Database Structures, Based on Tries, for Text, Spatial, and General Data

T. H. Merrett, Heping Shang[1] and Xiaoyan Zhao

School of Computer Science, McGill University
Montréal, Qué. H3A 2A7
{tim xiaoyan}@cs.mcgill.ca heping@sybase.com

Abstract

Digital trees, or *tries*, were introduced thirty years ago for sublinear-time retrieval of substrings from large texts. They were exploited for this, as a well-known example, by the University of Waterloo project to put the New Oxford English Dictionary onto CD-ROM. We have recently improved the performance of trie techniques for text and shown their use in searches for approximations to a given string.

We have also shown that tries have excellent retrieval properties for spatial data. We have shown how to use tries to represent, without redundancy, spatial data which can be displayed to any resolution, retrieving from disk or from network only the amount of data that will finally be displayed. We have done this particularly for two-dimensional vector data, such as makes up very large maps, but have also established that the trie techniques apply to raster data and to data of other than two dimensions.

These results are the basis for a claim that tries offer the best storage representations for large-scale multimedia databases. We are presently pursuing this claim by developing trie techniques for general data and queries. We give some results for multikey data.
Key words and phrases: file structures, multimedia databases, spatial data, text data.

1 Introduction

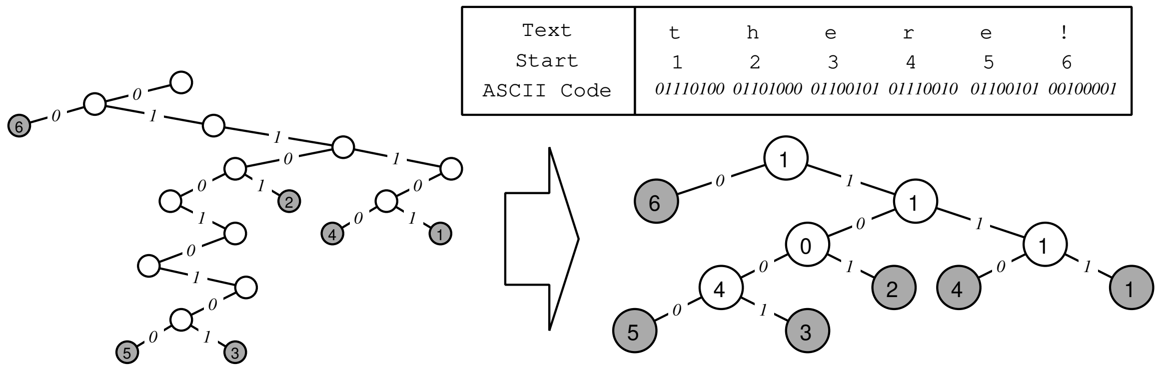
Digital trees, or *tries* [2, 3] are tree data structures which store data along the paths in the tree, not at the nodes, which is what the usual tree structure does. They thus require minimal storage — only a couple of bits — for each node, and they exploit the overlap of paths near the root to achieve considerable compression. Such compression is important in building an index to every substring in a large text, which will at least need a pointer to every character in the text [4]. Tries were first proposed for text indexing in 1968 [11].

In addition to the data compression that results from effective absence of nodes and from the sharing of storage space near the root, tries have a further useful property. Because they usually store the most significant digits or characters near the root, they can be used for *variable-resolution* queries. This was exploited [9] to display spatial data to any resolution using only one copy of the data and transferring from secondary storage only the amount of data needed for each display. (The only alternatives before this were to read the whole file and filter it to the resolution needed, or to keep multiple copies of the data, one at each potentially needed resolution.)

The above applications of tries highlight two properties which we should exploit: their ability to compress data for better storage and faster transfer; and their capability for variable resolution to speed up queries by starting with approximation and refining only where there is uncertainty. This paper reviews recent work done in this direction by the Aldat project at McGill University.

The next section discusses improvements in trie techniques for text searching and a recent algorithm for using tries for approximate searches using a distance measure. Section 3 discusses "zoom tries" for map applications. Section 4 makes a start at using the compression and approximation abilities of tries for queries on general data: we show that trie methods are superior to other known methods for high-activity queries on multikey data. Finally, in section 5, we discuss future steps which we or others can take.

[1] now at SYBASE, Inc., 6601-3 Bay Street, Emeryville, CA 94608



Text	t	h	e	r	e	!
Start	1	2	3	4	5	6
ASCII Code	01110100	01101000	01100101	01110010	01100101	00100001

Figure 1: Trie and PaTrie

2 Tries for Text Data

2.1 Tries

A *trie* is a tree structure whose internal nodes are empty. The decision which way to branch from an internal node at depth d is made according to the value of the dth position in the seach string. For *binary* tries, which we shall exclusively consider, this is the dth bit, and the trie node offers two options: branch left if `0` else right.

We will use three types of trie in this paper. The *full trie* records every right or left decision in an edge, and consequently may have many nodes with only one outgoing edge. If all bitstrings are of the same length, all leaves are at the same level. In the *ordinary trie*, which we shall call, simply, a *trie* when the context is clear, all leaves are moved up to the node below the closest bifurcation. Thus, there are no nodes with only one outgoing edge between a leaf and a node with two outgoing edges. Finally, the *Patricia trie*, or *PaTrie*, has *no* nodes with only one outgoing edge.

