# Fast Indexes and Algorithms for Set Similarity Selection Queries

Marios Hadjieleftheriou †, Amit Chandel ‡, Nick Koudas ‡, Divesh Srivastava †

†*AT&T Labs–Research*
*Florham Park, NJ 07932, USA*
marioh@research.att.com
divesh@research.att.com

‡*Department of Computer Science, University of Toronto*
*Toronto, ON M5S 2E4, Canada*
amit@cs.toronto.edu
koudas@cs.toronto.edu

***Abstract*— Data collections often have inconsistencies that arise due to a variety of reasons, and it is desirable to be able to identify and resolve them efficiently. Set similarity queries are commonly used in data cleaning for matching *similar* data. In this work we concentrate on *set similarity selection* queries: Given a query set, retrieve all sets in a collection with similarity greater than some threshold. Various set similarity measures have been proposed in the past for data cleaning purposes. In this work we concentrate on weighted similarity functions like TF/IDF, and introduce variants that are well suited for set similarity selections in a relational database context. These variants have special semantic properties that can be exploited to design very efficient index structures and algorithms for answering queries efficiently. We present modifications of existing technologies to work for set similarity selection queries. We also introduce three novel algorithms based on the Threshold Algorithm, that exploit the semantic properties of the new similarity measures to achieve the best performance in theory and practice.**

## I. Introduction

Data collections often have inconsistencies that arise due to a variety of reasons, such as typographic mistakes, formatting conventions, data transformation errors and more. Consistent or clean data are of high monetary significance for business practices; it is desirable to be able to identify and resolve such inconsistencies efficiently. For that purpose, various *set similarity join* operators have been proposed in the past [1], [2], [3]. The main idea behind such operators is to view operands as sets of tokens and evaluate the similarity of the operand sets. If the similarity is high enough the operand pair is flagged as being of interest (e.g., potential duplicate). Several such similarity operators have been proposed in the literature. The bulk of algorithm development in this area has concentrated on the efficient execution of join operations between data sets, incorporating such similarity operators as a join predicate (see Section IX). In this work we concentrate on *set similarity selection* queries. Informally, a set similarity selection query retrieves all sets from a data collection whose similarity with the query set exceeds a user specified threshold.

A large number of set similarity measures have been proposed in the past, like intersection, Jaccard and cosine similarity. It has been demonstrated that no single similarity function is best across all application domains [4]. In this work we concentrate on well known and largely deployed weighted similarity measures like TF/IDF cosine similarity and BM25 [5].

There are two main approaches in the design of efficient set similarity selection algorithms. The first is based on relational database technology, e.g., using tables, indexes, SQL and UDFs [6], [2]. The second approach is to design specialized disk resident indexes, in the form of inverted lists, and then deploy variants of the Threshold Algorithm (TA/NRA) to evaluate similarity [7], [3].

In this paper, for the problem of set similarity selection in a relational database context, we introduce simple variants of traditional TF/IDF, BM25 weighted measures that exhibit certain very desirable *semantic properties*; such properties can be exploited to design specialized algorithms for performing the search orders of magnitude faster than straightforward approaches. These semantic properties enable pruning a very large percentage of the search space instantaneously, resulting in huge performance benefits both for approaches utilizing indexes based on existing relational technology as well as for approaches utilizing inverted lists.

We first show how existing approaches (Section III) can take advantage of semantic properties (presented in Section IV). Next, we design three specialized list merging algorithms: The first (Section V) is an improved version of NRA, based on breadth first search. The second (Section VI) is based on depth first search and is characterized by very low bookkeeping cost. The third (Section VII) is a hybrid combination of the two, achieving the most efficient pruning in theory and practice, with a slightly increased bookkeeping cost. We experimentally demonstrate (Section VIII) that our algorithms achieve arbitrarily stronger pruning than NRA for certain instances, resulting in improved performance. This follows from the fact that NRA is instance optimal over a class of algorithms that does not capture the new techniques proposed herein.

## II. Preliminaries

Consider two strings $s_1$ ="Main St., Main" and $s_2$ = "Main St., Maine", mapped into token multi-sets {'Main', 'St.', 'Main'} and {'Main', 'St.', 'Maine'}. The two multi-sets share two tokens in common. Clearly, the larger the intersection of the two multi-sets, the larger the potential similarity. Nevertheless, not all tokens are equally important. Tokens that appear very frequently in the database (like 'Main' or 'St.') carry small information content, whereas rare tokens (like 'Maine') are more important semantically. Hence, the more important a token is, the larger the role it should play in overall similarity. For that reason, the TF/IDF similarity measure uses the Inverse Document Frequency (idf) as token weights. The idf of a token is the inverse of the total number of times that this token appears in the database.

In addition, TF/IDF uses a Term Frequency (tf) component, i.e., each token is also weighted by the total number of times it appears in the multi-set. For example, the partial weight of 'Main' in $s_1$ is doubled. From an information retrieval perspective, where we search for highly related documents to a given keyword set, the more times a token appears in a document, the higher the relevance of that document to the query is expected to be. For example, the higher the frequency of 'Main' within a document, the more relevant the document, in expectation, to either $s_1$ or $s_2$. Nevertheless, the tf component of the score does not carry over straightforwardly when evaluating similarity between two multi-sets in all application domains, e.g., when comparing similarity between two strings; the higher the frequency discrepancy of 'Main' between $s_1$ and $s_2$, the smaller the similarity of the two strings. Furthermore, in practice, in relational databases, such multi-sets usually have very small cardinality; a large percentage of tokens appear only once in each set (i.e., tf = 1). For example, after decomposing all strings in the IMDB [8] dataset into words, out of 950K distinct words, only 1861 words appeared in at least one string with tf $>$ 2. Overall, less than 4% of the words appeared in any set with tf $>$ 1. Moreover, only 1% of the strings contained at least one word with tf $>$ 1. Similar observations hold when decomposing strings into 3-grams, as well as when using other datasets, like DBLP [9]. Hence, specifically for set similarity queries in a relational database context, it is intuitive to drop the tf component of the TF/IDF score, essentially reducing multi-sets to sets. We term the modified similarity measure IDF. Table I presents *average precision* experiments for random set selection queries, comparing set similarity selections using TF/IDF, BM25, IDF and BM25′ (BM25 without tf information). We used the same data sets utilized in previous work [10] containing varying degrees of errors, from low ($cu8$) to high ($cu1$) (see [10] for a description of the data sets and error models). As is evident from the table, ignoring the tf component of the score does not affect the quality of the results in practice.

Another intuitive modification is to length normalize the similarity scores. Length normalization restricts similarity in the interval $[0, 1]$. First and foremost, we expect similar sets to have similar lengths. Hence, we can prune the search space based on set lengths alone (cf. Section IV). Second, expressing similarity in a closed, well defined interval is more intuitive; without length normalization similarity thresholds have to be expressed in terms of unbounded constants. Moreover, with length normalization an exact match always has score equal to 1. The same ideas can be applied to BM25 and other tf based weighted measures. For simplicity, in the rest of the paper we concentrate on TF/IDF.

