Indexing for String Queries using Generalized Search Trees

Jeff Foster
jfoster@cs.berkeley.edu

Megan Thomas
mct@cs.berkeley.edu

Abstract

We present five index trees designed for supporting string searches. We discuss *Counted Trees*, *Substring Trees*, and *Regular Expression Trees*, all of which suffer the same problem – their attempts at approximating a large set of data lead to an almost complete lack of information in the interior nodes. A variant of B-Trees, called a *Prefix-Suffix Tree*, avoids this difficulty by severely restricting the form of queries. Finally, we show that very high redundancy also avoids this pitfall, using a *Suffix B-Tree*.

1 Introduction

Modern benchmarks such as TPC-D [Tra96] contain many "like" queries, a restricted form of string pattern matching. However, current databases only index for prefix and suffix searches. As more users want to ask substring and possibly even regular expression queries, it is becoming important to index for these as well. The most common implementation of general "like" queries is a simple scan. Although this may be acceptable for small relations, we would like to do better for large databases.

We investigated several indexing strategies for supporting various kinds of string queries. We developed our indices in the GiST framework, a generalized search tree package. GiST is an abstraction of several common indexing techniques, including B-Trees and R-Trees, and provides support for concurrency and transactional semantics.

1.1 Description of GiST

A GiST is a balanced tree whose nodes contain sets of entries, which are pairs (k, p) where k is a key and p a pointer. For interior nodes, p points to a child node, and for leaves p contains a pointer to the corresponding tuple in the database. A key k represents a predicate that is true of any tuples reachable from p. For example, in B-Trees k is an interval meaning "all tuples reachable from p are within this interval." For R-Trees, k is two-dimensional rectangle.

GiST also requires the user to define six methods:

- *Consistent* takes an entry (k_1, p) and a predicate k_2 and determines whether both k_1 and k_2 could simultaneously be true.
- *Union* takes a set of entries P and returns a new predicate that holds for all tuples reachable from P.
- *Penalty* takes two entries e_1 and e_2 and returns some value indicating how expensive it would be to insert e_2 into the node pointed to by e_1.
- *PickSplit* takes a node and divides its entries into two groups, trying to cluster like entries together.

The other two methods are *Compress* and *Decompress*, which we will not discuss. They are used mainly for packing and unpacking keys for storage on disk. Although we will skip them, they are important in practice, because the fanout of a GiST is determined by the disk's page size. Thus, compact disk representations increase fan-out. In our experiments, we assume a page size of 8KB.

For a full description of GiST, see [HNP95].

1.2 Related Work

There is an extensive literature on string searching. A good general introduction is [Aho90]. One of the earliest indexing techniques for fast substring searches is the *suffix tree*, which is a compact representation that encodes every possible suffix of the dataset [McC76]. Krishnan, Vitter, and Iyer have shown how to use a pruned suffix tree to estimate selectivity of substring searches, though not how to do the actual searches [KVI96].

By combining suffix trees with other data structures, Baeza-Yates and Gonnet allow arbitrary regular expression searching in a relatively compact space [BYG89]. Unfortunately, though that work is designed for a very large dataset, the *Oxford English Dictionary*, it does not support transactional semantics. The *OED* does not often change.

All but one of the trees we describe are based on the following idea: Given a set of strings, we want some compact, approximate representation of that set. Thus, we want to find patterns in the strings. Learning patterns from sets of strings is an essential task in the world of genetic databases. See [BJEG95] for a good survey of this area. In this situation, it is often the case that approximate matching is good enough; [WM92] is a description of the `agrep` tool, which allows approximate searches for regular expressions. See [KM95] for a much more theoretical treatment of approximate matching.

Of course, searching for substrings and regular expressions is not all there is to textual databases. [Loe94] provides a fairly general overview of some techniques that treat data as structured text, rather than as byte strings. [GNU94] describes a framework adding string searching and manipulation primitives as part of database queries. That work may be too theoretical to be realistic, but it must be good because the paper uses the word "stringological."

2 The Trees

We have designed five kinds of GiST trees to support string queries. *Counted Trees* treat strings as bags of letters. *Substring Trees* use longest-common substrings as keys. *Regular Expression Trees* use approximate regular expressions as keys. *Prefix-suffix Trees* work like B-Trees but allow simultaneous prefix and suffix queries. Finally, *Suffix B-Trees* are B-Trees with replicated keys to provide fast substring queries.