Ordinary tries and Patricia tries require extra information to be stored at the leaves, to give the suffixes that have been cut off, or to point to the main data structure. Patricia tries also require "skip" information to be stored at internal nodes, to tell the search how many bits to skip before making the next comparison.

Full tries are good for fixed-length data, such as spatial coordinates. For text, the data are unique semi-infinite strings, or *sistrings* [5], which include all the text between a given start point and the end. As an example, the text `there!` has six sistrings: `there!, here!, ere!, re!, e!,` and `!`. Trie and PaTrie for `there!` are shown in figure 1. We have shown [8, 14] that Patricia tries are better. Indeed, the name is from Morrison's proposal [11] for text representation: Practical Algorithm To Retrieve Information Coded In Alphanumeric.

Tries can be represented using one or two bits per node, depending on the type of trie, plus the additional information, such as suffixes or pointers, and skips. Orenstein [13] developed a two-bit method for ordinary tries, which was the basis for a one-bit representation for Patricia tries [8]. Briefly, Orenstein's method assigns a bit for each of the two possible outgoing edges: `0` if absent, `1` if present. The trie is represented as a string of these bit pairs, and the position of each descendent node in the bitstring can easily be calculated, given the information about which preceeding nodes are present or absent. The bitstring is mapped onto pages of secondary storage in such a way that each level of pages corresponds to a predetermined number of levels of trie nodes.

So far, the search processing of this data structure must be sequential. Orenstein added two counters to each page. *Tcount* counts the number of trie links entering the level from above, up to but not including p. *Bcount* does the same for links leaving the bottom of the level.

In the example of Figure 2, the address (page) of the right descendent of the node marked "X" can be computed as follows. Bcount for "X" (i.e., the page "X" is on) is 2, which means that two links have already descended from earlier pages (to the left) on that level. The right

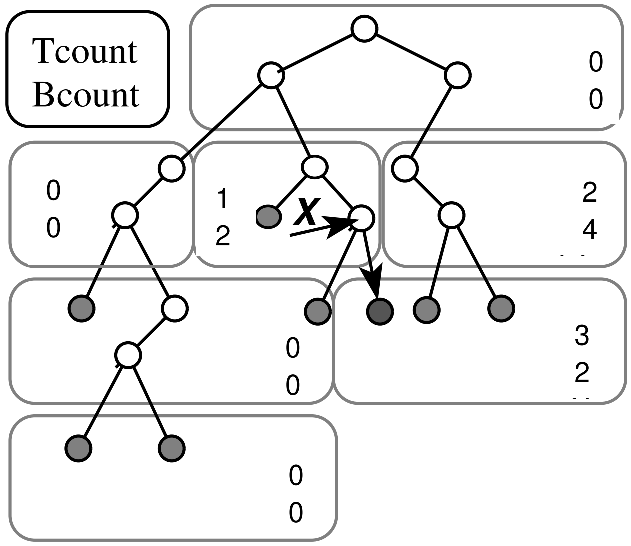


Figure 2: Paged OrTrie

descendent of "X" is the second link in its page. So in the next level, we must look for a Tcount which does not exceed 2+2=4. Candidates are 0 and 3, and 3 is closer, so we know that the descendent of "X" that we are looking for is on the second page of the next level.

Here we are still doing a sequential search on every level, so we must move the Tcounts of each level up into lists of Tcounts in the pages above. Then we can calculate directly which page to go to in the level below. We do not show this here.

2.2 Indexing Text

The major problem facing the implementor of a data structure to access every substring of a large text is the size of the index. At the very least, we require a pointer to every character in the text. A text of N bytes has N sistrings, and each pointer must be $\lg N$ bits, for a total index size of $\frac{1}{8}N\lg N$ bytes. This is just the index used by Gonnet for the New OED — their PAT array [5] — and it would seem to be as small as we can get.

For simplicity, we will suppose that we are considering texts the size of the New OED, namely about 2^{27} sistrings. We see that we already require $3.4N$ bytes for the index, potentially over three times the size of the text.

Using a Patrie, we [8] were able to improve this by 10 to 25%, bringing the index size for 2^{27} sistrings down to $2.7N$ to $3.1N$, depending on the statistics of the letters in the text. (These numbers depend on measured distributions in a number of representative texts.) The major improvement over the PAT array is that the trie method can shorten the pointer by, say, 12 bits, by having it point to a *page* of 4Kbytes (2^{12}) of text instead of to individual characters. The PAT array, on the other hand, must use a binary search, consulting the text data at every comparison, because it has no built in tree structure, and so must point to each character.

We were also able to speed up the expected access time by as much as tenfold, while keeping the worst case better than the PAT array. (We proposed an improvement to the PAT array, which still does not help it beat the PaTrie.)

Building these index structures is costly but feasible. Both PaTries and PAT arrays can be built over a weekend for a text of 2^{27} sistrings.

Both PaTries and PAT arrays support fancier searches than exact string match. They can match regular expressions by simulating the corresponding finite automaton on the trie. They can do proximity searching (e.g., finding adjacent occurrences of `t` and `h`). They can find longest repetitions, most frequent strings of a given length, or most common words of varying lengths.

