Adaptive Algorithms for Cache-efficient Trie Search

Anurag Acharya, Huican Zhu, Kai Shen
Dept. of Computer Science
University of California, Santa Barbara, CA 93106

Abstract

In this paper, we present cache-efficient algorithms for trie search. There are three key features of these algorithms. First, they use different data structures (partitioned-array, B-tree, hashtable, vectors) to represent different nodes in a trie. The choice of the data structure depends on cache characteristics as well as the fanout of the node. Second, they adapt to changes in the fanout at a node by dynamically switching the data structure used to represent the node. Third, the size and the layout of individual data structures is determined based on the size of the symbols in the alphabet as well as characteristics of the cache(s). We evaluate the performance of these algorithms on real and simulated memory hierarchies. Our evaluation indicates that these algorithms out-perform alternatives that are otherwise efficient but do not take cache characteristics into consideration. A comparison of the number of instructions executed indicates that these algorithms derive their performance advantage primarily by making better use of the memory hierarchy.

1 Introduction

Tries are widely used for storing and matching strings over a given alphabet. Applications include dictionary lookup for text processing [6, 7, 8, 14, 22], itemset lookup for mining association rules in retail transactions [1, 2], IP address lookup in network routers [26, 27] and partial match queries [13, 24]. There has been much work on reducing the storage requirement and the instruction count for tries – for example [3, 5, 4, 6, 19, 20, 22]. These algorithms, however, do not take the memory hierarchy into account. Given the depth of memory hierarchies on modern machines, good cache performance is critical to the performance of an algorithm.

In this paper, we present cache-efficient algorithms for trie search. There are three key features of these algorithms. First, they use different data structures (partitioned-array, B-tree, hashtable, vector) to represent different nodes in a trie. The choice of the data structure depends on cache characteristics as well as the fanout of the node. Second, they adapt to changes in the fanout at a node by dynamically switching the data structure used to represent the node. Third, the size and the layout of individual data structures is determined based on the size of the symbols in the alphabet as well as characteristics of the cache(s).

We evaluate the performance of these algorithms on real and simulated memory hierarchies. To evaluate their performance on real machines, we ran them on three different machines with different memory hierarchies (Sun Ultra-2, Sun Ultra-30, and SGI Origin-2000). To evaluate the impact of variation in cache characteristics on the performance of these algorithms, we simulated architectures that differed in cache line size and cache associativity.[1]

[1] Appendix A provides a brief introduction to caches in modern machines.

To drive these experiments, we used two datasets from different application domains and with different alphabet sizes. The first dataset was from the text processing domain and consisted of a trie containing all the words in the Webster's Unabridged Dictionary. Against this dictionary, we ran searches using all the words in Herman Melville's *Moby Dick*. This dataset had an alphabet of 54 symbols (lower and upper case English characters, hyphen and apostrophe). The second dataset was from the datamining domain. For this application, the alphabet consists of items that can be purchased in a grocery store (beer, chips, bread etc) and a string consists of a single consumer transaction (the set of items purchased at one time). The task is to determine the set of items (referred to as *itemsets* that are frequently purchased together. Tries are used in this application to store the candidate itemsets and to help determine the frequent itemsets. The dataset used in our experiments was generated using the Quest datamining dataset generator which we obtained from IBM Almaden [23]. This dataset generator has been widely used in datamining research [1, 2, 15, 28, 29]. The alphabet for this dataset contained 10,000 symbols (corresponding to 10,000 items).

Our evaluation indicates that these algorithms out-perform alternatives that are otherwise efficient but do not take cache characteristics into consideration. For the dictionary dataset, our algorithm was 1.7 times faster on the SGI Origin-2000 and 4 times faster on both the Sun Ultras compared to the ternary search tree algorithm proposed by Bentley and Sedgewick [7]. For the datamining dataset, our algorithm was 1.4-1.9 times faster than a B-tree-trie and 1.2-1.5 times faster than a hashtable-trie. A comparison of the number of instructions executed indicates that the algorithms presented in this paper derive their performance advantage primarily by making better use of the memory hierarchy.

