On Effective Multi-Dimensional Indexing for Strings

H. V. Jagadish*
University of Michigan
jag@eecs.umich.edu

Nick Koudas
AT&T Labs–Research
koudas@research.att.com

Divesh Srivastava
AT&T Labs–Research
divesh@research.att.com

Abstract

As databases have expanded in scope from storing purely business data to include XML documents, product catalogs, e-mail messages, and directory data, it has become increasingly important to search databases based on wild-card string matching: prefix matching, for example, is more common (and useful) than exact matching, for such data. In many cases, matches need to be on multiple attributes/dimensions, with correlations between the dimensions. Traditional multi-dimensional index structures, designed with (fixed length) numeric data in mind, are not suitable for matching unbounded length string data.

In this paper, we describe a general technique for adapting a multi-dimensional index structure for wild-card indexing of unbounded length string data. The key ideas are (a) a carefully developed mapping function from strings to rational numbers, (b) representing an unbounded length string in an index leaf page by a fixed length offset to an external key, and (c) storing multiple elided tries, one per dimension, in an index page to prune search during traversal of index pages. These basic ideas affect all index algorithms. In this paper, we present efficient algorithms for different types of string matching.

While our technique is applicable to a wide range of multi-dimensional index structures, we instantiate our generic techniques by adapting the 2-dimensional R-tree to string data. We demonstrate the space effectiveness and time benefits of using the string R-tree both analytically and experimentally.

1 Introduction

Conventional databases, and the index structures defined on these databases, have been designed for business data. Over the years, and particularly now, on account of the rapid growth of the Internet and XML, there has been a growing need to manage, and index, string data. Several index structures for string data have been proposed, beginning with the classical work in [2, 15, 16]. However, all of this work, to our knowledge, has dealt with indexing a single string attribute.

Supported in part by NSF under grant IIS-9986030

ACM SIGMOD 2000 5/00 Dallas, TX, USA

Multi-dimensional indexing of string data is important in a variety of contexts. Conjunctive term queries on document sets are standard, and supported by almost every information retrieval system. As we move to build similar functionality in database systems, we must support string matching queries effectively. In many contexts where strings are used, supporting partial match queries, such as prefix and substring matching, which are more common and more useful than exact matching, is crucial. In XML databases, many attributes and elements tend to be string-valued. Almost any complex enough XML query involves selections specified on multiple such string-valued attributes and elements. Even in relational databases, one may need multi-dimensional string querying. For example, in a data warehouse, one can imagine queries that select on a prefix match of the supplier name and a substring match of the product name. Our own work was motivated by the need for such indexing in the context of LDAP directories [10, 13]. We often find queries such as "find a person whose name begins with Sri and a telephone number in the 973 area code". A single index search on the name attribute in the AT&T corporate directory would find matches amongst the entire AT&T population rather than the much smaller population in the 973 area code. A single index search on the telephone number would find everyone in the 973 area code, rather than just the ones with the matching name. A simultaneous index on both attributes could greatly speed up such a search.

The need for multi-dimensional indexes has been recognized in the business and scientific contexts for some time now, and there is a rich history of research in multi-dimensional index structures [19, 7]. However, this research has tended to assume numeric data. This work does not carry over directly to string data for a number of reasons. First, most of these structures require partitioning of an attribute space using spatial distance and volume metrics, whose meaning is not obvious in the context of string data. Second, traditional index structures store attribute values in index pages for comparison purposes during

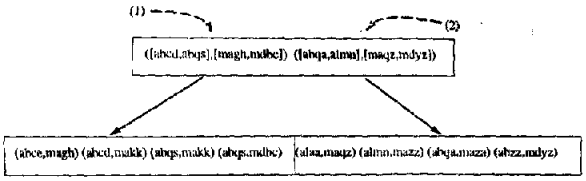


Figure 1: Applying R-tree concepts on a sample string data set

search, for page boundary demarcation, and so forth, implicitly assuming that these stored attribute values are small. This assumption is valid for most numeric attributes, which have values that can be stored in a small amount of space, frequently four bytes. Unfortunately, attributes with string values can often be very long: for instance, a 100 byte address field would not be unusual, nor would a 10,000 byte biography capsule. Storing such attribute values in index pages can be prohibitively expensive, leading us to search for special structures.

1.1 Contributions

In this paper, we present a generic technique for adapting a wide variety of multi-dimensional index structures for wildcard indexing of unbounded length string data. We choose to do so in this generic fashion because there is a plethora of structures that have been devised and little agreement on which is best, even in a well-specified application context [7]. Our generic technique addresses all of the issues raised in the foregoing. More specifically we make the following contributions:

- We map strings to a linear space in a manner that ensures strings that are extensions of a particular prefix are clustered, while preserving lexicographic ordering (Section 2).

  Such clustering is particularly important in a multi-dimensional context since one cannot, in multiple dimensions, use the well-known trick of keeping sideways pointers, as in 1-dimensional B-trees.

- We avoid storing long strings in index pages, and instead store only (small, fixed-length) references to the strings. To enable comparison during search, etc., *elided trie* (e-trie) structures, which are similar to Patricia trees [14], are used to organize these references on the index pages (Section 2).

  E-tries use space proportional to the number of string references stored in the index page, independent of the lengths of these strings.

- We present carefully designed index manipulation algorithms that exploit the e-tries on each index page to minimize the necessity of fetching the full string for any index operation (Section 3).

- To render the entire discussion concrete, we instantiate our generic techniques in the context of the well-known R-tree [9] and present both an analytical and an experimental evaluation of the *String R-tree* (Section 4).

2 The Generic String Multi-Dimensional Tree

We wish to create an index structure for database objects, indexed on d string-valued attributes, and support indexed retrieval of objects with exact match or prefix match, on some or all of the d attributes. We would like to use standard multi-dimensional index structures (see, e.g., [7]) for this purpose to the extent possible. To this end, there are two basic problems we need to address. First, how best to map strings to a number line so that standard techniques can be used. Second, how to avoid storing possibly very long strings in index pages, and avoid performing possibly costly comparisons between very long strings. We tackle each problem in turn, and then, based on our solutions to these problems, suggest a generic string multi-dimensional index.

2.1 Numeric Mapping

Let strings be comprised of symbols drawn from an alphabet of size α. Then each string can be considered to be a (fractional) number between 0 and 1 written out in base $\alpha + 1$. In other words, one can map each symbol to an integer in the range 1 to α. Let a string of length n be $s_1 s_2 \ldots s_n$, with each symbol s_i mapped to an integer t_i. The string as a whole is then mapped to $t_1/(\alpha+1) + t_2/(\alpha+1)^2 + \ldots + t_n/(\alpha+1)^n$. It is easy to show that there is a one-to-one mapping from (possibly infinite) strings to rational numbers, using the above technique.

