BASS: Approximate Search on Large String Databases

Jiong Yang
UIUC
jioyang@cs.uiuc.edu

Wei Wang
UNC Chapel Hill
weiwang@cs.unc.edu

Philip Yu
IBM
psyu@us.ibm.com

Abstract

In this paper, we study the problem on how to build an index structure for large string databases to efficiently support various types of string matching without the necessity of mapping the substrings to a numerical space (e.g., string B-tree and MRS-index) nor the restriction of in-memory practice (e.g., suffix tree and suffix array). Towards this goal, we propose a new indexing scheme, **BASS-tree***, to efficiently support general approximate substring match (in terms of certain symbol substitutions and misalignments) in sublinear time on a large string database. The key idea behind the design is that all positions in each string are grouped recursively into a fully balanced tree according to the similarities of the subsequent segments starting at those positions. Each node is labeled with a regular expression that describes the commonality of the substrings indexed through the subtree. Any search can then be properly directed to the portion in the database with a high potential of matching quickly. With the BASS-tree in place, wild card(s) in the query pattern can also be handled in a seamless way. In addition, search of a long pattern can be decomposed into a series of searches of short segments followed by a process to join the results. It has been demonstrated in our experiments that the potential performance improvement brought by BASS-tree is in an order of magnitude over alternative methods.*

1 Introduction

String data naturally exists in many applications including web documents, E-Commerce data, event sequences, and biological sequences, which generally involve very large databases and the database size still grows exponentially. Searching on these large string databases has become a common practice and the need for effective string indexes has been urgent. For example, biologists frequently need to search for similar samples of a certain DNA or protein region in a massive database of decoded sequences. Nowadays, the amount of mapped sequences exceeds 30 Giga base pairs and still grows at an exponential pace. However, the lack of an effective index makes the flat files continue to be used as the standard format to store the huge biological sequences. Searching on these sequences is usually carried out by sequentially scanning the entire database using a screening approach to identify the set of desired sequences (e.g., BLAST [1, 2]). This approach inevitably suffers from a prolonged response time when dealing with a large amount of sequences.

Similarity search on a string database can be classified into two categories: *exact* match and *approximate* match. The search of exact match looks for substrings in the database, which is exactly identical to the query pattern while the search of approximate match allows some types of imperfection such as substitutions between certain symbols, some degree of misalignment, and the presence of "wild-card" in the query pattern. We shall mention that supporting approximate match is very important to many applications. For instance, biologists have observed that mutations between certain pair of amino acids may occur at a noticeable probability in some proteins and such a mutation usually does not alter the biological function of the proteins.

Until very recently, many researchers have suggested using the suffix tree or the suffix array as an index of the string database [10, 14, 19, 25, 30]. However, this approach may not be the ideal choice due to the high I/O costs associated with constructing disk-reside structures [10, 14] and performing truly approximate match queries. Recently proposed indexes such as the String B-tree [11], the String R-tree [15], and the MRS-index [17] transformed the strings into a numerical space where a traditional multidimensional index structure is applied. Even though these techniques have been proved to be successful in their own application domains, the complicated relationship between symbols has not been taken into full consideration. The String B-tree and String R-tree were designed for exact match and hence did not provide efficient support for approximate match allowing symbol substitutions while the MRS-index did not allow the effects of different substitutions to be distinguished by associating with different scores.

Therefore, in this paper, we focus on the issue of supporting symbol substitution in searching approximate match. A score is assigned to each pair of symbols to represent the functional similarity between the pair of symbols. The higher the score, the more the similarity. A so-called *score matrix* is used to organize the set of scores. The score matrix is usually obtained through ample empirical studies and proper analytical inferences[1]. Figure 1 shows a score matrix, BLOSUM 50 [13], defined on the set of amino acids. Since their publication [13], the BLOSUM (BLOcks SUbstitution Matrix) score matrices have been the most popular scoring schemes used in evaluating the similarities between protein sequences. The entries on the diagonal correspond to identical amino acid pairs

[1]There is an extensive literature [8, 12] concerning the way how credible scores should be obtained and justified.

and have high positive scores. Entries in the rest of the matrix specify scores for substitution operations[2]. The similarity

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	5	-2	-1	-2	-1	-1	-1	0	-2	-1	-2	-1	-1	-3	-1	1	0	-3	-2	0
R	-2	7	-1	-2	-4	1	0	-3	0	-4	-3	3	-2	-3	-3	-1	-1	-3	-1	-3
N	-1	-1	7	2	-2	0	0	0	1	-3	-4	0	-2	-4	-2	1	0	-4	-2	-3
D	-2	-2	2	8	-4	0	2	-1	-1	-4	-4	-1	-4	-5	-1	0	-1	-5	-3	-4
C	-1	-4	-2	-4	13	-3	-3	-3	-3	-2	-2	-3	-2	-2	-4	-1	-1	-5	-3	-1
Q	-1	1	0	0	-3	7	2	-2	1	-3	-2	2	0	-4	-1	0	-1	-1	-1	-3
E	-1	0	0	2	-3	2	6	-3	0	-4	-3	1	-2	-3	-1	-1	-1	-3	-2	-3
G	0	-3	0	-1	-3	-2	-3	8	-2	-4	-4	-2	-3	-4	-2	0	-2	-3	-3	-4
H	-2	0	1	-1	-3	1	0	-2	10	-4	-3	0	-1	-1	-2	-1	-2	-3	2	-4
I	-1	-4	-3	-4	-2	-3	-4	-4	-4	5	2	-3	2	0	-3	-3	-1	-3	-1	4
L	-2	-3	-4	-4	-2	-2	-3	-4	-3	2	5	-3	3	1	-4	-3	-1	-2	-1	1
K	-1	3	0	-1	-3	2	1	-2	0	-3	-3	6	-2	-4	-1	0	-1	-3	-2	-3
M	-1	-2	-2	-4	-2	0	-2	-3	-1	2	3	-2	7	0	-3	-2	-1	-1	0	1
F	-3	-3	-4	-5	-2	-4	-3	-4	-1	0	1	-4	0	8	-4	-3	-2	1	4	-1
P	-1	-3	-2	-1	-4	-1	-1	-2	-2	-3	-4	-1	-3	-4	10	-1	-1	-4	-3	-3
S	1	-1	1	0	-1	0	-1	0	-1	-3	-3	0	-2	-3	-1	5	2	-4	-2	-2
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	2	5	-3	-2	0
W	-3	-3	-4	-5	-5	-1	-3	-3	-3	-3	-2	-3	-1	1	-4	-4	-3	15	2	-3
Y	-2	-1	-2	-3	-3	-1	-2	-3	2	-1	-1	-2	0	4	-3	-2	-2	2	8	-1
V	0	-3	-3	-4	-1	-3	-3	-4	-4	4	1	-3	1	-1	-3	-2	0	-3	-1	5

Figure 1. The BLOSUM 50 Score Matrix

search can then be posed by specifying a string/pattern and a similarity threshold, where the similarity threshold is used to define the separation between the approximate matches of the pattern and mismatches. The higher the similarity threshold, the more restrict the approximate match. This model of similarity search enables the presence of "wild card(s)" to be accommodated seamlessly.