2.3 Approximate String Match

Approximate string matching reports a match where the search string differs from the text by not more than k changes. A change can be a replacement (or substitution), an insertion, or a deletion. It can also be a transposition. (This solves different problems from regular expression matching, although the pattern, "`a#a##`", where `#` is a one-place wildcard, can be treated as either a regular expression or a 3-approximate match.)

The "edit distance", $D_{i,j}$, between a pattern of length i and a target string of length j, can be defined recursively in terms of

- $D_{i,j-1}$, because one character was added to the target;
- $D_{i-1,j}$, because one character was added to the pattern;
- $D_{i-1,j-1}$, because a different character was added to each; and
- $D_{i-2,j-2}$, because two characters were added in swapped order.

A brute force calculation of $D_{i,j}$ will thus be exponential in cost, but since there are only $(i+1) \times (j+1)$ possibilities, a table can be kept and dynamic programming used [16].

The columns of this table correspond to the characters of the target strings in the database. Since the database contains many target strings, there would normally be many tables. However, columns can be shared among tables where target strings have shared prefixes. Identifying shared prefixes is precisely what the trie representation gives us, and so we can drastically shorten the dynamic programming calculation.

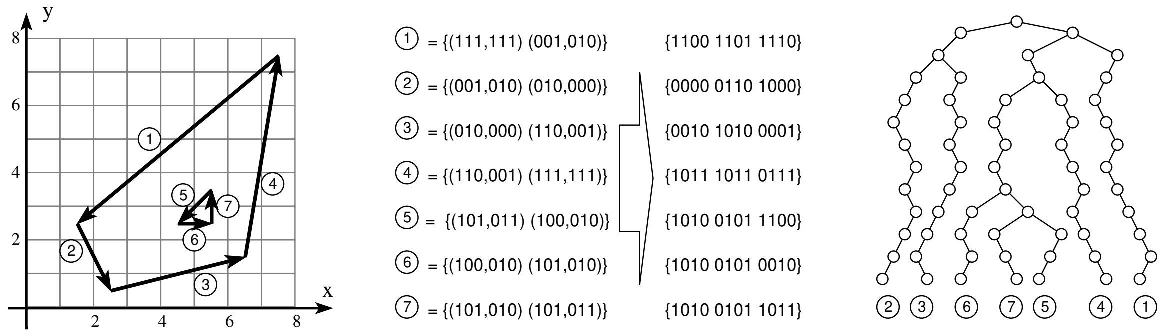
We take advantage of other prior improvements to the k-approximate match solution [15] to achieve a cost of evaluating $\mathcal{O}(k2^k)$ table entries. We made comparisons with *agrep* [17], which has linear complexity in k. The following table shows the factors by which tries beat *agrep* for five different texts of 4.5MB to 8.4MB. Thus, for exact matches, tries are an order of magnitude faster than *agrep*: 7.3 to 28.4. For the practically important case of single errors ($k = 1$), tries are still substantially faster. Only for two or more errors do tries begin to lose out. (The best case for *agrep* and worst for tries was a collection of file names, which are all very distinct from each other and force the trie to be very shallow.)

	k=0	k=1	k=2
min	7.3	1.6	0.2
max	28.4	9.5	0.7

The trie techniques for approximate searches can be extended in a number of directions. For example, *best match* is important for spelling checkers, and can be done by presetting k to some small value, and decreasing it monotonically each time the search succeeds. Different types of error can be given different weights. Approximate search can be combined with exact search when the user knows some characters for sure (e.g., telephone numbers or postal codes). Some regular expressions — alternative operator and Kleene star for single characters — can be extended to approximate search, although full regular expressions with approximation pose an open problem. The foregoing are also handled by *agrep*, but we have added something new. We can adapt the Kleene star to counters, so that a specified range of occurrences can be matched.

3 Tries for Spatial Data

For spatial data, we will use full tries. This particular discussion will be on two-dimensional vector data. Figure 3 shows how an edge, such as edge 1, is represented by four binary coordinates `((111,111), (001,010))`; how these are interleaved by taking the first bit from each coordinate in turn, then the second bit, then the third; and the resulting trie. This is called a *kd-trie*, after Bentley's kd-tree [1], because it uses the same principle of interleaving



