Approximate Searching in Large Collections of Text Data

Vlastislav Dohnal[1], Claudio Gennaro[2], and Pavel Zezula[1]

[1] Masaryk University, Brno, Czech Republic,
xdohnal/zezula@fi.muni.cz,
[2] IEI-CNR Pisa, Italy,
gennaro@iei.pi.cnr.it

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 task is the edit distance measure. However, its quadratic computational complexity prevents from applying naive storage organizations, such as the sequential search, and more sophisticated search structures must be used. We have investigated the properties of the D-index to approximate searching in text databases. The experiments confirm very good performance for retrieving close objects and sub-linear scalability to process large files.

1 Introduction

At present, digital collections of text data are probably the most important sources of information and their sizes are growing at enormous speed. Text collections are not strictly structured and individual text parts, such as words, sentences, paragraphs, etc. can have different significance for different individuals. Text collections are also becoming more heterogeneous and the same content can be expressed in different languages. Unfortunately, text collections are also becoming more error prone.

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, which requires a definition of the concept of *similarity*, as well as specification of algorithms to compute 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 [4], text typically contain about 2% of typing and spelling errors. Moreover, text collections digitalized via optical

character recognition (OCR) contain a non negligible percentage of errors ($7 - 16\%$). Such numbers are even higher for text obtained through conversion from a voice.

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 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 the 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. In order to perform such tasks, similarity rules are specified to determine whether specific pieces of data may actually be the same thing or not. However, when the database is large, 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 a similarity search index.

In this article, we use the Levenshtein or the *edit* distance to measure similarity of text strings and we apply the D-index to perform the similarity search fast. In Section 2, we define principles of the similarity search in metric spaces. Performance evaluation for large collections of 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* and *symmetric* function which satisfies the *triangle inequality* $d(x, y) \leq d(x, z) + d(z, y)$, $\forall x, y, z \in \mathcal{D}$. We assume that the maximum distance never exceeds d^+, thus we consider a *bounded metric space*. In general, we study the following problem: *Given a set $X \subseteq \mathcal{D}$ in the metric space $\mathcal{M}$, pre-process 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 k closest elements to q, that is a set $A \subseteq X$ such that $\mid A \mid = k$ and $\forall x \in A, y \in X - A, d(q, x) \leq d(q, y)$.

In the following, we define 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 computes the minimum number of atomic edit operations (i.e. the *insertion*, *deletion*, and *substitution* of one symbol) needed to transform a string into another. If all atomic operations have the same *costs* we talk about *unweighted* edit distance. On the other hand, the *weighted* modification assigns to each operation its own cost.

Definition 1. *Let Σ be a finite set of symbols, called an alphabet, Σ^* denotes the set of all finite strings over Σ, let ε be an empty symbol (not included in the alphabet). The* edit transformation *from the string X to the string Y is a sequence $S = S_1 S_2 \ldots S_n$ of atomic edit operations, i.e. $S_i = A_i \rightarrow B_i$, where $A_i, B_i \in \Sigma \cup \{\varepsilon\}$. A* weight function *$\delta$ assigns a non-negative cost ($\delta(A \rightarrow B) \geq 0$) to each operation $A \rightarrow B$. The total cost of an edit transformation S is the sum of costs for all S_i, $\delta(S) = \sum_{i=1}^{n} \delta(S_i)$. The weighted edit distance is defined: $d_{edit}[\delta](X, Y) = min\{\delta(S) | S$ is an edit transformation from X to $Y\}$.*

The edit distance is a metric function regardless of the specific weight function. The time and space complexity of algorithms implementing the edit distance $d_{edit}(X, Y)$ is $O(len(X) \times len(Y))$, that is evaluations of the edit distance are high CPU consuming operations. For more details, see for example the recent survey by Navaro [5].

In the rest of this article, we only use the unweighted edit distance. For illustration, consider the following examples:

- `d('lenght','length') = 2` – two substitutions of the two last letters, $h \rightarrow t$ and $t \rightarrow h$,
- `d('sting','string') = 1` – one insertion of r,
- `d('application','applet') = 6` – one substitution of the 5th letter, $i \rightarrow e$, two deletions of the 6th and 7th letters, three deletions of the three last letters (i, o, n).

2.2 D-Index

According to the survey [1], existing approaches to search general metric spaces can be classified in two basic categories:

Clustering algorithms: divide (recursively) the search space into partitions (clusters) characterized by some representative information.

Pivot-based algorithms: choose a certain number of distinguished elements from the data set and use pre-computed distances to these elements to speed up the retrieval.

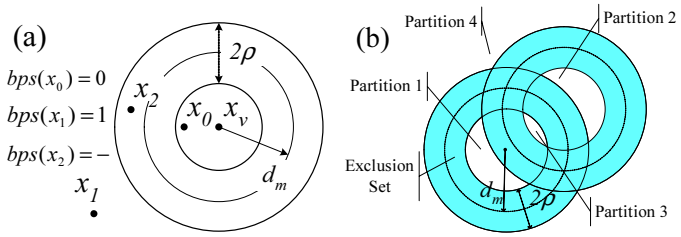

Fig. 1. The *bps* split function (a) and the combination of two *bps* functions (b).

Though the number of existing search algorithms is impressive, see [1], the search costs are still high, and there is no algorithm that is able to outperform the others in all situations. Furthermore, most of the algorithms are implemented as main memory structures, thus do not scale up well to deal with large data files.