TABLE I
Data sets and average precision

| Dataset | TFIDF | IDF | BM25 | BM25' |
|---|---|---|---|---|
| cu1 | 0.693 | 0.690 | 0.734 | 0.730 |
| cu2 | 0.759 | 0.759 | 0.811 | 0.809 |
| cu3 | 0.849 | 0.849 | 0.884 | 0.884 |
| cu4 | 0.949 | 0.947 | 0.954 | 0.953 |
| cu5 | 0.922 | 0.922 | 0.941 | 0.941 |
| cu6 | 0.969 | 0.967 | 0.976 | 0.976 |
| cu7 | 0.990 | 0.989 | 0.990 | 0.990 |
| cu8 | 0.995 | 0.995 | 0.995 | 0.995 |

Formally, consider a database $D$ of sets (e.g., a collection of strings where each string has been decomposed into q-grams, words, etc.), where every set consists of a number of elements from universe $\mathcal{U}$; Let set $s = \{s^1, \ldots, s^n\}, s^i \in \mathcal{U}$. Every $s^i$ is assigned an idf weight computed as follows: Let $N(s^i)$ be the total number of sets containing token $s^i$ and $N$ be the total number of sets in the database. Then, $idf(s^i) = \log_2(1 + N/N(s^i))$. The normalized length of set $s$ is computed as $len(s) = \sqrt{\sum_{s^i \in s} idf(s^i)^2}$. The IDF similarity of sets $q$ and $s$ is:

$$\mathcal{I}(q, s) = \sum_{s^i \in q \cap s} \frac{idf(s^i)^2}{len(s)len(q)}. \quad (1)$$

If $q = s$, the IDF score is equal to 1. Otherwise, as the number of common tokens grows the score becomes larger. Nevertheless, the contribution of every common token to the score is dampened as the length divergence between the two sets grows. Denote with $w_i(s)$ the contribution of $s^i$ to the overall score: $\mathcal{I}(q, s) = \sum_{s^i \in q \cap s} w_i(s)$. Notice that if $s^i \notin s$, then $w_i(s) = 0$. If $s^i \in s$, then $w_i(s) = idf(s^i)^2/len(s)len(q)$.

## III. Existing Solutions

The goal is to efficiently evaluate the IDF similarity of a given query set with all database sets, and report those that exceed a user defined threshold $\tau$. For this purpose, specialized indexes can be based either on relational database technology or inverted lists.

### A. Using Relational Database Technology

We can evaluate the IDF measure using pure relational database technology in a fashion similar to the processing described in [11], [2] for TF/IDF. First, we pre-process the database and store all sets in a relational table in First Normal Form; call this the Base Table. Figure 1 shows an example with strings decomposed into sets of 4-grams. Every row consists of

| id | 4−gram | idf | len |
|---|---|---|---|
| **Main St Maine** | Main | 7 | 31.7 |
| **Main St Maine** | ain_ | 10 | 31.7 |
| · · · | · · · | . . . | . . . |
| **Main St Maine** | aine | 6 | 31.7 |
| **Florham Park** | Flor | 4 | 29.3 |
| · · · | · · · | . . . | . . . |

Base Table (4−grams)

| id | 4−gram | idf | len |
|---|---|---|---|
| **Main Street** | Main | 7 | 29.3 |
| **Main Street** | ain_ | 10 | 29.3 |
| · · · | · · · | . . . | . . . |
| **Main Street** | reet | 6 | 29.3 |

Query

Fig. 1. Token base table and query table.

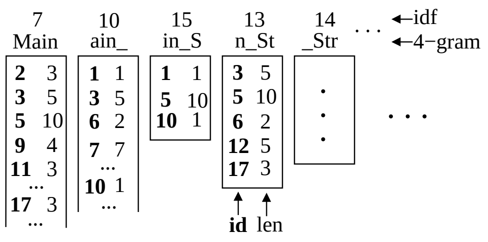
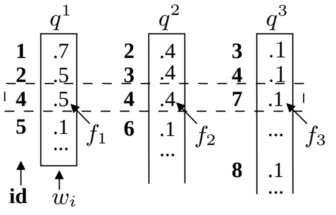


Fig. 2. Inverted lists of tokens in query "Main Street", sorted by id.



|  | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| **1** | .7 |  |  | × |
| **2** | .4 | .9 |  | ? |
| **3** | .1 | .5 |  | × |
| **4** |  | .1 | 1. | √ |
| **7** |  |  | .1 | × |
| **5** |  |  |  | × |

Fig. 3. Inverted lists sorted by decreasing token contribution.

a set id, a token, the token idf and the normalized length of the set. Given a query set $q$ we perform the same processing and store the result as a separate query table. Evaluating the IDF similarity between the sets in the base table and the query can be performed using standard SQL processing in the form of an aggregate/group-by/join statement. If an index on tokens is available, processing is expected to be very fast, pruning out immediately sets that do not contain any query tokens (a clustered index would be the best choice in this case). If an index is not available, a linear scan of the base table is unavoidable.

### *B. Using Inverted Lists*

An alternative solution is to design a specialized index. The straightforward approach is to create an inverted index on the tokens in $\mathcal{U}$ (see Figure 2). We construct one list per token $s^i$. The list is composed of one pair $\langle s, len(s)\rangle$ per set containing $s^i$. Let query $q = \{q^1, \ldots, q^n\}$ and length $len(q)$. Using the inverted index, we can compute $\mathcal{I}(q, s)$ for all $s$ by scanning the lists of tokens $q^i, 1 \leq i \leq n$ in one pass. Irrelevant sets (with $s \cap q = \varnothing$) are never accessed.

Assume that lists are sorted in increasing order of set id (for brevity, in the rest we associate with every set a unique natural number; see Figure 2). Computing $\mathcal{I}(q, s)$ for all $s$ can be performed using a multi-way list merging algorithm. We maintain an in memory heap containing the set ids at the head of the lists ($1, 2$ and $3$ in the example). We aggregate the score of any id that appears at the head of multiple lists. Obviously, the score of the smallest id (the one at the top of the heap) is complete (this id has either been encountered in all lists or does not appear in the rest of the lists). If its score exceeds $\tau$ it is reported as an answer, else it is discarded. The process is repeated after advancing the head of the lists pointing to the id last removed from the heap.

Alternatively, assume that lists are sorted first in increasing order of lengths and then in increasing order of set ids. Notice that $len(q)$ is constant across all lists, and for a given token $q^i$, $idf(q^i)$ is constant across list $i$. Hence, by sorting the list in increasing length order, we implicitly arrange the sets in decreasing $w_i$ order. Clearly, given that IDF is a monotonic score function, we can now use TA/NRA style algorithms to compute the scores incrementally, by using Equation (1) [7].

