DiskTrie: An Efficient Data Structure Using Flash Memory for Mobile Devices

(Extended Abstract)

N.M. Mosharaf Kabir Chowdhury, Md. Mostofa Akbar, and M. Kaykobad

Department of Computer Science & Engineering
Bangladesh University of Engineering & Technology
Dhaka-1000, Bangladesh
`mosharafkabir@gmail.com, mostofa@cse.buet.ac.bd, kaykobad@cse.buet.ac.bd`

Abstract. DiskTrie is an efficient external-memory data structure for storing strings in mobile devices using flash memory. It supports *Lookup* and *Prefix-Matching* operations with a constant internal memory and linear processing requirements. The number of disk accesses it takes to search for a string among n unique finite strings is bounded by $\Theta(\log^* n)$, while for a prefix-matching operation it takes $\Theta(\log^* n) + O(\frac{n}{B})$ disk accesses, where B is the size of one page in the flash memory.

1 Introduction

In recent years, there has been a large influx of technological gadgets into people's life due to the revolution in the mobile communication sector. Hand-held devices have become an essential part of everyday life with a lot of applications being developed for this platform and the complexity of the programs are also increasing rapidly. Moreover, technological advances in embedded systems and advanced research works in different sensor networks are also dependent on small-scale devices. As a result, low complexity algorithms and data structures, with small memory requirements, have become extremely important.

A very common problem addressed in computer literature is the storing of large amount of text and searching or retrieving them efficiently. Because of its multifarious applications, from searching substrings in a book or dictionary to large DNA sequence mapping or matching, it has got a lot of attention over the years. In *Stringology*, there are a lot of internal memory algorithms and data structures already available that efficiently address this problem; for example - binary search tree, hash, trie etc [1–3] to name some simple ones and Patricia trie [4, 5], suffix tree [6], suffix array [7], suffix cactus [8] etc. are more advanced ones. But when data get too large to fit in main memory, the question of using external memory algorithms arises. These data are kept in external storage systems and frequently accessed by the algorithm to perform desired operations. In that case, the main focus turns to number of disk accesses, as it is the costliest of operations. Vitter has surveyed on many such external memory implementations and discussed the findings in [9, 10]. Many of the internal memory data structures have also been extended to external memory as described in [11–13]. There are also some extremely efficient data structures for external memory like String B-tree [14], Hierarchy of indexes [13] etc. The performance of these data structures and accompanying algorithms have been extensively tested for external storage systems.

But this very common and well-addressed problem of *Stringology* is not so well discussed in case of mobile devices with very small internal memory and processing power. Because of its very small amount of internal memory it cannot cope with the amount of data that can easily be handled in a modern computer and hence it has to take help of external memory in most cases. On the other hand, in most of the recent hand-held devices *flash memory* is used as external memory and because of its cheap price and increasing size it is also gaining popularity even among computer systems. In recent years there has been some works on flash memory specific file systems [15, 16] and improvements over them [17]. Also, some conventional external memory data structures and algorithms e.g. B-tree, R-tree have been modified for this type of memory systems [18, 19] and different indexing data structures have been proposed for flash-based systems [20]. Nevertheless, a study of Gal and Toledo [21] in 2005 shows that, flash-aware data structures are still very rare and algorithms to exploit its characteristics are in their incipient phases.

In this paper a mobile-specific modification and flash-specific implementation of a static trie data structure is proposed. The idea is to create the trie in a computer, store the pre-built trie in the flash memory maintaining a particular order along with the actual strings and to retrieve the nodes of the trie from the flash to traverse in the required portion of the data structure. A path-compressed and level-compressed binary trie (LPC trie) as described by Nilsson and Tikkanen [22] is used because of its ease of implementation as a binary tree and expected average depth of $\Theta(\log^* n)$ [23] while containing n independent random strings from a distribution with density function $f \in L^2$ over $\Theta(\log n)$ [24] by normal trie. The function $\log^* n$ is the iterated logarithm function, which is defined as follows. $\log^* 1 = 0$. For any $n > 1$, $\log^* n = 1 + \log^*(\lceil \log n \rceil)$.