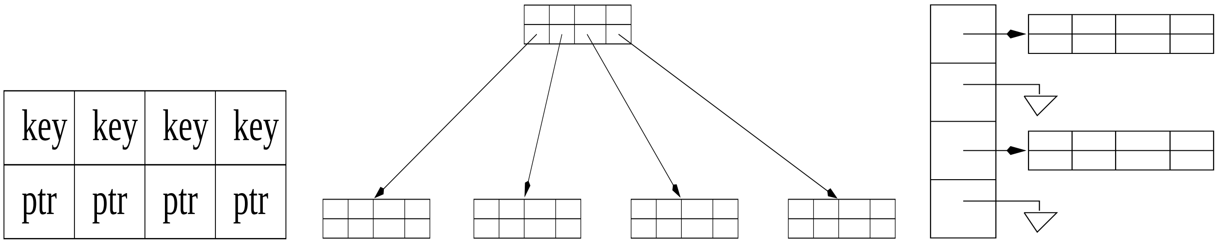
The paper is organized as follows. A brief review of tries and modern memory hierarchies can be found in Appendix A. Section 2 presents the intuition behind our algorithms and describes them in some detail. Section 3 describes our experiments and Section 4 presents the results. Section 5 presents related work and Section 6 presents a summary and conclusions.

2 Algorithms

The design of the algorithms presented in this paper is based on three insights. First, there is a large variation in the fanout of the nodes in a trie. The nodes near the root usually have a large fanout; the nodes further from the root have a smaller fanout. This suggests that different data structures might be suitable for implementing different nodes in a trie. Second, only the first memory reference in a cache line has to wait for data to be loaded into the cache. This suggests that the data structures should be designed so as to pack as many elements in a cache line as possible.[2] Third, at each level of a trie, at most one link is accessed (the link to the successor if the match can be extended, no link if the match cannot be extended). This suggests that for nodes with a large fanout, the keys and the links should be stored separately. This avoids reduces the number of links loaded and avoids loading any links for unsuccessful searches.

We present two algorithms. The first algorithm is for tries over large alphabets which require an integer to represent each symbol. The second algorithm is for tries over small alphabets whose symbols can be represented using a character.

[2]Some memory hierarchies return the first reference first and load the rest of the cache line later. For such memory hierarchies, subsequent memory references may also need to wait.



(a) Partitioned-array (b) B-tree (c) Hashtable

Figure 1: Data structures used to represent trie nodes for large alphabets.

2.1 Algorithm for large alphabets

This algorithm assumes that individual symbols are represented by integers and uses three alternative data structures, a *partitioned-array*, a bounded-depth B-tree and a hashtable, for representing trie nodes. It selects between them based on the fanout of the node and the cache line size.

The *partitioned-array* structure consists of two arrays – one to hold the keys and the other to hold the corresponding links. Each array is sized to fit within a single cache line. This assumes that integers and links are of the same size. For architectures on which links and integers are not the same size, the array containing the links is allowed to spread over multiple cache lines. The bounded-depth B-tree structure uses partitioned-arrays to represent individual nodes. The hashtable structure uses chains of partitioned-arrays to handle collisions. Figure 1 illustrates all three structures.

When a new node is created, the partitioned-array structure is used. As the fanout of a node increases, new keys are inserted into the partitioned-array. When the fanout of a node increases beyond the capacity of a partitioned-array, the partitioned-array is converted into a B-tree. Since partitioned-arrays are used to represent B-tree nodes, this conversion is cheap. Subsequent insertions into the node result in new keys being added to the B-tree. The bounded-depth B-tree structure is used as long as its depth is less than a threshold. In the experiments reported in this paper, we used a depth bound of four levels. When the fanout of the node increases such that the keys no longer fit within the bounded-depth B-tree, the B-tree is converted into a hashtable. This conversion requires substantial rearrangement. However, with a suitable depth bound for the B-tree, this conversion will be rare. Furthermore, since the hashtable uses the same basic data structure (the partitioned-array), it will be able to re-use the memory (and possibly cache lines) that were used by the B-tree.