Through these means, we may appear to have mapped strings to rational numbers and thus solved the problem of performing string matching using standard numeric techniques. However, several challenges remain. First, the representation of a (long enough) string can require a very large precision representation for the corresponding rational number, certainly well beyond the commonly used four or eight byte representations. In dealing with "true" numbers (real or rational) we are usually willing to sacrifice a small amount of precision (beyond 15 significant digits or so) in return for fast processing. The rational numbers resulting from the above string mapping require substantially more bits per symbol since α is likely to be much larger than 10. Furthermore, to be able to support partial

matches using this approach, we would need to keep the mapped numbers with high precision.

A second, and more subtle, problem has to do with the notion of distance. When numbers are drawn from a metric space, it is meaningful to speak of the distance between 10 and 12. It is also possible to use our intuition to estimate numbers of entries and numbers of queries that will fall within any numeric range, based upon reasonable distributions expected. These intuitions fall apart when dealing with strings. The difference between the numeric mappings of two strings represents some sort of lexicographic difference, and is not a very useful concept in terms of similarity or approximation. In particular, we would like the string az to be much closer to a than to b, for example, in a string domain with only the 26 Roman letters in the symbol alphabet.[1] This is because in a page partitioning, we would like to keep a and az together (along with ab, ac, etc.) to enable effective support for prefix match queries, rather than bundling az with b. We solve this problem next.

Distance Metric for Strings: Let the size of the alphabet be α, with an established lexicographic order on the symbols in the alphabet. Choose an integer $\beta > 2\alpha$. Let a string of length n be $s_1 s_2 \ldots s_n$, with each symbol s_i mapped to an integer t_i between 1 and α. The string as a whole is then mapped to $t_1/\beta + t_2/\beta^2 + \ldots + t_n/\beta^n$. Let us look at what is going on with an example, choosing β to be $2\alpha + 1$. In this case, let $[i, j]$ be the interval between two strings of some (equal) length that are adjacent to each other in the lexicographic ordering. Then, extensions of the first string by one character are equally placed in the open interval $(i, (i + j)/2)$. Thus, all extensions of the first string are still closer to i than to j in value. The role of β is to determine the "margin of victory". While it is technically sufficient to set β to be marginally greater than 2α, one can make absolutely certain of forcing page splits to occur only at desired points by choosing a much larger value of β. Our own belief is that a value of $\beta = 2\alpha + 1$ is sufficient.

Example 2.1 Consider an alphabet with 3 characters {a,b,c}. We have $\alpha = 3$ and choose $\beta = 7$. The mapping is shown in Figure 2. ∎

We are thus able to preserve the *prefix clustering property*: all strings that share a common prefix (of any length) occur within a numeric range that is smaller than the numeric distance between any string with this prefix and any string without this prefix. Further, it is straightforward to verify that this metric obeys the triangle inequality.

[1]For convenience of exposition, we use an alphabet comprising just the 26 lower-case Roman letters in all our examples. Of course, any real system will have a larger alphabet, including upper case letters, numerals, punctuation symbols, and perhaps even control characters.

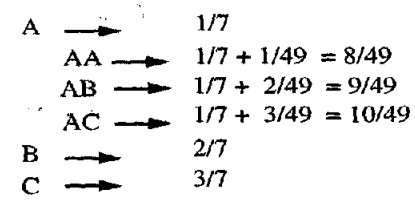


Figure 2: Mapping Strings to Rational Numbers to Preserve Prefix Properties

Categorical Attributes: We have thus far considered strings and assumed clustering based on shared prefixes, and lexicographic ordering, as the most natural clustering likely to be queried. Whereas these assumptions are reasonable for most string objects in databases, one important context where this is not true is with category labels. In such a case, we typically have (or can construct, based on observed queries) a hierarchy of categories and sub-categories, with queries likely to ask for partial matches corresponding to subtrees in this hierarchy, and with no regard to any prefix or lexicographic ordering in the labels. The simple technique of prefixing each category label string with the full path from the root in the category tree solves this problem. All categories in any subtree of the category tree now share as a common prefix, the labels on the path from the root of the category tree to the root of the specified subtree. Thus subtree match is converted to prefix match. Of course, the lengths of the strings involved is now considerably more. But this is exactly the problem that we address in this paper so that no category label strings are stored in the index tree. Moreover, even for the external data storage, with the help of some simple data structures, one can obtain the full effect of the longer label while physically storing only the shorter category label (see [12]).

2.2 External Keys

The strings that need to be indexed can be long. As such, it is inefficient both in terms of storage as well as in terms of processing time for long string comparisons, to store string values in index pages. Yet, almost all index structures require comparands of some sort, and typically these are drawn from the set of values being indexed. For instance, page boundaries are marked in terms of the smallest and the largest (in some dimension) values that occur in the page. Parent index pages have to store the boundaries for each of their children index pages. One way to avoid storing long strings is just to store references to them. Thus, instead of storing strings, we store pointers to the actual strings

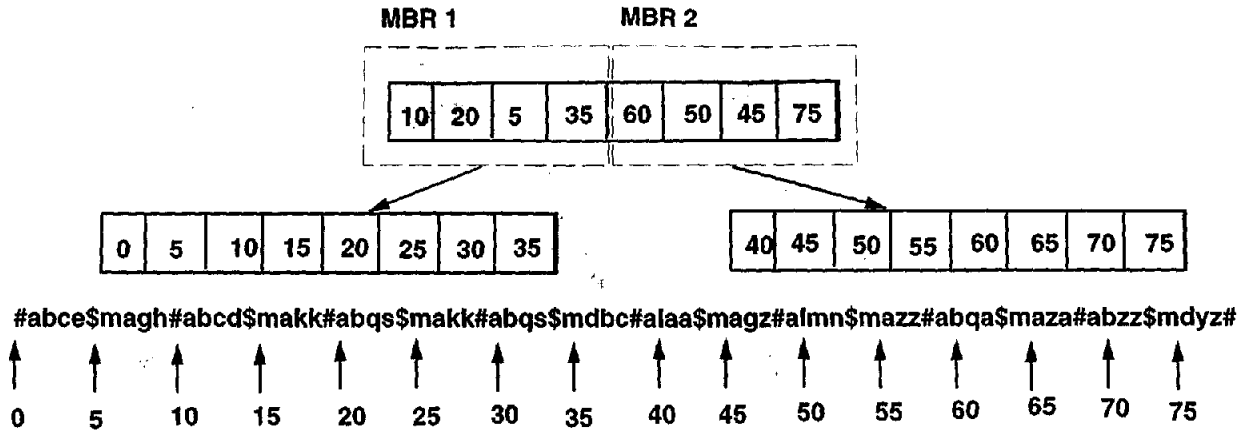


Figure 3: Using External Keys

stored elsewhere on disk. This idea has the virtue of providing us with the desired savings in memory, but now we have a challenge with respect to search (and other index operations).