(a) A Map (b) Bit Interleaving (c) *ZoomTrie*

Figure 3: Vector 4d-Trie

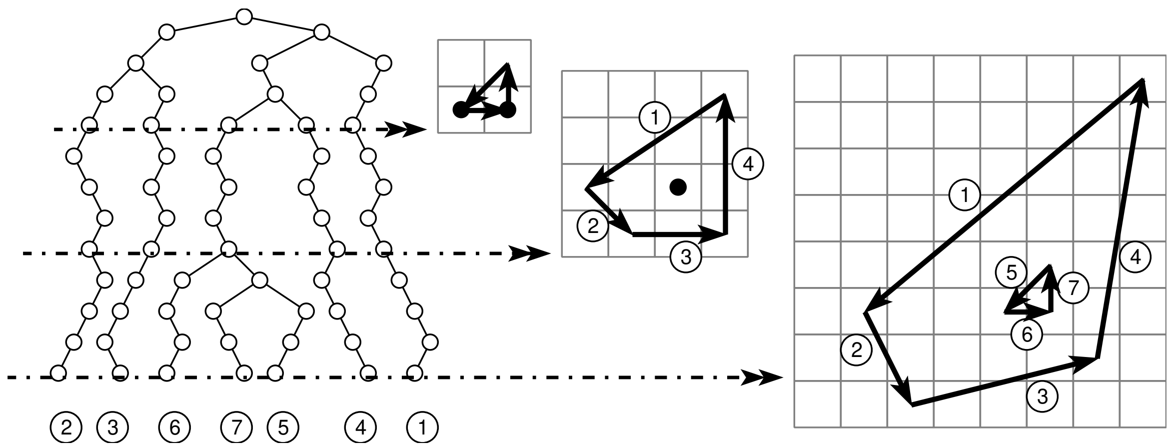


Figure 4: The Zoom Trie

coordinates. The four coordinates giving an edge in two dimensions are treated as though they were a point in four dimensions.

The same representation technique can be used for raster data (a 2d-trie in two dimensions, a 3d-trie in three, and so on) or for simplex data (a 6d-trie for triangles in two dimensions, a 12d-trie for 3-simplexes in three, etc.). Thus, the example we work here is very much a special case.

Tries give a variable-resolution, or *zoom*, capability for free [9, 14]. Because the trie stores the most significant bits near the root, we can look only part way down it to get an approximation to the vectors. Figure 4 shows the approximations resulting from the most significant bit of each coordinate (the first four levels of the trie), and from the most significant two bits of each coordinate (the first eight levels of the trie).

Figure 5 shows the expected result: that the cost of displaying a map stored with a zoomtrie is independent of the map size and linear in the amount of data displayed. (The map used was 31H1 in the 1:50,000 series from Energy, Mines and Resources Canada, with all coordinates 16 bit integers. We quote results for the contour map, which has 483,063 edges.)

The algorithms *Search* and *Scan* in figure 5 need brief explanation. $Scan(n, r)$ collects all paths from node n down $4r$ bits, and is used, with n set to the root node, to display the whole map to resolution r. $Search(n, r, \mathcal{P})$ finds all maximal subtries, working down from node n, that satisfy predicate $\mathcal{P}$, and invokes *Scan* to collect all paths of these subtries. *Search* is used, for example, to select a window in a map and display the contents to a desired resolution.

We built *priority* codes into the map, encoded with two bits at the root of the trie: the four levels permit us to select contours every 100 feet (priority 0), every 50 feet (priority 1), every 30 or 50 feet (priority 2), and every 10 feet (priority 3). It does not make sense to display a

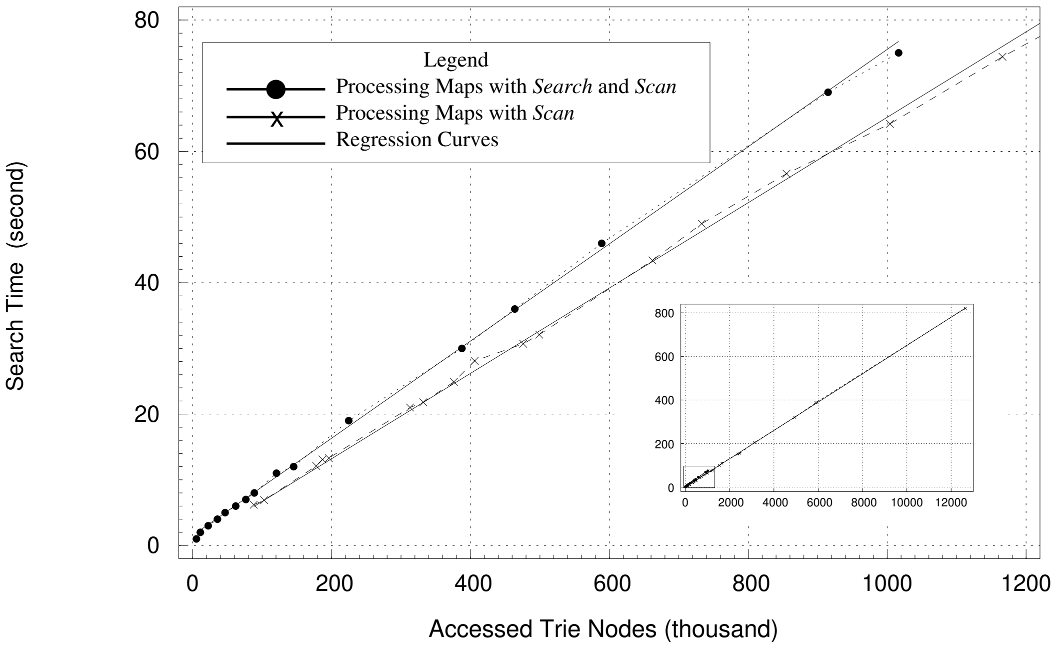


Figure 5: Search Time *vs.* Nodes Accessed

very small, low-resolution map of every single contour. In a highly contoured terrain, such as 31H1 shows, the result would be all black.