For simplicity in the following examples the lists appear already in sorted $w_i$ order, where the token idf, the length of the set and the length of the query have already been taken into account. Refer to the example query in Figure 3. We consider the NRA algorithm first. NRA performs sequential accesses only. The algorithm reads the lists in a round-robin fashion, and iteratively loads the next element from every list starting from the top. It maintains an *in memory hash table* with one entry per set id discovered so far. Each entry $s$ contains the aggregated score of the contributions of the lists in which $s$ has already appeared. It also contains a bit vector indicating the lists where $s$ has not been encountered yet. Denote the last (frontier) element read on each list with $f_i, 1 \leq i \leq n$. The lower bound $\mathcal{I}^{\vdash}(s)$ of the score of $s$ is computed as the sum of $w_i(s)$ for all $i$ where $s$ has been encountered so far. The upper bound $\mathcal{I}^{\dashv}(s)$ is computed as the sum of the lower bound and the contributions $w_i(f_i)$ for each $i$ where $s$ has not been encountered yet. On every iteration over the lists, after all $f_i$ have been updated, NRA scans the candidate set and discards all $s$ with upper bound smaller then $\tau$. It also reports sets whose score is complete and larger than $\tau$. The search terminates when the candidate set becomes empty. The algorithm appears as Algorithm 1. If an index on set ids is also available per inverted list, one can use the TA algorithm instead, to perform the search.

A running example of NRA is shown on the right side of Figure 3 for $\tau = 1$. After the first round through the lists, NRA discovers sets $1, 2, 3$ which are all viable, given current frontier $f_1 = .7, f_2 = .4, f_3 = .1$. After the second round, the scores of 2 and 3 are updated and 4 enters the candidate list. On the third round, the score of 4 is complete and is reported as an answer. Set 7 enters the list. On the fourth round 5 and 6 enter the list. Given the current frontier, $1, 3, 5$ and 7 cannot exceed the threshold and are deleted. The algorithm proceeds until set 2 is discovered or the lists are exhausted.

**Algorithm 1**: The NRA algorithm

```
Input  : Lists $q = \{q^1, \dots, q^n\}$, Threshold $\tau$
Output: Sets with $\mathcal{I}(q,s) \geq \tau$
1  Set $C = \varnothing$, $f_i$ = first element on list $i$
2  $\forall$ new $s \in C$, let $\mathcal{I}^{\vdash}(s) = 0$, $\mathcal{I}^{\dashv}(s) = 0$, $b_{[1,n]}(s) = 0$
3  repeat
4      forall $1 \leq i \leq n$ do
5          $f_i = s =$ pop next element from list $i$
6          If $s \notin C$ insert $s$ in $C$
7          Else retrieve $s$ from $C$
8          $\mathcal{I}^{\vdash}(s) += w_i(s)$, $b_i = 1$
9      forall $r \in C$ do
10         Update $b_i(r)$ according to new $f_i$
11         If $b_{[1,n]}(r) = 1$ and $\mathcal{I}(q,s) \geq \tau$ report $r$
12         If new $\mathcal{I}^{\dashv}(r) < \tau$ remove $r$ from $C$
13 until $C = \varnothing$ ;
```

## IV. Semantic Properties of IDF

The solutions presented so far take into account only the monotonicity property of `IDF`, needed for TA/NRA. In this section we present several other properties that can be exploited to design specialized algorithms that are considerably more efficient than straightforward solutions.

Recall that list entries are sorted in increasing order of lengths (and as a consequence in decreasing order of partial contributions $w_i$). The important observation is that the length of a set is constant across all lists. Hence, if two sets $s$ and $r$ appear in multiple lists, their sort order is preserved:

*Property 1 (Order Preservation):* For all $k \neq l$, if $w_k(s) \leq w_k(r)$ then $w_l(s) \leq w_l(r)$ and vice versa.

Property (1) is important for the following reason. If we know from list $k$ that $len(s) < len(r)$ and set $r$ has already been encountered in any other list $l$, then either set $s$ has been encountered in $l$ as well, or $s$ does not appear in $l$.

After the length of a set is known (e.g., after encountering the set in list $k$), we can immediately compute contribution $w_k$ and, in addition, all other contributions $w_l, l \neq k$ (since the idfs of all tokens are known). Hence, after encountering a set $s$ in any list $k$, a best case maximum score for $s$ can be determined by making the assumption that $s$ appears in all other lists:

*Property 2 (Magnitude Boundedness):* For any $s$ and $q$, after retrieving $len(s)$ from any list $k$, a best case upper bound $\mathcal{I}^{\dashv}(s)$ can be computed directly.

This gives a tight upper bound that can be used for more efficient pruning.

Intuitively, we expect similar sets to have similar lengths. Also, small sets tend to have small lengths, and large sets tend to have large lengths. [1] Clearly, it must be possible to perform pruning based on set lengths. The following holds:

[1]This is not always the case since, e.g., a large set may contain very frequent tokens only, which have low idfs, resulting in a small length.

*Theorem 1 (Length Boundedness):* Given query $q$, set $s$ and threshold $\tau$, if $\mathcal{I}(q,s) \geq \tau$ it follows that $\tau len(q) \leq len(s) \leq \frac{len(q)}{\tau}$.

*Proof:* There are three cases to consider:

Case 1. $q \cap s = q$. We have:

$$\mathcal{I}(q,s) = \frac{\sum_{q^i \in q} idf(q^i)^2}{len(s)len(q)} = \frac{len(q)^2}{len(s)len(q)} = \frac{len(q)}{len(s)} \geq \tau.$$

Case 2. $q \cap s = s$. We have:

$$\mathcal{I}(q,s) = \frac{len(s)^2}{len(s)len(q)} = \frac{len(s)}{len(q)} \geq \tau.$$

Case 3. $q \cap s \subset q$ and $q \cap s \subset s$. We have:

$$\frac{\sum_{q^i \in q \cap s} idf(q^i)^2}{len(s)len(q)} \geq \tau \quad \Rightarrow len(s) \leq \frac{\sum_{q^i \in q \cap s} idf(q^i)^2}{\tau len(q)} < \frac{\sum_{q^i \in q} idf(q^i)^2}{\tau len(q)} = \frac{len(q)}{\tau}.$$

This proves the upper bound. For the lower bound:

$$len(s)^2 = \sum_{s^i \in s} idf(s^i)^2 > \sum_{q^i \in q \cap s} idf(q^i)^2.$$

But:

$$\frac{\sum_{q^i \in q \cap s} idf(q^i)^2}{len(s)len(q)} \geq \tau \Rightarrow \sum_{q^i \in q \cap s} idf(q^i)^2 \geq \tau len(s)len(q).$$

Hence, $len(s)^2 \geq \tau len(s)len(q) \Rightarrow len(s) \geq \tau len(q)$. ■

Notice also that given cases 1 and 2 above, the range of lengths is tight. The importance of this theorem is self-evident. Given the inverted lists of the query tokens and a user defined threshold, we can immediately prune all sets whose lengths fall outside the given bounds. Hence, we only need to run any algorithm on a much reduced subset of the database. This property will significantly affect the performance of all existing and new solutions. It should be stressed here that `TF/IDF` and `BM25` follow looser versions of the aforementioned properties (by associating with every token a maximum tf component and boosting all bounds accordingly). Existing and novel algorithms for these metrics can also be optimized accordingly.

