A Metric Index for Approximate Text Management

Vlastislav Dohnal
Masaryk University
Brno, Czech Republic
xdohnal@fi.muni.cz

Claudio Gennaro
ISTI-CNR
Pisa, Italy
gennaro@iei.pi.cnr.it

Pavel Zezula
Masaryk University
Brno, Czech Republic
zezula@fi.muni.cz

Abstract

Text collections of data need not only search support for identical objects, but the approximate matching is even more important. A suitable metric to such a task is the edit distance measure. However, the quadratic computational complexity of edit distance prevents from applying naive storage organizations, such as the sequential search, and more sophisticated search structures must be applied. We have investigated the properties of the D-index to approximate searching and matching in text databases. The experiments confirm a very good performance for retrieving close objects and sub-linear scalability to process large files. Even the similarity joins can be performed efficiently.

Key Words:
metric data, similarity search, index structures, similarity join

1. Introduction

In the proliferation of web-based information systems, digital collections of text data are probably the most important sources of information. Contrary to traditional databases, text collections are not strictly structured and individual text parts – such as words, sentences, paragraphs, etc. – can have different significance to different individuals. Text strings are also more heterogeneous and, unfortunately, also more error prone. Consequently, string processing in databases has become a very fertile and important area of database research.

The *search* operation is the most prominent in any kind of databases, and the text databases are not exceptions. Since text is typically represented as a character string, pairs of strings can be compared and the *exact match* decided. However, the longer the strings are the less significant the exact match is: the text strings can contain errors of any kind and even the correct strings may have small differences. This gives a motivation to a search allowing errors, or *approximate search*, which requires a definition of the concept of *similarity*, as well as a specification of algorithms to evaluate it.

The problem of correcting misspelled words in written text is rather old, and the experience reveals that 80% of these errors are corrected allowing just one insertion, deletion, or transposition. But the problem is not only grammatical, because an incorrect word that is entered in the database cannot be retrieved anymore on the exact match bases. According to [8], text typically contain about 2% of typing and spelling errors.

There are many application areas for which the approximate matching is a relevant problem. Nowadays, virtually all text retrieval (or filtering) systems allow some extended facilities to recover from errors in text or patterns. Other text processing applications are spelling checkers, natural language interfaces, computer tutoring, and language learning, to name only a few of them.

But there are also applications that consider longer text units than words. Consider a database of sentences for which translations to other languages are known. When a sentence is to be translated, such the database can suggest a possible translation provided the sentence or its close approximation already exists in the database. Another application is the copy detection where the unit of comparison is the whole document.

The development of Internet services often requires an integration of heterogeneous sources of data. Such sources are typically unstructured whereas the intended services often require structured data. Once again, the main challenge is to provide consistent and error-free data, which implies the *data cleaning*, typically implemented by a sort of *similarity join*. In order to perform such tasks, similarity rules are specified to decide whether specific pieces of data may actually be the same things or not. However, when the database is large, the data cleaning can take a long time, so the processing time (or the performance) is the most critical factor that can only be reduced by means of convenient similarity search indexes.

For two strings of length n and m available in main memory, there are several dynamic programming algorithms to compute the *edit distance* of the strings in $O(nm)$ time and space. Refer to [9] for an excellent overview of the work and additional references. Index structures such as Suffix arrays and PAT-arrays allow for fast searches on strings, but they are difficult to update in secondary memory. Suffix trees are other classical indexes for strings, but they have unbalanced tree topology, which makes the dynamic maintenance in secondary storage difficult. These structures have also a lot of limitations to properly support the approximate string management.

The problem of approximate string processing has recently been studied in [4] in the context of data cleaning, that is removing inconsistencies and errors from large data sets such as occurring in *data warehouses*. A technique for building approximate string join capabili-

ties on top of commercial databases has been proposed in [6]. The core idea of these approaches is to transform the difficult problem of approximate string matching into other search problems for which efficient solution exists. A viable alternative is to explore an application of a general-purpose metric index, see [1] for a survey, to the problem of approximate string processing.

In this article, we use the edit distance to measure similarity of text strings, and we demonstrate that the D-index [3] metric structure can perform the similarity (approximate) search and join operations fast. In Section 2., we define principles of the similarity search in metric spaces. Performance evaluation for large collections of the Czech language sentences is reported in Section 3..