Throughout the rest of our discussion, we will use the following terminology. Let Σ be a finite alphabet from which the strings are drawn, and let ϵ be the empty string. Define $n = |\Sigma|$, and number the elements of Σ with 0 through $n-1$. If x is a string, x_i is the i^{th} character of x, $0 \leq i < n$. We use this same subscript notation for an element of an n-vector.

For each tree, we first describe the key data structure, then define the *Union*, *Penalty*, *PickSplit*, and *Consistent* methods. We define these in terms of keys rather than entries in order to make things slightly simpler. Whenever we use the word key where GiST requires an entry, we really mean an entry containing that key and some pointer.

3 Counted Trees

The simplest tree we considered is a *Counted Tree*. This tree treats strings as bags of letters, and at interior nodes simply stores the maximum number of times each particular letter appears in any single word below this node. Each key c is a mapping $\Sigma \rightarrow N$, where $c(a) =$ the maximum frequency of a in any sub-key. We represent this information as an vector $c = (c_0, \ldots, c_{n-1})$ of integers. To keep the vectors short, in the implementation we ignore case and punctuation.

The methods are as follows:

- *Union*(c,d) is the pointwise maximum of the two vectors: $(c \cup d)_i = \max(c_i, d_i)$.
- *Penalty*(k_1, k_2) is just $||(k_1 \cup k_2) - k_2||_1$, *i.e.* how much the sum of all the counts stored in e will increase by adding k_2.
- *PickSplit* uses Guttman's quadratic split algorithm, with the penalty metric as stated above [Gut84]. Let $P = \{k_1, k_2, \ldots, k_n\}$ be a set of keys, and let P_1 and P_2 be the output of *PickSplit*. We compute for each pair (k_i, k_j) the distance between the two keys, using the L_1 norm (the Manhattan distance). The keys that are farthest apart are chosen as seeds, and one is added to P_1 and the other to P_2. We then distribute the remaining keys to P_1 and P_2 according to *Penalty*.

This tree supports the sub-multiset query: *Consistent* takes a key k and a string x, and returns true if the frequencies of the letters in x are at most the frequencies in k. By storing the actual strings at the leaves, we can restrict this further to substring queries.

3.1 Performance

Clearly this tree is only a slight generalization of a tree that stores sets at the leaves and uses set union as the *Union* method. Given the theoretical inefficiency of such a workload, as documented in [HKP97], we knew from the beginning that this probably would not work. One potential saving grace might be that n, the size of Σ, is fixed, and thus the argument in [HKP97] about subset queries does not directly apply.

Unfortunately, it still fails to be efficient in much the same cases. The only time an interior node contains any useful information is when a string with an infrequent letter appears below that node. When tested on a file containing words from an English dictionary, nearly all of the keys contain the common letters and few contain the uncommon ones. Searching for "xylo" works very well; searching for "ate" requires scanning the entire index structure.

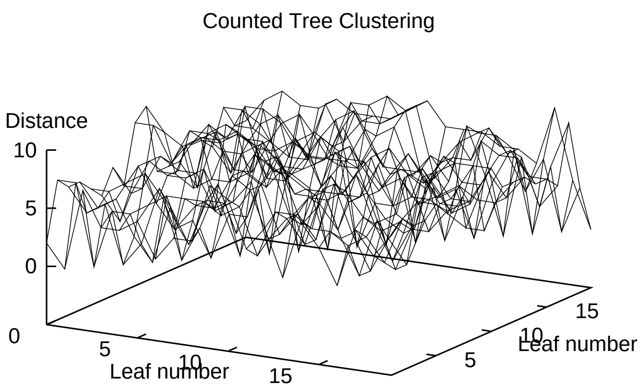

Figure 1: Clustering in a counted tree.

We had hoped that, while the Counted Tree itself would probably not be useful for searching, it would produce interesting clustering at the leaf nodes. Data clustered by this tree could be bulk-loaded into a different type of tree that might be good at searching but poor at clustering. Unfortunately, the Counted Tree had little success at clustering interesting datasets. Figure 1 shows a 3-dimensional plot of the distances, according to the *Penalty* metric, between keys stored on the same page. This dataset was a file of English words. If the clustering had worked well, then distances on this page should be relatively small. As the graph indicates, they vary between 0 and 10. Since the maximum word length on this page is only 11 and the average length of the whole dataset is 7, having keys on the same page be 10 apart is not very good.