2.2 Algorithm for small alphabets

This algorithm assumes that individual symbols are represented by characters and uses partitioned-arrays of different sizes (1/2/4/8) and an m-way vector (where m is the size of the alphabet) to represent trie nodes. When a new node is created, a partitioned-array with just a single entry is used. As new keys are inserted, the partitioned-array is grown by doubling its size. When the number of keys exceeds a threshold, the partitioned-array is converted into an m-way vector (in order to reduce search time). In our algorithm, the threshold is selected by dividing the cache line size by the size of each link. For this algorithm, we use the size of links to size the partitioned-array instead of the size of the keys. This is because a large number of keys fit into cache lines available on

Machine	Cache line size	L1 size	L2 size	Proc. Clock	Memory BW
Sun Ultra-2	32-byte	16KB (direct-mapped)	2MB (direct-mapped)	167MHz	200MB/s
Sun Ultra-30	32-byte	32KB (direct-mapped)	2MB (direct-mapped)	250MHz	200MB/s
SGI Origin-2000	64-byte	32KB (direct-mapped)	2MB (2-way assoc)	195MHz	296MB/s

Table 1: Details of the real memory hierarchies used in experiments. The associativity of each cache is in parentheses. The memory bandwidth numbers have been measured using the STREAM *copy* benchmark written by John. McCalpin (*http://www.cs.virginia.edu/stream*)

current machines (32-byte/64-byte) and using link-arrays with 32/64 slots would greatly increase the space requirement for these nodes. The m-way vector consists of a link array that is indexed by the key. Conversion between these alternatives is cheap and requires only re-allocation of the space required to hold the arrays and copying the contents.

3 Experimental setup

We evaluated the performance of these algorithms on real and simulated memory hierarchies. To evaluate their performance on real machines, we ran them on three different machines with different memory hierarchies – a Sun Ultra-2, a Sun Ultra-30, and an SGI Origin-2000. Table 1 provides details about these machines. We ran each experiment five times and selected the lowest time to account for potential interference due to daemons and other processes running during the experiments.

To evaluate the impact of variation in cache characteristics on the performance of these algorithms, we simulated architectures that differed in cache line size (32/64-byte) and cache associativity (direct-mapped/2-way/4-way/8-way). We used the *msim* simulator [25] and assumed a 32KB L1 cache and a 2MB L2 cache. We assumed zero delay fetching data from L1 cache, a 5 cycle delay from L2 cache and a 36 cycle delay from main memory.

To drive these experiments, we used two datasets from different application domains and with different alphabet sizes. The first dataset was from the text processing domain and consisted of a trie containing all the words in the Webster's Unabridged Dictionary.[3] The words were inserted in the order of their occurrence in the dictionary. Against this dictionary, we ran searches using all the words (212933) in Herman Melville's *Moby Dick*.[4] The words were searched for in their order of occurrence. This dataset had an alphabet of 54 symbols (lower and upper case English characters, hyphen and apostrophe). The average length of the words in the trie was 6.7 characters and the average length of the words in *Moby Dick* was 5.4 characters.

For this dataset, we used the algorithm for small alphabets and compared its performance to that of the ternary search tree proposed by Bentley and Sedgewick [7].[5] Bentley et al demonstrate that ternary search trees are somewhat faster than hashing for an English dictionary and up to five times faster than hashing for the DIMACS library call number datasets [7]. Clement et al [10]

[3] An online version is available at *ftp://uiarchive.cso.uiuc.edu/pub/etext/gutenberg/etext96/*.

[4] An online version is available at *ftp://uiarchive.cso.uiuc.edu/pub/etext/gutenberg/etext91/moby.zip*.

[5] We obtained the code for the ternary search tree algorithm from *http://www.cs.princeton.edu/~rs/strings.*.

analyze the ternary search tree algorithm as a form of trie and conclude that it is an efficient data structure from an information-theoretic point of view.