2. Metric Space Searching

A convenient way to assess similarity between two objects is to apply metric functions to decide the closeness of the objects as a distance, that is the objects' dis-similarity. A *metric space* $\mathcal{M} = (\mathcal{D}, d)$ is defined by a domain of objects (elements, points) $\mathcal{D}$ and a total (distance) function d – a *non negative* ($d(x, y) \geq 0$ with $d(x, y) = 0$ iff $x = y$) and *symmetric* ($d(x, y) = d(y, x)$) function, which satisfies the *triangle inequality* ($d(x, y) \leq d(x, z) + d(z, y), \forall x, y, z \in \mathcal{D}$).

In general, the problem of indexing in metric spaces can be defined as follows: *given a set $X \subseteq \mathcal{D}$ in the metric space $\mathcal{M}$, preprocess or structure the elements of X so that similarity queries can be answered efficiently*. For a query object $q \in \mathcal{D}$, two fundamental similarity queries can be defined. A *range query* retrieves all elements within distance r to q, that is the set $\{x \in X, d(q, x) \leq r\}$. A *nearest neighbor* query retrieves the h closest elements to q, that is a set $R \subseteq X$ such that $|R| = h$ and $\forall x \in R, y \in X - R, d(q, x) \leq d(q, y)$. For two sets $A \subseteq \mathcal{D}, B \subseteq \mathcal{D}$, the *similarity join* operation retrieves all pairs $(x, y) \in A \times B$ such that the distance between x and y is less than or equal to a predefined value k. If $A = B$, the operation is called the *similarity self-join* – in this article, we only concentrate on this version of similarity joins.

For the space constraints of this article, we only consider here the similarity range and self-join operations. In the following, we introduce the edit distance as a convenient metric function for comparing text strings. We also outline the principles of the D-index, which supports indexing of metric data.

2.1 Edit distance

The *edit distance*, also known as the *Levenshtein distance*, is a distance function which measures similarity between two text strings. In fact, it computes the minimum number of *atomic edit operations* needed to transform one string into the other. The atomic operations are *insertion*, *deletion* and *replacement* of one character. For illustration, consider the following examples:

- `d('lenght','length')=2` – two replacements of the two last letters, $h \rightarrow t$ and $t \rightarrow h$,

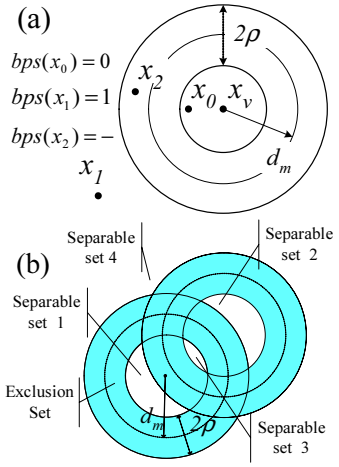

Figure 1. The bps split function (a) and the combination of two bps functions (b).

- `d('sting','string')=1` – one insertion of r,
- `d('application','applet')=6` – one replacement of the 5th letter, $i \rightarrow e$, two deletions of the 6th and 7th letters and three deletions of the three last letters (i, o, n).

The time complexity of algorithms implementing the edit distance function $d_{edit}(x, y)$ is $O(len(x) \times len(y))$, that is evaluations of the edit distance function are high CPU consuming operations. For more details, see the recent survey [9].

2.2 D-Index

The D-Index is a multi-level metric structure, consisting of the *search-separable* buckets at each level. The structure supports easy insertion and bounded search costs because at most one bucket needs to be accessed at each level for range queries up to a predefined value of search radius ρ. At the same time, the applied *pivot-based strategy* significantly reduces the number of distance computations in accessed buckets. In the following, we provide a brief overview of the D-Index, more details can be found in [5] and the full specification, as well as performance evaluations, are available in [3].

The partitioning principles of the D-Index are based on a multiple definition of a mapping function, called the ρ-split function, as illustrated in Figure 1a. This function uses one reference object x_v and the *medium distance* d_m to partition a data set into three subsets. The result of the following bps function gives a unique identification of the set to which the object x belongs:

$$bps(x) = \begin{cases} 0 & \text{if } d(x, x_v) \leq d_m - \rho \\ 1 & \text{if } d(x, x_v) > d_m + \rho \\ - & \text{otherwise} \end{cases}$$

The subset of objects characterized by the symbol '$-$' is called the *exclusion set*, while the subsets of objects characterized by the symbols 0 and 1 are the *separable sets*, because any range query with radius not larger than ρ cannot find qualifying objects in both the subsets.