We believe this happens because, after accumulating tens of strings in a node, the key for that node contains a vector that is near (in the L_1 sense) almost any word. All of the common letters are present, as are scattered uncommon ones. Therefore virtually any word can be added to the node without changing the key very much. Although this word may be close to the *key*, it is not necessarily close to specific words in the node.

4 Substring Trees

Among the most common string queries are substring searches. Therefore, it seems natural to try to index for these by storing substrings as keys. In a *Substring Tree*, each key is the longest common substring (LCS) of the sub-keys.

One possible definition of $LCS(x, y)$ would be a string z such that $x = \alpha z \beta$ and $y = \gamma z \delta$, where Greek letters stand for arbitrary strings. However, that definition seems overly restrictive. Instead, we use the longest common *subsequence* definition from [CLR90]. Rather than restating this formally, we shall just say that $LCS(x, y)$ is z where all the letters of z appear in sequence in both x and y, except that an arbitrary number of other letters are allowed in between. For example, $LCS(joe, jeff) = je$.

This improved definition is still impractical because if the two strings are not exactly identical, then we lose all information – anything could appear in the gaps. Accordingly, we record the maximum distance between any two sequential letters of the LCS, plus distances from the beginnings and endings of words. Thus, in our system, $LCS(joe, jeff) = 0, j, 1, e, 2$, meaning that j always appears first, there is at most one letter between between j and e, and at most two letters to the right of e. Our algorithm for computing this information is a straightforward modification of the one given in [CLR90].

We write an LCS as a pair (x, d), where x contains the letters of the longest common subsequence and d contains the distances between the letters, *e.g.* $LCS(joe, jeff) = (je, (0, 1, 2))$. The GiST methods for this tree are

- *Union*$(k_1, k_2) = LCS(k_1, k_2)$, as just described.
- *Penalty*(k_1, k_2). Let $k = (x, d) = LCS(k_1, k_2)$. Penalty returns $|x| \times 2^l$, where $l = \sum_{i=0}^{|d|} 2^{d_i}$. The idea here is that longer strings are better, but that a larger gap between letters is exponentially bad, since any letter could fill in the gap, or there might be no letter in the gap.
- *PickSplit* again uses the quadratic split algorithm as we described before. This time the distance between two keys k_1 and k_2 is measured as *Penalty*$(k_1 \cup k_2)$ − *Penalty*(k_1), the increase in the penalty value when k_2 is added to k_1.

The query that this tree supports is substring search. *Consistent* takes a key k and a string x, and returns true if x could appear as a substring of any of the strings k represents. At the leaves, this is the usual substring search, but we need to take the gaps into account in the interior nodes. Fortunately, a minor, and slightly more efficient, modification of the LCS algorithm can handle this.

4.1 Performance

Unfortunately, although the *Consistent* method is efficient, the keys it operates on contain little information. In a moderately varied dataset, the probability of having a long LCS for an arbitrary pair of keys is low. Because each key is the LCS of many other keys, there is little chance that the LCS will, in fact, contain any string.

Figure 2 shows a series of histograms capturing the trend towards less and less information in LCSs as the number of sub-keys increases. Each diagram shows how many LCSs of n distinct strings have a given length. When n is 1, the LCS is simply the string length. The data is 102 random words from an English dictionary. As can clearly be seen, the fraction of LCSs containing a non-empty string is negligible once more than a few words are unioned. When taking the LCS of an 8KB page, which may contain hundreds of words, there is little clustering any penalty metric can do. In general interior nodes contain no information.

