Database index for approximate string matching

Alexander Korotkov
National Research Nuclear University "MEPhI"
Moscow, Russia
email: aekorotkov@gmail.com

Abstract — **In this work the database index for approximate string search is proposed. In particular the task of finding strings from some data domain which have a distance from given string less than given number is considered. Some kind of editorial string distance is used in capacity of string distance. Some subclass of regular expressions is used in the capacity of tree node predicates. The analysis of performance tests was performed and the areas of further researches were surveyed.**

Index Terms — **database systems, approximate string searching, search tree, dictionary search**

I. Introduction

Traditional database queries support a limited number of predicates. The limitation of standard search queries is caused in part by lack of implementation of required data types and search predicates in DBMS. This limitation is also caused by finite data structures on which database indexes are based. Search predicate support by database index frequently is required in order for a database to be scalable in terms of data amount incensement. The traditional search query predicates which are supported by database indexes are equality and linear range predicates [1].

Modern database applications tend to extend their functionality. Part of this functionality can be implemented without extension of DBMS, but the other part is not. Modern database applications frequently require the support of nonstandard data types and nonstandard search query predicated from a DBMS. An example of an application of nonstandard data types and nonstandard search query predicates is geographical informational systems (GIS). In these systems the geometrical data types which are nonstandard in DBMS are used. Also, the search predicates like overlap and containment are non-standard on DBMS. The spatial indexes are used for search optimization of these predicates [2].

This work considers the implementation of a database index for approximate stirng searching. The search predicate is based on the editorial distance between strings. The database index was implemented as an extension of GiST, which is a universal framework for database index implementation.

II. Approximate string search

An approximate string search is implied as a string search when search pattern or search domain can suffer from some kind of distortion. Some examples of approximate string search are finding DNA subsequences after possible mutations [3, 4, 5] and searching for typing and spelling errors in text [6, 7, 8].

In this work the searching data domain is the set S of strings s_i; $S = \{s_1, s_2, \dots, s_n\}$. The search predicate is the assertion that the editorial distance from the element of domain s_i to the search string p is less or equal than fixed number d, i.e. $ed(s_i, p) \leq d$. The editorial distance between string s_1 and string s_2 is the minimal number of editorial actions required to transform s1 to s_2. In this work the Levenshtein distance [9] is used in the capacity of editorial distance. In the Levenshtein distance there are three editorial actions: character insertion, character deletion, and character replacement. This type of search predicate can be applied to search for a misspelled word in the dictionary. However, the application of this search predicate is not limited by described case.

Various implementations of database indexes for approximate string search in this definition already exist [10]. In this work the search index implementation based on a generalized search tree is presented. The pg_trgm module which is an implementation of approximate string search indexes already exists for GiST [11]. However in the pg_trgm the amount of matching trigrams is used in the capacity of string distance whereas in this work the Levenshtein distance is used.

III. Generalized search tree

Generalized search tree presents a very general solution of the generalization of database access methods. GiST is the data structure which is extensible in terms of search queries as well as in terms of indexing data types. GiST defines the set of interface functions, for which implementation defines search indexes. These interface functions only depend on indexing data type and search predicates, but these functions are abstracted from data pages, records, query processing, etc. Thus to implement a search index using GiST it is not

required to write a code which maintains data structure [12]. Additionally, GiST generalizes the majority of currently existing search trees. For example B+-tree and R-tree can be implemented as GiST extensions [13].

At this moment GiST is fully implemented in open source postrelational DBMS PostgreSQL, though the result of GiST researches is used in the majority of commercial DBMSs such as Oracle and DB2. Several reasons for implementation of search index based on GiST in this work can be noted:

- To provide open source and license free solutions for approximate string searching
- To see completely new application of GiST
- Simplicity of GiST extension implementation

IV. Using GiST for approximate string search

As it was noted before the Levenshtein distance is used in the capacity of string distance. The Levenshtein distance is the minimum number of elementary operations needed to transform one string to another one. There are the following elementary operations:

- Insertion of arbitrary character to arbitrary position of string
- Replacement of arbitrary character of string with another arbitrary character
- Deletion of arbitrary character of string

The two sequences alignment algorithm [14, 15] can be used in order to calculate the distance between strings **a** and **b**. The two modifications of this algorithm were introduced in this work. Next let's consider this algorithm in detail.