The second dataset was from the datamining domain and consisted of sets of retail transactions generated by Quest datamining dataset generator which we obtained from IBM Almaden [23]. For this application, the alphabet consists of the items that can be purchased in a grocery store and a string consists of a single retail transaction. Recall that the task is to determine the set of items that are frequently purchased together. Tries are used in this application to store the candidate itemsets and to help determine the frequent itemsets. The alphabet for this dataset contained 10,000 symbols (corresponding to 10,000 items); the average length of each string (i.e., the average number of items purchased per transaction) was 4 with a maximum length of 10. We created four pairs of transaction-sets. One transaction-set in each pair was used to construct the trie and the other was used for searching the trie. In each pair, the transaction-set used to search the trie was four times as large as the transaction-set used to create the trie and included it as a subset. The trie for the first pair of transaction-sets contained 20,000 transactions; the trie for the second pair of transaction-sets contained 40,000 transactions; the trie for the third pair of transaction-sets contained 80,000 transactions; and the trie for the fourth pair of transaction-sets contained 160,000 transactions. Compactly represented, these transactions can be stored in 1MB, 2MB, 4MB and 8MB respectively. For the machines used in our experiments, this corresponds to 0.5*cache_size, cache_size, 2*cache_size and 4*cache_size. This allowed us to explore the space of ratios between the cache size and the dataset size.

For this dataset, we used the algorithm for large alphabets and compared its performance against two alternatives. The first algorithm was a non-adaptive variant of our algorithms which used B-trees for all nodes (we refer to this as the B-tree-trie algorithm). This algorithm does not bound the depth of the B-tree. Our goal in this comparison was to determine how much advantage is provided by adaptivity. The second algorithm used variable-sized hashtables for all nodes (we refer to this as the hashtable-trie algorithm). The hashtable used for the root node had 1024 buckets; the hashtables used for nodes at every subsequent level reduced the number of buckets by a factor of two. We selected this algorithm for our experiments as an algorithm similar to this is proposed by several researchers for maintaining frequent itemsets for mining association rules [1, 2, 15].

4 Results

Figure 2 compares the performance of the adaptive algorithm with that of the ternary search tree for the dictionary dataset. The adaptive algorithm significantly outperforms the ternary search tree on all three machines – by a factor of 1.7 on the Origin-2000 and by a factor of 4 on the Ultra-2 and the Ultra-30.

Figure 3 compares the performance of the adaptive algorithm with that of the B-tree-trie and the hashtable-trie algorithms for the datamining dataset. We note that the adaptive algorithm is faster than both the alternatives for all inputs (by a factor of 1.4-1.9 over B-tree-trie and a factor of 1.2-1.5 over hashtable-trie). Note that the speedup falls off as the dataset size grows to several times the cache size. This indicates that the input is not localized to a small part of the trie.

There are three possible reasons for the performance advantage of the adaptive algorithms over the non-adaptive algorithms: (1) they execute fewer instructions, or (2) they spend less time waiting for data to be loaded into cache, or (3) both. For each of the dataset-platform-algorithm combination, we determined the number of instructions executed. For this, we built a cost model for each platform-algorithm combination and inserted code to its implementation to maintain an instruction counter. To build the cost model for each platform-algorithm combination, we disas-

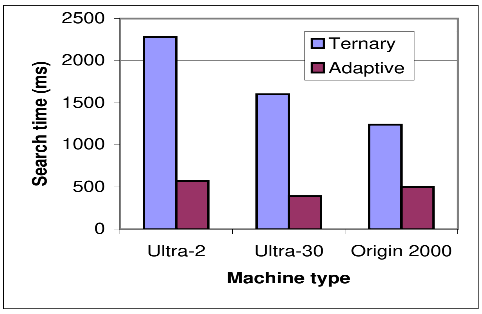
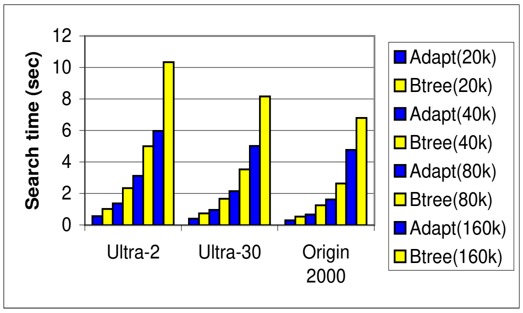
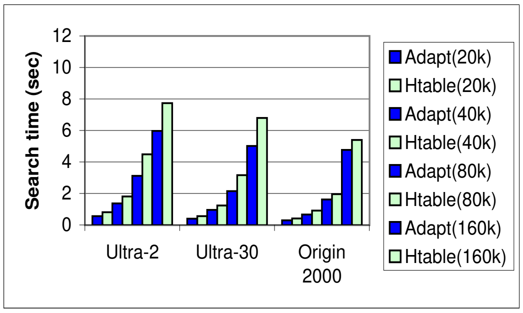