## V. The Improved NRA Algorithm (iNRA)

The above observations can help significantly improve NRA. Given query $q$ and threshold $\tau$, we use Length Boundedness to determine which part of the lists we need to scan. If no index on lengths exists, we follow the normal NRA algorithm and simply ignore list entries outside the length bounds during sequential scans. If an index on length exists (e.g., in the form of a skip list) we skip directly to the first entry with length equal to $\tau len(q)$ in every list. We also stop reading a list after encountering the last element with length equal to $len(q)/\tau$.

Next, we use Order Preservation to directly determine if a given element appears in a list or not. Given set $s$, if $len(s) < len(f_i)$ for any $i$, and $s$ has not appeared in list $i$ yet, we update its upper bound accordingly; $s$ will never appear on list $i$.

**Algorithm 2**: The iNRA algorithm

```
   Input  : Lists $Q = \{q^1, \dots, q^n\}$, Threshold $\tau$
   Output: Sets with $\mathcal{I}(q,s) \geq \tau$
1  Set $C = \varnothing$, $f_i$ = first element on list $i$
2  $\forall$ new $s \in C$, let $\mathcal{I}^{\vdash}(s) = 0$, $\mathcal{I}^{\dashv}(s) = 0$, $b_{[1,n]}(s) = 0$
3  Skip to first entry with $len(s) > \tau len(q)$ in all lists
4  repeat
5      forall $1 \leq i \leq n$ do
6          $f_i = s =$ pop next element from list $i$
7          If $len(s) > len(q)/\tau$ mark list as complete
8          If $s \notin C$ and ($\mathcal{F} < \tau$ or $\sum_{1\leq j\leq n} w_j(s) < \tau$)
           continue
9          Else insert $s$ in $C$ or retrieve $s$ from $C$
10         $\mathcal{I}^{\vdash}(s) += w_i(s)$, $b_i = 1$
11     if $\mathcal{F} < \tau$ then   forall $r \in C$ do
12         Update $b_i(r)$ according to new $f_i$
13         If $b_{[1,n]}(r) = 1$ and $\mathcal{I}(q,s) \geq \tau$ report $r$
14         If new $\mathcal{I}^{\dashv}(r) < \tau$ remove $r$ from $C$
15         Else break
16 until $C = \varnothing$ ;
```

We can use Magnitude Boundedness to directly compute the best case upper bound for any encountered set id. If the upper bound is less than $\tau$ we can immediately discard the set. This computation requires time linear to the number of lists per element access. To reduce the overhead, we can use the following observation as a pre-condition. The frontier elements $f_i$ define a conceptual best possible score of a yet unseen element. Assume that the same set id appears in all lists exactly after elements $f_i$. The score of this unseen element is at most $\mathcal{F} = \sum_{1\leq i\leq n} w_i(f_i)$. If $\mathcal{F} < \tau$ no unseen element can exceed the threshold. Hence, after this condition is satisfied we do not need to insert any new elements in the candidate set, but only complete the scores of already discovered elements. Threshold $\mathcal{F}$ is computed only once per round robin iteration.

Another observation is that NRA performs one scan of the candidate set per round robin iteration. If the candidate set is large, the cost is overwhelming. First, notice that iNRA cannot terminate unless $\mathcal{F} < \tau$. Hence, scanning the candidate set before this condition is satisfied is not necessary. Second, a conservative approach for reducing the scanning cost is to terminate the scan once the first viable candidate has been encountered (i.e., a candidate with $\mathcal{I}^{\dashv}(s) \geq \tau$). The iNRA algorithm appears as Algorithm 2. The correctness of the algorithm is not hard to prove. It follows directly from that of NRA and the correctness of the novel pruning decisions.

Running this algorithm on the example of Figure 3 illustrates the significant benefits of the improvements. After the second round through the lists the algorithm immediately discards set 1 using the Order Preservation property, since clearly if set 1 appeared in lists $q^2, q^3$ it should appear before entries 2 and 3 respectively. Overall, the algorithm terminates at round four after also discarding set 2. The original NRA, in the worst case, could do a complete scan of the lists:

*Lemma 1:* In the worst case, the NRA algorithm reads arbitrarily more elements than iNRA.

Notice that this observation does not contradict the instance optimality of NRA, since our algorithm is taking informed decisions based on a property other than monotonicity. This class of algorithms is not considered in the proof for NRA (they are referred to as algorithms that take “lucky” guesses). To prove Lemma 1 we use the Order Preservation property. Additionally, we can also prove that *any algorithm* that utilizes the Length Boundedness property runs arbitrarily better than NRA for certain instances. This is not hard to see in special cases. Assume that set lengths are unique and $\tau = 1$. The Length Boundedness property will restrict the search space to only one set, the one with length equal to the length of the query. Clearly, in this case we can construct examples where NRA will have to examine every single set in the database instead. It should be noted here that we can apply similar optimizations to the TA algorithm. Modifications are straightforward, and omitted for brevity. We call the optimized version iTA.

## VI. The Shortest-First Algorithm (SF)

The iNRA algorithm uses the semantic properties of `IDF` to speed-up the search but adheres to the round robin processing of lists in the original NRA. In that sense, the NRA algorithm can be viewed as a breadth-first approach. A different strategy is a depth-first approach. The SF algorithm scans lists in decreasing idf order. By the definition of idf, frequent tokens (with low idf) are associated with long lists and rare tokens (with high idf) are associated with short lists. By reading shorter lists first, the search discovers a smaller number of false positive candidates, improving pruning bounds faster and, hence, obviating the need to exhaustively scan longer lists.

Let query $Q = \{q^1, \dots, q^n\}$, and without loss of generality assume that $idf(q^1) > idf(q^2) > \cdots > idf(q^n)$. Denote by $\lambda_i$ the maximum length a candidate $s$ in list $q^i$ can have, in order to exceed threshold $\tau$ assuming that $s$ appears in all subsequent lists $j \geq i$ (recall that the length of $s$ is constant across all lists). Hence:

$$\sum_{i\leq j\leq n} \frac{idf(q^j)^2}{\lambda_i len(q)} = \tau \Rightarrow \lambda_i = \sum_{i\leq j\leq n} \frac{idf(q^j)^2}{\tau len(q)}. \qquad (2)$$

Length $\lambda_i$ is a natural cutoff point in list $i$ beyond which no yet unseen element $s$ can be a viable candidate. An important observation is that $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n$. (Notice that iNRA implicitly evaluates $\lambda_i$ in line 8 on a per element access basis as opposed to computing it in advance.)