To efficiently support this type of similarity search, we propose a new index structure called BASS-tree. The key idea behind the design is that all *position points*[3] are grouped recursively into a fully balanced tree according to the similarities of the *subsequent* substrings. Each node in the tree has a bounded size and is designed to be fit in a disk block. Leading positions of similar substrings would share a common ancestor and each node is labeled with a regular expression that describes the commonality of the substrings indexed through the subtree. Each leaf node contains a collection of position points and each internal node holds links to a list of children nodes. While search of short patterns can be performed directly on the BASS-tree (via a recursive traversal of *relevant* nodes[4]), search of a long pattern can be decomposed into a series of searches of short segments (which may or may not overlap with each other) followed by a process to *join* the results. Many properties of the BASS-tree, including the balanced height and compactness, provide the theoretical foundation of the efficiency of the search as well as the maintenance.

[2]Note that, a negative score means that the pair of amino acids are highly unlike. While most substitution operations are associated with negative scores, a few entries have positive scores. This is because mutations between certain pair of amino acids are frequently observed in practice and such mutations usually do not alter the biological function of proteins.

[3]We will define later that a position point is the combination of a string ID and an offset relative to the beginning of the string. The notation of position point is introduced to uniquely identify a specific position in the string database.

[4]A node is believed to be *relevant* if one of its leaf descendants may contain a pointer to a similar substring.

We have applied the BASS-tree index on a protein database and have been able to achieve at least an order of magnitude speed-up over other schemes. In this paper, we will use the protein database as an example to present the structure of BASS-tree and its potential advantage in supporting approximate match search, although the BASS-tree can be applied to other type of strings.

The remainder of this paper is organized as follows. Section 2 gives an overview of some related work. The approximate match query is formally introduced in Section 3 and the model of BASS-tree is presented in Section 4. Section 5 describes the process of approximate queries using the BASS-tree. An extensive experimental study is provided in Section 6 before the final conclusions are drawn in Section 7.

2 Related Work

In this section, we give a brief overview of previous research on approximate matching and on indexing string databases. Interested readers please refer to the individual papers for detail discussions.

Most previous work on approximate string match falls into three categories. The first category of approaches utilize a filtering algorithm to quickly eliminate large parts of the strings. The filter used is usually a simpler and *necessary* condition of the matching criterion. A family of algorithms that fall in this category use the notion of q*-gram* [22, 26, 27, 28]. A q-gram is a (short) substring of length q. Matches of q-grams in the target database may be stored and indexed [22] to facilitate the search. In general, such indices need linear space and can be built in linear time, with respect to the size of the target database, and may be able to provide sublinear search time on average if the required similarity is very high [24]. However, the q-gram based approaches are not suitable for approximate search allowing symbol substitutions with moderate similarity requirement.

Algorithms belonging to the second category [1, 2, 17, 21, 25] try to reduce the problem to a set of *approximate* searches of (short) segments of the original query pattern. The BLAST [1, 2] is a representative and has been the most popular tool used by biologists. The BLAST (Basic Local Alignment Search Tool) package is designed for finding high scoring *local alignments* between a query pattern and a target database, both of which can be either DNA or protein. The idea behind the BLAST algorithm is that true match alignments are very likely to contain within them a short stretch of identities, or very high scoring matches. Therefore, short segments of the query pattern are first taken as the "seeds" to search the database and the results are then "extended out" in search of good longer alignments.

The third category of approaches run on a persistent index structure instead of on the raw strings. Suffix tree [7, 16, 25, 29] and suffix array [18] are the most popular choices. Since every substring appearing in the string database can be found by traversing the suffix tree from the root, it is sufficient to explore every path starting at the root, descending by every branch until the point where it can be seen that the branch does not represent the beginning of a potential match. Some further

investigation has been taken in limiting the traversal by only visiting nodes that represent *viable prefixes* [7, 16, 25, 29]. Even though these structures typically take linear space and construction time [30] theoretically, they are very inefficient in both construction time and space requirements [10]. Most previous work [3, 10, 14, 19, 23, 25, 30] focused on fast algorithms for building suffix trees on a string database, though many of them failed to break the barrier of memory bottleneck [10]. One successful attempt on building a truly disk-reside suffix tree has been made recently in [14]. A new construction algorithm is proposed to trade the linear time performance for locality of access. A large persistent suffix tree can be built in $O(N \log N)$ time on average and in $O(N^2)$ time in the worst case.

Parallel to the above achievements, the model of String B-tree [11] and its multidimensional version, String R-tree [15], were developed to index string databases. The idea is to first convert strings to numerical domain and then apply the traditional indexing schemes. Although this approach is good for exact substring search, it does not support approximate search that allows symbol substitutions because the distance in the mapped numeric space does not reflect the similarity between symbols. The authors of [6] proposed an index structure, RE-tree, on regular expressions, but the queries are still limited to exact match.

More recently, the authors of [17] proposed a novel index structure, MRS-index, built on an integer space of wavelet coefficients to support range query and k-nearest neighbor query. Substrings of fixed length are mapped to a coefficient vector via wavelet transform and multiple lengths may be employed to generate local frequencies for different resolutions. A new distance function is also defined on the coefficient space, which is also proved to be a lower bound of the original distance (e.g., *edit distance* used in [17]). The efficiency of this approach largely depends on the tightness of the bound provided by the new distance function, which in turn depends on the choice of the original distance function. The performance is expected to degrade significantly if different weights are associated with different symbol substitutions. Such degradation amplifies for applications involving larger alphabet (e.g., the alphabet of protein sequences is 20 in contrast to the DNA sequences used in [17], which only have alphabet of size 4).

In additional to these advances, some interesting work has also been done on indexing regular expressions [6] under the assumption that no symbol mutation is allowed.

3 Query of Approximate Match

Let $\Im$ be the alphabet. For example, the alphabet for protein sequences are the set of 20 amino acids. Let $S = \{S_1, S_2, \ldots, S_n\}$ be a set of strings over the alphabet where $S_i = s_1^i s_2^i \ldots s_{l_i}^i$ and $s_j^i \in \Im$ for $1 \leq j \leq l_i$. l_i is referred to as the **length** of the string S_i. i is also referred to as the **ID** of the string S_i. Given a string $S_i = s_1^i s_2^i \ldots s_{l_i}^i$, a consecutive portion $s_{j_s}^i \ldots s_{j_e}^i$ $(1 \leq j_s \leq j_e \leq l_i)$ is referred to as a **segment** whose **length** is $j_e - j_s + 1$. We define $S_i[j_s..j_e] = s_{j_s}^i \ldots s_{j_e}^i$ in the following discussion. Each individual symbol s_j^i in a string $s_1^i s_2^i \ldots s_{l_i}^i$ can be regarded as a segment of length 1. In this case, we use $S_i[j]$ as an abbreviation of $S_i[j, j]$. It is easy to see that each distinct position in a string serves as the starting position of some segment(s). Each of these positions can be uniquely identified by the combination of string ID and offset within the string. Each distinct combination (of string ID i and offset j) is also referred to as a **position point** $\langle i, j \rangle$. We sometimes say that a position point $\langle i, j \rangle$ **leads** a segment $t_1 \ldots t_k$ if the segment starts at the jth position in the string S_i (i.e., $s_j^i = t_1, \ldots, s_{j+k-1}^i = t_k$). The total number of position points is equal to the size of the string database (i.e., $\sum_{i=1}^{n} l_i$). Our index structure is built upon the set of all position points in the database.