2 Preliminaries

2.1 Characteristics of Flash Memories

Flash memory is a type of EEPROM (Electronically Erasable Programmable Read-Only Memory). It is non-volatile and, as a result, is used for storing data permanently in hand-held devices and mobile phones. It has a unique read/write/erase behavior than other programmable memories and can be written a limited number of times, ranging from 10,000 to 1,000,000, after which it wears out and becomes unusable. It comes in two flavors - NAND and NOR. In both types, *write* operations can only clear bits and the only way to set bits is to *erase* entire region of memory.

NOR Flash Memory. NOR flash is the older of the two types and provides *random access*. It can be addressed at byte level from CPU and can be used as RAM, if needed. It is very slow to write and hence, normally used as storage for static data. This property of addressing at byte level makes it suitable for storing the nodes of the trie.

NAND Flash Memory. NAND is the recent addition to the family, and it works much like the block devices such as hard disks. It is normally partitioned and used as file systems. Normally, NAND flashes are divided into blocks that work as erase units and each block consists of several pages. Read/write operations are handled per page and hence, while considering NAND flashes for the proposed data structure the only concern is the page size. Typical page size of NAND flash is 512 bytes; where block size is about 16K bytes.

2.2 Trie Data Structure

In its original form the trie [2] is a data structure where a set of strings from an alphabet containing m characters is stored in an m-ary tree and each string is represented by a unique path from the root to a leaf node. A search is performed in the following way: The root uses the first character to choose a node and then that node uses the next character, and so on. An example is shown in Fig.1(a). In this paper binary encoding is used and thus the m-ary tree is converted to a binary one (Fig.1(b)). This binary encoding simplifies the structure in two ways. First, every internal node now has the same degree, which is two. Second, no space is needed to remember the character corresponding to an outgoing edge. The obvious problem of using binary encoding is that each string will become longer and as a result search will take longer time. The solution is *path compression* as shown in Fig.1(c). A path compressed binary trie is a trie where all sub-tries with an empty child have been removed.

Patricia Trie. The result of *path compression* can be further optimized by a structure known as Patricia trie [5]. In this case, at each internal node only two values are remembered. One is the first character of the compressed string and the other is called *Skip* value- number of bits that are same in the edge toward the descending subtrie. As binary encoding is used, remembering the first bit is also redundant. It is also observed that the size of the Patricia trie is not dependent on the length of the strings, rather it depends on the total number of strings. A Patricia trie storing n strings has exactly $2n - 1$ nodes. The Patricia trie form of the compressed trie of Fig.1(c) is shown in Fig1.(d).

Level Compression. *Level Compression* [23] can further reduce the size of the Patricia trie. In this case, if i highest levels of the trie are complete and $(i + 1)$ is not, then i highest levels are replaced by a single node of degree 2^i. And this procedure is repeated in a top-down manner until total trie is level