This is not to say that there is *no* clustering in this tree, but simply that a vast majority of the nodes have none. We loaded a 38470-word spelling dictionary into the suffix tree and looked at the root node. Out of 20 entries, only four contained entries longer than three letters: *superc*, *ion*, *lies*, and *consus*. For each,

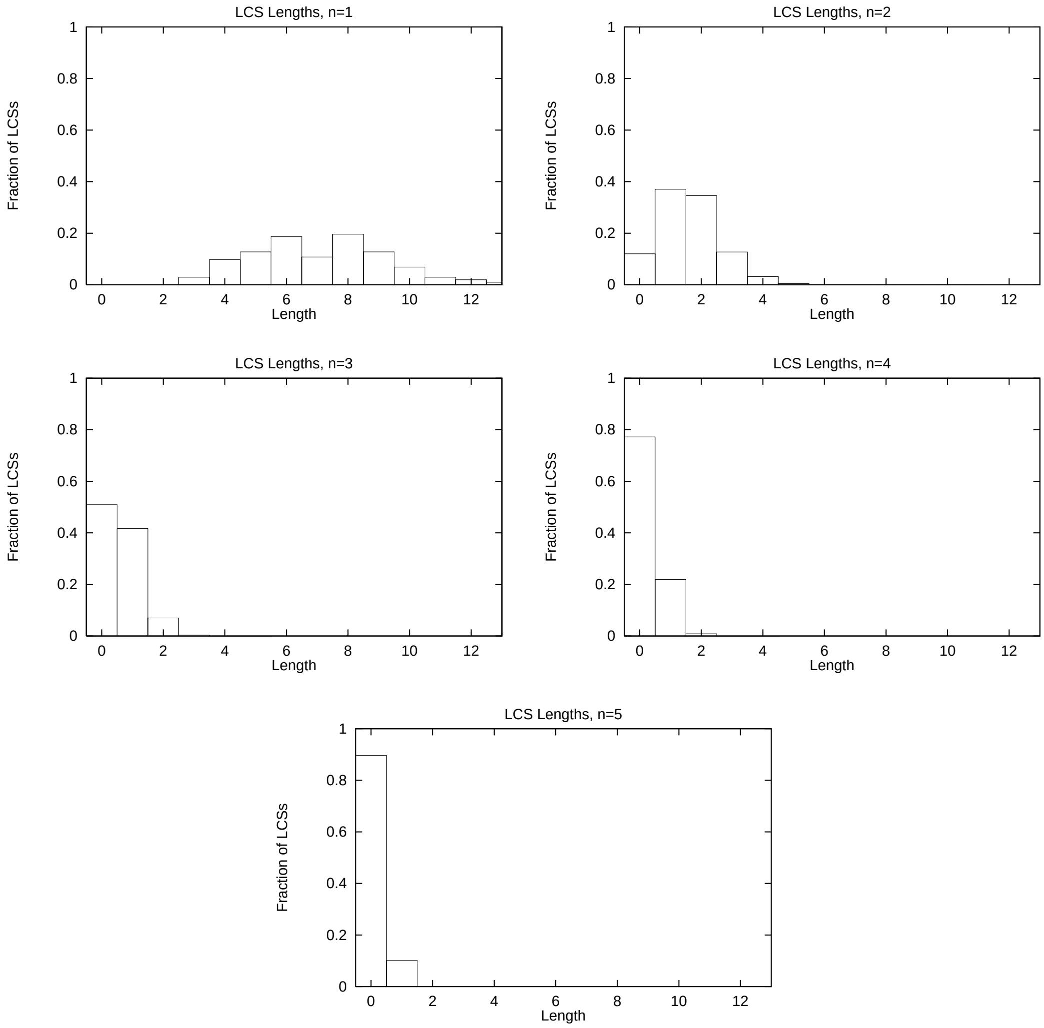

Figure 2: Histograms illustrating how the length of the LCS of n strings decreases as n increases. The data is 102 random words from an English dictionary.

the largest gap between letters was, respectively, nine, five, seven, and six. Thus, for substring searches of five or fewer letters, there would be no pruning of the search at the root note. Considering that the average word length in this file is eight letters, having substring searches of fewer than five letters descend every path from the root is unacceptable. The results were even worse when we tried a data file from the TPC-D benchmark [Tra96] with longer strings – the average length in this dataset was 19 letters. In this case, loading the data yielded *no* substrings longer than one letter at the root.

5 Regular Expression Trees

The most general string searches commonly available use regular expressions as patterns. The properties of regular expressions and their correspondence with deterministic finite automata (DFAs) are well known [HU79]. Within the GiST framework, using DFAs as keys struck us as being a natural technique for indexing both general regular expression and substring queries. A *Regular Expression Tree* is a tree whose keys contain DFAs. The DFA at a key accepts a superset of all the languages accepted by its subkeys. To check whether a key matches a regular expression query, we intersect the DFA for the query with the DFA at the key and test whether the result is non-empty.