Given a score matrix that defines the similarity $sm(s, t)$ between each pair of symbols s and t ($s, t \in \Im$), we adopt the *weighted edit distance* to measure the similarity. Let $sm(-)$ be the score of inserting a gap in one string. The value of $sm(-)$ is typically negative. For example, $sm(-) = -8$ is a typical score used by many biologists for BLOSUM Score Matrix. The similarity between two strings is defined as the *maximum* aggregated similarity of any alignment of these two strings.

Definition 3.1 *Let $S_1 = s_1^1 s_2^1 \ldots s_l^1$ and $S_2 = s_1^2 s_2^2 \ldots s_{l'}^2$ be two strings. Their* **similarity** *is*

$$Sim(S_1, S_2) = \max \begin{cases} Sim(S_1[1..l-1], S_2[1..l'-1]) + sm(s_l^1, s_{l'}^2) \\ Sim(S_1, S_2[1..l'-1]) + sm(-) \\ Sim(S_1[1..l-1], S_2) + sm(-) \end{cases}$$

For example, using the score matrix in Figure 1, the similarity between **FILVM** and **LIVLM** is 2. One gap is introduced in each string in this best alignment.

	F	**I**	**L**	**V**	**–**	**M**	
	L	**I**	**–**	**V**	**L**	**M**	
sm	$+1$	$+5$	-8	$+5$	-8	$+7$	$= 2$

The query for searching approximate match can be formalized as follows.

Definition 3.2 *Given a string database and a score matrix, an* **approximate match query** *is defined in terms of two parameters: a query pattern σ and a similarity threshold τ. The query returns the set of substrings in the database whose similarity to σ is greater than or equal to τ. Each of such substrings is also referred to as an* **approximate match** *of σ.*

The threshold τ controls the set of symbol substitutions allowed in an approximate match. For example, **FILVM** is considered an approximate match of **LIVLM** if the similarity threshold is set to $\tau = 0$.

Given a set of strings of equal length, a regular expression (of the same length) can be used to describe the symbol(s) taken at each position of the strings. For example, the set of three strings **FIL**, **LIV**, and **LVF** can be described by the regular expression **(F+L)(I+V)(F+L+V)**. We also say that the

length of the regular expression is 3. Note that the regular expression of this form only serves as a necessary condition of the set of strings described. We choose this simple form (instead of using a precise yet more complex regular expression) because it is easy to generate, to store, and to operate. The measure of similarity can also be defined between a string and a regular expression and between two regular expressions.

Definition 3.3 *Given a string* $S_1 = s_1^1 s_2^1 \ldots s_l^1$ *and a regular expression* $\psi_2 = (t_1^1 + \ldots t_{n_1}^1)(t_1^2 + \ldots t_{n_2}^2) \ldots (t_1^{l'} + \ldots t_{n_{l'}}^{l'})$, *their* **similarity** *is defined as*

$$Sim(S_1, \psi_2) = \max \begin{cases} Sim(S_1[1..l-1], \psi_2[1..l'-1]) + \max_{1 \le i' \le n_{l'}} sm(s_l^1, t_{i'}^{l'}) \\ Sim(S_1, \psi_2[1..l'-1]) + sm(-) \\ Sim(S_1[1..l-1], \psi_2) + sm(-) \end{cases}$$

Definition 3.4 *Given two regular expressions* $\psi_1 = (t_1^1 + \ldots t_{n_1}^1)(t_1^2 + \ldots t_{n_2}^2) \ldots (t_1^l + \ldots t_{n_l}^l)$ *and* $\psi_2 = (v_1^1 + \ldots v_{m_1}^1)(v_1^2 + \ldots v_{m_2}^2) \ldots (v_1^{l'} + \ldots v_{m_{l'}}^{l'})$, *their* **similarity** *is*

$$Sim(\psi_1, \psi_2) = \max \begin{cases} Sim(\psi_1[1..l-1], \psi_2[1..l'-1]) + max\_sm(l, l') \\ Sim(\psi_1, \psi_2[1..l'-1]) + sm(-) \\ Sim(\psi_1[1..l-1], \psi_2) + sm(-) \end{cases}$$

where $max\_sm(l, l') = \max_{1 \le i \le n_l, 1 \le i' \le m_{l'}} sm(t_i^l, v_{i'}^{l'})$.

For example, the similarity between the string **FLM** and the regular expression **(F+L)(I)(K)** is 8. Through a similar calculation, we can also obtain that the similarity between **(F+L)(I)(K)** and **(F)(I+L)(L+M)** is 11. By definition, the similarity measure satisfies the following property.

Property 3.1 *The similarity between a string* σ *and the regular expression of a set of strings* Π *is always greater than or equal to the similarity between* σ *and any string in* Π.

To facilitate the following discussion, we also introduce the concept of generalization/specialization on regular expressions.

Definition 3.5 *Given two regular expressions* ψ_1 *and* ψ_2, ψ_1 *is said to be a* **specialization** *of* ψ_2 *iff, for every segment* σ *satisfying* ψ_1, *either* σ *also satisfies* ψ_2 *or a prefix of* σ *satisfies* ψ_2. *In this case,* ψ_2 *is called a* **generalization** *of* ψ_1.

For example, both **(A+S)** and **(D)(I+L)** are considered specializations of **(A+D+S+T)**. In addition, a string can also be regarded as a specialization of some regular expression. Property 3.1 can be regarded as a special instance of the following properties, which provide the motivation and justification of our approach.

Property 3.2 *The similarity between a string* σ *and a regular expression* ψ *is always greater than or equal to the similarity between* σ *and any specialization of* ψ.

Property 3.3 *The similarity between a string* σ *and a regular expression* ψ *is always less than or equal to the similarity between* σ *and any generalization of* ψ.

We also adopt the notion of *common generalization (specialization)*. For a set of regular expressions, Ψ, a regular expression ψ is a common generalization (specialization) of Ψ iff it is a **generalization (specialization)** of every expression in Ψ; and it is called the **minimum (maximum) common generalization (specialization)** iff there does not exist another regular expression ψ' ($\neq \psi$) such that ψ' is a specialization (generalization) of ψ and ψ' is also a common generalization (specialization) of Ψ.