Example 2.2 We illustrate the idea in Figure 3, using R-tree concepts. The original string data set, from Figure 1, is laid on disk. The representation has been enhanced with two characters '$' and '#' to denote boundaries between the elements of a 2-dimensional string pair and individual string pairs, respectively. The choice of the characters is arbitrary; any character can be used as long as it is not a member of the alphabet from which the strings were derived.

The index leaf pages contain references to individual string elements. This representation requires only a pointer (fixed size, usually 4 bytes) for each string. In each index leaf page, in this example, we store four 2-dimensional strings; we thus have two index leaf pages. In an index non-leaf page, we store the MBRs of the data elements of child index pages. The MBRs are also represented using offsets, not the actual strings, and take a fixed amount of space, that of storing 4 external offsets. ∎

2.3 Elided Tries

In most disk-based index structures, not much attention is paid to how one computes with the information on a page. Typically, one just performs a linear scan through the entries in a page to find the one(s) of interest. This is considered acceptable since most database applications are I/O bound, and the time required to scan a disk page linearly in memory is small compared to the time required to fetch a page from disk. However, with external keys we have a problem. Such a scan means that every pointer to an external key would have to be dereferenced, causing a horrendous amount of I/O. A solution that has been proposed to this problem, in the 1-dimensional case, is to store a small in-memory structure: an *elided trie* [4, 6].

An elided trie for a set of strings is obtained as follows. First, construct a compacted trie on the strings in question [14]. Many edges in this trie will have multiple symbols, in situations where the first of these symbols determined the path to be taken down the trie, and there was no decision point associated with the rest. Obtain an elided trie (*e-trie*) by *eliding* (not recording in the trie) all but the first symbol in every such case, and instead just keeping a count of the elided symbols. The concept is similar to that used in Patricia trees [16, 14]. It is important that no string in an elided trie be a proper prefix of another string in the same e-trie. This can be ensured by appending to each string a character that does not occur in the alphabet. In our examples, we do not show this appended character, for simplicity, and make sure that our data sets satisfy the above property. It has been shown that an elided trie on n strings can be constructed to require only $O(n)$ space, irrespective of the lengths of the strings in question [6].

Example 2.3 Figure 4(a) shows a sample data set. Figure 4(b) shows a compacted trie constructed on this data set. Notice that strings need not be unique in the data set. Duplication is possible and the duplicate strings can be recorded (as references to the string collection) in the leaf nodes of the compacted trie. Figure 4(c) presents the elided trie after all but the first character on each edge has been removed and the count of the number of characters on the path from the root has been inserted at each vertex.

The structure in Figure 4 has a number of interesting properties. Consider the search for string Q = abqe in Figure 4(d). Starting from the root, we can match the first character 'a'. The length of the string till the next node is 2; thus we visit this node and we test the third character 'q'. Following the branch for 'q' we reach a

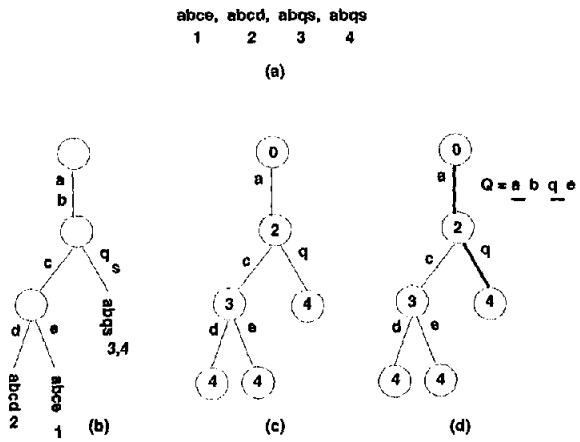


Figure 4: Elided Tries

node n, and we cannot branch further. Node n is the node in the elided trie where the maximal match of Q occurs. Let us choose a descending leaf l from the node n, in this case the node n itself, since it has no children. Leaf l has a very important property: it stores one of the strings that share the longest common prefix with Q [6]. We next retrieve that string. By testing it against Q, we can find Q's position in the collection of strings in the trie. The way to perform this test is to start again from the root of the e-trie and follow the correct path according to the values of the missing characters in the maximal match of Q. For the specific example, string Q should be located as a left child of node n. We can also test for Q's membership in the collection of strings in an e-trie in a similar way.

Consider now the problem of finding the positions of all strings having a given query Q, say ab, as a prefix. We can follow a similar procedure for this problem. The maximal match of Q on the e-trie matches only character 'a'. Then by retrieving any one of the strings corresponding to the leaves below, we can determine that (in this example) all the strings below share the prefix ab. ∎

We say that a string s is *maximally* matched in the e-trie if each character of s is matched with either elided or non-elided characters along a path in the e-trie. We say that a string s is *completely* matched in the e-trie if it is maximally matched with no elided characters. We say that a string s is *partially* matched in the e-trie if it is maximally matched with elided characters. E-trie structures can be efficiently stored, by linearizing them, and efficiently implemented to require minimal storage, independent of the string lengths [5].

The notion of a trie and a compacted trie can be extended to string-tuples in a multi-dimensional space [11]. Such multi-dimensional tries can also be elided in a manner similar to regular 1-dimensional tries, and it was our first thought to use this data structure. However, it turned out that one could always do at least as well through keeping multiple 1-dimensional e-tries, one for each dimension. The reason is that any specified search predicate can be evaluated against all the tries, sets of candidate matches obtained from each, and then disk accesses performed only to verify candidates in the intersection. This is as good as multi-dimensional trie indexing could possibly get. As such, we focus on multiple 1-dimensional e-tries per index page in what follows.