Figure 2: Performance of the adaptive algorithm and the ternary search tree for the dictionary dataset.



(a) Adaptive vs B-tree-trie



(b) Adaptive vs hashtable-trie

Figure 3: Performance of the adaptive algorithm, the B-tree-trie and the hashtable-trie for the datamining dataset.

sembled the optimized binary code for that platform-algorithm combination and determined the number of instructions in each basic block.

For the dictionary dataset, we found that the ternary search tree executes about 64 billion instructions whereas the adaptive algorithm executes about 62 billion instructions. Figure 4 compares the number of instructions executed by all three algorithms (adaptive, B-tree-trie and hashtable-trie) for the four datamining inputs. We note that the adaptive algorithm executes about the same number of instructions as the hashtable-trie and close to half the number of instructions executed by the B-tree trie. We also compared the amount of space used by the different algorithms. For the dictionary dataset, the ternary search tree used 6.2 MB (388K nodes) and the adaptive algorithm used 6.0MB (217K nodes).[6] Table 2 presents the space used by different algorithms for the datamining dataset. From these results, we conclude that the adaptive algorithms derive their performance advantage primarily by making better use of the memory hierarchy.

One of the main features of the adaptive algorithms is that they change the representation of a node as its fanout changes. Figure 5 presents the distribution of the different representations used

[6] These numbers are for the Sun Ultras.

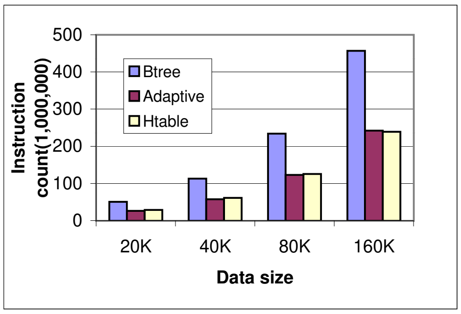
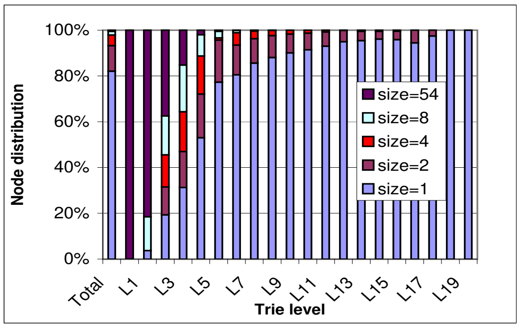


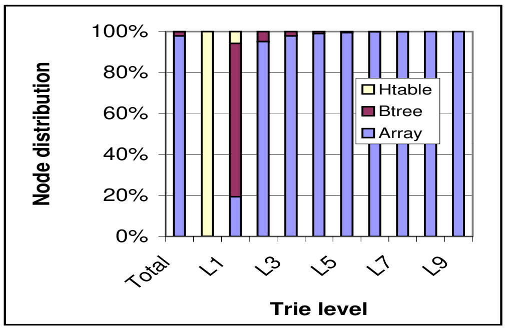
Figure 4: Number of instructions executed by different algorithms for the datamining dataset.