The D-Index structure [2, 3] synergistically combines more principles into a single system to minimize the amount of accessed data and the number of distance computations. In general, it is a new technique which recursively clusters data in *separable* partitions and also applies the pivot-based strategy to decrease search costs.

The partition principle is based on the concept of ρ-split function. In order to illustrate, we show an example of the *ball partitioning* function *bps*. This function uses one object x_v and the *medium* distance d_m to partition the data set into three subsets – the result of the following function gives the index of the set to which the object x belongs. The subset associated to the symbol '$-$' is called exclusion set (see Figure 1a).

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

Since the *bps* function divides the objects in just three subsets, we can combine more functions with the purpose of generating ρ-split functions of higher order (see Figure 1b). Moreover, several of these ρ-split functions can be used in order to obtain the multilevel access structure of D-Index. In particular, on the first level, the D-index divides the whole data set into buckets using a "combined" ρ-split function. For any other level, objects mapped to the exclusion set of the previous level are the candidates for storage in separable buckets of this level. Finally, the exclusion set of the last level forms the exclusion bucket of the whole D-index structure. The ρ-split functions of individual levels use the same ρ.

Each bucket of the D-index is implemented as a list of fixed size blocks (pages) of capacity being able to accommodate any object of the processed data set.

3 Performance Evaluation

We have evaluated the D-index performance on metric data sets consisting of sentences form the Czech language corpus compared by the edit distance mea-

sure. For illustration, see Figure 2 for the edit distance density of our data set. Observe that the most frequent distance was around 100 and the longest distance was 500, equal to the length of the longest sentence.

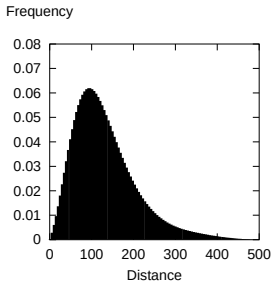

Fig. 2. Distance density for the data set

In all experiments, the search costs are measured in terms of block reads and the number of distance computations. All presented cost values are averages obtained by execution of queries for 50 different query objects (sentences) and the constant search radius, that is the maximum edit distance to the query object. 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 this short article, we only present results for the range queries. 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 our approach.

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 query radius.

Figure 3 shows global trends of costs spent by evaluating 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

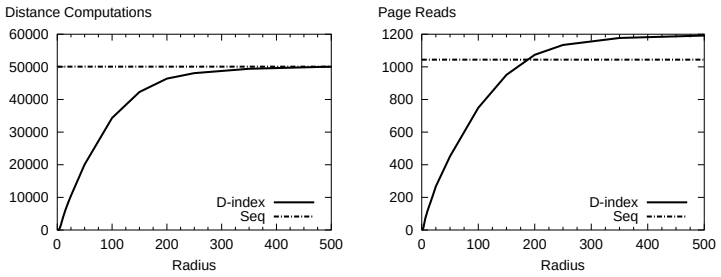

Fig. 3. Range queries.

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 block utilization of the D-index is lower than the average block utilization of the SEQ. As Figure 3 illustrates, 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 milliseconds while the distance computations between a query object and objects in only one block can take nearly one second.

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.

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 0.25 second. 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 less than 1 second. This is in a sharp contrast with 5 minutes 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
succ	1.14	12.42
unsucc	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.

Scalability. To analyze the scalability of D-index, we have considered collections of sentences ranging from 50,000 up 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.

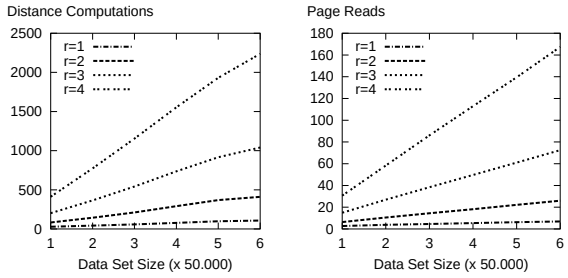

Fig. 4. Range search scalability.

From the figure we can see that the search costs scale up with slightly sublinear 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 files.

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.

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 its quadratic computational complexity, the edit distance is an expensive operation and a sequential search is prohibitive on large files. We have observed by experiments that a sequential search on 50,000 sentences takes about 5 minutes, which would result in a half an hour for a 6 times larger file of 300,000 sentences.

We have applied the D-index, a metric index structure, and we have performed numerous experiments to analyze its search properties. We can conclude that 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.

Due to the lack of space, we do not present results for the similarity nearest neighbor queries. These queries are suitable when the users are interested in a specific number of closest objects. For example, find six sentences with the shortest edit distance to a given reference. Our experiments also confirm high efficiency of the D-index for this type of queries, and we will report these results in some other publication in a short time.

Finally, 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.

References

1. E. Chavez, G. Navarro, R. Baeza-Yates, and J. Marroquin. Proximity Searching in Metric Spaces. To appear in *ACM Computing Surveys*. A preliminary version available as: *Technical Report TR/DCC-99-3*, Dept. of Computer Science, Univ. of Chile, 1999. `ftp://ftp.dcc.uchile.cl/pub/users/-gnavarro/survmetric.ps.gz`.
2. V. Dohnal, C. Gennaro, P. Savino, P. Zezula: D-Index: Distance Searching Index for Metric Data Sets. Submitted for publication.
3. 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.
4. K. Kukich: Techniques for automatically correcting words in text. *ACM Computing Surveys*, 1992, 24(4):377-439.
5. G. Navaro: A guided tour to approximate string matching. *ACM Computing Surveys*, 2001, 33(1):31-88.