The $\mathbf{a} = a_1a_2\ldots a_n$ and $\mathbf{b} = b_1b_2\ldots b_m$ are two strings of length n and m. The alignment is produced when a null character «-» is inserted into the strings; the new strings must have the same length L. After insertion of «–» the $\mathbf{a} = a_1a_2\ldots a_n$ becomes $\mathbf{a}^* = a_1^*a_2^*\ldots a_n^*$ and $\mathbf{b} = b_1b_2\ldots b_m$ becomes $\mathbf{b}^* = b_1^*b_2^*\ldots b_m^*$. The alignment is the two sequences which are written one over the other.

$$\begin{matrix} a_1^* & a_2^* & \ldots & a_L^* \\ b_1^* & b_2^* & \ldots & b_L^* \end{matrix}$$

The distance between strings **a** and **b** is introduced as:

$$D(\mathbf{a},\mathbf{b}) = \min \sum_{i=1}^{L} d(a_i^*, b_i^*)$$

The d(a,b) represents the distance between characters a and b. In the case of Levenshtein distance d(a,b) is defined below:

$$d(a,-) = d(-,a) = 1$$
$$d(a,b) = \begin{cases} 0, a = b \\ 1, a \neq b \end{cases}$$

The matrix D is introduced as the distance between prefixes of strings **a** and **b**.

$$D_{i,j} = D(a_1a_2\ldots a_i, b_1b_2\ldots b_j)$$

There are following rules of matrix filling:

$$D_{0,0} = 0$$

$$D_{0,j} = \sum_{k=1}^{j} d(-, b_k) \quad D_{i,0} = \sum_{k=1}^{i} d(a_k, -)$$

$$D_{i,j} = \min\{D_{i-1,j} + d(a_i,-), D_{i-1,j-1} + d(a_i,b_j), D_{i,j-1} + d(-,b_j)\}$$

TABLE 1. THE MATRIX OF ALIGNMENT

	–	b_1	b_2	…	b_m
–	$D_{0,0}$	$D_{0,1}$	$D_{0,2}$	…	$D_{0,m}$
a_1	$D_{1,0}$	$D_{1,1}$	$D_{1,2}$	…	$D_{1,m}$
a_2	$D_{2,0}$	$D_{2,1}$	$D_{2,2}$	…	$D_{2,m}$
…	…	…	…	…	…
a_n	$D_{n,0}$	$D_{n,1}$	$D_{n,2}$	…	$D_{n,m}$

The bottom right element of the matrix represents the distance between strings.

$$D_{n,m} = D(a_1a_2\ldots an, b_1b_2\ldots bm) = D(\mathbf{a},\mathbf{b})$$

A. Search predicate

In this work the optimizable search predicate is P(x) = (levenshtein (s,x) ≤ d), where levenshtein – the function of Levenshtein distance calculation, s – given string, d – given nonnegative number. Thus this predicate is true for strings which have Levenshtein distance to a given string less than or equal to a given number. Considering Levenshtein distance as metrics (it is possible because this distance has metrics properties) set of strings satisfying this predicate can be represented as solid sphere with center in s string and d radius.

B. Tree node predicate

The selection of tree node predicate is critical for GiST extension implementation. All the characteristics of the resulting tree generally depend on selected tree node predicate. In this work the predicate of matching to some class of regular expressions was selected. The description of a selected class of regular expressions is below. Each expression of a selected class can be represented as a concatenation of n (n is nonnegative integer) sub-expression. Each sub-expression can be defined in one of the ways below:

- One character from set of m characters (format of sub-expression is "$[a_1a_2\ldots a_m]$")
- One character from set of m characters or empty string (format of sub-expression is "$[a_1a_2\ldots a_m]?$")
- Any character or empty string (format of sub-expression is ".?")

In this work when the term "regular expression" is used this class of regular expressions is mentioned.

C. GiST interface methods implementation

The GiST interface consists of 7 methods. The purpose of these methods is considered below.

1) compress and decompress – these two methods are responsible for key compression and decompression (in keys should be suitable to work with them but it is frequently reasonable to compress a key before storing it to disc)
2) consistent – this method calculates compatibility of

tree node key and search query (The search optimization performs at the expense of this method. If the predicate of the tree node is incompatible with the search predicate then all the sub-tree should be skipped)

3) union – this method returns the union of two keys (all the values which conform to any of source keys should conform to the resulting key)
4) penalty – this method returns the measure of growth of the source key after addition of another key to it (this value should represent the measure of growth of values set which conforms to the key predicate)
5) picksplit – this method splits an array of keys into two arrays. It is desirable that union keys of the resulting two arrays have a minimal size (the size of the key is assumed to be the size of set of values which conform to the key predicate)
6) same – this methods checks if two keys are the same

In this work the compression of keys before writing them to the disc is not used. This is why the implementation of the compress and decompress methods was trivial. The implementation of same method also was trivial because all the regular expressions are stored in same manner. The penalty and picksplit methods were implemented using the keys union function and key size measurement function. The penalty method calculates keys union and calculates the difference between keys union size and source key size. The picksplit method is based on the Guttmann's clusterization algorithm. The union and consistent methods use the modification of two strings alignment algorithm.