Num Transactions →	20K	40K	80K	160K
Adaptive	3.6MB	7.1MB	13.6MB	41.3MB
B-tree-trie	8MB	14.9MB	27.9MB	62.3MB
Hashtable-trie	36MB	60MB	95MB	190MB

Table 2: Space used by different algorithms for the datamining dataset.



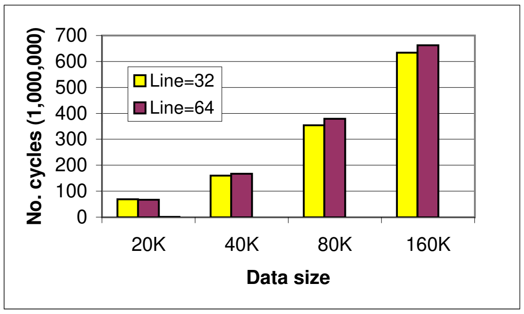
(a) Dictionary dataset



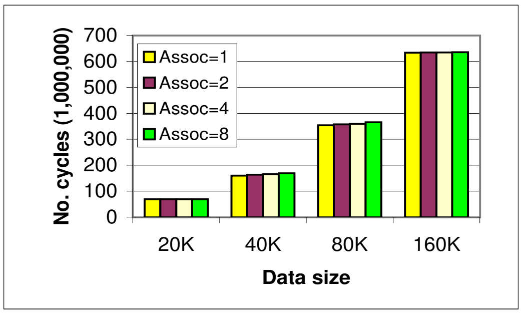
(b) Datamining dataset

Figure 5: Distribution of the data structures used by the adaptive algorithms. For the datamining dataset, the graphs present the distribution for the input with 160K transactions.

for both datasets. The first bar in both graphs depicts the distribution of the three data structures in the entire trie; the bars to its right provide a level-by-level breakdown. We note that there is a wide variation in the distribution of data structures (and therefore fanout) on different levels and that on the whole, tries for both datasets are dominated by nodes with very small fanouts.



(a) Variation in line size



(b) Variation in associativity

Figure 6: Impact of variation in cache characteristics on the performance of the adaptive algorithm. For the results presented in (a), both L1 and L2 are assumed to be direct-mapped; for the results presented in (b), the cache line size is assumed to be 32-bytes and L1 is assumed to be direct-mapped.

4.1 Impact of variation in cache characteristics

To evaluate the impact of variation in cache characteristics, we used the *msim* memory hierarchy simulator. Figure 6(a) examines the variation in simulated execution time as the cache line size changes. We used 32B and 64B as the alternatives as these are considered the most suitable cache line sizes. Smaller cache lines result in more frequent fetches from the main memory; larger cache lines require the processor to wait longer. In addition, larger cache lines are more likely to result in unnecessary data movement from/to the main memory. For this experiment, we assumed that both L1 and L2 caches were direct-mapped. We note that the performance of the adaptive algorithm does not change much with a change in the cache line size. This is not surprising as the line size is an explicit parameter of the algorithm.

Figure 6(b) examines the variation in simulated execution time as the cache associativity changes. For this experiment, we assumed 32-byte cache lines, and a direct-mapped L1 cache. We varied the associativity of the L2 cache with direct-mapped, 2-way, 4-way and 8-way associative configurations. Note that most modern L1 caches are direct-mapped and L2 caches are either direct-mapped or 2-way associative. We note that the the performance of the adaptive algorithm is insensitive to changes in cache associativity.

5 Related work

Previous work on adapting trie structures has taken one of four approaches. The first approach focuses on reducing the number of instructions executed by reducing the number of nodes and levels in the trie [5, 4, 6, 12, 19, 22]. The second approach views tries as collections of m-way vectors and focuses on reducing the space used by these vectors using a list-based representation or a sparse-matrix representation [3, 17, 20]. The third approach focuses on the data structures used to represent trie nodes with the goal of reducing both the space required and the number of instructions executed [7, 9]. Finally, there has been much recent interest in optimizing the tries used for address lookups in network routers [11, 21, 26]. The algorithms proposed by these researchers focus on reducing the number of memory accesses by reducing the number of levels in the trie and the fanout at individual nodes. Our approach is closest to these in that we share their goal of reducing memory accesses; however, there are two main differences. First, these algorithms assume that the "symbols" to be matched at each level (so to speak) are not fixed and that the strings that form the trie can be arbitrarily subdivided. This assumption is correct for application that they focus on – IP address lookup in network routers.[7] We assume a fixed alphabet which is applicable to most other applications tries are used for. Second, these algorithms focus on re-structuring the trie, while we focus on selecting the data structure for individual nodes.