We then compared trie performance with a simple filter implementation, which reads all the data and displays only what can actually be seen on the screen. This filter used highly compressed data, in which only *differences* between coordinates and their predecessors in a contour are stored. (The compression is by a factor of about ten over the data supplied to us.) The filter moves down the sequence of differences until the displacement spans at least one pixel, then it draws the resulting edge.

The filter compression is about twice as good as the compression provided by the zoom trie, which gives the filter a strong advantage. For drawing the whole map at full resolution, the filter required 112 seconds on our machine, and was much faster than the trie at the same application. However, the trie drew figure 6 in 1.3 seconds, and figure 7 in 22 seconds. The zoomtrie was better than the filter for all practical displays, namely those of a megapixel or less.

The zoom trie offers impressive savings. For a really large map, these savings would be phenomenal. For instance, all 13,000 of the 1:50,000 series for Canada would require 40GB. But displaying all of it, or any selected part of it, to resolutions such as those of figures 6 and 7, would cost exactly the same as it did to draw those figures. The filter technique would need six hours on our machine.

The only alternative to filters or tries is to store multiple copies of the data, at predetermined resolutions. This has a clear cost in extra storage, and the flexibility is determined solely by the number of extra copies one is willing to store.

Zoom tries also permit similar savings for data transmission across a network.

4 Tries for General Data

We have shown the advantages of tries for special data, such as text, for which trie compression makes possible indexes which would otherwise be impractical, and such as spatial data, for which the variable resolution capability of tries gives tremendous improvements in the ablity