Definition 3.6 *Given two regular expressions* $\psi_1 = (t_1^1 + \ldots t_{n_1}^1)(t_1^2 + \ldots t_{n_2}^2) \ldots (t_1^l + \ldots t_{n_l}^l)$ *and* $\psi_2 = (v_1^1 + \ldots v_{m_1}^1)(v_1^2 + \ldots v_{m_2}^2) \ldots (v_1^{l'} + \ldots v_{m_{l'}}^{l'})$, *where* ψ_1 *is a specialization of* ψ_2, *the* **relaxation** *from* ψ_1 *to* ψ_2 *is defined as*

$$R(\psi_1, \psi_2) = \sum_{i=1}^{l'} \left(\min_{1 \le j, k \le n_i} sm(t_j^i, t_k^i) - \min_{1 \le j, k \le m_i} sm(v_j^i, v_k^i) \right) + \sum_{i=l'+1}^{l} \left(\min_{1 \le j, k \le n_i} sm(t_j^i, t_k^i) - min\_sm \right)$$

where $min\_sm = \min_{s_j, s_k \in \Im} sm(s_j, s_k)$ *is the minimum score between any pair of symbols in the score matrix.*

For example, the relaxation from **(F+L)(I)(K)** to **(F+L)(I+V)** is $(\min\{sm(\mathbf{F},\mathbf{F}), sm(\mathbf{F},\mathbf{L}), sm(\mathbf{L},\mathbf{L})\} - \min\{sm(\mathbf{F},\mathbf{F}),\ sm(\mathbf{F},\mathbf{L}),\ sm(\mathbf{L},\mathbf{L})\}) + (\min\{sm(\mathbf{I},\mathbf{I})\} - \min\{sm(\mathbf{I},\mathbf{I}),\ sm(\mathbf{I},\mathbf{V}),\ sm(\mathbf{V},\mathbf{V})\}) + (\min\{sm(\mathbf{K},\mathbf{K})\} - min\_sm) = (\min\{8,1,5\} - \min\{8,1,5\}) + (\min\{5\} - \min\{5,4,5\}) + (\min\{6\} - (-5)) = 12$.

By definition, the relaxation from a regular expression to a generalization is always non-negative while the relaxation from a regular expression to itself is zero. The monotonicity property of the relaxation makes it a desirable measure to assess the "impact" of inserting a position point into a subtree, in terms of how much generalization of the label is needed at a node in order to include the position point in the subtree. Therefore, the relaxation measure is used during the construction of the BASS-tree to guide the insertion of position points.

4 Model of BASS-Tree

In this section, we introduce a novel indexing structure, namely BASS-tree[5], on string databases. A BASS-tree is a fully *balanced* tree that organizes all position points by recursively grouping together position points that lead *similar* segments in the string database. The structure of the BASS-tree is in spirit similar to that of the B+-tree [9] in the sense that all leaf nodes are on the same level and each internal node has a bounded number of children. The fundamental difference between the BASS-tree and those structures proposed for indexing numerical space is that the relationship between two position points can only be defined in a much looser fashion: (1) Their similarity largely depends on the length of the subsequent segments used in the calculation; and (2) Even if the segment length is given, the similarity does not preserve the property of metric as the transitivity does not hold. This characteristic poses a great challenge to the construction

[5] **BASS** stands for Balanced Approximate Substring Search.

and maintenance of the BASS-tree, especially when an overflowed node needs to be split. Figure 2(a) shows a portion of a BASS-tree built on a protein database using the score matrix BLOSUM 50 depicted in Figure 1. Each node in the BASS-tree is **labeled** with a regular expression that describes a set of *similar* segments in the string database. More specifically, the minimum common generalization of these segments is used as the label of the node. Each leaf node (also referred to as **data node**) contains a collection of position points that lead a set of similar segments described by the regular expression that serves as the **label** of the leaf node. In Figure 2(a), the node X_7 contains three position points that lead the segments with prefix **FIL**, **LIV**, and **LVF**. Each data node in Figure 2(a) is represented by a rectangle, which contains a list of entries, each of which stores a position point. As illustrated by a dashed arrow in the figure, the first entry of node X_7 stores the position point $\langle i, 1\rangle$. The **label** of each internal node describes a set of similar segments led by position points indexed through the subtree below. In Figure 2(a), the *label* of node X_5 is **(F+L)(I+V)** which indicates that the set of position points indexed in the subtree lead a set of segments (of length 2) that comply with the regular expression **(F+L)(I+V)**. For instance, the position point $\langle i, 1\rangle$ leads the segment **FI** that satisfies the label of node X_5. Each internal node, X, has a varying but bounded number of children, each of which is labeled with a *specialization* of the regular expression that labels X. This can be clearly observed from the example in Figure 2(a). According to Property 3.2, a node (and its subtree) can be excluded from further examination if the similarity between its label and the query pattern falls below the threshold τ. In such a way, the search of approximate matches can be well confined to a (small) portion of the BASS-tree and hence a (small) portion of the underlying string database.

4.1 Data Structures

As shown in Figure 4(a), each data node consists of a head field and a list of data entries, each of which stores a position point. The head field usually maintains the number of position points currently stored in the node and the segment length[6] currently used to assess the similarity between position points during node split. The maximum number of entries that can be stored at each data node is bounded by a parameter M_D whose value can be determined as

$$M_D = \lfloor \frac{size_b - size_h}{size_p} \rfloor \quad (1)$$

where $size_b$, $size_h$, and $size_p$ are the size of a disk block, the space occupied by the head of the data node, and the space used to store each position point, respectively. We have[7] $size_h = 8$ Bytes, $size_p = 8$ Bytes, and $size_b = 8$ KBytes in most cases. Then $M_D = 1023$. An example of the data node (e.g., node X_7 in Figure 2(a)) is also shown in Figure 4(b). This node currently has three position points and the segment length used in similarity calculation is 3.

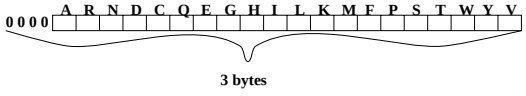
[6]This length should be equal to the length of the label of this data node.

[7]Assume that all parameters are of integer type.

