performing orderings,

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



(a) Runtime, DBLP dataset.

(b) Runtime, IMDB dataset.

Fig. 5. Increasing number of secondary indexes.

O^{cm} is about 4.4x times faster on DBLP dataset (Figure 4(a)) and about 1.7. faster on IMDB (Figure 4(b)). In the following experiments, the set attributes were ordered according to O^{cm}.

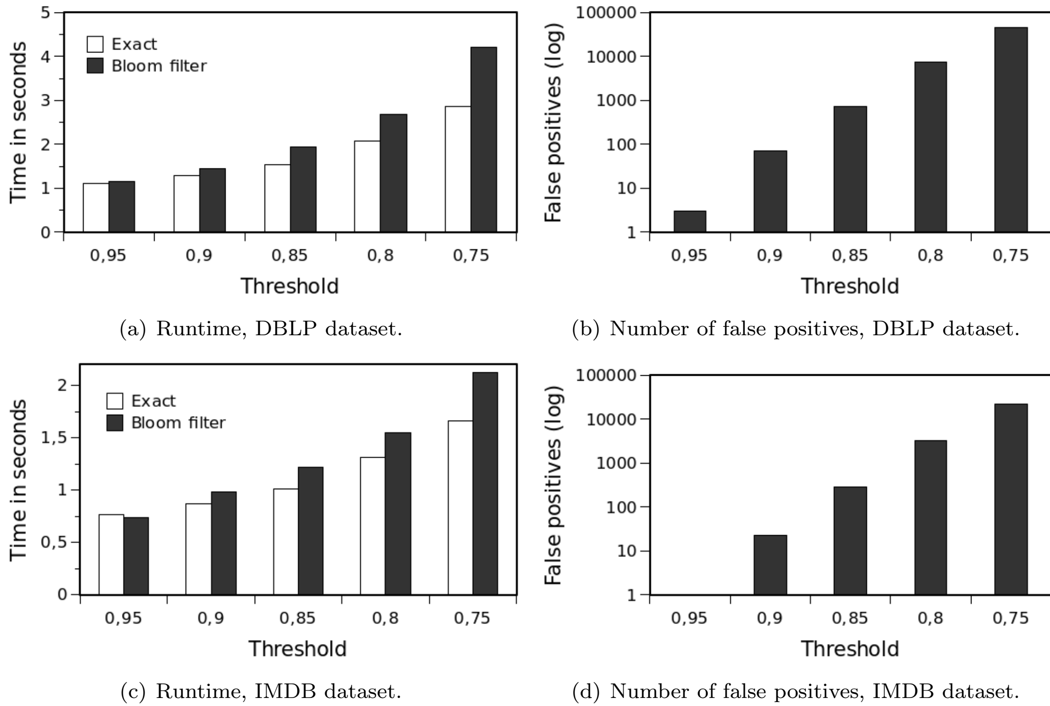
Next, we evaluate the effect of secondary indexes on performance. Figure 5 shows the results for an increasing number of secondary indexes on datasets with 5 attributes. The best configuration on these datasets used a single index and performance drops as more indexes are added. The reason for this behavior is that candidate pairs are filtered mostly due to the first index. For this experiment, more than one secondary index yielded diminishing returns, i.e., they did not pay off the overhead of checking and maintaining them. For example, more than 2.6M pairs are filtered by the first index on DBLP and only about 32K additional pairs are filtered with the inclusion of the second index. Another aspect is that the verification procedure is already highly optimized (recall Section 2.3). Thus, much more pairs need to be pruned to compensate for the additional filtering cost. The following experiments used a single index.

We now compare two implementations of secondary indexes: an exact index implementation based on a hash table and an approximate alternative based on Bloom filter (with an expected false positive probability of 3%). Figure 6 shows the results. While performance is comparable at high thresholds, the Bloom filter variant is noticeably slower at low threshold values on both datasets (Figures 6(a) and 6(c)). The number of false positives produced by the approximate implementation steadily increases as the threshold decreases (Figures 6(b) and 6(d)). As a result, much more unnecessary similarity comparisons are performed at lower thresholds. The following experiments used the exact implementation of secondary indexes.