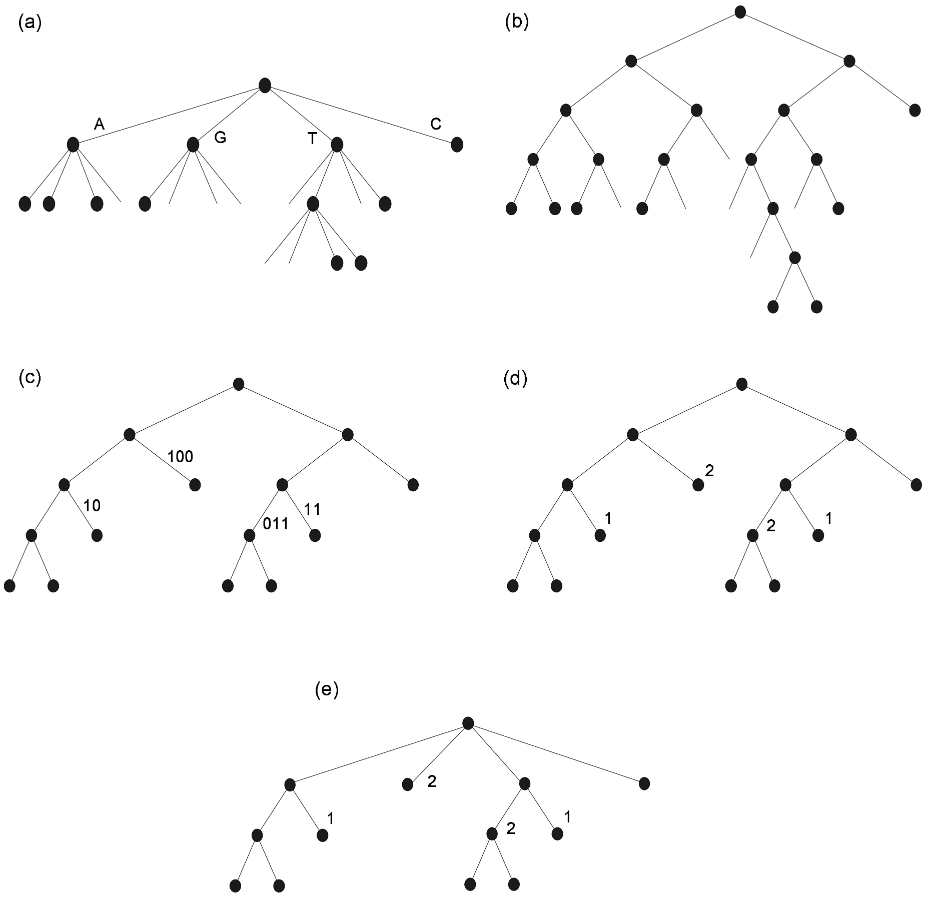

Fig. 1. Eight strings AA,AG,AT,GA,TGT,TGC,TC,C stored in (a) a trie: (b) a binary trie: (c) a trie after path compression: (d) a Patricia trie: (e) a LPC trie:

compressed. The resulting trie is known as LPC trie. Figure 1(e) presents the final state of the trie from Fig.1(a) after path and level compression. It results in an expected average depth of $\Theta(\log^* n)$. This exceedingly reduced depth will play a significant role in the proposed implementation for flash memories. It is shown in [23] that the degree of a node can be at most $2^{\lceil \log n \rceil}$, where n is the number of elements.

A detailed example of these compression techniques of trie is provided by Andersson and Nilsson in [25].

3 DiskTrie Implementation

In this section, the proposed data structure DiskTrie is described in details. It is an external memory implementation of an LPC trie taking advantage of *random access* provided by NOR flash memory for storing the nodes of the trie and using NAND flash memory for storing the actual strings. Along with the use of NOR flash for storing trie nodes, a NAND flash storage structure of trie nodes is also described. The implementation is discussed in detail by dividing the total process into three distinct phases, namely- Creation and Placement in Flash Memory and Retrieval.

3.1 Creation and Placement in Flash Memory

As DiskTrie is a static implementation, its content will not change during the operation of the data structure and as a result, it can be pre-built in a computer and then transfered to the flash memory afterward. This results in a two-pass algorithm.

- In the first pass, the data structure is built in RAM based upon the basic construction algorithms suggested in [23] and [25] with certain changes in the node structure for placing it in external storage. Another important part in this phase is the in-memory lexicographical sorting of all the unique strings stored in the DiskTrie. After sorting, all the strings are placed in NAND flash sequentially delimited by a specific separator symbol and the pointers are stored alongside the strings. This separator or end-marker also ensures that no string is a prefix of another one, which means that in every case only leaf nodes will contain pointer to the actual string residing in the disk.
- In the second pass, the necessary updates of pointers are done in the in-memory representation of the DiskTrie and the nodes of it are placed in the flash memory.