The SF algorithm is shown as Algorithm 3. It proceeds as follows. First, it skips to the first entry in every list with length $len(s) \geq \tau len(q)$. Then, it computes $\lambda_1, \dots, \lambda_n$ and scans lists from high idf to low idf order, reading all elements from length $\tau len(q)$ up to and including sets with length $\min(\lambda_i, len(q)/\tau)$. Potential candidates are stored in a sorted list $C$ in increasing length order. When scanning list $q^1$, $C$ is empty and it is populated with all new elements from $q^1$.

**Algorithm 3**: The SF Algorithm

```
Input : Lists $Q = \{q^1, \ldots, q^n\}$, Threshold $\tau$
Output: Sets with $\mathcal{I}(q, s) \geq \tau$
1  Let $idf(q^1) > idf(q^2) > \ldots, idf(q^n)$
2  $C = \varnothing$, $\max len(C) = 0$
3  for $1 \leq i \leq n$ do
4      Skip to first entry with $len(s) \geq \tau len(q)$
5      Compute $\lambda_i$ using Equation (2)
6      Let $\mu_i = \min(\lambda_i, len(q)/\tau)$
7      repeat
8          $s$ = pop next element from list $i$
9          If $s \in C$ then $\mathcal{I}^{\vdash}(s) += w_i(s)$
10         Else if $len(s) \leq \lambda_i$ insert $s$ in $C$
11         $\forall$ skipped $r \in C$ reevaluate $\mathcal{I}^{\dashv}(r)$ and discard
12     until $len(s) > \max(\max len(C), \mu_i)$ ;
```

Notice that any element with length larger than $\lambda_1$ cannot exceed the threshold, even if it appeared at the top of every subsequent list. When scanning $q^2$, since both $C$ and $q^2$ are sorted by increasing lengths, a merge-sort algorithm is performed to combine the new elements read with the existing list. The partial score of elements in $C$ contributed from list $q^1$ is updated; new elements from list $q^2$ are inserted in $C$ in sorted length order; elements contributed by previous lists not present in the current list are reevaluated for potential pruning (e.g., sets from list $q^1$ that did not appear in list $q^2$, and thus have smaller potential maximum score than initially computed). Once again, new elements with length larger than $\lambda_2$ cannot exceed $\tau$. But clearly, there might be elements $s$ from list $q^1$ with $len(s) > \lambda_2$ already in $C$. Hence, to guarantee that no partial score components of elements in $C$ have been omitted, the SF algorithm continues to scan list $q^2$ until it encounters an element with length larger then the largest length in $C$ (denote this by $\max len(C)$). Pruning non viable candidates is important since it reduces $\max len(C)$ and, consequently affects how deep the algorithm needs to scan subsequent lists. The algorithm continues sequentially with all remaining lists. It terminates when the score of all elements in $C$ is complete. The correctness of the algorithm follows from the correctness of the $\lambda_i$ values for identifying possible new candidates in a list, given that consecutive lists are sorted in decreasing idf order. It also follows from the fact that the algorithm will read a list until all the scores of existing candidates have been updated or it has been deduced that a candidate does not appear in this list.

In the example of Figure 3, setting $idf(q^1) = 15$ yields $idf(q^1)^2 = 225$, $idf(q^2)^2 = 180$ and $idf(q^3)^2 = 45$, which are consistent with the partial scores in the lists. Now, we can compute $len(q) = 21.21$, $len(1) = 15.15$, $len(2) = len(3) = len(4) = 21.21$, $\lambda_1 = 21.21$, $\lambda_2 = 10.6$ and $\lambda_3 = 2.12$. The SF algorithm first reads elements $1, 2, 4$ from list $q^1$ (up to length 21.21), and proceeds with list $q^2$. Clearly, $len(2) > \lambda_2$ and no new candidates can exist in $q^2$. Nevertheless, currently $\max len(C) = 21.21$ and the algorithm reads all elements with $len(s) \leq 21.21$ in order to complete the scores of $1, 2, 4$. Hence, sets $2, 3, 4$ are read; 3 is discarded since it does not appear in $C$; 1 can also be discarded due to the Order Preservation property; the scores of 2 and 4 are updated. Next, list $q^3$ is processed. Once again, no new candidates can exist in this list since $len(3) > \lambda_3$. Nevertheless, elements with $len(s) \leq 21.21$ need to be read for score completion. After set 3 is processed, candidate 2 can be discarded due to the Order Preservation property. The score of 4 is completed and reported as an answer. The candidate set becomes empty and the algorithm terminates.

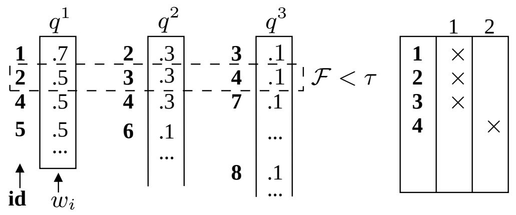


Fig. 4. In this example iNRA is arbitrarily better than SF.

We can see that SF reads even fewer entries than iNRA for this particular instance. More specifically:

*Lemma 2:* Let $Q = \{q^1, \ldots, q^n\}$ and $d_{max}$ be the maximum depth that SF descents over all lists. In the worst case iNRA will read $(d_{max} - 1)(n - 1)$ elements more than SF.

Nevertheless, Figure 4 shows a slightly different example where iNRA performs arbitrarily better than SF. Once again, consider $\tau = 1$ and set $idf(q^1) = 15$. We get $idf(q^1)^2 = 225$, $idf(q^2)^2 = 135$ and $idf(q^3)^2 = 45$. Now, we can compute $len(q) = 20.12, len(1) = 15.97, len(2) = len(3) = len(4) = \ldots = 22.36$, $\lambda_1 = 20.12$, $\lambda_2 = 8.94$ and $\lambda_3 = 2.23$. SF reads elements $1, 2, 4, 5, \ldots$ from $q^1$, $2, 3, 4$ from $q^2$ and $3, 4$ from $q^3$ as before, deducing that no exact matches exist. A running example of iNRA is shown at the right side of the figure. Here, iNRA reads elements $1, 2, 3$ in the first round, deduces that 1 can be discarded due to Order Preservation. Given $\mathcal{I}^{\dashv}(2) = \mathcal{I}^{\dashv}(3) = 0.9$, both 2 and 3 are also discarded. The algorithm continues with the second round since $\mathcal{F} > \tau$. Sets 2 and 3 are ignored once again and 4 is discarded similarly. Now, $\mathcal{F} < \tau$ and the algorithm terminates. iNRA reads arbitrarily fewer elements than SF. In the worst case SF will read all elements in list $q^1$, before reading list $q^2$.

