KP-Trie Algorithm for Update and Search Operations

Feras Hanandeh[1], Izzat Alsmadi[2], Mohammed Akour[3], and Essam Al Daoud[4]
[1]Department of Computer Information Systems, Hashemite University, Jordan
[2, 3]Department of Computer Information Systems, Yarmouk University, Jordan
[4]Computer Science Department, Zarqa University, Jordan

Abstract: *Radix-Tree is a space optimized data structure that performs data compression by means of cluster nodes that share the same branch. Each node with only one child is merged with its child and is considered as space optimized. Nevertheless, it can't be considered as speed optimized because the root is associated with the empty string. Moreover, values are not normally associated with every node; they are associated only with leaves and some inner nodes that correspond to keys of interest. Therefore, it takes time in moving bit by bit to reach the desired word. In this paper we propose the KP-Trie which is consider as speed and space optimized data structure that is resulted from both horizontal and vertical compression.*

Keywords: *Trie, radix tree, data structure, branch factor, indexing, tree structure, information retrieval.*

Received January 14, 2015; accepted March 23, 2015; Published online December 23, 2015

1. Introduction

Data structures are a specialized format for efficient organizing, retrieving, saving and storing data. It's efficient with large amount of data such as: Large data bases. Retrieving process retrieves data from either main memory or secondary memory. Using data structures vary depending on the nature or the purpose of the structure. The main characteristics that use for assessing the quality of any data structure depending on the performance or the speed of storing and accessing data in those data structures. Any current natural language such as: English, French, Arabic, etc., can have number of words up to one million words although dictionaries couldn't contain all such words especially as languages continuously grow to add new words or borrow words from other languages. Current versions of Oxford English dictionary may have up to half million words. As such, a software product or web application that needs or uses a dictionary should have efficient data structures for effective: Storage, access and expansion of data or words.

The concept of data structures such as trees has developed since the 19th century. Tries evolved from trees. They have different names such as: Radix tree, prefix tree compact Trie, bucket Trie, crit bit tree and Practical Algorithm to Retrieve Information Coded in Alphanumeric (PATRICIA) [9].

In the existing general form, it is believed that tries was first proposed by Morrison [14]. Radix tree, prefix tree compact Trie, bucket Trie, crit bit tree and PATRICIA those different names may have some differences in the detail structure. For example, unlike PATRICIA tree nodes that store keys and words, with the exception of leaf nodes, nodes in the trie work merely as pointers to words.

A trie, also called digital tree, is an ordered multi-way tree data structure that is useful to store an associative array where the keys are usually strings, that comes in the form of a word or dictionaries. The word "Trie" comes from the middle letters of the word "retrieval" that was coined by Fredkin [7]. Their structure nature facilitates the process of searching or retrieving queried words quickly. A common use-case for tries was finding all dictionary words that start with a particular prefix (e.g., finding all words that start with "data"). In tries, keys were stored in the leaves and the search method involves left-to-right comparison of prefixes of the keys.

In each trie, nodes form the children that could further be parents for lower nodes. Nodes contain letters that represent keys or pointers to words (or the rest of the words) at the lowest leaf levels. In principle, each node can contain the searched for word (if it is a leaf). This can dynamically change if more words are added. Finding a word in a trie depends on the size of the tree or the number of words along with its structure. The depth of the nodes that a query could go depends on the number of words in the searched for word. If the word does not exist in the tree, the longest node sequence is performed.

Certain trie structures nature can facilitate not only quicker access and retrieval of information; they can further facilitate smooth expansion of such structures. In many cases, it is necessary for a dictionary to accept the addition of new words and hence the structure should facilitate this expansion smoothly and dynamically. This can be for either fragmentation or

for allocation/reallocation [9]. In some other cases, it may be necessary to compress, encode or encrypt data in those tree structures.

In computer science, a radix tree also Known as PATRICIA Trie or compact prefix tree, is a compressed version of a trie and space optimized data structure, in which any node that is an only child is merged with its parent. Unlike regular trees, the key at each node is compared part-of-bits by part-of-bits; where the quantity of bits in that part at that node is the radix r of the radix trie, Instead of compare the whole keys from their beginning up to the point of inequality.

Insertion, deletion, and searching operations were supported by radix trees with O(k) where k is the maximum length of all strings in the set. Unlike in regular tries, edges could be labeled with sequences of elements as well as single elements.