The union of two keys M_1 and M_2 is a new DFA M such that $L(M) \supseteq L(M_1) \cup L(M_2)$, where $L(M)$ is the language accepted by DFA M. The easiest way to generate such a DFA is to use the standard union algorithm [HU79] and then minimize the result. If we apply only this algorithm the $\supseteq$ becomes an equality. Unfortunately, recall that the fanout of GiST trees is determined by the number of keys that can fit on a disk page. Even compactly represented DFAs tend to be bulky. Thus, if we only use the standard algorithms for minimization, it may not be possible to fit a DFA within a page, which defeats the purpose of having an index. Additionally, the union and minimization algorithms are, respectively, $O(n^2)$ and $O(n^3)$ in the number of states. While CPU time is not normally important, if intermediate DFAs are allowed to get large this time can become quite significant.

In order to produce small DFAs, we resort to approximation. Suppose that DFA M takes up too much space. Then create a DFA with one fewer state by picking a pair of states of M and collapsing them. Repeat until the DFA is small enough. If this still is insufficient, we can also make the following approximation: Suppose some state s has two transitions to state t, one when the input symbol is a and one when the input symbol is c. Rather than store two separate transitions, we can store one transition with label $[a-c]$, meaning take the transition on symbols a, b, and c.

In order to pick the states to collapse, we use the following simple-minded heuristic: We pick a state q at random, and then we pick a state r such that q has a transition to r. If there is no such state, we collapse q with a random state. This is clearly not optimal. Even so, it shows absolutely no signs of working, even on a relatively small set of words.

Basically, this tree suffers from the same difficulties as Counted Tree and the Substring Tree. In order to have a small, accurate regular expression accepting a large set of strings, the strings have to have many commonalities. For a database like a set of words from the English language, that does not happen. Figure 3 shows three snapshots of the union method taken after four, 64, and 128 strings were unioned and then minimized. These strings were the first 128 from a list of English words in alphabetical order. The topmost DFA has perfect information, but by the time only 64 strings have been thrown in, the union method started making large approximations. By the time we get to 128 strings, the DFA contains no information.

6 Prefix-Suffix Trees

A B-Tree [Com79] can efficiently support prefix queries on strings. The keys of the B-Tree are ranges of strings, and a prefix x matches a range $[y,\ z]$ if either x is lexically ordered between y