5.3 Comparison with Competing Algorithms and Scalability Tests

We now compare FSSJoin with PrefTreeJoin and PrefInvJoin. Figures 7(a) and 7(b) show the results for varying threshold values on DBLP and IMDB, respectively. FSSJoin is the faster algorithm in all settings. Even though record prefixes have more pruning power than prefixes derived from a single attribute, the cost of their generation outweighs the benefits of a reduced workload for the verification procedure (which is very fast, as already mentioned). For PrefTreeJoin, the construction of the partial prefix tree is very expensive computationally and, indeed, most of the execution time is spent in this phase. PrefInvJoin is faster than PrefTreeJoin because it identifies records with overlapping prefixes using an inverted index, which is cheaper to construct than a prefix tree. However, the computation of the Cartesian product over the prefixes of all set attributes is nevertheless onerous and negatively impacts runtime performance.

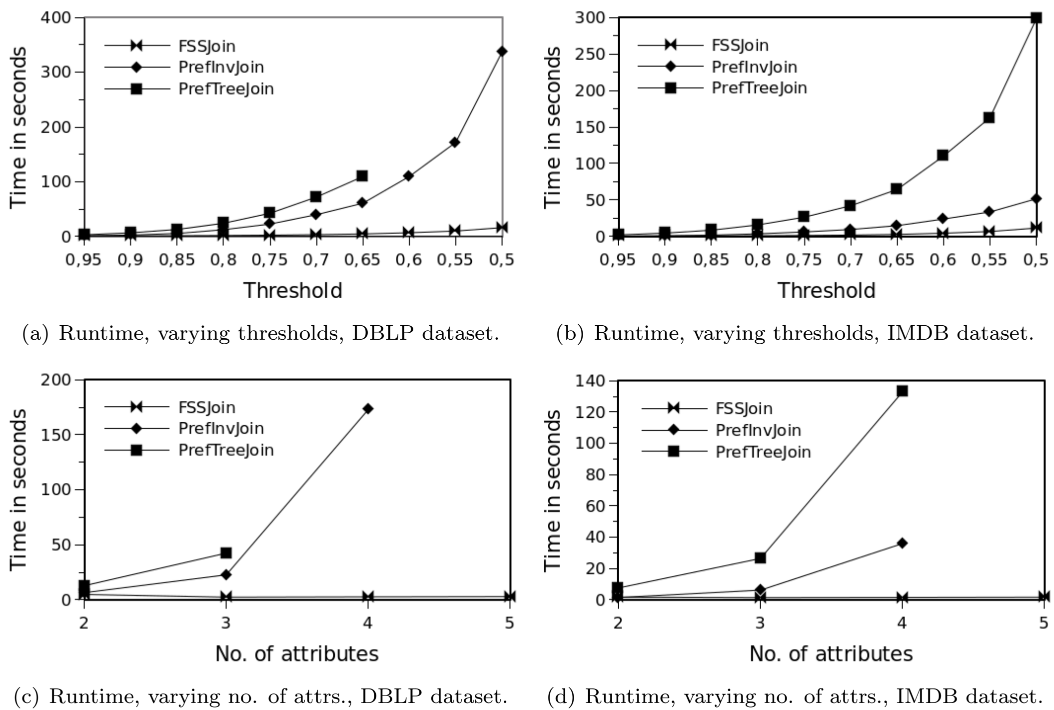
Furthermore, the performance gap between FSSJoin and its competitors increases as the threshold decreases. The reason for this behavior is that lower thresholds translate into larger prefixes, which leads to an explosion in the number of generated record tokens and a corresponding drop in both



(a) Runtime, DBLP dataset. (b) Number of false positives, DBLP dataset.