The branching factor was an essential in compute the time complexity of the searching tree, the branching factor is the number of children in each node of the tree, where usually every node had the same branching factor, but the difficulty occurs when different nodes at the same level of the tree have different numbers of children. In this situation, branch factor with a given depth of the tree could be defined by computing the ratio of the number of nodes at that depth with the number of nodes at the next insignificant depth.

The time complexity also depends on solution depth. Solution depth is the length of a shortest solution path. Computing the branch factor and solution depth depends on the given problem instance.

The rest of this paper is organized as follows: Section two presents' related studies to the paper subject, section three shows the proposed KP-Trie, section four presents evaluation and experimental results, Last section presents the conclusions.

2. Related Works

Some recent researches suggested that new data structures such as: Hash tables or linked structures can be suitable alternatives for tries in terms of flexibility and performance.

Askitis and Sinha [1] suggested that HAT-Trie is better alternative for trie representation, this is based on the previous approach: Bucket-trie where Buckets are sectioned using B-tree splitting. This research tends to improve Burst-tries through caching and using hash tables. Programmers used several datasets of texts for the comparison of their data structure with some known ones (i.e., known design structures for tries) and proved well in performance and memory size. Askitis and Zobel [2] declared efficient optimizing of data structures such as: Hash tables and burst tries using caching. Authors assumed that storage can be significantly reduced. Bando and Chao [3] claimed also to use trie compression for enhancing IP lookup, the proposed Flash trie data structure followed compression techniques to minimize the total size of the Trie. Behdadfar and Saidi [4] applied Trie search for IP routing table, search optimization where they tried to adjust tries to deal with large size data structures such as those of IP addresses. They tried to arrange nodes through encoding their addresses with numbers which can be reached faster based on the numbers encoded. They concluded that some methods can be optimized for search or query whereas similar methods can be optimized for an addition or update process to the Trie table.

Bodon and Ronyal [5] prepared a more modified version of trie for solving frequent itemset mining which is assumed to be the most important data mining fields. Experiments proved that the performance of tries is close to the performance of hash-trees with optimal parameters at high support threshold; however, at lower threshold the trie-based algorithm does better than the hash-tree. Moreover, tries are mainly suitable for useful implementation of candidate generation because pairs of items that produce candidates have the same parents. Thus, candidates can be easily obtained by a simple scan of a part of the trie. It gives simpler algorithms, which are faster for a range of applications and escapes the need of fine tuning by employing a self-adjusting method. Ferragina and Grossi [6] introduced the string B-Tree that is link between some traditional external-memory and string-matching data structure. It's a combination of B-Trees and Patricia Tries for internal-node indices that is made more helpful by adding extra pointers to accelerate search and renovate operations.

Fredkin [7] described the Trie structure. Fredkin said "As defined by me, nearly 50 years ago, it is properly pronounced "tree" as in the word "retrieval", at least that was my intent when I gave it the name "Trie", the idea behind the name was to combine reference to both the structure (a tree structure) and a major target (data storage and retrieval)". Fu *et al.* [8] introduced a modified LC-Trie lookup algorithm in order to enhance its performance. The idea depends on expansion and collapsing of the routing prefixes which turns them into disjoints, complete and minimal set. Subsequently, the base and prefix vectors are removed from the data structure. This result in a smaller data structure and less number of executed instructions, which in turn improves the performance, a technique called prefix transformation. Thereafter, the LC-Trie's performance is analyzed for both the earliest and modified algorithm using real and synthetically produced traces.

Heinz *et al.* [10] claimed that burst-Trie version of prefix trees is the fastest, this trie tried to further reduce the number of nodes by breaking up similar nodes that share same prefixes, the buckets or nodes were pointed out using linked lists, later papers claimed that bursts can be further reduced using

caches. The main goal or enhancement of burst Trie over traditional Trie is in minimizing the number of required search cycles to get back subject or query. Knuth [11] suggested improving performance in tries through flexible size pointers of array lists in contrast with original fixed size pointers. This requires non-root nodes to be highly occupied in order for such substitute to be competitive, nodes can share keys with neighbors rather than divide when get full. Leis *et al*. [12] produced the Adaptive Radix Tree (ART), a fast and space-efficient indexing structure for main-memory database system. A high fan-out, path compression and lazy expansion reduce the tree height and therefore lead to excellent performance. The worst-case space consumption, a common problem of radix trees, is limited by energetically choosing compact internal data structures. They compared ART with other state-of-the-art main-memory data structures. The results show that ART is much faster than a red-black tree, a cache sensitive B+ Tree and the Generalized Prefix Tree (GPT). Even though ART's performance is comparable to hash tables, it keeps the data in sorted order, which allows additional operations like range scan and prefix lookup. Mihăescu and Burdescu [13] presented a study usage of an implementation of M-tree building algorithm, the handling of M-tree structure for classification of the learners based on their final marks obtained in their respective courses. The classical building algorithm of M-tree with an original accustomed clustering procedure was implemented. The data that are managed within M-tree structure are represented by orders.