In a multi-dimensional structure, we could have the same key value occur in multiple string tuples, in conjunction with different other strings each time. However, the e-tries are constructed on only one dimension at a time. As such, there could be multiple string tuples that match perfectly on the chosen dimension, and are all pointed to from the same e-trie leaf node. A property of these external references from a common node is that all the objects retrieved have exactly the same value for the string attribute of focus in the externally stored string tuple.

Example 2.4 We are now able to demonstrate the structure of a leaf index page (Figure 5(a)) and an index non-leaf page (Figure 5(b)) in the 2-dimensional string R-tree. E-tries are constructed for each dimension on the actual string representations. The e-tries in both the leaf and the index non-leaf pages point to the offsets in the index page, which point to the actual data on disk. Thus, the associations between the strings corresponding to the different dimensions in a string tuple are implicitly recorded as shared index page offsets. Similarly, the associations between the minimum bounding rectangles are also kept implicitly in the e-trie nodes of index non-leaf pages. ∎

Enhancing existing multi-dimensional index trees to effectively deal with strings essentially requires careful manipulation of elided tries, for the various index operations of search, insert, split, merge, etc. How to do so efficiently in multiple dimensions is the subject of the rest of this paper.

2.4 Distance Computation with Elided Tries

The mapping introduced in Section 2.1 can be used to compute the distance between any two strings, and hence determine the volume of any hyper-rectangle. The mapping assumes the exact string representation is available. However, with e-tries, the common case is that several characters have been elided. The mapping can, of course, always be computed by accessing the exact string representations from disk, and the distances and volumes subsequently determined. It turns out that one does not have to incur this cost if one

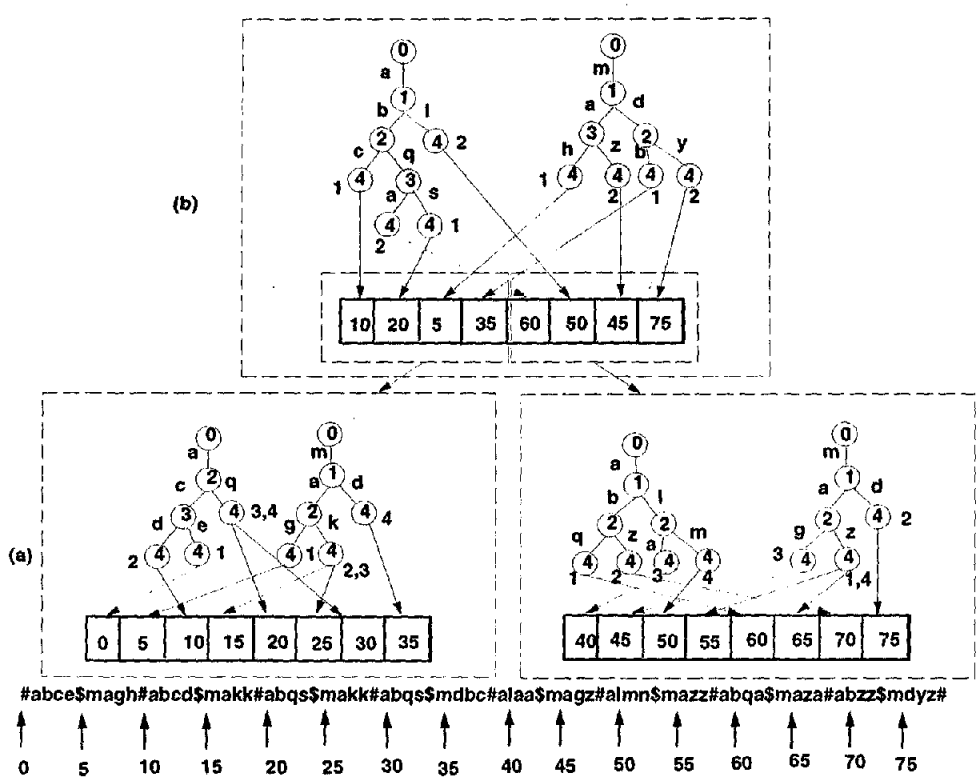


Figure 5: String R-tree: Index Leaf and Non-Leaf Pages

is willing to accept *approximate* distances and volume measures, as illustrated below.

Example 2.5 Suppose we want to compute, using the e-trie in Figure 4, the distance between strings abcd (i.e., offset 2) and abqs (i.e., offset 4). From the elided representation of a string we have complete knowledge of the length of the string as well as the positions of the elided characters. Although we do not know the second character of the strings in the e-trie, it does not affect the distance between the two strings (since they are guaranteed to have the same second character), even though it affects the actual numeric values, under the mapping, of the two strings.

While the missing fourth character of the string at offset 4 does impact the exact distance, we can derive *lower* and *upper* bounds for this distance. The lower bound is obtained by assuming that the missing fourth character is 'a' (i.e., the string at offset 4 ends with qa), and the upper bound is obtained by assuming that this missing character is 'z' (i.e., the string ends with qz). An approximate distance between the strings can then be estimated as the average of the lower and the upper bounds. The error introduced in the approximate distance computation is very small. ∎

2.5 The Generic Index Structure

Based on the above ideas, we propose a generic string multi-dimensional index as follows:

- Build the multi-dimensional tree index of choice, partitioning either the attribute space or the data points in the manner selected.
- At each page in the index tree maintain d e-tries to describe the children pages, where d is the dimensionality of the index structure.

  The role of the e-tries is to establish partition boundaries that are $O(1)$ per partition, rather than using a possibly unbounded length string.
- Follow standard algorithms to search, insert, delete, and otherwise manage the index tree, with appropriate modifications on account of the changes discussed above.

The nature of these modifications is the subject of the next section.

3 The Algorithms

For reasons of space, update algorithms are not presented here. Algorithms for insertion and deletion, including page split and merge, as well as bulk-load, are available in the full version of this paper [12].

3.1 Indexed Data Access

We describe here modifications of the standard algorithms on multi-dimensional index structures to deal with prefix match queries on strings in 2-dimensions, for ease of exposition. Extensions to higher dimensions, and for exact match and range match queries can be derived analogously. Our presentation is in two parts. First, we describe search in index non-leaf pages. Then, we elaborate on search in index leaf pages.