As in [25], the fields in each node are *bits*, *Skip* and *pointer*. The number *bits* indicates the number of bits used for branching at each node. A value of $bits \geq 1$ means the node has 2^{bits} child nodes and $bits = 0$ indicates that it is a leaf node. *Skip* is the Patricia skip value and *pointer* refers to the pointer to the text for a leaf node and pointer to the left-most child for an internal node. While placing these nodes into the secondary storage the pointer values are changed by actual addresses in the flash. Another important characteristic followed in case of NAND flash is that page boundaries are always maintained i.e. every single node is contained totally within a single page resulting in a certain amount of wastage in each page. As there is only one pointer to the left-most child of an internal node, all the child nodes of an internal node are also placed in sequence. This type of representation is suitable for both NAND and NOR type of flash.

3.2 Retrieval

The retrieval process deals with two types of operations on the data structure - *lookup* and *prefix-matching* in separate ways. These are discussed in the following.

Lookup. The `Lookup` algorithm presented in Fig.2 is almost similar to the one in [23], but extended to include the Patricia *Skip* value and the necessary access to flash memory. Here `getFromFlashDrive(address)` procedure takes the physical address and returns the node. The boundary maintaining property included in the previous section comes handy in case of NAND memory to prevent multiple access to retrieve a single trie node. `getValue(S, k, m)` procedure takes the zero-indexed binary string, `S` and returns the integer that starts at the k^{th} index and stretches up to `m` bits. And `calcAddress(n, p)` takes the offset of the child `n` and pointer to the first child and returns the actual physical address of the node. Finally, `stringCompare(S, rS)` compares the binary string `S` with the stored string `rS` and reports a boolean answer. This comparison at the end is absolutely necessary as skipped bits can induce errors while looking for a key string. `k`, `currentNode` and `temp` are three temporary variables used in the procedure.

Prefix-Matching. The problem is defined as follows: given a prefix P, all the strings that start with P must be returned. In a normal trie prefix-matching is a single pass procedure. All one has to do is to traverse to the node where P ends and the subtrie rooted at that node will contain all the strings that start with P. For example, in Fig.1(a) all the strings that start with a 'T' are found under the node that can be reached following the 'T' edge from the root. This is also true for the binary encoded trie in Fig.1(b) and the path compressed trie shown in Fig.1(c).

But when the Patricia *Skip* value is introduced, the chance of making errors while traversing arises. It can give rise to situations when the compared bits are equal but the search string differs in the omitted bits along the path to the leaf node. A two-phase search, given in [14], can effectively solve this problem. The first one identifies a prospective leaf node l, and in the second phase the *longest common prefix* between the string stored in l and P is identified. If the common prefix is not equal to P, then the search fails.

```
procedure Lookup(S) returns boolean
{
  currentNode = getFromFlashDrive(ROOT_ADDRESS);
  k = currentNode.Skip;
  while (currentNode.bits > 0 & k < S.length) do
    temp = getValue(S, k, currentNode.bits);
    k += currentNode.bits;
    currentNode = getFromFlashDrive(calcAddress(temp,
     currentNode.pointer));
    k += currentNode.Skip;
  endwhile

  if (currentNodes.bits > 0 | k > S.length) then
    return false;
  endif

  return stringCompare(S, getFromFlashDrive(currentNode.pointer));
}
```

Fig. 2. Lookup algorithm

As DiskTrie uses similar Patricia *Skip* value, it is also prone to problems just described. Moreover, because of level compression it has become an m-ary tree in effect and hence the prefix-matching is quite different from the previously described procedure of prefix-matching in Patricia tries. In this case, situation can arise when strings containing the same prefix P, may be located in different subtries. This happens when Patricia trie is level compressed using the procedure described in Section 2.2. But the problem eases as all the strings are stored in the flash sequentially. Because of that ordering, the problem reduces to finding the first string that starts with the given prefix P - all others will follow sequentially. A range, defining the starting and ending points of the locations of the strings in the storage, is also provided to reduce the number of string comparisons.