(c) Runtime, IMDB dataset. (d) Number of false positives, IMDB dataset.

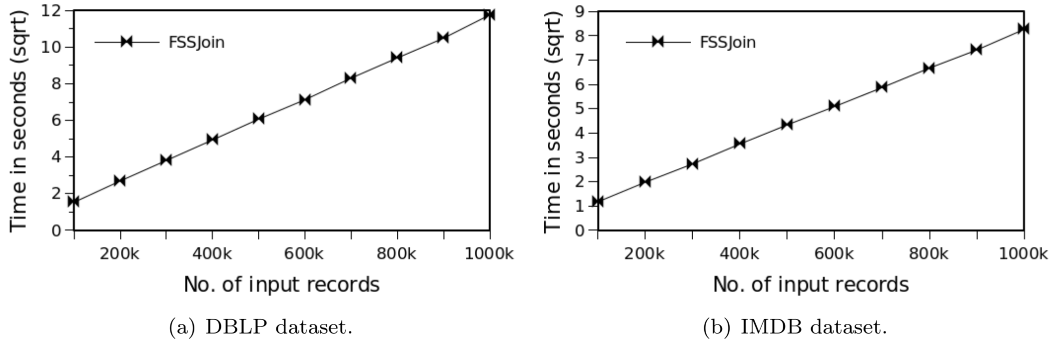
Fig. 6. Comparison between exact and approximate implementations.



(a) Runtime, varying thresholds, DBLP dataset. (b) Runtime, varying thresholds, IMDB dataset.

(c) Runtime, varying no. of attrs., DBLP dataset. (d) Runtime, varying no. of attrs., IMDB dataset.

Fig. 7. FSSJoin vs. competing algorithms.



(a) DBLP dataset.

(b) IMDB dataset.

Fig. 8. Scalability results.

time and space efficiency. For example, PrefInvJoin is 276x slower at threshold 0.5 on DBLP than at threshold 0.95, whereas FSSJoin is only 18x. PrefTreeJoin is even worse, as it ran out of memory for thresholds lower than 0.65.

The size of record prefixes grows exponentially with the number of set attributes in the similarity expression. Figures 7(c) and 7(d) show the results for varying number of attributes on DBLP and IMDB, respectively (2 to 5 attributes, threshold fixed at 0.75). PrefTreeJoin and PrefInvJoin did not finish with 5 attributes on both datasets owing to the combinatorial explosion in the number of generated record tokens; the former also did not finish with 4 attributes on DBLP. In contrast, the runtime of FSSJoin does not depend on the number of set attributes as prefixes are generated from set attributes in isolation. In fact, FSSJoin is sometimes even faster with more set attributes in the similarity expression because more candidate pairs are pruned in the filtering phase.

Finally, we evaluate the scalability of FSSJoin. For this experiment, we used datasets with an increasing number of records: from 100K to 1000K. Figure 8 shows the results. The runtime of FSSJoin on both datasets exhibits a quadratic growth (note that we show the square root of the runtime). This behavior is expected because the number of candidate pairs also grows quadratically with the input size.

5.4 Experimental Summary

We conducted an extensive experimental analysis to evaluate our proposal and compare it with other approaches. Our cost model allowed us to successfully identify the set attribute ordering leading to the best results. On the datasets considered, a single secondary index provided the best trade-off between the number of filtered candidate pairs and the additional cost in the filtering phase; additional indexes yielded diminishing returns. The approximate implementation of secondary indexes was competitive in terms of runtime performance only at high threshold values; at lower thresholds, the number of false positives increases substantially causing many unnecessary comparisons.

FSSJoin was the best-performing algorithm in all settings both in terms of runtime efficiency and memory consumption. PrefTreeJoin performed worse than PrefInvJoin due to the high cost associated with the construction of the prefix tree. As the input size increases, the runtime of FSSJoin follows the quadratic growth of the number of candidate pairs.