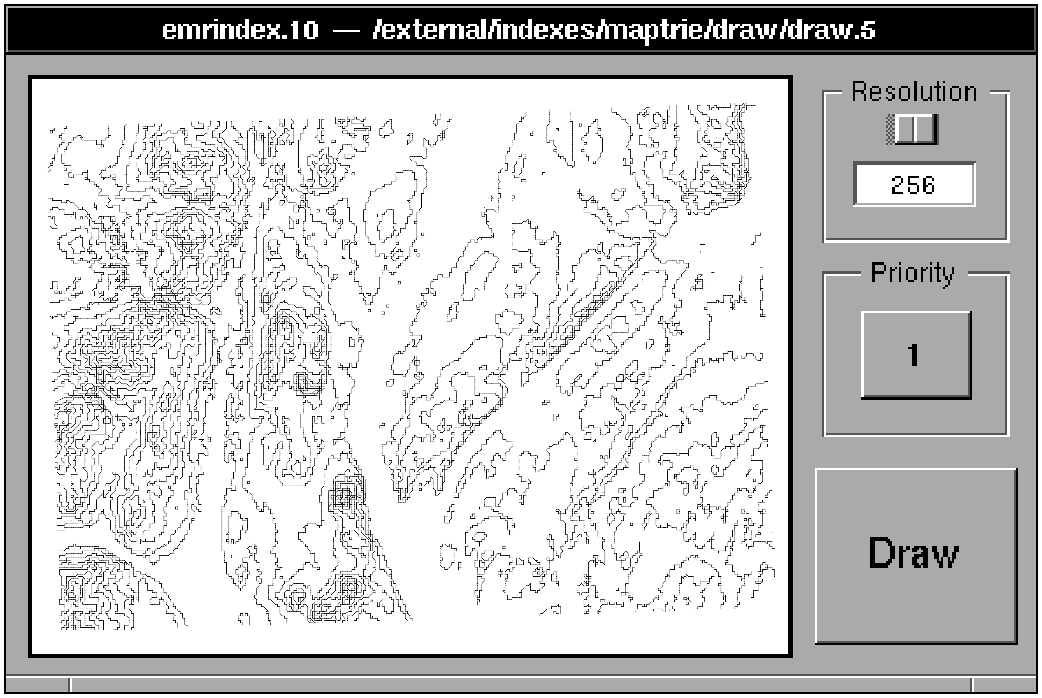


Figure 6: Full Contour Map at Low Resolution

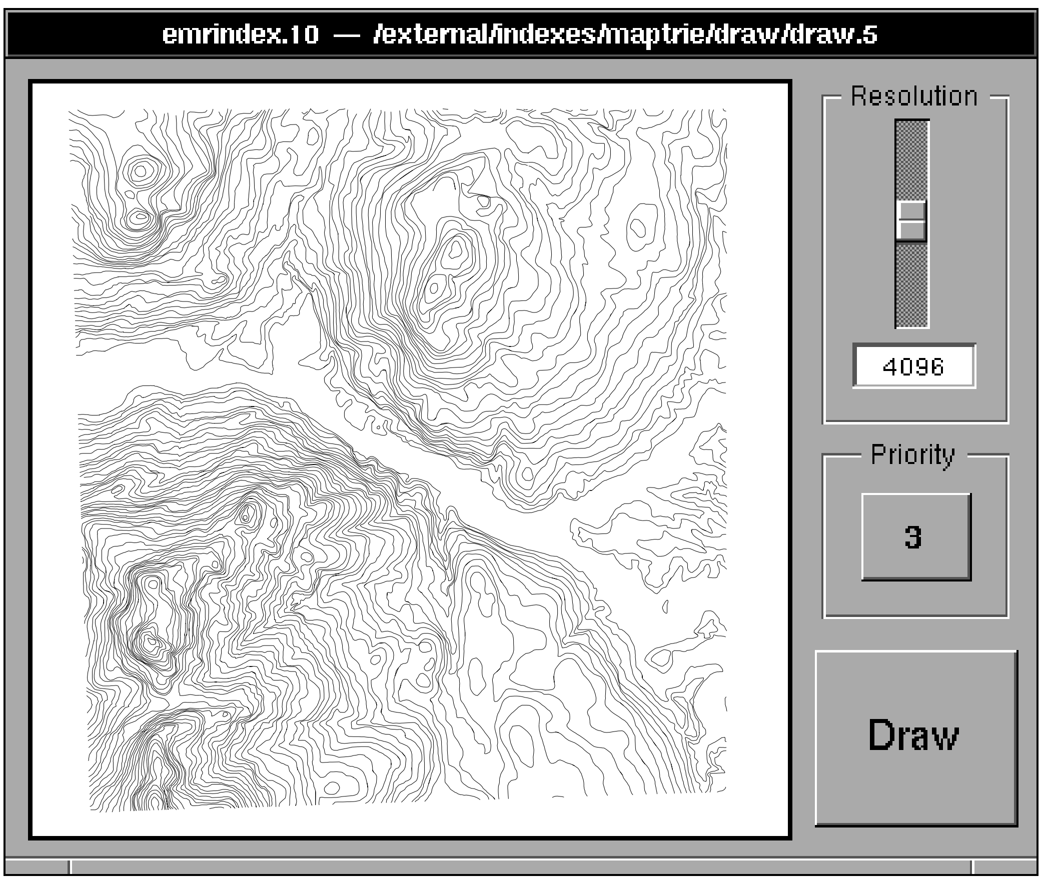


Figure 7: Contour Map Zooming

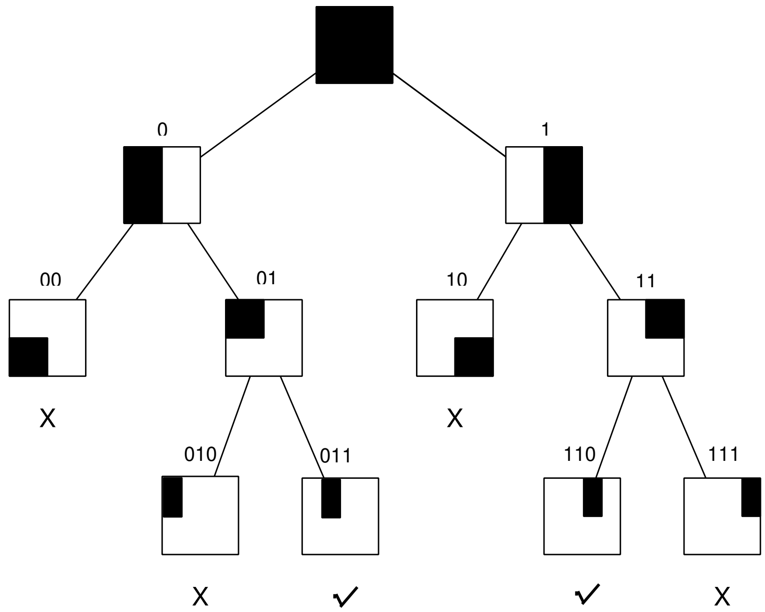


Figure 8: Variable-Resolution Orthogonal Range Query

to transmit data differentially from disk or across a network. We now go on to apply these advantages to less specialized data and more general queries.

We discuss orthogonal range queries on multikey data [10], which is the first step from spatial data. The data need not be spatial, but it is multidimensional. The queried attributes are thus not limited to numeric data, but can be generally alphanumeric. Figure 8 shows a trie for a two-dimensional space, with each node schematically indicating the region of space it represents. The top levels shown represent the first two bits of one attribute, say *Name*, and the first bit of the second attribute, say, *Age*. The labels of the paths leading to each node are also shown at the node, for convenience.

Suppose that we have a range query for all *Name*s starting with bits `01` to all *Name*s starting with bits `10` *and* all *Age*s starting with bit `1`.

The range query is processed as follows.

1. If the current node has no overlap with the query, it is *rejected*, along with all its descendents.
2. If it is entirely contained in (or equals) the query, it and its descendents are *accepted*.
3. Otherwise, the search proceeds recursively to the descendents of the current node.

Thus, in the example, case (3) holds at the root and at its two children. At node `00`, (1) holds, so this node is rejected and the search along that branch stops. At node `01`, (3) holds, so we proceed to `010`, which is rejected, and `011`, which is accepted (case (2)). Similar considerations apply to the descendents of node `1`. The crosses and checks show where the search ends in rejections or acceptances. Once a node is accepted, we know that all leaves that descend from it will contain data answering the query.

For our experimental measurements, we used ordinary tries, and we compared the results with grid files [12] and with multipaging [6, 7]. Two parameters proved to be important. Figure 9 shows the dependence of access cost on *activity*, which is the ratio of the number records returned by the answer to the number of records in the file. It shows that tries perform up to four times better than grid files, with multipaging in between. These results can be explained in terms of page capacity. A page of a trie holds more records than a multipaged page of the same size, and this in turn is better than a grid file page.