The main aim of the structure is to provide information to students and course managers regarding the knowledge level attained by students. The proposed clustering process that is used for splitting full M-tree nodes are designed to properly categorize learners; the tree manages real data representing the tests are classified according to their rank of difficulty: Low, medium and high. Mount and Park [15] presented a simple, randomized dynamic data structure for storing multidimensional point sets, called a quadtree. This data structure is a randomized, balanced variant of a quadtree data structure. In particular, it defines a hierarchical decomposition of space into cells, which are based on hyperrectangles of bounded aspect ratio, each of constant combinatorial complexity. It can be considered as a multidimensional generalization of the treap data structure of Seidel and Aragon. When inserted, points are assigned random priorities, and the tree is restructured through rotations as if the points had been inserted in priority order. Nilsson and Tikkanen [16] presented a study of order-preserving general purpose data structure for binary data, level- and path-compressed trie or LPC-trie. The structure is a compressed trie, using both level and path compression. The LPC-Trie is suitable to modern language applications with efficient memory allocation and garbage collection. Park *et al*. [17] presented the outline of a trie that is a parameter represents the number of nodes (either internal or external) with the same distance from the root. It is a function of the number of strings stored in a trie and the distance from the root. Patel and Garg [18] explained and compared different variants of B-tree and R-tree. B-tree and its variants are support point query and single dimensional data efficiently while R-tree and its variants support multidimensional data and range query efficiently. BR-tree support single dimensional, multi-dimensional and all four type of query. Reznik [19] studied the modification of digital trees (or tries) with adaptive multi-digit branching. That assumes method for selecting that helps the tries to adjust the degrees of their nodes that such selection can be accomplished by examining the number of strings remaining in each sub-tree and estimating parameters of the input distribution. This is called class of digital trees Adaptive Multi-Digit tries (or AMD-tries) and provides an initial analysis of their expected behavior in a shorter memory model.

Shafiei [20] presented a new concurrent implementation of Patricia tries (non-blocking Patricia trie) for an asynchronous shared-memory system that store binary strings using single-word Compare And Swap (CAS). This implementation provided wait-free find operations and non-blocking insertions and deletions. Also provide a non-blocking replace operation that made two changes to the trie atomically. If all pending updates were at disjoint parts of the trie, they did not interfere with one another.

Yan *et al.* [21] showed evidence how that a column is more possible to have sub-regularity than to have global-regularity. Therefore, they proposed a new compression scheme, called VParC. Moreover, evaluation experiments were performed.

3. KP-Trie

This section proposes KP-Trie data structure. We started with a brief description of indexed trees data structure and then describe the KP-Trie, its structure, and operations. Indexed trees are specialized trees data structure used to arrange data in such a way that is fast to access. Like binary search trees, indexed trees use keys to perform their operations. But unlike binary search trees which use the keys to test the different branched paths until they get into the desired node, indexed trees extract traversing path from the key itself, so the time complexity of indexed trees is constant corresponding to the length of the key. The old version of indexed trees is called 'Trie'. Radix tree is a variation of indexed trees based on Trie in which every node with only one child is merged with its child to the previous level so this will reduce the number of levels making tree traversal faster and reduce the space occupied by the tree. Like conventional trie, radix tree

structure will start with a null root, which is a little disadvantage of it because this will increase the height of the tree; increasing the height of the tree results in decreasing the speed of the process.

Unlike binary search trees family in wich each node have to keep only up to two pointers to its children nodes, Indexed trees , tries, nodes have many children to keep track of, so they have to keep more memory space to store pointers of their children nodes. In this paper we introduce a KP-Trie indexed tree data structure as an innoviative solution for this problem.