Prefix Search In Index Non-Leaf Pages: As is common in all tree-based index structures, a search starts from the root of the index structure and proceeds down in the tree. The objective of this search in all multi-dimensional index structures is to obtain a list of pointers to lower level index pages, and proceed applying the same search procedure to each of these lower level index pages in turn, until we reach the index leaf pages. This is in contrast to search in 1-dimensional tree-based index structures that typically follow a single path down the index tree both for point and for range queries. We describe this generic step in terms of our modified index pages containing e-tries for each dimension. Let $Q = (s_x, s_y)$ be a 2-dimensional prefix query. Then the answer set of Q will contain all string-pairs (S_x, S_y) in the database that match (s_x*, s_y*). · Let e_x and e_y denote the elided tries on the x and y dimensions respectively. Consider an index non-leaf page, I; searching I is algorithmically described in the full version of the paper [12]. Intuitively, we are interested in identifying the minimum bounding rectangles on index page I, such that the extent of the rectangles overlaps with the prefix match query (s_x*, s_y*). Clearly, this can be achieved by doing multiple disk accesses to retrieve all the strings corresponding to the boundaries of each of the minimum bounding rectangles stored on page I. However, this approach is unnecessarily expensive. Algorithm IndexNonLeafPagePrefixSearch shows how this set of MBRs can be identified with at most one disk access per dimension.

First, the query strings s_x and s_y are used to obtain the longest matches against the e-tries e_x and e_y. In extreme cases, $n1_x$ and/or $n1_y$ may be the roots of the elided tries. Note that these longest matches are not required to be maximal matches. To see why, consider the prefix match query `Q1 = (abd*, makk*)`, and the index pages depicted in Figure 5. There is no maximal match for the query string `makk` against the e-trie of the y dimension in the index non-leaf page (b). However, there is a prefix match to this string in the first index leaf page, whose MBR's y range is `[magh, mdbc]`. Similarly, for the query string in the x dimension. In this example, node $n1_x$ is the parent of the left-most leaf of the elided trie e_x, and node $n1_y$ is the parent of the left-most leaf of the elided trie e_y.

Next, elided characters along the longest matching paths are determined, as described in Section 2.3. In our example, there is an elided character (the third character) in the longest matching path in the y dimension, which results in an external disk access; any of the strings beginning at offsets 5 or 45 may be fetched. Since the elided (third) character is 'g', $n2_y$ is the parent of $n1_y$ in e_y. No disk access is performed in the x dimension, since the query string `abd` in the x dimension is matched only against non-elided characters in e_x; in this case $n2_x$ is the same as $n1_x$.

Next, relevant minimum bounding rectangles along the x and the y dimensions are identified, by invoking Algorithm IdentifyOverlappingMBRs. In that algorithm, L_s identifies the MBRs whose left-endpoint is to the "left" of the query string s, and R_s identifies the MBRs whose right-endpoint is to the "right" of the query string s. Their intersection is the desired set. In our example, of the two minimum bounding rectangles stored in the index non-leaf page of Figure 5, $MBR_x = \{1\}$, and $MBR_y = \{1, 2\}$. We first illustrate the rationale for the x dimension. From the discovered elided characters, it can be determined that the x range of the first MBR is `[abc_, abqs]`, and the x range of the second MBR is `[abqa, al__]`, where '_' denotes an elided character. Independent of the values of the elided characters, prefix matches to query string `abd` can only fall in the first range. Next, consider the y dimension. From the discovered elided characters, it can be determined that the y range of the first MBR is `[magh,mdb_]`, and the y range of the second MBR is `[magz,mdy_]`. Independent of the value of the elided character, prefix matches to the query string `makk` can fall in both the ranges. This rationale is algorithmically captured by Algorithm IdentifyOverlappingMBRs.

Finally, the intersections of the MBRs in the two dimensions are computed. In our example, the result is that only the left-most index leaf page will be searched subsequently for query matches.

Theorem 3.1 *Given an index non-leaf page I, and a prefix query Q in d dimensions, Algorithm IndexNonLeafPagePrefixSearch* correctly *identifies the minimum set of children index pages of I that need to be further searched to find answers to Q. Further, it does so using at most d external disk accesses.* ∎

Prefix Search in an Index Leaf Page: An index leaf page stores the offsets of the actual strings on disk as well as e-tries on each string dimension. Since no string ranges are present, the search on an index leaf page follows a different strategy. Algorithm IndexLeafPagePrefixSearch presents the approach.

Algorithm IndexNonLeafPagePrefixSearch($I, Q = (s_x, s_y)$) {

1. Retrieve e-tries e_x and e_y from I.
2. Match s_x against e_x, and let $n1_x$ be the node in e_x corresponding to the longest match with s_x, based on elided and non-elided characters.

   Define node $n1_y$ analogously, based on matching s_y against e_y.
3. If there are any elided characters along the longest match path p_x from the root of e_x, then:

   (a) Choose any leaf l_x in the subtree rooted at $n1_x$.

   (b) Make an external disk access to retrieve the corresponding string in the x dimension, and determine all the elided characters along the path p_x.

   (c) Based on the known values of the elided characters along p_x, let $n2_x$ be the node along path p_x that has the longest match with s_x.

   If there are no elided characters along the path p_x, let $n2_x$ be the same as node $n1_x$.

   Define p_y, l_y and $n2_y$ analogously on the y dimension.
4. Determine the relevant minimum bounding rectangles in the x and y dimensions. MBR_x = IdentifyOverlappingMBRs(I, e_x, p_x, s_x). MBR_y = IdentifyOverlappingMBRs(I, e_y, p_y, s_y).
5. Let $MBR_{xy} = MBR_x \cap MBR_y$. For each of the child index pages I_c of I whose minimum bounding rectangle is included in MBR_{xy}, do either IndexNonLeafPagePrefixSearch(I_c, Q) or IndexLeafPagePrefixSearch(I_c, Q), based on where I_c is a non-leaf or a index leaf page.

}

Algorithm IdentifyOverlappingMBRs(I, e, p, s) {

/* p identifies the string, including both elided and non-elided characters, along the longest path in e that matches with s */

1. Insert string s into a local copy of e.
2. Let L_s denote the MBRs pointed to by trie nodes that precede s in a pre-order traversal of (the modified) elided trie e.
3. Let R_s denote the MBRs pointed to by trie nodes that succeed s in a post-order traversal of (the modified) elided trie e.
4. Return $L_s \cap R_s$.

}

Figure 6: Prefix Searching of an Index Non-Leaf Page

Algorithm IndexLeafPagePrefixSearch($I, Q = (s_x, s_y)$) {

1. Retrieve e-tries e_x and e_y from I.
2. Maximally match s_x against e_x, and let n_x be the node in e_x corresponding to this maximal match, based on elided and non-elided characters. Define node n_y analogously.
3. If either s_x or s_y did not match maximally, there are no matches. Return $\emptyset$.
4. Let C_x and C_y denote the sets of string-pair offsets in the subtrees rooted at n_x and n_y.