Each internal node consists of a head field and a number of entries, each of which holds the information of a child node. The head field contains a counter to track the actual number of children and the maximum label length for children nodes. The essential information kept in each entry of the main body includes label of a child and the link to the child node. While it usually takes 4 Bytes to store the link of a child node, the label of the child may be of varying lengths and hence may take varying amount of space. To store the label efficiently, we use a bit array to encode the regular expression. Each bit represents the presence of a given amino acid at a given position of the regular expression. Since there are 20 distinct amino acids, three Bytes are allocated to encode each position of a regular expression as shown in Figure 3(a). (The first 4 bits are always set to zero.) If an amino acid is present at some position in the regular expression, the corresponding bit flips to 1. Figure 3(b) shows the bit array representation of the label of nodes X_3, X_5, and X_7 in Figure 2(a). The space needed to store a regular expression is $3 \times \ell$ Bytes[8] where ℓ is the length of the regular expression. It is possible that the labels of children of a node may be of variable length. For simplicity, we only use one number (maximum label length in a node) for all its children. Thus, the maximum label length is allocated for every children. For an internal node, let ℓ_{max} be the maximum length of the child node labels. The *maximum* number of children of the internal node, M_I, can be determined as

$$M_I = \lfloor \frac{size_b - size_h}{size_c} \rfloor \quad (2)$$

where $size_b$, $size_h$, and $size_c$ are the disk block size, the space allocated for the head field, and the space taken by each child entry. We have $size_b = 8$ KBytes, $size_h = 8$ Bytes, $size_c = 3 \times \ell_{max} + 4$ Bytes, and hence $M_I = \lfloor \frac{8184}{3 \times \ell_{max}+4} \rfloor$. For instance, for $\ell_{max} = 1, 2, 3, 4, 5$, M_I would be 1169, 818, 629, 511, 430, respectively. Figure 4(b) shows the information stored at node X_5 in Figure 2(a). The bit array for each child label is also shown.



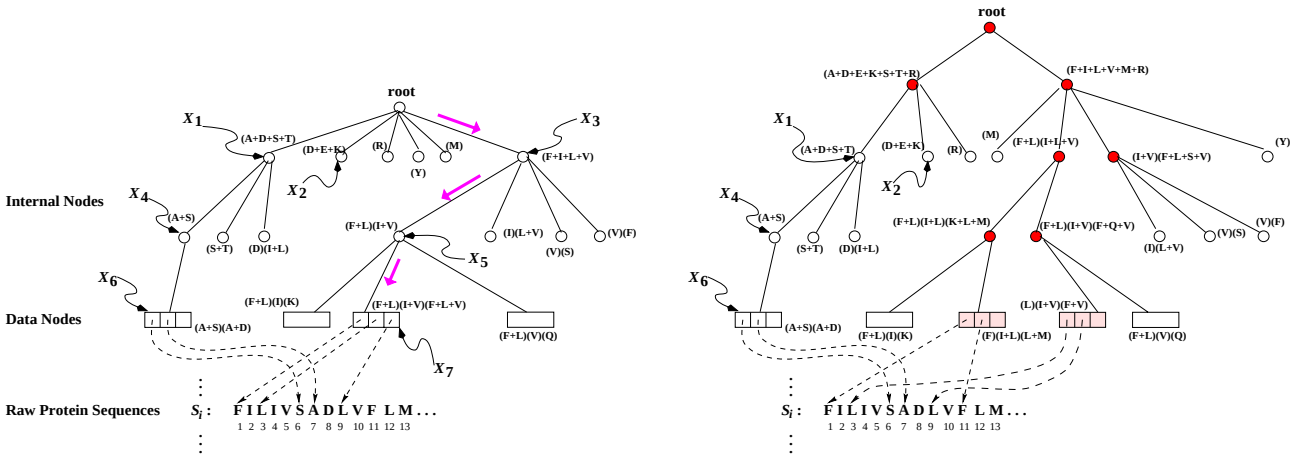
(a) The bit array for each position

Label	Bit Array
(F+I+L+V)	00000000000011001000001
(F+L)(I+V)	00000000000001001000000 00000000000010000000001
(F+L)(I+V)(F+L+V)	00000000000001001000000 00000000000010000000001 00000000000001001000001

(b) Label encoding

Figure 3. Bit Array Encoding of Regular Expression Label

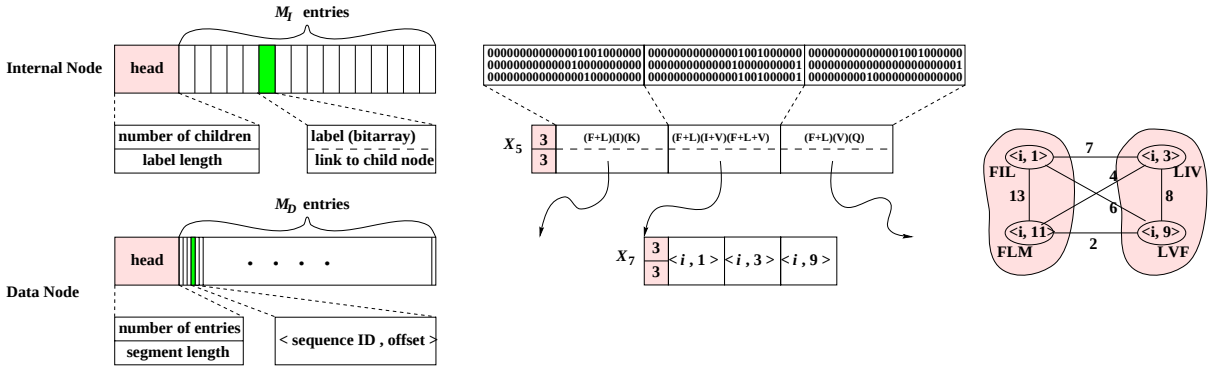
[8]Additional saving can be obtained by storing only the difference between the label of a child and its parent and applying similar techniques used in [11, 15].



(a) Part of a BASS-tree after Inserting Position Points <i, 1> ... <i, 10>

(b) Part of a BASS-tree after Inserting the Position Point <i, 11>

Figure 2. An Example of BASS-Tree



(a) Data Structures of the BASS-Tree

(b) Examples of Nodes X5 and X7 in Figure 2(a)

(c) Split Node X7 in Figure 2(a)

Figure 4. Data Structures of BASS-Tree

4.2 Building the BASS-Tree

The general procedure to build a BASS-tree on a string database is to insert each distinct position point into an initially empty tree. At the beginning, the BASS-tree only consists of a root node whose only child is an empty data node. An insertion procedure is invoked to insert each position point into an appropriate data node (determined by examining the segment led by this position point) and to maintain the relevant internal nodes accordingly.

4.2.1 Insertion of a Position Point

Given a BASS-tree, the procedure of inserting a position point $\langle i, j \rangle$ consists of two phases: (1) locating the proper data node and (2) updating the data node and its ascendant internal node(s). Intuitively, the insertion of this position point should be performed in such a data node that the least overall relaxation of the node's label and its ancestors' labels is incurred as a result. The routine of locating the desired data node can be done by, beginning from the root, recursively picking the child whose label needs minimum relaxation to "spell out" the segment α starting from the jth position in S_i and of the same length as the child label. If there is a tie among several children, the one with the *highest similarity* to α is picked and the procedure continues. For example, in order to locate an appropriate data node for the position point $\langle i, 11 \rangle$ in Figure 2(a), since every child of the root has label of length 1, we first compute the label relaxation needed to include the segment **F** for every child of the root. It is easy to see that node X_3 needs minimum relaxation (i.e., zero relaxation) to **F**. The same procedure is then invoked again to examine X_3's children where the segment **FL** is used in evaluating the relaxation. This procedure continues until the data node (e.g., X_7) is reached. (The path is indicated by bold arrows in Figure 2(a).)