KP-Trie is a speed and space optimized trie data structure, where each node with only one child will be merged to the parent, like radix tree, also KP-trie added the concept of level grouping, also could be called horizontal grouping, where each node with N children will merge its children into just one child, which lead to minimize the required space to 1/N times. In KP-Trie, the space optimization resulted from both horizontal and vertical compression. Vertical compression is the result of merging the unique child of the node to the node itself and the horizontal compression is the result of merging any N children into one child node, making them to share the same memory space occupied by only one child to store pointers to their children in the next level. This eventually will lead to shorten the path and lead for speed optimization. In addition, the root node in KP-Trie is not null as it in the other variations of indexed trees; which also leads to shorten the number of tree level by one level more. The KP-Trie starts with a root node containing an array of characters with size 26, which have all the alphabetic from (a-z), the reason of using this array is when adding a new word to the tree the letters of the word already present in the array, this will reduce the time of searching for characters on other types of trees. KP-Trie considered the disadvantage of the radix tree.

The complexity of KP-Trie is (B+N+M), when B is the branching factor, N is the number of children and M is another needed space to store data and keep track of keys; Then it's obvious that the KP-Trie is more space optimized than radix tree, when the number of children is equal to the branching factor, then the space occupied by KP-tree for N children is 2N+M.

In the indexed trees like the variations of the Radix tree and the proposed KP-Trie; the transformation from the parent node to which child node would be deterministic, because it's indexed by the partition of key corresponding to the child node and there is no need for comparison or for testing different nodes. So the traversal path length would be as maximum as the length of the key, in some times would be less as result of optimization process resulting from vertical compression by which the number of levels would be minimized .

The KP-Trie complexity is O(1) as it in the radix tree [12], because it represents the depth of the tree which is constant related to the length of the key and independent of the variable of amount of data in the KP-Trie or radix tree. Moreover, the number of iterations for any operation in the KP-Trie is less than the number of iterations in the radix trie by one; because the number of levels in KP-Trie is less than those in radix tree by one resulting slightly in more speed.

Insert and search operations will be considered in the following descriptions:

1. Insert: Unlike radix tree the root in KP-Trie is not empty; it contains an array of 26 characters so, this resulted in minimizing the memory space. First case is if the root is empty, insert an array of 26 character in the root node, so when insert new word there is no need to initiate new node because the root is already has all needed characters. KP-Trie starts searching for the character of the inserted word begins from the first char, second char and so on, Figures 1 and 2 refers to example 1 where the word "romane" is inserted to the root:

```
1  insert (key, data)
2
3  set I= 0
4  if root is empty then
5      set root of type Node
```

Figure 1. Algorithm of first case.

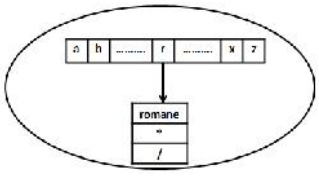

Figure 2. Representation of example 1.

Second case, the root is not empty in this case KP-Trie structure has two sub-cases; The first case insert a new word that has a common characters with an existing word in the Trie, KP-Trie look at the common characters and separated them in inner node attached with the first character that was located inside array, Figure 3 refers to example 2 where the word "romulus" is inserted to the trie and it has a common characters "rom":

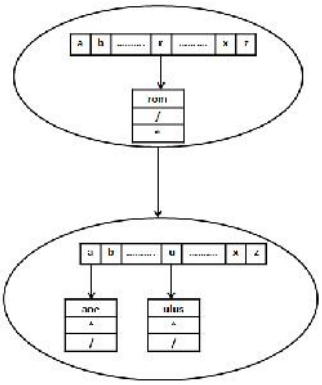

Figure 3. Representation of example 2.

The second case is insert a new word that has no-common characters in the Trie, KP-Trie separates it in other inner node; In this case KP-Trie doesn't need to create new node because all the needed characters are already exist in the root, Figure 4 refers to example 3 where the word "Jordan" is inserted to the trie:

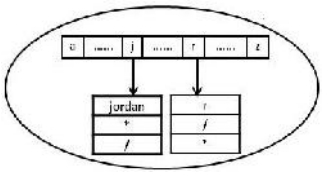

Figure 4. Representation of example 3.

2. Search: The search operation in KP-Trie starts from the root node by checking if the letters are matched together and then check the length of the matching result if the length is not equal then it will recursively search the tree again until the length of the key and the founded sub-key are equal each other. Figure 5 shows example 4 where the search operation is performed to find the word "romane":

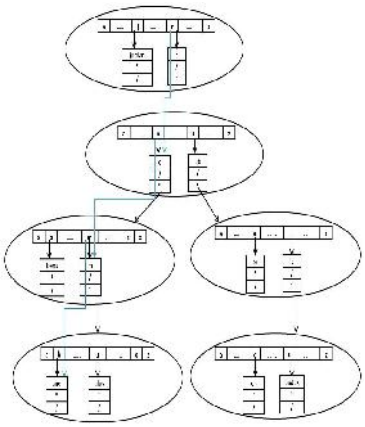

Figure 5. Representation of example 4.