   (a) If $C_x \cap C_y = \emptyset$, there are no matches. Return $\emptyset$.

   (b) If both s_x and s_y matched completely (i.e., no elided characters), no additional disk accesses are required. Return $C_x \cap C_y$.

   (c) Else, choose any string-pair in $C_x \cap C_y$. Perform a disk access to retrieve this string-pair. This can be used to determine the elided characters on the paths to n_x and/or n_y. If any of the determined elided characters does not match s_x or s_y, return $\emptyset$. Else, return $C_x \cap C_y$.

}

Figure 7: Prefix Searching of an Index Leaf Page

Essentially, maximal matches (not just longest matches, as in the case of Algorithm IndexNonLeafPagePrefixSearch) need to be performed against the elided tries. The intersection of the string-pairs in the two subtrees are the candidate matches. Essentially, the key property that holds is that either all of them are matches, or none of them are matches, depending on whether or not the elided characters along the maximal match paths agree with the query strings. Since the two components of a string-pair are assumed to be stored contiguously, either or both components of a string pair can be obtained using a single disk access, and the elided characters determined.

Consider, again, our example prefix match query `Q = (abd, makk)`, and the left-most index leaf page. While there is a maximal match in the y dimension, there is no maximal match in the x dimension. Consequently, we can determine, without any additional disk access, that the query has no answers. If, instead, the prefix match query had been `Q = (abc, makk)`, then the maximal match of abc with e_x in the left-most index leaf page would have determined that the string-pairs 1 and 2 are potential matches. Similarly, the maximal match of makk with e_y would have determined that string-pairs 2 and 3 are potential matches. Computing the intersection, the algorithm would have determined that only string-pair 2 is a potential match. Retrieving this string-pair, it is determined that it is indeed a query answer.

In general, if disk accesses are independently performed to determine the elided characters for each of the e-tries, one can perform as many disk accesses per index leaf page as the number of dimensions. However, we have the following result:

Theorem 3.2 *Given an index leaf page I, and a prefix query Q in d dimensions, Algorithm IndexLeafPagePrefixSearch* correctly *identifies the string-tuples that are answers to Q. Further, it does so using at most 1 external disk access, independent of the number of dimensions d.* ∎

Exact Match Search: Given an exact match query $Q = (s_x, s_y)$, we wish to retrieve all string-pairs from the database, that exactly match Q on each dimension. The search on an index non-leaf page proceeds in exactly the same way as in the case of prefix match on an index non-leaf page. The search on an index leaf page is similar to the case of prefix match, except that we search for exact matches in the e-tries, instead of searching for prefix matches.

4 Experimental Evaluation

We implemented string R-trees and string B-trees and we report preliminary experiments on their comparative performance. We evaluate the performance of prefix and range queries using real data sets, for a range of query selectivities.

4.1 Description of the Index Structures

In our implementation, the page size was 8KB. The e-tries in both data structures use a `short` (2 bytes) to represent the node counts. In addition we use `char` (1 byte) for the representation of the non-elided characters. In the case of string R-trees, we record the associations between the leaf nodes of the e-tries using `short`. Disk offsets and pointers to pages used `long` (4 bytes). A small fraction of the page size is used for auxiliary information (counts etc). The resulting fanout is 200 for the string R-tree and 400 for the string B-tree using the above values.

In each disk page, we pre-allocate size for the maximum size of an e-trie. E-tries are linearized with a traversal and stored on a page; they are reconstructed on demand.

4.2 Description of Data Sets

We used two real data sets, extracted from an AT&T data warehouse. Both data sets contain 200,000 2-dimensional strings. As in any access method, skew in the underlying data space is an important parameter in the evaluation of comparative performance. Let N be the total number of multi-dimensional strings. We divide the string domain into a number, n, of lexicographically equidistant segments. This results in n^2 lexicographic buckets, in 2-dimensional space. The expected number of strings in bucket i is $\hat{C}_i = \frac{N}{n^2}$. Let C_i be the real count of strings in bucket i. We then perform the χ^2 goodness of fit test:

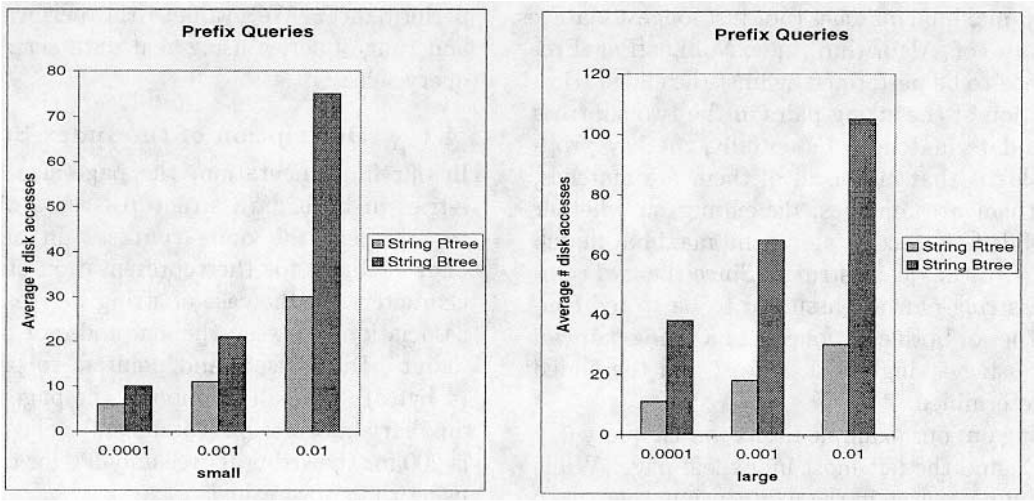
$$\chi^2 = \sum_{i=1}^{n^2} \frac{C_i - \hat{C}_i}{\hat{C}_i} \quad (1)$$

The first data set, which we refer to as D1, contains last names in the first column and computer generated login id's in the second, and has a small value of χ^2, or inter-column correlation between strings in this data set. The second data set, which we refer to as D2, contains first names in the first column and last names in the second. D2 has a large value of χ^2, implying high inter-column correlation.

4.3 Query Description

We evaluate the performance of prefix, and range queries[2] on both data sets for low, medium and high selectivities. We generate prefix queries by uniformly selecting tuples from the data sets and uniformly (on

[2]Although we have not described the evaluation of range match queries in this paper, this evaluation can be performed by simple extensions of our evaluation strategy for prefix match queries.