As a general conclusion, the experimental results showed that approaches based on the concept of record tokens do not scale at low threshold values and with the number of set attributes due to the explosion in the number of such tokens that are produced in the filtering phase. Finally, our results suggest a fast verification procedure calls for lightweight filters to improve overall runtime.

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.

A number of distributed algorithms for set similarity have been proposed [Fier et al. 2018]. A common approach consists of applying a partition scheme to send dissimilar strings to different processing nodes and, thus, avoid unnecessary similarity calculations; to some extent, the partition scheme plays the role of the filtering phase in centralized algorithms. Almost all proposals considered single-attribute data. One exception is work in [Oliveira et al. 2017; Oliveira et al. 2018]. Evaluating our filters in a distributed setting is an interesting line of future research.

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 as a physical operator [Chaudhuri et al. 2006].

All algorithms discussed above are exact, i.e., they always return all similar pairs. Approximate set similarity joins may miss some valid results to trade accuracy for query time. Locality Sensitive Hashing (LSH) is the most popular technique for approximate set similarity joins [Indyk and Motwani 1998], which is based on a probabilistic scheme of hashing functions that are approximately similarity-preserving. Minhash [Broder et al. 1998] is a popular LSH scheme for the Jaccard similarity.

String similarity join can also employ constraints based on the edit distance, which is defined by the minimum number of character-editing operations — insertion, deletion, and substitution — to make two strings equal [Navarro 2001]. As for token-based similarity, we can map strings to sets and derive set overlap bounds for the edit distance [Ribeiro et al. 2016] and apply prefix filtering to reduce the number of expensive distance computations in the verification phase. Therefore, our filters can be straightforwardly used with similarity predicates based on edit distance.

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 article, 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 instantiated with an algorithm called FSSJoin based on a lightweight filtering technique supported by a simple, yet effective index. 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. We also investigated alternative approaches and presented a new algorithm addressing some shortcomings of previous work. Our performance study demonstrated that FSSJoin outperformed its competitors by a large margin.

REFERENCES

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

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

Broder, A. Z., Charikar, M., Frieze, A. M., and Mitzenmacher, M. Min-Wise Independent Permutations (Extended Abstract). In *Proceedings of the STOC Symposium*. Dallas, USA, pp. 327–336, 1998.

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

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

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

Deng, D., Tao, Y., and Li, G. Overlap Set Similarity Joins with Theoretical Guarantees. In *Proceedings of the SIGMOD Conference*. Houston, USA, pp. 905–920, 2018.

Fier, F., Augsten, N., Bouros, P., Leser, U., and Freytag, J. Set Similarity Joins on MapReduce: An Experimental Survey. *Proceedings of the VLDB Endowment* 11 (10): 1110–1122, 2018.

Indyk, P. and Motwani, R. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the STOC Symposium*. Dallas, USA, pp. 604–613, 1998.

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

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

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

Navarro, G. A Guided Tour to Approximate String Matching. *Communications of the ACM* 33 (1): 31–88, 2001.

Oliveira, D., Borges, F. F., and Ribeiro, L. A. Uma abordagem para processamento distribuído de junção por similaridade sobre múltiplos atributos. In *Proceedings of the Brazilian Symposium on Databases*. Uberlãndia, Minas Gerais, Brazil, pp. 300–305, 2017.

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

Ribeiro, L. A., Borges, F. F., and Oliveira, D. A Framework for Set Similarity Join on Multi-Attribute Data. In *Proceedings of the Brazilian Symposium on Databases*. Porto Alegre, Brazil, pp. 61–72, 2020.

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

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

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

Sarawagi, S. and Kirpal, A. Efficient Set Joins on Similarity Predicates. In *Proceedings of the SIGMOD Conference*. Paris, France, pp. 743–754, 2004.

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

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