Notice that, as the process proceeds from the root to the data node, the segment used in calculating the similarity also extends to include additional symbol(s). When the leaf level is reached, the segment used in relaxation assessment is of identical length to the label of the data node, which is also equal to the segment length used to perform a node split when this data node overflows. This would ensure that the exactly

same pool of segments would be evaluated during a further node split, and make the structure of BASS-tree more consistent and robust. Once the proper data node has been located, the process enters the second phase where two scenarios need to be considered.

- *The data node still has empty entries.* This scenario is relatively easy to handle since it does not involve structural change of the BASS-tree. The position point is stored in an empty entry and the head field of the data node is updated to reflect the inclusion of the new data entry. In addition, the node label (physically stored at the parent node) is updated accordingly and this update also propagates up towards the root if necessary. The label of the data node is the minimum common generalization of the corresponding segments of all position points, and can be updated incrementally after a position point is inserted. The label of an internal node is the minimum common generalization of the labels of its children, and can also be updated easily in an incremental fashion.

- *The data node is full.* This is the scenario when we try to insert the position point $\langle i, 11\rangle$ in X_7 in Figure 2(a), assuming that each data node can hold at most 3 entries (i.e., $M_D = 3$). In this case, additional node(s) need to be created in the BASS-tree. A partition procedure is invoked to split this node into two data nodes. All position points in this node and the new position point will be divided into two portions of roughly equal size, one portion will remain in this data node while the other portion will be stored in a newly created data node. A new child entry (corresponding to the newly created data node) should be inserted in the parent node of the (original) data node. If the parent node **overflows** as a result of the insertion of a new child entry, a splitting procedure is invoked to split this node into two internal nodes containing roughly same number of child entries. A new child entry is then added in the parent node of the internal node. This split may propagate all the way up to create a new root node and hence a new level of the BASS-tree. In the above example, assume that an internal node can take up to 54 Bytes (for illustration purpose only). Then the maximum number of children that X_5 can have is $M_I(X_5) = \lfloor \frac{54-8}{3\times 3+4} \rfloor = 3$. Similarly, we have $M_I(X_3) = 4$ and $M_I(root) = 6$. After the data node X_7 is split into two nodes, the additional child entry causes X_5 to overflow. This triggers the split of node X_5 into two internal nodes, which happens to cause the overflow of its ancestor node(s). In this example, this chain reaction results in cascading split of all internal nodes along the path up to the root and leads to the growth of the tree height (from 3 to 4). The resulting BASS-tree is shown in Figure 2(b). The shaded nodes (including both data nodes and internal nodes) are those nodes affected in this process.

We shall mention that, in most cases, inserting a position point would not cause any structural change (in terms of node split) and hence can be performed very efficiently. Specifically, the chance that inserting a position point would trigger a node split is $\frac{1}{M_D}$ where M_D is the capacity of the data node and is typically in the range of hundreds to thousands. In addition, the probability that such a data node split may cause an internal node to be split is $\frac{1}{\max M_I}$ where $\max M_I$ is the maximum capacity of an internal node and often ranges up to hundreds. The chance that the split propagates to the root and increases the height of the BASS-tree is very rare. As a matter of fact, the height of the BASS-tree grows at a very slow pace ($O(\log(\sum_{i=1}^{n} l_i))$). For example, a BASS-tree of height 3 is sufficient to index the entire SWISS-PROT protein database [4].

4.2.2 Node Split

The node split is a very crucial procedure to ensure the quality of the resulting BASS-tree, even though it is invoked only when a node is full. There certainly exist many choices on how to split a node (into two portions). A poor split may result in loss of vital information that plays a deterministic role in screening out unnecessary candidates during an approximate match and significantly prolong the response time. The optimal partition should be the one that maximize the similarities within each portion resulted from the partition but minimize the similarities across portions. Ideally, we would like that position points leading similar segments should be stored together in the BASS-tree so that only a small number of data nodes need to be retrieved eventually. In addition, the position points should be organized in such a way that the label of each node is as specific (i.e., containing less variant) and distinct (i.e., having less similarity to others) as possible. In general, the more distinct the label of each node, the more likely, on average, a node (and its subtree) may be eliminated from further investigation at a very early stage during an approximate search, and hence the faster the overall response time.

The problem of generating the optimal partition is in spirit similar to the **min-cut** problem [20]. Each entry in the node to be partitioned as well as the new entry can be mapped to a vertex, and the similarity between two entries is regarded as the weight on the edge between the two corresponding vertices. More specifically, the similarities between the labels of child entries are used as the weights on edges during an internal node split, whereas the similarities between segments led by position points when splitting a data node and the length of the segments used is specified in the head field of the node.

The objective then turns to partition the graph into two portions, each consists of roughly half of the vertices, and the maximum weight on the edges across portions is minimized. Consider the previous example of split node X_7 in Figure 2(a) due to the insertion of position point $\langle i, 11\rangle$. The mapped graph is shown in Figure 4(c). The length of segment used in similarity assessment is 3. The segment annotated beside each vertex is the segment used in similarity computation. Each shaded area represents a portion after the partition. In this case, all edges across portions have weights less than that of the edges within each portion. This graph partition can be solved in $O(M^2 \log M)$ time via a randomized algorithm where M is the number of entries considered [20] (e.g.,

$M_I + 1$ for internal node and $M_D + 1$ for data node). Due to space limitations, interested readers please refer to [20] for the detail algorithm.

We also want to mention that the length of a node label does not necessarily corresponds to its level in the BASS-tree, even though the label length of a node is always greater than or equal to that of its parent. In fact, the length of the regular expression used to label a node is driven by the needs. The label of a data node has the same length as the segment length used to assess similarity between position points. The only time it might be incremented is when the current length fails to produce a good partition during a data node split (e.g., when the similarities between every pair of position points are uniformly high using the current segment length). During an internal node split, lengthening of any child label may propagate up to the split internal nodes.

The size of the BASS-tree is linear in proportion to the number of position points, which is also the size of the string database. As a fully balanced tree, the height is guaranteed to be $O(\log N)$ where N is the size of the string database. The BASS-tree is also a dynamic index structure in the sense that update to the string database can be accommodated efficiently. The insertion of a position point takes $O(\log N)$ time and the deletion can be carried out in an analogous procedure to the insertion and also takes $O(\log N)$ time. Due to space limitations, we omit the detail algorithm for deletions.