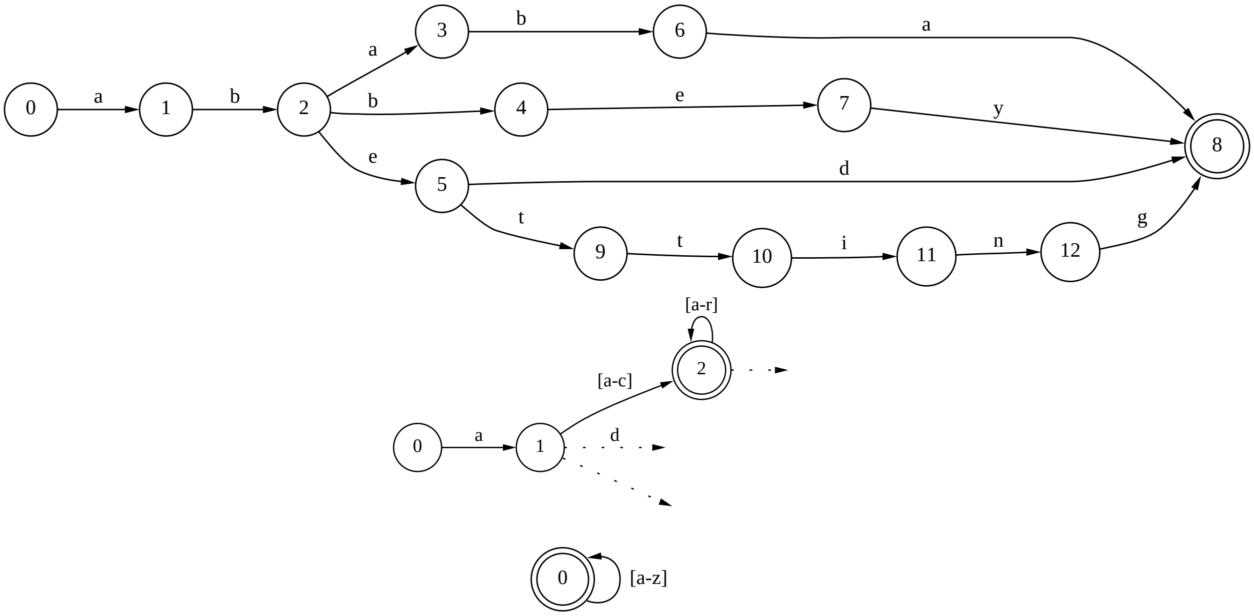

Figure 3: A progression of three finite automata created by the union method. The topmost DFA is the union of four strings, the middle the union of 64 strings, and the bottom the union of 128 strings.

and z or if x is a prefix of y. By reversing strings before inserting them, B-Trees can handle suffix queries in the same fashion.

We propose a new kind of tree, a *Prefix-Suffix Tree*, which can handle simultaneous prefix and suffix queries. Rather than storing plain strings or reversed strings, we store *folded* strings. Let x be a string of length m. Define *fold(x)* as a string of length m such that

$$\begin{aligned} \mathit{fold}(x)_{2i} &= x_i \\ \mathit{fold}(x)_{2i+1} &= x_{m-i-1} \quad \text{if } 2i+1 < m \end{aligned}$$

for $0 \leq i < \lfloor n/2 \rfloor$. For example, $\mathit{fold}(megan) = mneag$.

Aside from this modification, the tree works just like a B-Tree:

- *Union* takes two intervals and returns the smallest interval containing both.
- $\mathit{Penalty}(k_1, k_2)$ returns how much the interval k_1 will expand if k_2 is added.
- $\mathit{PickSplit}(P)$ divides P into P_1 and P_2, where the first half of P goes in P_1 and the second half goes in P_2.

A more thorough description of an implementation of B-Trees with GiST appears in [HNP95].

The *Consistent* method for this tree takes a key k and a query of the form $x(\Sigma^*)y$. Strings x and y are folded together, and *Consistent* returns true if the folded string is a prefix of k. The only tricky aspect is that x and y may be different lengths, in which case the folded string must be truncated. For example, the query $t(\Sigma^*)as$ is folded into ts. There is no point to representing it as $ts\Sigma a$, since in a prefix search this matches anything ts matches. Only when comparing a query to a leaf node do we necessarily do a full comparison on x and y.

6.1 Performance

In contrast with the last three ideas we mentioned, this tree does work well for many queries. In general, as long as x and y are long enough, the tree will be able to eliminate interior nodes from consideration. Specifically, the length of the *fold* of a query $x(\Sigma*)y$ is $2 \times min(|x|, |y|) + (|x| \ mod \ 2)$. The last term takes into account that the folding begins with the prefix. If this quantity is too small, *e.g.* if $x = \epsilon$, then, even if y is long, the folded query string will match any interior node.

However, if all the queries were to omit the prefix, then presumably a B-Tree with reverse strings would be the right index to use. If we restrict the queries to have $|x| = |y|$, then the new tree is optimal. The moral here is that by sacrificing the generality of full substring search, we were able to achieve good performance. Further, while this tree cannot handle full regular expression queries, it is capable of answering queries a normal B-Tree cannot.

7 Suffix B-Trees

Our last tree is what we term a *Suffix B-Tree*. As we stated previously, B-Trees can efficiently support prefix queries. In a Suffix B-Tree, we take advantage of this efficiency by turning all substring searches into prefix searches.

Rather than inserting each string once into a B-Tree, we insert it many times. For every string x in the original data file, we insert x and all suffixes of x, keeping the same pointer for each insertion. For example, if the original key was foo with pointer 42, then we insert the entries $(foo, 42)$, $(oo, 42)$, and $(o, 42)$. Aside from that, we make no changes to the regular B-Tree implementation.

After doing this, substring searches are easy. A substring s appears in x iff s is a prefix of some suffix of x. Thus, to search for a substring s, we do a prefix search. Since prefix searches in B-Trees are extremely efficient, now so are substring searches.

The reason this tree works where our first three trees failed is redundancy. If the original dataset contained m strings with an average length of l, then the Suffix B-Tree will contain lm leaf-level entries. The height of the tree will also increase by a corresponding amount. In general this is unacceptable for large values of l, though for a small, fixed l it works wonderfully.