*Lemma 3:* In the worst case, the SF algorithm reads arbitrarily more elements than iNRA.

Choosing to access longer, low idf lists last has important advantages. In practice, it is expected that only a small fraction of long lists will need to be accessed, since $\max len(C)$ and $\lambda_i$ keep decreasing as the algorithm proceeds. Another advantage is that SF requires only one scan of the candidate set per list, in contrast with iNRA that requires one scan for each round robin iteration. Clearly, the bookkeeping cost of SF will be significantly smaller than that of iNRA. Still, a hybrid approach that combines the small I/O cost of both algorithms for all problem instances would be desirable.

**Algorithm 4**: The Hybrid algorithm

```
...
forall $1 \le i \le n$ do
7   ...
    If $len(s) > \max len(C)$ mark list as complete
8   ...
```

## VII. The Hybrid Algorithm

Clearly the SF algorithm has very small bookkeeping cost due to its sorted data structure and is expected to achieve high element pruning on average. On the other hand, iNRA has significantly higher bookkeeping cost due to the required candidate set scans, but may access arbitrarily fewer elements than SF in special cases. We expect SF to work best in practice, but it would be desirable to design an algorithm that accesses the least possible number of elements in theory. We call this algorithm Hybrid. Hybrid reads elements in a round robin fashion like iNRA but uses $\max len(C)$ as a stopping condition for a particular list. This condition restricts the Hybrid algorithm from descending in any list deeper than SF, hence making Hybrid at least as efficient as SF in terms of element accesses for all instances. In addition, since the algorithm follows the iNRA strategy, it reads no more elements than iNRA in all cases, combining the best of both previous techniques:

*Lemma 4:* The Hybrid algorithm reads at most as many elements as either SF or iNRA for all problem instances.
The proof for SF is based on the fact that no candidates with length larger than $\lambda_1$ will ever be inserted in $C$. Thus, $\max len(C)$ is in the worst case the same for both algorithms. The rest depends on tightly pruning candidates from $C$, hence not allowing Hybrid to exceed the depth of SF in any list.

The algorithm is the same as Algorithm 2, except from the addition of the $\max len(C)$ condition between lines 7 and 8. In the present form the algorithm has higher bookkeeping cost than either iNRA or SF, since, first, it needs to maintain $C$ as a hash table on set ids for efficient access, second, it needs to identify the current $\max len(C)$ per list access, which necessitates *a full scan* of $C$. (Notice that $\max len(C)$ cannot be maintained incrementally, since elements are deleted occasionally from the set.)

A special candidate set organization can reduce both the cost of scanning $C$ and identifying $\max len(C)$. We partition candidates into lists sorted by length; one sorted list $c_i$ per inverted list $q^i$ along with a hash table on set ids. A candidate $s$ first discovered in $q^i$ is inserted into candidate list $c_i$. Notice that since candidates from $q^i$ are discovered in increasing length order by construction, they can simply be appended to the end of $c_i$ for a constant insertion cost. Each candidate is also inserted in the hash table, along with a pointer to its location in list $c_i$, needed for efficient deletion when elements are pruned from the hash table. With this combined structure $\max len(C)$ can be computed by peeking at the last element of every list, for a cost linear to $n$ (as opposed to linear to the number of candidates). Moreover, deleting all non-viable candidates from the candidate set is accomplished by dropping elements repeatedly from the back of all lists until a viable candidate is found in every list (once a viable candidate is found all subsequent elements are guaranteed to be viable as well). Notice that the same structure can be used with iNRA in order to minimize memory requirements by removing all non-viable candidates after a scan, instead of terminating the scan once the first viable candidate is found.

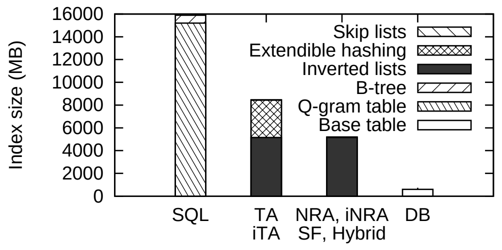


Fig. 5. Index size.

## VIII. Experimental Evaluation

### A. Experimental Setup

We run experiments on an Intel(R) Xeon 2.66 MHz, with 16 GB RAM. Two real datasets were used for the experimental evaluation; the DBLP citations [9] and the IMDB database [8]. In the rest, we concentrate on the IMDB data. Results for DBLP followed identical trends. Our data table consists of two fields, Actor and Movie, it is stored in first normal form, and consists of 7 million rows. We tokenize tuples into words, and convert each word into a set using 3-grams. Every word/set is associated with a unique, 8 byte long identifier encoding the row/column/location of the word in the data table. Word similarity queries are translated to set similarity using the `IDF` measure. The result of a query is the locations of words with similarity larger than the user specified threshold. We compare set similarity indexes based both on inverted lists and DBMS technology. Algorithms are evaluated according to overall *processing cost* (measured in average wall-clock time over repeated runs), *pruning power* (measured as the percentage of words examined over the total number of words), and *storage efficiency*. We create query workloads of 100 words each, by randomly extracting words between lengths 1-5, 6-10, 11-15, and 16-20 3-grams from the base table. Clearly, every word has at least one exact match. We also apply a fixed number of random letter insertions, deletions and swaps (termed modifications from now on) on every word in the query workloads to create words with potentially close but not exact matches.

We implemented the set similarity query processing techniques proposed in [11], [2] using MS SQL Server 2005. The q-gram table consists of fields: word id ($s$), 3-gram ($i$), word length ($len(s)$), and partial weight ($w_i(s)$). It contains one tuple per word per 3-gram, for a total of 453 million tuples. We also need to build a composite B-tree index on 3-gram/length/id/weight, which we build as a clustered index to save space. We run queries without using the composite

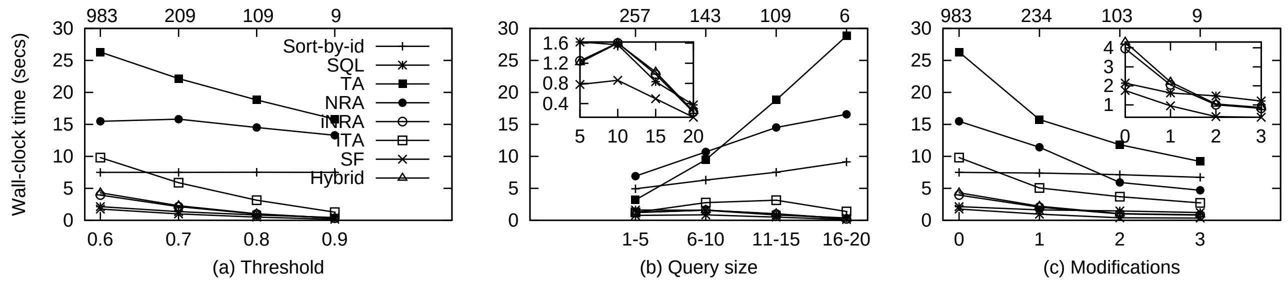


Fig. 6. Performance as a function of wall-clock time.