More separable sets can be obtained as a combination of bps functions, where the resulting exclusion set is the union of the exclusion sets of the original split functions. Furthermore, the new separable sets are obtained as the intersection of all possible pairs of separable sets of the original functions. Figure 1b gives an illustration of this idea for the case of two split functions. The separable sets and the exclusion set form the separable buckets and the exclusion bucket of one level of the D-index structure, respectively.

Naturally, the more separable buckets we have, the larger the exclusion bucket is. For large exclusion bucket, the D-index allows an additional level of splitting by applying a new set of split functions on the exclusion bucket of the previous level. The exclusion bucket of the last level forms the exclusion bucket of the whole structure. The ρ-split functions of individual levels should be different but they must use the same ρ. Moreover, by using different number of split functions (generally decreasing with the level), the D-Index structure can have different number of buckets at individual levels. In order to deal with overflow problems and growing files, buckets are implemented as *elastic buckets* and consist of the necessary number of fixed-size blocks (pages) – basic disk access units.

Due to the mathematical properties of the split functions precisely defined in [3], the range queries up to radius ρ are solved by accessing at most one bucket per level, plus the exclusion bucket of the whole structure. This can intuitively be comprehended by the fact that an arbitrary object belonging to a separable bucket is at distance at least 2ρ from any object of other separable bucket of the same level. With additional computational effort, the D-Index executes range queries of radii greater than ρ. The D-index also supports the nearest neighbor(s) queries.

3. Performance Evaluation

In order to demonstrate suitability of the D-index to the approximate management of text data, we have used sentences from the Czech language corpus compared by the edit distance measure. For illustration, see Figure 2 for the edit distance distribution of our data sets. Observe that the most frequent distance was around 100 and the longest distance was 500, equal to the length of the longest sentence. In all experiments, the search costs are measured in terms of block reads and the number of distance computations. The basic structure of D-index was fixed for all tests and consisted of 9 levels and 39 buckets. However, due to the elastic implementation of buckets with a variable number of blocks, we could easily manage data files of different sizes. In the following, we report results of our experiments separately for the similarity range queries and the similarity self-join operations.

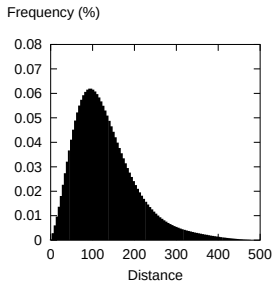

Figure 2. Distance distribution for the data set

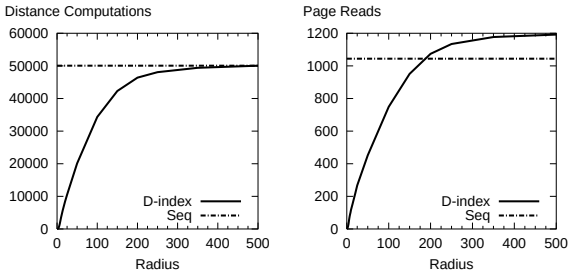

Figure 3. Range queries.

3.1 Range Search

All presented cost values are averages obtained by execution of 50 queries with different query objects (sentences) and a constant search radius.

The objective of the first group of experiments was to study the relationship between the search radius (or the selectivity) and the search costs considering a data set of constant size. In the second group, we concentrated on small query radii and by significantly changing the cardinality of data sets, we studied the scalability of the query execution.

Selectivity-cost ratio. In order to test the basic properties of the D-index to search text data, we have considered a set of 50,000 sentences. Though we have tested all possible query radii, we have mainly focused on small radii. That corresponds to the semantically most relevant query specifications – objects that are very close to a given reference can be interesting. Recall that a range query returns all objects whose distances from the query object are within the given query radius.

Figure 3 shows global trends of costs needed to evaluate range queries, separately for the number of distance computations and the block reads. As a reference, the figures also include the costs of the sequential approach (SEQ) using the exhaustive scan. This does not depend on query radii so the search costs are constant. The search costs of the D-index are very low for small query radii, but for very large radii, the number of distance computations can become as high as the number that is needed for the SEQ – all objects must be tested to solve the query. Notice that the number of disk reads can even become higher than the number of disk reads of SEQ, because the average utilization of a block of the

D-index is lower than the average block utilization of the SEQ. As Figure 3 illustrates, the D-index requires less page reads than SEQ up to the radius 190 where such queries return about 80% of the whole database which is typically to much to be interesting from the search point of view. Due to the computational complexity of the edit distance, the page reads are not so significant because one page read from a disk takes less than one millisecond while the distance computations between a query object and objects in only one block can take 15 milliseconds.