8 Future Work

In [HKP97], it has been shown that set inclusion workloads have potentially the worst possible access overhead, while B-Trees have the best. We believe that the substring queries we have looked at are an important intermediate point on this spectrum, and we plan to look for some theoretical, rather than statistical, justification for why low redundancy does not work unless the form of queries is severely restricted. Additionally, there should be some middle ground. It should be possible to restrict the form of queries a little and add a little redundancy to get, from an engineering perspective, the optimal balance between the expressiveness and redundancy.

There is a large body of string data for which $|\Sigma|$ is very small: genetic data. Although in theory a small finite number is no different from a large finite number, in practice there is quite a difference. Perhaps by restricting $|\Sigma|$ to be small we could build better indices.

Finally, it may just be that GiST is not the right framework for building string indexing. As we stated before, there is a significant body of related work, though none of the solutions we have seen support transactional semantics. Another approach would be to start from one of those solutions and try to add concurrency and locking.

9 Conclusions

We have described five kinds of index trees that were designed for string searches. The first three trees we discussed – Counted Trees, Substring Trees, and Regular Expression Trees – all fell into the same trap. Counted Trees only work for queries containing infrequent letters and fail completely at clustering. Substring Trees and Regular Expression Trees both contain virtually no information in the interior nodes.

The only trees that work well are the last two we described: Prefix-Suffix Trees and Suffix B-Trees. Prefix-Suffix Trees succeed by severely restricting the form of queries. Suffix B-Trees avoid the trap through very high redundancy.

These last trees sidestep a fundamental problem with the others: small approximations of large sets of strings contain little information.

Acknowledgments

Many thanks to Joe Hellerstein and Steve Fink for listening to our crazy ideas and making good suggestions.

References

[Aho90] A. V. Aho. Algorithms for Finding Patterns in Strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, pages 257–300. Elsevier Science Publishers, 1990.

[BJEG95] A. Brāzma, I. Jonassen, I. Eidhammer, and D. Gilbert. Approaches to the automatic discovery of patterns in biosequences. Technical Report Report No 113, Department of Informatics, University of Bergen, Bergen, Norway, December 1995.

[BYG89] R. Baeza-Yates and G. Gonnet. Fast Text Searching for Regular Expressions or Automaton Searching on Tries. In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *ICALP '89*, volume 372 of *Lecture Notes in Computer Science*, pages 46–62. Springer-Verag, Stresa, Italy, July 1989.

[CLR90] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. McGraw Hill, 1990.

[Com79] D. Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, June 1979.

[GNU94] G. Grahne, M. Nykänen, and E. Ukkonen. Reasoning about Strings in Databases. In *Proceedings of the 13th ACM Symposium on Principles of Database Systems*, pages 303–314, Minneapolis, MN, 1994.

[Gut84] A. Gutman. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proc. 1984 ACM-SIGMOD Conference on Management of Data*, Boston, MA, 1984.

[HKP97] J. Hellerstein, E. Koutsoupias, and C. Papadimitriou. On the Analysis of Indexing Schemes. In *PODS '97*, 1997. To appear.

[HNP95] J. Hellerstein, J. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. In *Proceedings fo the 21st VLDB Conference*, Zurich, Switzerland, 1995.

[HU79] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 1979.

[KM95] J. Knight and E. Myers. Approximate Regular Expression Pattern Matching with Concave Gap Penalties. *Algorithmica*, 14(1):85–121, July 1995.

[KVI96] P. Krishnan, J. Vitter, and B. Iyer. Estimating Alphanumeric Selectivity in the Presence of Wildcards. In *Proceedings of the 1996 ACM SIGMOD Internaional Conference on Management of Data (SIGMOD '96)*, pages 282–293, Montreal, Canada, May 1996.

[Loe94] A. Loeffen. Text Databases: A Survey of Text Models and Systems. *ACM SIGMOD Record*, 23(1), March 1994.

[McC76] E. M. McCreight. A Space-Economical Suffix Tree Construction. *J. of the ACM 23*, pages 262–272, 1976.

[Tra96] Transaction Processing Performance Council, San Jose, CA. *TPC Benchmark D Standard Specification*, 1996. Revision 1.2.2.

[WM92] S. Wu and U. Manber. Fast Text Searching Allowing Errors. *Communications of the ACM*, 35(10):83–91, October 1992.