The total procedure of *Prefix-Matching* is basically divided into two separate phases.

– In the first phase, the trie is traced using P in a fashion similar to *Lookup* procedure until P is exhausted. If the tracing procedure reaches a leaf node before P ends, the result is a trivial failure. Otherwise, the last visited node is selected.
– In the second phase, the range where the matching strings may be located is identified. If the traversal in the first phase ended in a node then the subtrie rooted at this node contains the results. A single comparison must be made with the first of them to ensure whether all these strings start with P indeed.

  But if the traversal ended in an arc, then the selected node is the last node visited before reaching that arc. This is a particularly interesting case where only a portion of the subtrie located in that node should be selected instead of the entire one. The extra bits after reaching that node are used to make this selection. The left-most and right-most leaf nodes define the range of the location of the desired strings. Similar to the last case, a single comparison will ensure that the resultant strings are the correct ones.

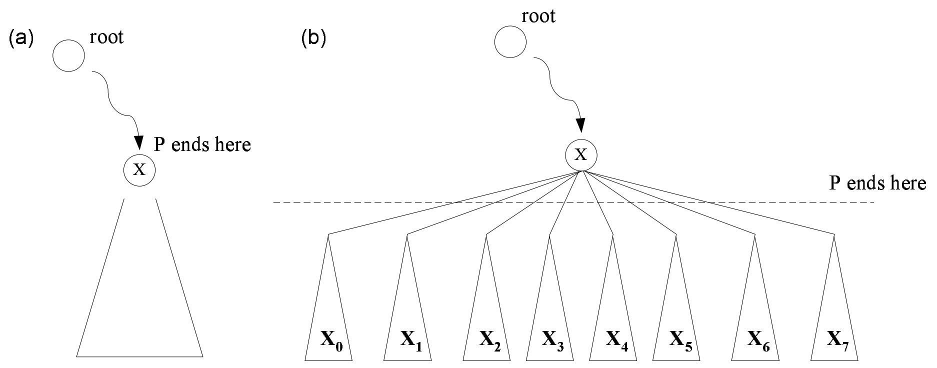

Fig. 3. Second phase of the Prefix-Matching algorithm, when (a) ‘P’ ends in a node (b) ‘P’ ends in an arc

An example of the two cases that may arise in the second phase of the algorithm is shown in Fig.3. Here X is an internal node of the DiskTrie with degree 8 and the search to find all the strings that has prefix P starts from the root. In the first case (Fig.3(a)), when P is exhausted in node X, the subtrie rooted at X holds the strings that start with P. But in the second case (Fig.3(b)), P does not end in X; instead it has some more bits, for example ‘01’, remaining after the traversal reaches X. Then the subtries X_2 and X_3 will hold the result strings as the indices of the subtries - 2 and 3 start with ‘01’ in 3-bit binary representation. Similarly, if only one bit is remaining, say 1, then the result strings will be in the subtries X_4 to X_7 in order.

Most of the procedures used in the algorithm (Fig.4) are described before while discussing the *Lookup* algorithm. The only new one is `startsWith(str1, str2)`, which checks if `str1` is a prefix of `str2`. `lNode` and `rNode` variables hold the range and `previousNode` is used to find the range when in the second phase traversal does not end in a node.