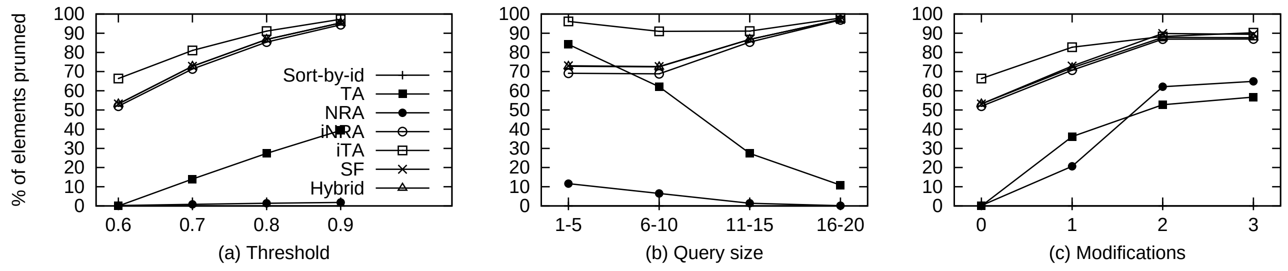


Fig. 7. Performance as a function of pruning power.

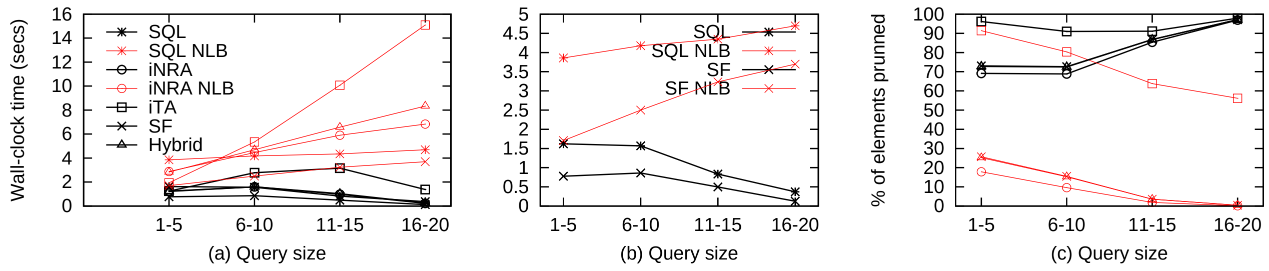


Fig. 8. Effect of Length Bounding on performance.

index, and since they did not terminate in a reasonable amount of time we omit the results. We refer to this algorithm as SQL. For the inverted list indexes we constructed two lists per q-gram, and stored them on disk. One list is sorted by increasing word id, the other by decreasing partial weights. The first is used for the multi-way merge algorithm according to ids (referred to as sort-by-id), and the second for TA/NRA style algorithms. Lists sorted by weights are also associated with a skip list for efficiently identifying an entry with a specific weight (needed for employing the Length Bounding property) and a secondary index on word ids for efficiently deducing whether a given id appears in the list or not (needed for running TA style algorithms that are based on random accesses). We use extendible hashing as the index on word ids since it can answer set containment queries with at most one random I/O in the worst case. The total size of the inverted lists (either sorting) is 5 GB. Skip lists are restricted to at most 10 MB per inverted list, for a total size of 42 MB over all lists. Finally, the extendible hashing indexes occupy 3.2 GB (after tuning, 1 KB page sizes appeared to be the best choice). The total index size for all algorithms, in comparison to the size of the data table, is shown in Figure 5. Clearly, all indexes are significantly larger than the data table (explosion due to 3-gram decomposition). The inverted lists approach is 9 times larger, while SQL is 26 times larger. Notice the significant overhead of extendible hashing, which is necessary for running TA. As a basis of comparison, we also run the original TA and NRA algorithms. We employ a few of the modifications proposed in Section V to NRA, for reducing the overwhelming bookkeeping cost. We use early termination for candidate set scans, and also avoid scanning while $\mathcal{F} \geq \tau$. NRA did not terminate in a reasonable amount of time without these optimizations. Finally, we leave caching up to the operating system and the disk drive, disabling all other software buffers. More aggressive buffering will certainly favor TA and iTA.

## *B. Wall Clock Time Improvement*

The first set of experiments measures performance as a function of wall-clock time in terms of variable query thresholds. We use a query workload with 11-15 3-grams per word and 0 modifications (every word has an exact match). Figure 6(a) clearly shows that SF has the best overall performance, with SQL, iNRA and Hybrid being slightly slower. The sort-by-id algorithm has constant computation cost since it needs to do a full scan of the inverted lists irrespective of query threshold. For large thresholds it is up to 70 times slower than SF. The traditional TA and NRA algorithms are clearly *not competitive* (up to 92 times slower). We implemented an improved version of the TA algorithm (iTA) by applying ideas similar to iNRA. iTA is slower than iNRA due to the random I/Os per element access per list, as a result of probing the hash table to complete candidate scores. Notice that the performance of the algorithms is also related to the selectivity of the query workload. The top of the graph displays the average number of results returned per query word. For smaller thresholds, a larger number of results is returned as expected. For highly selective queries, our algorithms achieve sub-second answers (0.17 secs on average for SF and $\tau = 0.9$).

Figure 6(b) plots performance as a function of query size. Here $\tau = 0.8$ and we use a workload with 0 modifications per word. The first important observation is that all algorithms that employ the Length Bounding property have improved performance for increased query sizes (from 0.77 secs for SF to 0.12 secs, as shown in the detailed graph). Since large queries have larger lengths, the Length Bounding property enables the algorithms to skip a much larger prefix of the inverted lists, and restrict the search space significantly. The second observation is that the performance of TA deteriorates sharply with increasing query sizes, since a larger number of lists implies an increased number of hash probes per element access. Once again, the total number of results returned are shown on the top of the graph.

Finally, performance as a function of query modifications is shown in Figure 6(c). We use $\tau = 0.6$, and query workloads with 11-15 3-grams per word. Notice that as the number of modifications per word increases, the average number of results per query decreases sharply, as expected. The processing cost decreases accordingly, since queries become highly selective and as a consequence pruning efficiency increases.