1) Consistent method

The consistent method implementation uses the modification of two strings alignment algorithm[14] which makes it possible to find the minimal Levenshtein distance between any string which conforms to regular expression and the search query string. The resulting minimal distance can be represented by the expression:

$$d = \min\{\text{levenstein}(\mathbf{s},\mathbf{x}) | \mathbf{x} \sim \mathbf{r}\}$$

where **s** – search query string, **r** – regular expression, "~" – operator of regular expression conformance.

The decision on compatibility of search query and regular expression is made by comparing the resulting value and maximum distance of the search query.

The modification of two strings alignment algorithm is used in calculations of minimal distance. In this modification alignment between $\mathbf{s} = s_1 s_2 \ldots s_n$ and $\mathbf{r} = r_1 r_2 \ldots r_m$ is produced. The minimal distance between a string which conforms to **r** and **s** is calculated by expression:

$$D(\mathbf{s},\mathbf{r}) = \min \sum_{i=1}^{L} d(\mathbf{s}_i^*, \mathbf{r}_i^*)$$

There is following definition d(s,r):

$$d(s,r) = \begin{cases} 0, \text{if r allows s} \\ 1, \text{if r doesn't allow s} \end{cases}$$

$$d(s,-) = 1$$

$$d(-,r) = \begin{cases} 0, \text{if r allows empty string} \\ 1, \text{if r doesn't allow empty string} \end{cases}$$

TABLE 2. THE EXAMPLE OF ALIGNMENT MATRIX FOR FINDING MINIMAL DISTANCE BETWEEN STRING AND REGULAR EXPRESSION

	-	[dk]	[uzm]	[oc]	.?
-	0	1	2	2	2
d	1	0	1	1	1
o	2	1	1	1	1
m	3	2	1	1	1

Other parts of this algorithm is similar to the original algorithm. Let's consider an example. Let's find minimal distance between "dom" word and "[dk][uzm][oc]?.?" expression.

2) Union method

In the union method some other modification of the two strings alignment algorithm was used. The following distance function between two sub-expressions was used:

$$d(r_1, r_2) = \frac{u_1}{c + u_2} + \frac{u_2}{c + u_1},$$

There u_1 – the number of unique characters in the first sub-expression (the number of characters which are allowed by the first sub-expression and are not allowed by the second sub-expression), u_2 – the number of unique characters in the second sub-expression and c – number of common characters in sub-expressions. The empty string is assumed to be a separate character.

The case of equality of one sub-expression to ".?" should be considered separately (when the both sub-expressions are equal to ".?", it is evident that distance should be assumed as zero). In this case following measure was used:

$$d(".?",".?") = 0$$

$$d(".?", r_2) = \frac{n - c_2}{n}$$

There c_2 – the number of characters of second sub-expressions and n – the total number of characters in the alphabet used.

In the case of one sub-expression being skipped, the following measure was used:

$$d(r,-) = d(-,r) = 1 + \frac{u}{u + c},$$

There u = 0, when an empty string was allowed by sub-expression, u = 1, otherwise; c – the number of characters in sub-expression.

In this modification of alignment it is not only necessary to calculate the distance but also to find the union expression. Let's consider alignment of two expressions $\mathbf{a} = a_1 a_2 \ldots a_n$ and $\mathbf{b} = b_1 b_2 \ldots b_m$.

$$\begin{matrix} a_1^* & a_2^* & \ldots & a_L^* \\ b_1^* & b_2^* & \ldots & b_L^* \end{matrix}$$

The resulting expression $\mathbf{c} = c_1 c_2 \ldots c_m$ can be calculated by $c_i = u(a_i, b_i)$, where u – the function of two subexpressions

unification.

$$u(".?", a) = u(a, ".?") = ".?"$$

$$u("[a_1a_2\ldots a_n]","[\ b_1b_2\ldots b_m]") = "[a_1a_2\ldots a_nb_1b_2\ldots b_m]"$$

$$u("[a_1a_2\ldots a_n]","[\ b_1b_2\ldots b_m]?") =$$

$$u("[a_1a_2\ldots a_n]?","[\ b_1b_2\ldots b_m]")$$

$$= u("[a_1a_2\ldots a_n]?","[\ b_1b_2\ldots b_m]?") = "[a_1a_2\ldots a_nb_1b_2\ldots b_m]?"$$

In the operation of unification of sub-expressions if the number of characters in the final sub-expression exceed the threshold value k then this sub-expression is replaced by ".?". This replacement is performed in order to decrease the length of sub-expression and to improve the performance.