6 Summary and conclusions

In this paper, we presented cache-efficient algorithms for trie search. The key features of these algorithms are: (1) they use multiple alternative data structures for representing trie nodes; (2) they adapt to changes in the fanout at a node by dynamically switching the data structure used to represent the node; (3) they determine the size and the layout of individual data structures based on the size of the symbols in the alphabet as well as characteristics of the cache(s).

Our evaluation indicates that these algorithms out-perform alternatives that are otherwise efficient but do not take cache characteristics into consideration. A similar conclusion is reached by Lamarca&Ladner in their paper on cache-efficient algorithms for sorting [18].

For the dictionary dataset, our algorithm was 1.7 times faster on the SGI Origin-2000 and 4 times faster on both the Sun Ultras compared to the ternary search tree algorithm proposed by Bentley and Sedgewick [7]. For the datamining dataset, our algorithm was 1.4-1.9 times faster than a B-tree-trie and 1.2-1.5 times faster than a hashtable-trie. A comparison of the number of instructions executed indicates that the algorithms presented in this paper derive their performance advantage primarily by making better use of the memory hierarchy.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large data bases. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 207–16, Washington, D.C., May 1993.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Databases*, 1994.

[3] E. Ai-Sunwaiyel and E. Horowitz. Algorithms for trie compaction. *ACM Transactions on Database Systems*, 9(2):243–63, 1984.

[4] J. Aoe, K. Marimoto, and T. Sato. An efficient implementation of trie structure. *Software Practice and Experience*, 22(9):695–721, 1992.

[5] J. Aoe, K. Morimoto, M. Shishibori, and K. Park. A trie compaction algorithm for a large set of keys. *IEEE Transactions on Knowledge and Data Engineering*, 8(3):476–91, 1996.

[6] A. Appel and G. Jacobson. The world's fastest scrabble program. *Communications of the ACM*, 31(5):572–8, 1988.

[7] J. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of SODA'97*, 1997.

[7] IP addresses (currently) are 32-bits long and arbitrary-length prefixes can appear in network routing tables.

[8] A. Blumer, J. Blumer, D. Haussler, and R. McConnel. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–95, 1987.

[9] H. Clampett. Randomized binary searching with tree structures. *Communications of the ACM*, 7(3):163–5, 1964.

[10] J. Clement, P. Flajolet, and B. Vallee. The analysis of hybrid trie structures. Technical Report 3295, INRIA, Nov 1997.

[11] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. *Computer Communication Review*, October 1997.

[12] J. Dundas. Implementing dynamic minimal-prefix tries. *Software Practice and Experience*, 21(10):1027–40, 1991.

[13] P. Flajolet and C. Puech. Partial match retrieval of multidimensional data. *Journal of the ACM*, 33(2):371–407, 1986.

[14] G. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures: in Pascal and C*. Addison-Wesley, second edition, 1991.

[15] E. Han, V. Karypis, and V. Kumar. Scalable parallel data mining for association rules. In *Proceedings of SIGMOD'97*, 1997.

[16] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, second edition, 1996.

[17] D. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 1973.

[18] A. Lamarca and R. Ladner. The influence of caches on the performance of sorting. In *Proceedings of SODA'97*, 1997.

[19] C. Lucchesi and T. Knowaltowski. Applications of finite automata representing large vocabularies. *Software Practice and Experience*, 23(1):15–30, 1993.

[20] K. Maly. Compressed tries. *Communications of the ACM*, 19:409–15, 1976.