Searching for typing errors or duplicates results in range queries with small radii and the D-index solves these types of queries very efficiently. The following table shows the average numbers of page reads and distance computations needed by the D-index structure to evaluate queries of small radii. Due to the pivot-based strategy applied to the D-index, only some objects of accessed pages have to be examined and distances to them computed, this technique saves a lot of distance computations.

Radius	#pages	#dist
1	2.74	29.08
2	6.34	83.34
3	14.88	201.38
4	30.64	411.60

Searching for objects within the distance 1 to a query object takes less than 3 page reads and 29 distance computations – such queries are solved in 9 milliseconds. A range query with radius $r = 4$ is solved using 411 distance computations, which is less than 1% of the sequential scan (it needs 50,000 computations), and 30 page reads, which is about 2.5% of all pages – all objects are stored in 1192 pages. But even queries with radii 4 take at average about 0.125 seconds. This is in a sharp contrast with 16 seconds of the SEQ access method.

We have also compared the costs for the execution of the *successful* and *unsuccessful* exact match queries, i.e. executing queries with radius 0 separately for query objects present and not present in the database. The following table shows the average results for 50 different queries of both types.

Exact match	#pages	#dist
successful	1.14	12.42
unsuccessful	0.46	11.38

On average, the D-index needs 0.46 page reads for answering the unsuccessful exact match query and 1.14 page reads for successful query. This means that the D-index typically needs only 1 (rarely 2) block access to evaluate the exact match queries. Due to the construction of D-index, the absence of the searched object can even be inferred from the D-index structure, so in many cases no block access is needed for the unsuccessful search. The distance computations are mostly due to the evaluation of split functions, i.e. strictly related to the D-index structure. Provided the (chosen) reference objects of split functions are short sentences, the costs of such distance computations can be significantly reduced.

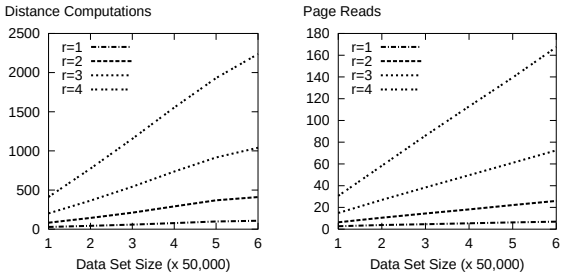

Figure 4. Range search scalability.

Scalability. To analyze the range search scalability of D-index, we have considered collections of sentences ranging from 50,000 to 300,000 elements. We have conducted tests to see how the costs for range queries (from $r = 1$ to $r = 4$) grow with the size of the data set. The obtained results are reported in graphs in Figure 4, where individual curves correspond to different query radii.

¿From the figure we can see that the search costs scale up with slightly sub-linear trends. This is a very desirable feature that guarantees the applicability of the D-index not only on small data sets but also on large collections of data.

As the following table demonstrates, the search scale up for exact match queries, both successful and unsuccessful, is more or less constant, i.e. the D-index needs nearly the same number of distance computations and page reads for different data set sizes.

Data Size	#pages	#dist
50,000	1.14	12.42
100,000	1.22	13.04
150,000	1.30	13.58
200,000	1.38	13.90
250,000	1.58	14.58
300,000	1.60	14.78

This is an important property because an efficient exact match is needed in dynamic files where new objects are inserted and old objects deleted – specific object instances are usually located through exact match queries.

3.2 Similarity Self-join

The implementation of the similarity joins is not an easy task. The traditional hash joins and sort merge joins do not carry over easily to the similarity join problem, as the join predicate makes the use of a distance threshold which requires an evaluation of a distance function between pairs of objects. The main problem is that there is no ordering of objects in a metric space. The *naive* algorithm strictly follows the definition of similarity join and computes the *Cartesian product* between two sets to decide the pairs of objects that must be checked on the threshold k. This algorithm has the time complexity $O(N^2)$, where $N = |X|$. A more efficient implementation, called the *nested loops*, uses the symmetric property of metric distance functions for pruning some

pairs. The time complexity is $O(\frac{N \cdot (N-1)}{2})$. More sophisticated methods use pre-filtering strategies to discard dissimilar pairs without actually computing distances between them. Such pre-filtering rules can save many useless (and expensive) distance computations and make algorithms more efficient. The pre-filtering rules are typically based on a simplified or an approximate distance measure, which is much cheaper to compute.