4 Analysis

As DiskTrie is an external memory data structure, the main focus of the analysis of the performance is on the number of disk accesses it takes. In addition to this, use of internal memory and processing power is also a very important issue for DiskTrie as it is specifically developed for mobile devices with very low memory

```
procedure Prefix-Matching(P) returns CollectionOfStrings
{
  previousNode = currentNode = getFromFlashDrive(ROOT_ADDRESS);
  k = currentNode.Skip;
  while (currentNode.bits > 0 & k < P.length) do
    temp = getValue(P, k, currentNode.bits);
    k += currentNode.bits;
    previousNode = currentNode;
    currentNode = getFromFlashDrive(calcAddress(temp,
     currentNode.pointer));
    k += currentNode.Skip;
  endwhile

  if (k > P.length) then
    return {};
  endif

  lNode = left-most node in the probable region;
  rNode = right-most node in the probable region;

    if (startsWith(lNode.pointer, P)) then
      return {all strings in between lNode.pointer and rNode.pointer};
    else
      return {};
    endif
}
```

Fig. 4. Prefix-Matching algorithm

and processing power. In this section, theoretical bounds are given on the space and time complexity of this data structure along with an analysis of the use of internal memory and flash storage.

Storage Requirement

First of all, DiskTrie needs a linear storage space of $\sum_{i=0}^{n} L(i) = O(n)$ bytes to save the n finite strings stored in it, where $L(i)$ is the length of the i^{th} string. Next comes the question, *how much space does it actually take to store this data structure with all its nodes into the flash?* It is known that a Patricia trie with n strings needs exactly $2n-1$ nodes. So, the number of nodes after level compression will be less than or equal to $2n-1$. In the proposed implementation each node has three fields *bits*, *Skip* and *pointer*. Of them, *pointer* with 4 bytes will be able address up to 4GB of space which is enough at present and it is known that if an LPC tree has n leaf nodes $\lceil \log\log(n) \rceil$ bits will be enough to store the number of child. And it can be shown that for storing independent random strings *Skip* will need at most $O(\log n)$ bits. Considering all these, it is clear that even after storing the DiskTrie the storage requirement is still *linear*. It should be remembered that if NAND flash is used to store the nodes, the page boundary must be maintained while storing trie nodes resulting in wastage of a few bytes per page depending on the size of the DiskTrie nodes. Without this guarantee, there is a probability that some nodes might reside across page boundaries, which will need multiple block reads in a NAND flash.

Lookup

The *Lookup* algorithm accesses disk only to fetch the nodes that are on the path to a leaf node. It means that the number is equal to the depth of that node. Hence, for random independent strings the number of disk access is bounded by $\Theta(\log^* n)$.

The storage requirement in the internal memory is minimal - only one node per iteration and some other temporary variables. And all the procedures used in the algorithm are either constant time or linear on the length of the string resulting in a linear complexity.

Prefix-Matching

In the *Prefix-Matching* algorithm, disk access occurs to retrieve the nodes of the data structure similar to *Lookup*, which is bounded by $\Theta(\log^* n)$. And in the successful case, it takes only $O(\frac{n}{B})$ disk accesses in NAND flash to retrieve the strings with given prefix P, where B is the page size of the flash memory. So, total the number of disk access is bounded by $\Theta(\log^* n) + O(\frac{n}{B})$.

The internal memory requirement is still minimal. The time complexity of the algorithm is linear and this is the combined result of placing the strings in lexicographically sorted order and providing range to search for prefix-matched strings. This effectively eliminates $O(n)$ string comparisons that would have been needed otherwise to find out where to stop retrieving.

5 Conclusion

DiskTrie is a simple but efficient implementation of trie that exploits the possibilities offered by a mobile device and its accompanying flash disk. It has a satisfactory time and space complexity. The number of disk accesses is also very low. Further improvement of its storage requirement can be achieved by removing the space wastage per page while storing the nodes. Considering the current shortage of flash-specific data structures [21], DiskTrie can be regarded as a significant step toward future.