(a) Data Set: D1 (b) Data Set: D2

Figure 8: Performance of Prefix Queries on String R-trees and String B-trees

the length of the strings) selecting the query size. For range queries, we uniformly choose the start and end ranges from the data sets.

We ask 100 queries of each kind and we report the average number of disk accesses (including both index non-leaf and leaf pages) performed in order to retrieve the query answer. High selectivity queries, medium selectivity queries and low selectivity queries retrieve fractions of approximately 10^{-4}, 10^{-3} and 10^{-2} of the database, respectively.

4.4 Experimental Results

Figure 8 presents the performance of prefix match queries on string R-trees and string B-trees that have been bulk loaded, using the the low-x bulk loading method for R-trees [18] and sorting on the indexed dimension for B-trees, for various selectivities for data sets D1 and D2. We report on their comparative performance using the metric of average number of disk accesses, which is the common metric of choice for the evaluation of index structures. For data set D1 and high selectivity the performance of the string B-tree is close to that of the string R-tree. This is somewhat expected as the number of strings retrieved is small and small inter-column correlation exists in D1. It is evident that as the selectivity increases, string R-trees show large performance benefits compared to string B-trees. The overhead of retrieving strings with common prefix per dimension is too high for string B-trees and it is the main reason for their poor performance. Figure 8(b) presents results of the same experiment, on data set D2. The gap between the performance of string B-trees and string R-trees increases even further, due to the higher inter-column correlation of D2.

Figure 9 presents the performance of range queries on the data sets. The overall trends in performance are similar to prefix queries.

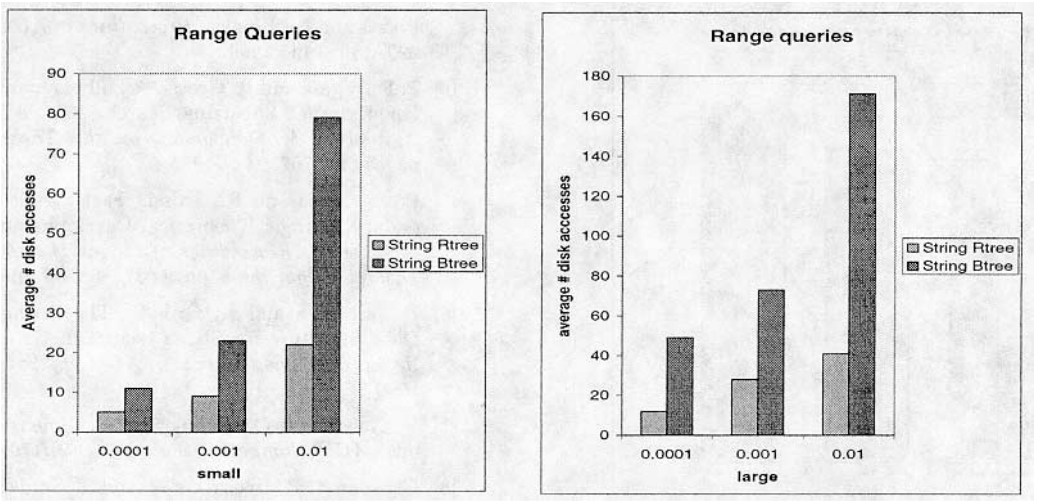
4.5 Space Considerations

One issue that is often overlooked in the performance of multi-dimensional versus 1-dimensional index structures is that of space. For example, the space required by two B-trees to index a 2-dimensional space is smaller than that required by an R-tree to index the same space. Thus, by using an R-tree we are essentially trading space for search time, since an R-tree can complete the search for most common queries much faster than querying two independent B-trees. We wish to carry out a similar evaluation in the case of string B-trees and string R-trees.

In the case of B-trees and R-trees over numerical domains, using 4 bytes per index entry and 4 bytes for a pointer, we expect a fanout of 400 and 1000 for an R-tree and a B-tree, respectively, for an 8KB page. Using a 16KB page, the numbers become 800 for an R-tree and 2000 for a B-tree.

In the case of string B-trees and string R-trees, an efficient implementation of e-tries is crucial, since e-tries are stored in index pages and their size limits the fanout of the index page. As mentioned earlier, in our implementation, we use `short` and `char`, where possible, for a concise encoding of the e-tries.

Let us compute the space requirements of the various entities in a string R-tree page. Assuming n minimum bounding rectangles, we require $16 * n$ bytes for their storage, since each offset is 4 bytes. We also require $4*n$ bytes for the pointer to index pages. For a single e-trie, using the encoding technique described above would require 2*n bytes for the storage of the characters and 4*n bytes for the storage of the node counts. Ideally,



(a) Data Set: D1 (b) Data Set: D2

Figure 9: Performance of Range Queries on String R-trees and String B-trees

only one of the tries would need an additional 4*n bytes for the storage of the trie associations. The complete storage requirements (for the case of a 2-dimensional string R-tree) is $36 * n$ bytes.

String B-trees, as defined in [6] require the index structure to store the start and the end offset (offset of minimum and maximum string in the page below) as opposed to the traditional B-tree approach of storing only the maximum key value in the child page. Carrying out a similar computation for the case of string B-trees, the total space requirements is $18 * n$ bytes.

Using these numbers we can obtain values of the fanout of the index pages for various disk page sizes. For example, with a page size of 8KB, we expect a fanout of 227 for a string R-tree and 455 for a string B-tree (202 and 449 in our "sub-optimal" implementation). Similarly, with a page size of 16KB, we expect a fanout of 455 for a string R-tree and 910 for a string B-tree (404 and 899 in our "sub-optimal" implementation).

The fanout ratio of the R-tree in the numeric domain and the B-tree in the numeric domain is 2.5. This ratio becomes 2 in the string domain. Figure 10 presents the average ratio of the space required by two B-trees and one R-tree for page sizes of 8KB and 16KB, for data sizes ranging from 10^4 to 10^7 elements. There are three categories, one for the ratio in the numeric domain, one for the ratio achieved by our implementation and one for the ideal ratio. It is evident that due to the design of the string B-tree, the space efficiency of R-trees improves when used in the string domain.