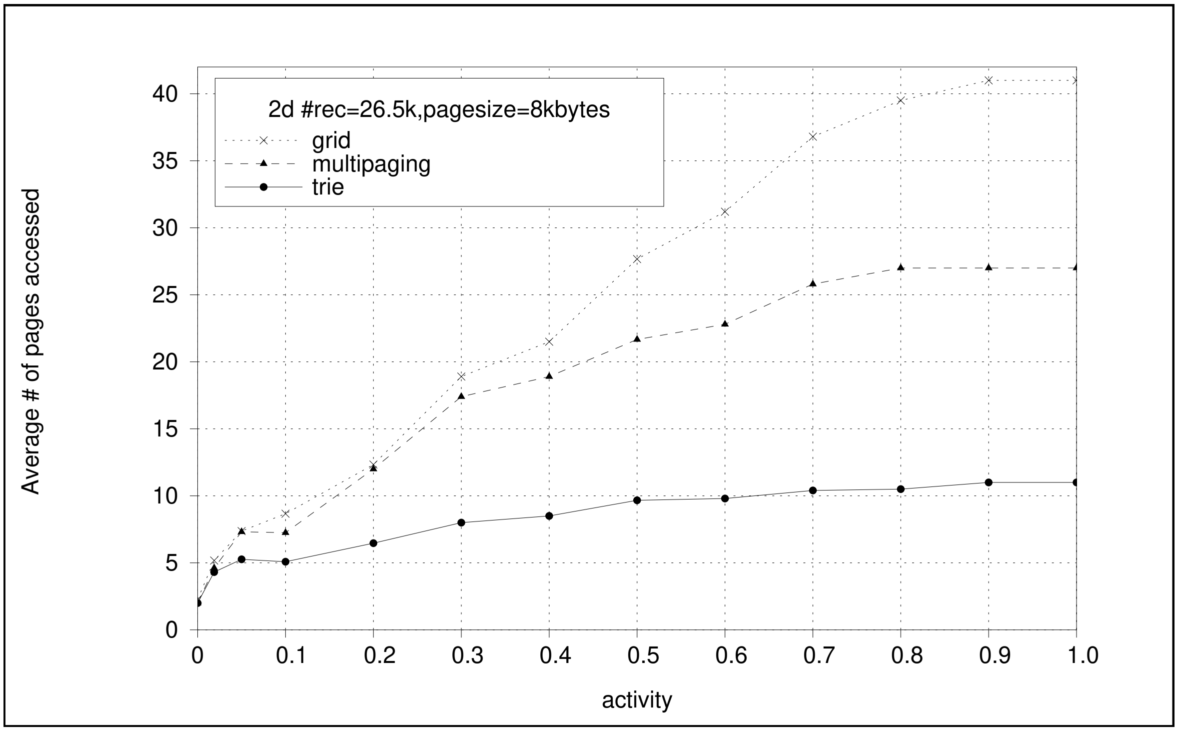


Figure 9: Access Cost vs. Activity

The second important parameter is the *distribution* of the data. Poorly distributed data can penalize grid files immensely in storage overheads, and multipaging in either storage or speed (but not both). We needed a good way to quantify the notion of poor and good distribution, and found that the *mutual information, I* of information theory is suitable. Figure 10 shows the dependence of storage costs on distribution, with the distributions most favouring grid files and multipaging at $I=$ -9.5, and the poorest distribution (for these techniques) at $I=$ -4.5. (The best distribution has each axis independent of the others, and examples of poor distributions in two dimensions have the data lying on a diagonal straight line or on a large circle.)

We see that the grid file starts with a 50% overhead, which rises to 350% as the distribution becomes poorer, and we see that this is mainly due to the expansion of the directory. The version of multipaging used suffers speed losses at poor distributions, but not storage penalties. We see that tries offer over 50% *compression*, and improve as the distribution gets bad for the other methods. (In fact, we have shown that tries compress more as the file grows, and over 90% for files of 10^7 records.)

We also established that trie access speeds are an order of magnitude better than grid files for activities of 20% to 100% and distributions in the poorer half of the range.

5 Conclusions and Future Work

It is evident that tries are a promising technique for all kinds of data on secondary storage. For text data, we have demonstrated a 10% to 25% improvement over earlier methods for exact searching, and improvements of typically several *times* the single-error approximate search of *agrep*. For spatial data, we have produced a new capability, variable-resolution zooming, with time *improvements* proportional to the file size over the alternatives. Finally, for general data, tries can perform orthogonal range queries up to an order of magnitude faster than the usual technique, the grid file, with up to an order of magnitude less storage.