Let's consider the process of unification of "[abc][def][hg]?.?" and ".?[ad][bef]?h?h?" expressions as an example. The final matrix is presented below.

TABLE 3. THE EXAMPLE OF ALIGNMENT MATRIX UNIFICATION OF TWO REGULAR EXPRESSION

	-	.?	[ad]	[bef]?	h?	h?
-	0,00	1,00	2,33	3,33	4,33	5,33
[abc]	1,25	0,88	1,88	2,88	3,88	4,88
[def]	2,50	2,13	1,75	2,75	3,75	4,75
[hg]?	3,50	3,13	2,75	2,63	3,63	4,63
.?	4,50	3,50	3,75	3,59	3,53	4,53

The resulting alignment is

expression 1	[abc]	[def]	[hg]	.?	–
expression 2	.?	[ad]	[bef]?	h?	h?
union	.?	[adef]	[befhg]	.?	h?

The union expression is ".?[adef][befhg]?.?h?".

V. THE PERFORMANCE TESTING

Two tasks should be completed in order to perform synthetic testing of a database index. These tasks are to prepare test data domain and to prepare the test set of queries. The English dictionary with a volume of 61 505 words was used as test date domain.

After that the test was generated. There are two kinds of generated tests. The first kind of test is with random generated words. The second kind of test is with random distortion in existing words.

In the tests with random generated words the sequence of random characters of English alphabet with a length between 3 and 18 was generated. After that a random number between 1 and $\lceil n/5 \rceil$ was generated. This number was used as the radius of the search query. The expression $\lceil n/5 \rceil$ was used as the upper boundary in order to prevent the radius of the search query from being too high in comparison with word length.

In the tests with random distortion in existing words the random word from the dictionary was selected. Let's assume the length of this word as n. After that the random distortions (insertion, replacement and deletion of character) with number between 1 and $\lceil n/5 \rceil$ was applied to the selected word. Eventually the random number between 1 and $\lceil n/5 \rceil$ is selected as the radius of search query.

The results of the tests are presented in the tables.

TABLE 4. THE RESULTS OF INDEX TESTING WITH RANDOM DISTORTIONS IN EXISTING WORDS

	Search query radius									Average		
	1			2			3					
Dist.	S	WOI	WI	S	WOI	WI	S	WOI	WI	S	WOI	WI
0	2,29	124	66	1,36	141	115	1,529	186	132	1,72	150	104
1	2,91	124	58	1,54	143	108	1,751	180	126	2,07	149	98
2	3,52	142	63	1,62	142	102	1,851	184	121	2,38	156	95
3	10,7	180	36	6,66	187	063	2,549	182	116	6,64	183	72
Avg.	4,85	142	56	2,80	153	097	1,920	183	124	**3,19**	**160**	**92**

In the table 4 the results of testing of search queries with random distortion in existing words. The dependence of average speedup (S), average search time without index (NI) and average search time with index(I) on radius of search query and number of distortion in source word is presented. The speedup (S) calculates as $S = T_{woi} / T_i$, where T_{woi} and T_i are the time of search without using of index and the time of search with using of index respectively. As it is shown in the table the quotient of NI and WI is less then S as the rule. There is no contradiction because the average of quotient is not the quotient of average. This argues that the faster queries have higher speedup than slower ones.

In the table 5 the results of testing of search queries with random generated words are shown. In this table the same data as in the table above is presented but it depends on the length of generated word and radius of search query. As it is shown in the table the speedup increases as the length of generated word increases, and speedup decreases as the search radius increases.