References

1. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. Prentice Hall (1998)
2. Fredkin, E.: Trie Memory. Communications of the ACM **3**(9) (1960) 490–499
3. Reingold, E.M., Hansen, W.J.: Data Structures. Addison Wesley Publishing Company (1998)
4. Knuth, D.E.: The Art of Computer Programming. 2nd edn. Volume 3. Addion Wesley, New Delhi, India (2000)
5. Morrison, D.R.: PATRICIAPractical Algorithm To Retrieve Information Coded in Alphanumeric. J. ACM **15**(4) (1968) 514–534
6. Mccreight, E.M.: A Space Economical Suffix Tree Construction Algorithm. Journal of the ACM **23**(2) (1976) 262–272
7. Manber, U., Myers, G.: Suffix Arrays: A New Method for Online String Searches. SIAM Journal of Computing **22**(5) (1993) 935–948
8. K̈arkk̈ainen, J.: Suffix Cactus: A Cross between Suffix Tree and Suffix Array. In: Proceedings of the Annual Symposium on Combinational Pattern Matching. (1995) 191–204
9. Vitter, J.S.: Online Data Structures in External Memory. In: Lecture Notes In Computer Science, vol. 1644, London, UK, Springer-Verleg (1999) 119–133
10. Vitter, J.S.: External Memory Algorithms and Data Structures: Dealing with Massive Data. ACM Computing Surveys **33**(2) (June 2001) 209–271
11. Arge, L. In: External Memory Geometric Data Structures, Handbook of Massive Data Sets. Springer (2002)
12. Clark, D.R., Munro, J.I.: Efficient Suffix trees on Secondary Storage. In: SODA'96: Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms. (1996) 383–391
13. K̈arkk̈ainen, J., Rao, S.S.: 7. In: Full-Text Indexes in External Memory, Algorithms for Memory Hierarchies. Springer-Verlag, Heidelberg (2003) 149–170
14. Ferragina, P., Grossi, R.: The String B-Tree: A New Data Structure for String Search in External Memory and its Applications. Journal of the ACM **46**(2) (March 1999) 236–280
15. Baker, M., Asami, S., Deprit, E., Ousterhout, J., Seltzer, M.: Non-Volatile Memory for Fast, Reliable File Systems. ACM SIGPLAN Notices (1992)
16. Rosenblum, M., Ousterhout, J.K.: The Design and Implementation of a Log-Structured File System. ACM Transactions of Computer Systems **10**(1) (February 1992) 26–52

17. Chang, L.P., Kuo, T.W., Lo, S.W.: Real-Time Garbage Collection for Flash-Memory Storage Systems of Real-Time Embedded Systems. ACM Transactions on Embedded Computing Systems **3**(4) (November 2004) 837–863
18. Wu, C.H., Chang, L.P., Kuo, T.W.: An Efficient R-Tree Implementation over Flash-Memory Storage Systems. In: Proceedings of the 11th ACM international symposium on Advances in geographic information systems, New Orleans (2003) 17–24
19. Wu, C.H., Chang, L.P., Kuo, T.W. In: An Efficient B-Tree Layer for Flash-Memory Storage Systems, Real-Time and Embedded Computing Systems and Applications. Volume 2968 of Lecture Notes in Computer Science. Springer Berlin/Heidelberg (2004) 409–430
20. Lin, S., Zeinalipour-Yazti, D., Kalogeraki, V., Gunopulos, D., Najjar, W.A.: Efficient Indexing Data Structures for Flash-Based Sensor Devices. ACM Transactions on Storage (2006)
21. Gal, E., Toledo, S.: Algorithms and Data Structures for Flash Memories. ACM Computing Surveys (2005) 149–170
22. Nilsson, S., Tikkanen, M.: Implementing a Dynamic Compressed Trie. In Mehlhorn, K., ed.: Proceedings WAE'98, Germany (August 20-22 1998) 25–36
23. Andersson, A., Nilsson, S.: Improved Behaviour of Tries by Adaptive Branching. Information Processing Letters **46** (1993) 295–300
24. Devroye, L.: A note on the average depth of tries. Computing **28**(4) (1982) 367–371
25. Andersson, A., Nilsson, S.: Efficient Implementation of Suffix Trees. Software-Practice and Experience **25**(2) (February 1995) 129–141