This paper merely opens the door to further adventures. The obvious next step is range queries more general than orthogonal, say involving general boolean conditions, or algebraic

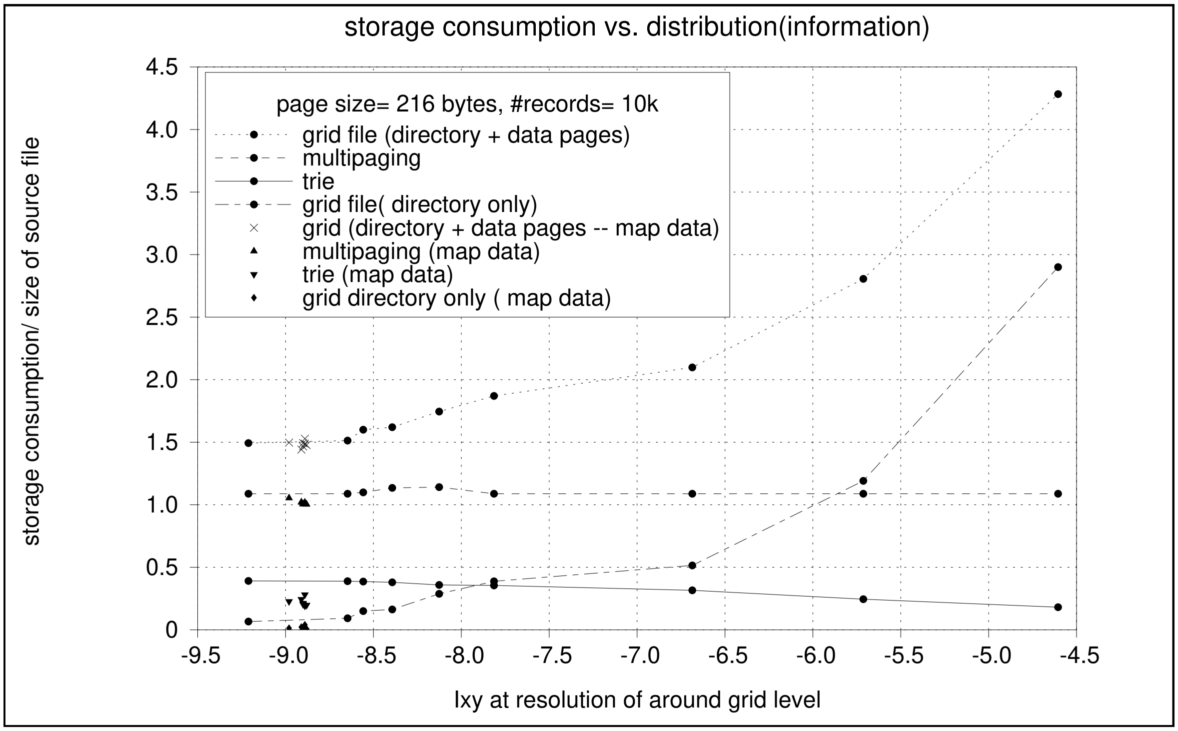


Figure 10: Storage Cost vs. Distribution

expressions on numeric data. Space and discretion preclude any more hints, but there are exciting possibilities for new work in this direction.

6 Acknowledgements

This work has been supported by the Natural Sciences and Engineering Research Council of Canada under grant NSERC OGP0004365, and by the national Networks of Centres of Excellence through the Institute of Robotics and Intelligent Machines, adminstered by PRECARN Associates.

References

[1] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–17, September 1975.

[2] R. de la Briandais. File searching using variable-length keys. In *Proc. Western Joint Computer Conf.*, pages 295–8, San Francisco, March 1959.

[3] E. H. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–9, Sept. 1960.

[4] G. H. Gonnet. Efficient searching of text and pictures. Technical Report OED-88-02, Centre for the New Oxford English Dictionary, University of Waterloo, Waterloo, Ont., Canada, 1988.

[5] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. *Information Retrieval: Data Structures and Algorithms*, pages 66–82, 1992.

[6] T. H. Merrett. Multidimensional paging for efficient data base querying. In *International Conference on Data Base Management Systems*, pages 277–90, Milano, June 1978.

[7] T. H. Merrett and E. J. Otoo. Dynamic multipaging: a storage structure for large shared data banks. In P. Scheuermann, editor, *Improving Database Usability and Responsiveness*, pages 237–56, New York, 1982. Academic Press.

[8] T. H. Merrett and H. Shang. Trie methods for representing text. In D. B. Lomet, editor, *Foundations of Data Organization and Algorithms*, pages 130–145, Chicago, Illinois, October 13–15 1993.

[9] T. H. Merrett and H. Shang. Zoom tries: A file structure to support spatial zooming. In T. C. Waugh and R. G. Healey, editors, *Advances in GIS Research Proceedings, Volume 2*, pages 702–804, Edinburgh, Scotland, September 5–9 1994. Taylor & Francis.

[10] T. H. Merrett and Xiaoyan Zhao. Zoom tries: Using variable resolution for orthogonal range queries. Submitted to ACM SIGMOD'97, International Conference on the Management of Data, Tucson, Arizona, May 13–15, 1997, May 1997.

[11] D. R. Morrison. PATRICIA: Practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–34, 1968.

[12] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: an adaptable, symmetric, multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, March 1984.

[13] J.A. Orenstein. Blocking mechanism used by multidimensional tries. Unpublished Letter, February 1983.

[14] H. Shang. Trie methods for text and spatial data on secondary storage. Ph.D. Dissertation, School of Computer Science, McGill University, January 1995.

[15] H. Shang and T. H. Merrett. Tries for approximate string matching. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):540–7, August 1996.

[16] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–78, 1974.

[17] S. Wu and U. Manber. Fast text searching. *Comm. ACM*, 35:83–91, 1992.