TABLE 5. THE RESULTS OF INDEX TESTING WITH RANDOM WORDS

	Search query radius												Average		
	1			2			3			4					
Len.	S	NI	I	S	NI	I	S	NI	I	S	NI	I	S	NI	I
3	5,3	80	16										5,3	80	16
4	4,5	89	24										4,5	89	24
5	4,6	99	29										4,5	99	29
6	4,6	109	33	1,8	109	68							3,2	109	51
7	5,7	119	30	2,2	119	71							4,0	119	50
8	6,6	128	27	2,7	128	63							4,7	128	45
9	8,4	139	22	3,2	138	58							5,8	138	40
10	11	148	16	6,1	148	35							8,5	148	26
11	12	157	14	7,3	156	28	3,4	157	58				7,7	157	33
12	16	166	11	9,8	167	20	5,4	167	47				10,5	166	26
13	21	175	9,4	13,2	174	14	8,5	175	25				14,1	175	16
14	28	183	7,5	15,4	184	13	9,3	184	23				17,6	184	14
15	49	193	4,9	20,3	192	10	12,7	194	17				27,4	193	11
16	88	201	2,8	32,3	202	7,4	16,1	200	13	10,58	202	20	36,7	201	11
17	201	211	1,6	63,6	210	4,4	25,5	209	10	13,74	211	16	76,1	210	7,9
18	353	220	0,8	116	218	2,2	43,6	218	5,8	17,90	220	13	132	219	5,4
Avg.	51,3	151	16	22,6	165	30	15,6	188	25	14,07	211	16	**25,9**	**179**	**22**

VI. CONSLUSION

In this work the development of a new search index for the approximate string search based in GiST was considered. The new search index which allows searching in the domain of

strings S = (s_1, s_2, … s_n) such s_i that levenshtein(si, p) ≤ d was developed. The index testing with the English dictionary with the volume of 61 505 words in the capacity of a data domain was performed. The average speedup in the tests with random distortions in existing words was 3.19 times. The average speedup in the tests with random words was 25.88 times.

There are following directions of further researches:

- To research the developed index behavior on the different data domains. To understand which domain can be used with considerable performance improvement and which is not.
- To improve the performance of the index. There are two ways to improve the performance. The first way is to change the implementation of some GiST interface methods (in particular the PickSplit method). The second way is to change the class of regular expressions used in the capacity of tree node predicates.
- To apply the developed index for other search predicates. These predicates are the following: predicate based on the editorial distance different than Levenshtein distance, the regular expressions in the capacity of search predicate.

REFERENCES

[1] Douglas Comer, "The Ubiquitous B-Tree", Computing Surveys 11(2), June 1979, pp. 121–137.

[2] Antonin Guttman, "R-Trees: A Dynamic Index Structure For Spatial Searching", In Proc. ACM SIGMOD International Conference on Management of Data, June 1984, pp. 47–57.

[3] Miller , and DJ Lipman, "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs", Nucleic Acids Res., Sep. 1997, pp. 3389-3402.

[4] Zemin Ning, Anthony J. Cox, and James C. Mullikin, "METHODS: SSAHA: A Fast Search Method for Large DNA Databases", Genome Res., Oct. 2001, pp. 1725-1729.

[5] Maria B. Chaley, Eugene V. Korotkov, and Konstantin G. Skryabin, "Method Revealing Latent Periodicity of the Nucleotide Sequences Modified for a Case of Small Samples", DNA Res, June 1999, pp. 153-163.

[6] Sun Wu, Udi Manber, "Fast text searching: allowing errors", Communications of the ACM, Oct. 1992, pp. 83 – 91.

[7] James L. Peterson, "Computer programs for detecting and correcting spelling errors", Communications of the ACM archive, Dec. 1980, pp. 676 – 687.

[8] Fred J. Damerau, "A technique for computer detection and correction of spell-ing errors", March 1964, pp. 171 – 176.

[9] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals", Soviet Physics 10, 1966, pp. 707–710.

[10] G. Navarro, R. Baeza-Yates, E. Sutinen, and J. Tarhio, "Indexing methods for approximate string matching", IEEE Data Engineering Bulletin 24, April 2001, pp. 19-27.

[11] R. C. Angell, G. E. Freund, P. Willett, "Automatic spelling correction using a trigram similarity", Information Processing and Management 19(4), 1983, pp. 255-262.

[12] M. Kornacker, C. Mohan, J.M. Hellerstein, "Concurrency and recovery in generalized search trees", In Proceedings of the ACM-SIGMOD Conference, May 1997, pp. 62-72.

[13] Paul M. Aoki, "Generalizing "search" in generalized search trees" in Proc. 14th Int'l Conf. on Data Engineering, Feb. 1998.

[14] Robert A. Wagner, Michael J. Fischer, "The String-to-String Correction Problem", Journal of the ACM, Jan. 1974, pp. 168 – 173.

[15] Michael S. Waterman, "Introduction to computational biology: maps, sequences and genomes", 1995.