In order to demonstrate how the D-index can be used for the similarity self-join, we assume the data set $X \subseteq \mathcal{D}$ organized by this structure and apply the search strategy as follows: for $\forall x \in X$, perform $range\_query(x, k)$.

Join-cost ratio. To analyze basic properties of the D-index to process similarity self-joins, we have conducted our experiments on a set of 50,000 sentences. Though this approach can be applied to any threshold k, we have mainly concentrated on small k, which are used in the data cleaning area. The following table shows costs of similarity self-join queries with different k.

k	#pages	#dist
1	13,257	794,866
2	91,850	1,746,877
3	331,310	4,073,230
4	760,291	9,122,964

As expected, the number of distance computations (or the processing costs in general) increases quite fast with growing k. However, the nested loops algorithm (NL) is much more expensive. For $k = 4$, the NL algorithm is 137 times slower, and for $k = 1$, the NL algorithm uses even 1500 times more distance computations. In this respect, the performance of our approach is quite comparable to the approximate string join algorithm proposed in [6]. This approach is based on segmenting strings into q-grams and introduces an additional overhead for building lists of q-grams. However, the reported search speedup with respect to NL, is practically the same.

Another recent paper [4] proposes the *neighborhood join* algorithm (NJ), which uses the difference of lengths of compared strings ($abs(|x| - |y|)$) as the simplified distance function for the pre-filtering rule. The authors of [4] have tested NJ algorithm on bibliographic strings with k equal to 15% of the maximum distance. The NJ has saved from 33% to 72% of distance computations with respect to the NL algorithm, depending on the distance distribution. In order to contrast such approach with our technique, we have also set k to the 15% of the maximum distance of our data set, specifically $k = 35$. For this case, our algorithm saved 70% of distance computations.

Note that the experiments in [6] and [4] were conducted on data sets of comparable sizes, but quite shorter strings, which makes the problem of similarity joins easier. We can conclude that the implementation of the self-join through the general index structure, D-index, is very competitive to the specialized algorithms for strings.

Scalability. Considering the web-based dimension of data, scalability is probably the most important issue to

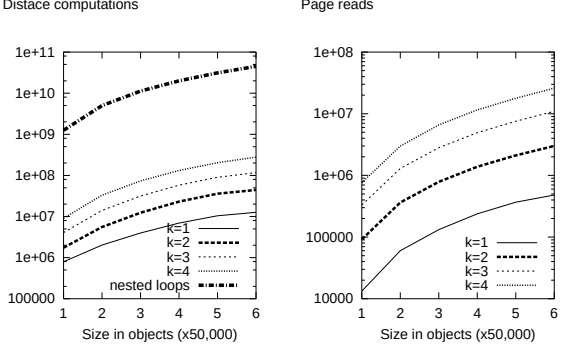

Figure 5. Join queries.

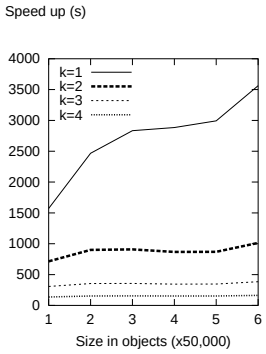

Figure 6. Join speed-up.

investigate. In the elementary case, it is necessary to investigate what happens with the performance when the size of data doubles. We have experimentally investigated the behavior of the D-index on data sets with sizes from 50,000 to 300,000 objects (sentences). Figure 5 presents results for $k = 1, 2, 3$ and 4. Considering $k = 1$, the number of distance computations has increased only 15 times when the data set was enlarged 6 times (from 50,000 to 300,000 objects). Though this is not linear, it is in a sharp contrast with the nested loops algorithm, which performed 36 times more slowly. For $k = 4$, the results are not so promising, the execution has consumed 30 times more distance computations but it was still better than the nested loops algorithm. In [4], the authors have executed experiments on two different sizes of the same data and their NJ specialized algorithm deteriorated about 35 times while the data set has been enlarged 5 times only.