5 Related Work

The work presented in this paper is based on the idea of a string B-tree proposed in [6, 5, 4]. In these papers, the authors introduce the notion of an index structure designed for unbounded length strings, and use elided tries in the leaf pages of a B-Tree for this purpose. However, the work described is very specific to a (1-dimensional) B-tree. Our contribution in this paper is to extend such a scheme to apply to a wide variety of index structures in multiple dimensions. Multidimensional indexing has a long history in database research [19, 7]. There are numerous proposals for indexing techniques in multiple dimensions. A very good recent survey can be found in [7]. There exist several structures for indexing strings in one dimension. Prefix B-trees [2] is a classic structure, capable of dynamically handling 1-dimensional variable length strings. String B-trees [6] provide better performance guarantees for variable length strings, than Prefix B-trees however. Suffix arrays [22] and PAT-arrays [8] allow for fast searches on strings but they are difficult to update in secondary storage. Suffix trees [15] are another classical index for strings; they have an unbalanced tree topology, which makes the dynamic maintenance in secondary storage difficult. We are not aware of any dynamic structures for multi-dimensional string indexing in secondary storage.

6 Conclusions

More and more frequently, databases have to index strings rather than numbers. It has been believed tacitly by many that well-understood index structures for numeric data can be adapted in a straightforward manner to deal with string data. In this paper, we have explored several of the pitfalls in this process, and proposed generic techniques that can be used to

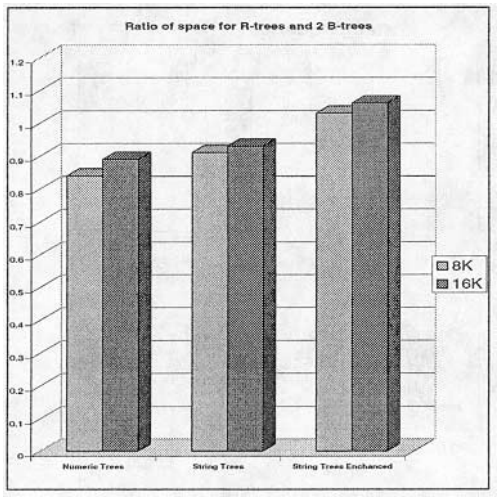


Figure 10: Average Space Ratios

adapt most multi-dimensional index structures to the string indexing problem. The proposed techniques, being generic, are open for various optimizations in specific application contexts. We have also evaluated the benefits experimentally through implementing the string R-tree. Several open problems remain. Foremost among them is effective support for substring match, rather than just the prefix match described in this paper. Of course, there is an obvious technique, of indexing all suffixes, rather than just the original strings, and indeed this is the basis of the well studied suffix-tree. However, we would like to believe that greater efficiency is possible in a multi-dimensional context, and this is the subject of our on-going research.

A second open problem worth mentioning is one of index structure choice. The use of our generic techniques to adapt index structures also changes the costs and performance of the structure. As such, any performance comparisons between multi-dimensional indexes in the numeric domain may not carry over directly to the string domain. Even for a given index structure, different algorithms may now become preferred for various index maintenance operations. Furthermore, we also have the possibility of multi-dimensional indexes with some dimensions being numeric and others string. A careful engineering assessment of the options is called for.

References

[1] L. Arge, P. Ferragina, R. Grossi, and J. S. Vitter. On Sorting Strings In External memory. *Proceedings of STOC, El Paso, Texas*, pages 540–548, June 1997.

[2] R. Bayer and K. Unterauer. Prefix B-trees. *ACM Transactions on Database Systems, 2,1*, pages 11–26, Jan. 1977.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R* - tree: An Efficient and Robust Access Method for Points and Rectangles. *Proceedings of ACM SIGMOD*, pages 220–231, June 1990.

[4] P. Ferragina and R. Grossi. A Fully Dynamic Data Structure For External Substring Search. *Proceedings of the 27th Annual ACM Symposium on the Theory of Computing*, pages 693–702, May 1995.

[5] P. Ferragina and R. Grossi. Fast String Searching In Secondary Storage: Theoretical Developments and Experimental Results. *Proceedings of the ACM SIAM Symposium on Discrete Algorithms*, pages 373–382, Jan. 1996.

[6] P. Ferragina and R. Grossi. The String B-Tree: A New Data Structure for String Search in External Memory and Its Applications. *Journal of ACM 46,2*, pages 237–280, Mar. 1999.

[7] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[8] G. Gonnet, R. Baeza-Yates, and T. Snider. *New Indices for Text: PAT Trees and PAT Arrays.* Information Retrieval: Data Structures and Algorithms, Prentice Hall, 1992.

[9] A. Guttman. R-trees : A Dynamic Index Structure for Spatial Searching. *Proceedings of ACM SIGMOD*, pages 47–57, June 1984.

[10] T. Howes, M. Smith, and G. S. Good. *Understanding and deploying LDAP Directory Services.* Macmillan Technical Publishing, Indianapolis, Indiana, 1999.

[11] H. V. Jagadish, O. Kapitskaia, R. T. Ng, and D. Srivastava. Multi-dimensional substring selectivity estimation. In *Proceedings of the International Conference on Very Large Databases*, Edinburgh, Scotland, UK, Sept. 1999.

[12] H. V. Jagadish, N. Koudas, and D. Srivastava. On effective multi-dimensional indexing for strings. AT&T Labs–Research Technical Report, 2000.

[13] H. V. Jagadish, L. V. S. Lakshmanan, T. Milo, D. Srivastava, and D. Vista. Querying network directories. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, Philadelphia, PA, June 1999.

[14] D. Knuth. *The Art of Computer Programming: Volume 3 Sorting and Searching.* Addison Wesley, Aug. 1998.

[15] E. M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM Vol 23.*, pages 262–272, Dec. 1976.

[16] D. R. Morrison. PATRICIA: Practical Algorithm to Retrieve Information Coded in Alphanumeric. *Journal of ACM, 15,4*, pages 514–534, Oct. 1968.

[17] J. Robinson. The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes. *Proceedings ACM SIGMOD*, pages 10–18, 1981.

[18] N. Roussopoulos and D. Leifker. Direct Spatial Search on Pictorial Databases Using Packed R-Trees. *Proceedings of ACM SIGMOD*, May 1985.

[19] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison Wesley, June 1990.

[20] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+ -tree : A Dynamic Index for Multi-dimensional Data. *Proceedings of VLDB 1987*, pages 507–518, Sept. 1987.

[21] K. C. Sevcik and N. Koudas. Filter Trees for Managing Spatial Data Over a Range of Size Granularities. *Proceedings of VLDB*, pages 16–27, Sept. 1996.

[22] M. U and G. Myers. Suffix Arrays: A New Method For Online String Searches. *SIAM Journal On Computing, 22,5*, pages 935–948, Jan. 1993.