5 Approximate Match on BASS-tree

Once the BASS-tree has been constructed on a string database, the cost of processing an approximate match of any given pattern can be dramatically reduced. One type of commonly asked query is, given a pattern $\sigma = s_1 s_2 \ldots s_l$ and a similarity threshold τ, find the set of substrings in the database whose similarities to σ are at least τ. A depth-first traversal of the BASS-tree is able to locate all position points that lead the qualified segments. Starting from the root, a recursive procedure is invoked to examine the label of every child and determine the set of children that may hold index to at least one segment potentially possessing sufficient similarity to the query pattern. This can be achieved by estimating the maximum possible similarity between the query pattern and any segment indexed through the child node. The maximum similarity is equal to the summation of (1) the highest similarity between the label of the child node and any prefix of the query pattern and (2) the highest possible similarity that can be produced by the remaining part of the query pattern. For example, the maximum similarity between **FALM** and node X_5 in Figure 2(a) can be computed as follows. First, the prefix of **FALM** which has the highest similarity to the node label **(F+L)(I+V)** is **FAL** (Sim(**FAL**,**(F+L)(I+V)**) $= 5$).

	F	**A**	**L**	
	$(\mathbf{F}+\mathbf{L})$	$-$	$(\mathbf{I}+\mathbf{V})$	
sm	$+8$	-8	$+5$	$= 5$

Then, the highest possible similarity that can be produced by the remaining portion, **M**, is sm(**M**, **M**) $= 7$. Finally, the maximum similarity between **FALM** and X_5 is $5 + 7 = 12$. If the maximum similarity to a child node is less than τ, then the child node and its subtree can be excluded from further evaluation according to Property 3.3. If the maximum similarity to a child node is greater or equal to τ, the child node is considered *relevant* to the query and the same procedure will be invoked on the child node. The process continues until the set of relevant data nodes are located.

The computational complexity of the approximate match query depends on the number of data nodes actually involved. In most cases, this number can be regarded as a constant (with respect to the size of the string database). Then each query can be answered in $O(\log N)$ time where N is the size of the database.

6 Experimental Results

We evaluate the BASS-tree on an IBM-AIX computer with a 333MHz CPU and 128 MB main memory. All programs are implemented in C. The SWISS-PROT protein database [4] is used in our experiments, which consists of 122,550 protein strings with an average length of 367 amino acids. An amino acid is encoded using 1 byte.

To build a BASS-tree, we set the data page size to be 8KB, which is the same as a disk I/O page. The maximum number of position points in a data node is 1023. To fit an internal node in a data page, the number of children in an internal node is calculated using Equation 2. In this experiment, we find that the average fan-out of an internal node is 301.2 and the height is 3, which confirm with our previous analysis. The structure of BASS-tree is very compact. The space occupied by the internal nodes is roughly 5% of the database size.

6.1 Effectiveness of the Node Split Algorithm

The efficiency of answering an approximate match query using a BASS-tree depends on the number of nodes visited. The more the distinction between labels of sibling nodes in the BASS-tree, the less the expected number of nodes to be visited during an approximate match query. To evaluate the effectiveness of our node split algorithm, we compare the average similarities between every pair of sibling nodes using different node split policies. Table 1 shows a comparison of the average similarity of sibling nodes in the BASS-tree using our node split algorithm and the average similarity of sibling nodes if random split is performed on overflowed nodes. BLOSUM 50 is used in this experiment. The average similarity between siblings decreases as we proceed from the root to the leaf level. This is due to the fact that the label of a node is always a specialization of its parent. It can be induced from the definition that the similarity between two regular expressions is always greater than or equal to the similarity between specializations of these two regular expressions. The BASS split algorithm, however, enables the average similarity to decrease as the node level increases, while the average similarity surges when the random split is employed. This difference is owe to the split policy employed in BASS, which intentionally minimizes the similarity between portions during each node split.

Table 1. Avg. Similarity of Two Split Algorithms

node level	1	2	3
BASS split	2.11	-1.06	-2.53
random split	3.32	7.98	12.61

6.2 Query Response Time

The most important metric to determine how good an index structure is the query response time. We compare the query time of four search schemes, the BASS-tree, the suffix tree (with suffix links) [**?**], the MRS-index [17], QUASAR [5], BLAST [1], and the linear scan (Boyer-Moore's algorithm [12]). Due to the space limitations, we do not furnish the comparison with String B-tree [11] and RE-tree [6] since they are designed for exact match only. Interested readers may refer to [31] for a detailed comparison.

The query response time with respect to the query pattern length is shown in Table 2. BLOSUM 50 is used in this set of experiments. The time is measured in second and is the average of 20 queries. The BASS-tree is clearly the winner. This performance attributes to the factors that the BASS-tree is very compact and can be easily held in memory all together, and that the BASS-tree tends to group similar segments together and hence confines the search to a very limited number of branches.

Table 2. Response Time (sec.) on SWISS-PROT Protein Database as a Function of Query Pattern Length

Pattern Length	5	10	20	40
BASS-tree	0.08	0.1	0.13	0.18
BLAST	0.81	1.12	1.36	2.05
QUASAR	0.63	0.95	1.45	2.35
MRS-index	2.9	4.6	7.5	12.7
Suffix tree	2.1	3.8	7.3	15.2
Linear scan	6.4	13.8	22	29.5

Secondly, we study the response time with different score matrices. Table 3 shows the response time of these six schemes using BLOSUM 30, BLOSUM 40, BLOSUM 50, BLOSUM 60, and BLOSUM 70, respectively. The average pairwise similarity between amino acids for each score matrix (i.e., $\frac{\sum_{s_i \in \Im, s_j \in \Im} sm(s_i, s_j)}{|\Im|^2}$) is also reported. BASS-tree performs uniformly well for all score matrices. As a method designed for highly selective queries, QUASAR is most vulnerable to the increase of average pairwise similarity because it may substantially weaken the filtration ratio. A comprehensive study can be found in [31]. To further evaluate the performance of BASS-tree, we also carry out a thorough study on the dependencies of the query response time to the database size, the average string length, and the number of distinct symbols, respectively. A set of synthetic data sets are

Table 3. Response Time (sec.) on SWISS-PROT Protein Database for Different Score Matrices

BLOSUM	30	40	50	60	70
Avg Similarity	-0.88	-1.17	-1.46	-1.28	-1.72
BASS-tree	0.21	0.17	0.13	0.16	0.10
BLAST	3.12	2.33	1.36	2.07	1.24
QUASAR	4.93	2.75	1.45	2.35	1.23
MRS-index	17.2	11.7	7.5	10.4	5.8
Suffix tree	18.6	12.5	7.3	10.5	5.8
Linear scan	23.1	22.4	22	22.1	21.6

generated for this purpose. The default values of the parameters are set as follows. The pattern length is 20. For a given symbol, the average pairwise similarity is -1.2. The average string length is 400. The database size is 2GB. The number of distinct symbols is 20. To isolate the effect of each parameter, we vary the value of only one parameter while the remaining parameters are fixed to the default vales in each set of experiments. The results are presented in Figure 5. The BASS-tree performs consistently well under all parameter settings while other schemes have mixed performance. The response time of BASS-tree increases in a logarithmic pace with respect to the database size and is insensitive to the changes of string length and vocabulary size. This advantage mainly attributes to the balanced and stable structure of the BASS-tree and the unique feature that similar segments are always grouped together. We also notice that the performance of the MRS-index degrades substantially when the vocabulary size increases. A detailed comparison is available in [31].