[21] S. Nilsson and G. Karlsson. Fast address lookup for internet routers. In *Proceedings of IEEE Broadband Communications'98*, 1998.

[22] J. Peterson. *Computer Programs for Spelling Correction*. Lecture Notes in Computer Science, Springer Verlag, 1980.

[23] IBM Quest Data Mining Project. The Quest retail transaction data generator[8], 1996.

[24] R. Rivest. Partial match retrieval algorithms. *SIAM Journal on Computing*, 5:19–50, 1976.

[25] S. Sharma and A. Acharya. The *msim* memory hierarchy simulator. Personal Communication, 1997.

[26] S. Venkatachary and G. Varghese. Faster IP Lookups Using Controlled Prefix Expansion. In *Proceedings of SIGMETRICS'98*, pages 1–10, 1998.

[27] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed IP routing lookups. In *Proceedings of SIGCOMM'97*, 1997.

[28] M. Zaki, M. Ogihara, S. Parthasarthy, and W. Li. Parallel data mining for association rules on shared-memory multi-processors. In *Proceedings of Supercomputing'96*, 1996.

[29] M. Zaki, S. Parthasarathy, and W. Li. A localized algorithm for parallel association mining. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1997.

[8] Available at *http://www.almaden.ibm.com/cs/quest/syndata.html.*

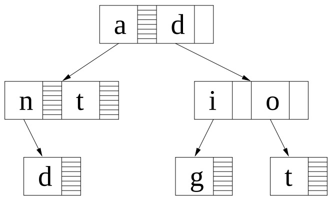


Figure 7: Example of a trie. A shaded box next to a key indicates that a string ends at that key. For example, the shaded box next to n at the second level indicates that the string "an" ends at that node.

A Background

A.1 Tries

Knuth [17] defines a trie as an m-way tree whose nodes are m-place vectors with components corresponding to individual symbols in the alphabet. Each node on level l represents the set of all strings that begin with certain sequence of l symbols; the node specifies an m-way branch depending on the $(l + 1)th$ symbol. Figure 7 presents a trie that contains the words *and, an, at, dig* and *dot*.

The search for a string in a trie starts from the root node and proceeds as a descent in the tree structure. At each level, the fanout of the node is examined to determine if the the next symbol in the string appears as a label for one of the links. The search is successful is all the symbols in the strings are matched; the search is unsuccessful if at any node, the next symbol in the string does not appear as a label for one of the outgoing links.

A.2 Modern memory hierarchies

Modern memory hierarchies attempt to bridge the difference between processor cycle time and memory access time by inserting one to three caches (referred to as L1, L2 and L3) between the processor and the main memory. In contemporary systems, data in the L1 cache can typically be accessed in a single processor cycle, data in the L2 cache in 5-13 processor cycles and data in the main memory in 25-150 processor cycles.

The size of caches grows with their distance from the processor – the caches closer to the processor are smaller than the caches further in the hierarchy. In contemporary systems, the L1 cache is typically between 8KB and 32KB; the L2 cache is between 96KB and 4MB and the L3 cache (if it exists) is multiple megabytes. Modern memory hierarchies satisfy the *inclusion* property – i.e., the cache at level i contains all the data contained in the cache at level $(i - 1)$.

Caches are divided into *lines*; a reference to any memory location in a cache line results in the entire line being brought in from the next level in the memory hierarchy. Newer implementations include an optimization that brings in referenced memory location first and allows the processor to resume execution while the rest of the cache line is being brought in. Larger cache lines are useful for programs with strong spatial locality; however, each cache miss takes longer to satisfy. Typically, cache lines are 32 or 64 bytes long.

Associativity of a cache is defined to be the number of locations in this cache that a memory location in the next level in the memory hierarchy can be mapped to. A cache that has m such locations is referred to as *m-way associative*. Caches with $(m = 1)$ are referred to as *direct-mapped*; caches with $(m = num\_lines\_in\_cache)$ are referred to as *fully associative*. Caches in modern memory hierarchies are, typically, either direct-mapped or 2-way associative.

For more information on modern memory hierarchies, see [16].