Figure 6 reports the *speed-up*, s, which is defined as

$$s = \frac{N \cdot (N-1)}{2 \cdot n},$$

where N is the number of objects stored in the D-index and n is the number of distance evaluations needed by our algorithm. In fact, the speed-up says how many times our algorithm is faster than the NL algorithm. The figure demonstrates that the speedup never deteriorates with growing files and for very low values of k, it actually grows quite fast. For higher values of k, the speedup is not so high, and for $k \geq 3$ it is practically constant with respect to the data set size. This implies that similarity self-join with the D-index is also suitable for large and

growing data sets.

We have observed the same trends both for distance computations and page reads. But as explained before, the page reads aren't so important compared to the distance computations.

4. Conclusions

Approximate string matching in text databases is an important search operation and the edit distance is a convenient way how such approximation can be specified. However, due to the quadratic computational complexity of the edit distance, the execution costs are high on large files, and indexing techniques must be applied. We have observed by experiments that a sequential similarity range search on 50,000 sentences takes about 16 seconds. But to perform the nested loop similarity self-join algorithm on the same file would take 25,000 times more, which is about 4 days and 15 hours.

We have applied the D-index, a metric index structure, and we have performed numerous experiments to analyze its search properties. Whenever possible, we have also contrasted our results with recent specialized proposals for approximate string processing. In general, we can conclude that the application of the D-index is never worse in performance than the specialized techniques. The D-index is strictly sub-linear for all meaningful search requests, that is search queries retrieving relatively small subsets of the searched data sets. The D-index is extremely efficient for small query radii where practically on-line response time is guaranteed. The important feature is that the D-index scales up well to processing large files and experiments reveal even slightly sub-linear scale up for similarity range queries. Though the scale up for the similarity self-join processing is not linear, it is still better than the scale up reported for the specialized techniques.

In principle, any metric index structure can be used to implement our algorithm of the similarity join. Our choice of the D-index is based on an earlier comparison of index structures summarized in [3]. Besides the D-index, the authors also studied the performance of the M-Tree [2] and a sequential organization – according to [1], other metric index structures do not support disk storage and keep objects only in the main memory. Presented experiments reveal that for small range query radii, typical for similarity join operations, the D-index performs at least 6 times faster than the M-Tree, and it is much more faster than the sequential scan. We plan to systematically investigate this issue in the near future.

We have conducted our experiments on sentences. However, it is easy to imagine that also text units of different granularity, such as individual words or paragraphs with words as string symbols, can be handled in a similar way. However, the main advantage of the D-index is that it can also perform similar operations on other metric data. As suggested in [7], where the problem of similarity join on XML structures is investigated, metric indexes could be applied for approximate matching of tree structures. We consider this challenge as our second major future research direction.

References

[1] E. Chavez, G. Navarro, R. Baeza-Yates, and J. Marroquin: Searching in Metric Spaces. *ACM Computing Surveys*, 33(3):273-321, 2001.

[2] P. Ciaccia, M. Patella, and P. Zezula: M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. *Proceedings of the 23rd VLDB Conference*, pp. 426-435, 1997.

[3] V. Dohnal, C. Gennaro, P. Savino, P. Zezula: D-Index: Distance Searching Index for Metric Data Sets. To appear in *Multimedia Tools and Applications*, Kluwer, 2002.

[4] H. Galhardas, D. Florescu,D. Shasha, E. Simon, and C.A. Saita: Declarative Data Cleaning: Language, Model, and Algorithms. Proceedings of *the 27th VLDB Conference*, Rome, Italy, 2001, pp. 371-380.

[5] C. Gennaro, P. Savino, and P. Zezula: Similarity Search in Metric Databases through Hashing. Proceedings of *ACM Multimedia 2001 Workshops*, October 2001, Ottawa, Canada, pp. 1-5.

[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. Proceedings of *the 27th VLDB Conference*, Rome, Italy, 2001, pp. 491-500.

[7] S. Guha, H.V. Jagadish, N. Koudas, D. Srivastava, and T. Yu: Approximate XML Joins. Proceedings of *ACM SIGMOD 2002* to appear.

[8] K. Kukich: Techniques for automatically correcting words in text. *ACM Computing Surveys*, 1992, 24(4):377-439.

[9] G. Navaro: A guided tour to approximate string matching. *ACM Computing Surveys*, 2001, 33(1):31-88.