7 Conclusions

In many applications, a symbol in a string may be substituted by another symbol without change the natural characteristic of the string. In this paper, we proposed a new index structure, BASS-tree, for approximate search on large string databases. In order to expedite the search process, the BASS-tree localizes the access and computation to a small portion of the index and the database by grouping position points according to the similarities of following segments. Both symbol substitutions and gap penalties are supported by this scheme and querying patterns with wild card(s) can also be handled seamlessly. Empirical studies show that the BASS-tree outperforms all existing algorithms by a wide margin.

References

[1] S. Altschul, W. Gish, W. Miller, W. Myers, and J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, vol. 215, pp. 403-410, 1990.

[2] S. Altschul, T. Madden, A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and J. Lipman. Gaped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, vol. 25, pp. 3389-3402, 1997.

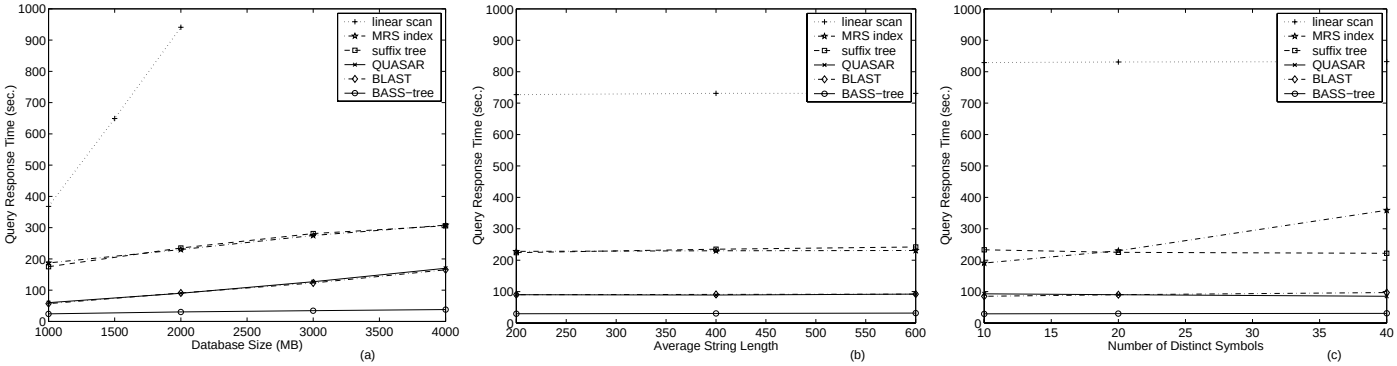


Figure 5. Query Response Time on Synthetic Data

[3] A. Andersson and S. Nilsson. Efficient implementation of suffix trees. *Softw. Pract. and Exp.*, vol 25(2), pp. 129-141, 1995.

[4] The SWISS-PROT protein sequence data bank. *Nuci. Acids Res.*. http://www.ebi.ac.uk/swissprot/

[5] S. Burkhardt, A. Crauser, P. Ferragina, H. Lenhof, E. Rivals, and M. Vingron. q-gram based database searching using a suffix array (QUASAR). *Proc. of Recomb*, pp. 77-83, 1999.

[6] C. Chan, M. Garofalakis and R. Rastogi. RE-tree: An efficient index structure for regular expressions. *VLDB*, pp. 263-274, 2002.

[7] A. Cobbs. Fast approximate matching using suffix trees. *Proc. 6th Annual Symp. Combinatorial Pattern Matching*, pp. 41-54, 1995.

[8] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*, Cambridge, 1998.

[9] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*, The Benjamin/Cummings Publishing Company, 1994.

[10] M. Farach, P. Ferragina, and S. Muthukrishnan. Overcoming the memory bottleneck in suffix tree construction. *Proc. FOCS*, pp. 174-185, 1998.

[11] P. Ferragina and R. Grossi. The string B-tree: a new data structure for string search in external memory and its applications. *J. ACM*, vol 46(2), pp. 236-280, 1999.

[12] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.

[13] S. Henikoff and J. G. Henikoff. Amino acid substitution matrices from protein blocks. *Proc. Natl. Academy Science*, vol 89(10), pp. 915-919, 1992.

[14] E. Hunt, M. Atkinson, and R. Irving. A database index to large biological sequences. *Proc. of VLDB*, pp. 139-148, 2001.

[15] H. V. Jagadish, N. Koudas, and D. Srivastava. On effective multi-dimensional indexing for strings. *Proc. ACM SIGMOD*, 2000.

[16] P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. *Proc. 2nd Annual Symp. Mathematical Foundations of Computer Science*, vol 16, pp. 240-248, 1991.

[17] T. Kahveci and A. K. Singh. An efficient index structure for string databases. *Proc. 27th VLDB Conference*, pp. 351-360, 2001.

[18] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Computing*, vol 22(5), pp. 935-948, 1993.

[19] E. M. McCreight. A space-economic suffix tree construction algorithm. *J. ACM*, vol 23(2), pp. 262-272, 1976.

[20] R. Motwani and P. Raghavan. *Randomized Algorithms*, Cambridge University Press, 1995.

[21] E. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, vol.12, pp. 345-374, 1994.

[22] G. Navarro and R. Baeza-Yates. A practical q-gram index for text retrieval allowing errors. *CLEI Electronic Journal*, vol 1(2), 1998.

[23] G. Navarro and R. Baeza-Yates. A new indexing method for approximate string matching. *Proc. 10th Annual Symp. Combinatorial Pattern Matching*, pp. 163-185, 1999.

[24] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, vol 33(1), pp. 31-88, 2000.

[25] G. Navarro and R. Baeza-Yates. A hybrid indexing method for approximate string matching. *J. of Discrete Algorithms*, vol. 1, no. 1, pp. 21-49, 2000.

[26] G. Navarro, E. Sutinen, J. Tanninen, and J. Tarhio. Indexing text with approximate q-grams. *CPM2000*, LNCS 1848, pp. 350-365, 2000.

[27] E. Sutinen and . Tarhio. Filtration with q-samples in approximate string matching. *Proc. 7th Annual Symp. Combinatorial Pattern Matching*, pp. 50-61, 1996.

[28] E. Ukkonen. Approximate string matching with q-grams and maximal matches. *Theor. Comput. Sci.*, vol 92(1), pp. 191-202, 1992.

[29] E. Ukkonen. Approximate string matching over suffix trees. *Proc. 4th Annual Symp. Combinatorial Pattern Matching*, pp. 228-242, 1993.

[30] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, vol 14(3), pp. 249-260, 1995.

[31] J. Yang, W. Wang, Y. Lee, and P. Yu. Approximate search on large string databases. *UNC Technical Report TR03-024*, 2003.