## C. List Element Pruning

Figure 7 plots the same set of experiments but as a function of the percentage of list elements pruned by each algorithm. We focus on inverted list approaches only. Sort-by-id does not perform any pruning. iTA has the largest pruning power since it uses random accesses to complete element scores directly and avoids using looser lower bounds. Nevertheless, the random I/Os come at a cost. SF, Hybrid and iNRA exhibit close to 95% pruning power for large thresholds, and significantly faster performance due to sequential I/Os. Once again, algorithms that use the Length Bounding property exhibit increasing pruning power with increasing query lengths, in contrast to TA and NRA.

Figure 8 shows the importance of Length Bounding on the efficiency of the algorithms. We run the same set of experiments for all algorithms by disabling Length Bounding. In some cases, for a given algorithm, Length Bounding yields a 4-fold improvement, both in terms of wall-clock time and pruning power. A more detailed graph for SQL and SF is shown in 8(b).

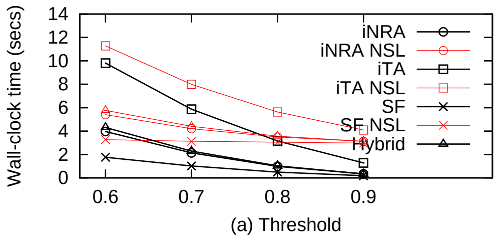


Fig. 9. Effect of Skip Lists on performance.

Finally, we measure the effect of using skip lists on the performance of inverted list algorithms. Without the skip lists, algorithms employing Length Bounding need to sequentially scan and immediately discard large prefixes of the inverted lists. Skip lists yield almost a two-fold improvement for all algorithms (as shown in Figure 9). The improvement increases for increased query sizes. Skip lists offer an excellent improvement, given their small space overhead (especially when compared with extendible hashing, needed for running TA).

## D. Experimental Summary

Overall, SF is a clear winner both in terms of computation cost (due low bookkeeping), and index size (it uses only the inverted lists and the skip lists). Notice that even though Hybrid has higher pruning power, on average the more involved data structures increase the computation cost. Hybrid is expected to perform better only in very special cases.

## IX. Related Work

The Prefix Filter [2] technique was proposed for evaluating joins using pure relational processing. It is designed for edit/hamming distance, Jaccard and some simple weighted variants thereof. It can be modified to work for all weighted similarity measures for selection queries (i.e., in the degenerate case where one side of the join contains only one entry), but it is subsumed by the SQL based approach described in Section III-A, when a B-tree index exists. Arasu et al. [1] design a signature scheme that can be used as a filter for identifying candidate sets with hamming distance smaller than $k$ from a query set. It was used for answering set similarity joins based on edit distance and Jaccard. It is not clear how to extend this work for weighted metrics and selection queries.

Both exact [12] and approximate [13] algorithms for set similarity joins between sets with unweighted elements have been proposed as well. Specialized set similarity join algorithms using cosine similarity between sets have also been considered [14].

Kahveci et al. [15] propose an index for substring matches within edit distance $k$ from a query. The same problem has been addressed in, among others, [16], [17]. In [18] the authors use VP-trees for answering nearest neighbor queries for edit distance. An exhaustive comparison of methods based on edit distance and variants appears in [19]. None of these techniques can be applied for `TF/IDF`, `BM25` style metrics.

## X. Conclusions

We argued that special semantic properties of some weighted similarity measures can be exploited to design very efficient index structures and algorithms for set similarity retrieval. We proved a Length Bounding property that, if employed, yields orders of magnitude speed up for all algorithms. We proposed three new algorithms based on TA/NRA style processing on inverted lists. The Shortest-First algorithm achieved truly interactive responses in all practical cases. In the future, we plan to extend our techniques for top-k processing, devise parallel versions of all algorithms, and explore semantic properties of a wide variety of similarity measures.

## References

[1] A. Arasu, V. Ganti, and R. Kaushik, "Efficient exact set-similarity joins," in *Proc. of Very Large Data Bases (VLDB)*, 2006, pp. 918–929.

[2] S. Chaudhuri, V. Ganti, and R. Kaushik, "A primitive operator for similarity joins in data cleaning," in *Proc. of International Conference on Data Engineering (ICDE)*, 2006, p. 5.

[3] S. Sarawagi and A. Kirpal, "Efficient set joins on similarity predicates," in *Proc. of ACM Management of Data (SIGMOD)*, 2004, pp. 743–754.

[4] S. Sarawagi and A. Bhamidipaty, "Interactive deduplication using active learning," in *Proc. of ACM Knowledge Discovery and Data Mining (SIGKDD)*, 2002, pp. 269–278.

[5] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Addison Wesley, May 1999.

[6] 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 *Proc. of Very Large Data Bases (VLDB)*, 2001, pp. 491–500.

[7] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," in *Proc. of ACM Symposium on Principles of Database Systems (PODS)*, 2001, pp. 102–113.

[8] IMDB, "IMDB database," http:// www.imdb.com/ interfaces.

[9] M. Ley, "DBLP database," http://dblp.uni-trier.de/xml.

[10] A. Chandel, O. Hassanzadeh, N. Koudas, M. Sadoghi, and D. Srivastava, "Benchmarking Declarative Approximate Selection Predicates," in *Proc. of ACM Management of Data (SIGMOD)*, 2007.

[11] L. Gravano, P. G. Ipeirotis, N. Koudas, and D. Srivastava, "Text joins in an RDBMS for web data integration," in *WWW*, 2003, pp. 90–101.

[12] N. Mamoulis, "Efficient Processing of Joins on Set-valued Attributes," in *Proc. of ACM Management of Data (SIGMOD)*, 2003.

[13] A. Gionis, D. Gunopoulos, and N. Koudas, "Efficient and Tunable Similar Set Retrieval," in *Proc. of ACM Management of Data (SIGMOD)*, 2001.

[14] R. Bayardo, Y. Ma, and R. Srikant, "Scaling Up All-Pairs Similarity Search," in *WWW*, 2007.

[15] T. Kahveci and A. K. Singh, "Efficient index structures for string databases," in *Proc. of Very Large Data Bases (VLDB)*, 2001, pp. 351–360.

[16] S. Wu and U. Manber, "Fast text searching: allowing errors," *Communications of the ACM (CACM)*, vol. 35, no. 10, pp. 83–91, 1992.

[17] R. A. Baeza-Yates and G. Navarro, "Faster approximate string matching," *Algorithmica*, vol. 23, no. 2, pp. 127–158, 1999.

[18] S. C. Sahinalp, M. Tasan, J. Macker, and Z. M. Özsoyoglu, "Distance based indexing for string proximity search," in *Proc. of International Conference on Data Engineering (ICDE)*, 2003, pp. 125–135.

[19] G. Navarro, "A guided tour to approximate string matching," *ACM Computing Surveys*, vol. 33, no. 1, pp. 31–88, 2001.