4. Evaluation and Results

KP-Trie and Radix tree have been implemented using java programming language to compare the space needed for both algorithms, a text file containing (67044) words was used with the algorithms to perform the testing process, the following results have been found out:

Figure 6 and Table 1 shows that radix tree will consume more memory space than KP-Trie when insert large amount of data.

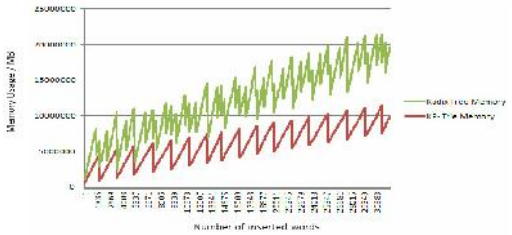

Figure 6. Memory usage for KP-Trie and radix tree.

Table 1. Memory usage for KP-Trie and radix tree.

Number of Inserted Data	KP-Trie Memory Usage	Radix Tree Memory Usage
1	551408	502080
1000	3084112	3758424
5000	5292168	4526976
10000	4551312	4376912
50000	1.39E+07	1.47E+07
67044	1.81E+07	2.00E+07

Also a comparison of the number of nodes that each algorithm takes has been done, to determine which algorithm will be more space and speed optimized and which will reduce the branching factor, Figures 7 and 8 refers to example 5 which shows the final results after inserting the words ("romane", "romulus", "robens", "ruber", "rubicon", "rubicundus", "Jordan") to both algorithms:

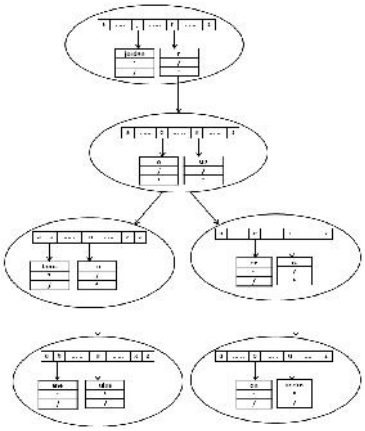

Figure 7. Representation of example 5 (KP-Trie).

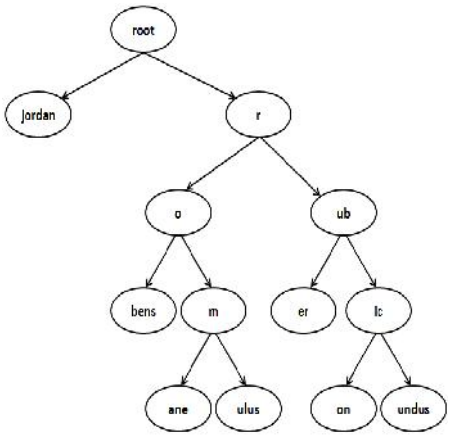

Figure 8. Representation of example 5 (radix tree).

It is obvious that radix tree has created more nodes that KP-Trie. Also a test on large amount of data for both algorithms has been performed, Table 2 represent the final result.

Table 2 shows that the number of nodes created in KP-Trie is much lesser than the nodes created in radix tree, that's because KP-Trie does not need to store data in different nodes; it stores all the children of a parent in one child which will lead to minimize the space, the number of created nodes and the branching factor.

Table 2. Number of created nodes for KP-Trie and radix tree.

Number of Word	KP-Trie Node	Radix Tree Node
1	1	2
500	275	687
1000	537	1351
5000	2716	6800
10000	5085	13092
15000	7701	19682
20000	10335	26186
25000	12969	32728
30000	15309	39048
35000	18082	45821
40000	20486	52194
45000	22976	58784
50000	25603	65444
55000	28202	71955
60000	30664	78344
65000	33179	84911
67044	34276	87695

5. Conclusions

This paper has introduced the KP-trie, a speed and space optimized trie data structure for storing a set of strings, KP-trie is an advanced version of radix tree that solve the disadvantage of radix tree, and modify it to be more space optimized. Vertical and horizontal compressions have been employed. Experiments proved that performance of the KP-trie is more efficient than radix tree and more space and speed optimized. Moreover, KP-trie is particularly more efficient with large amount of data comparing with radix tree.

References

[1] Askitis N. and Sinha R., "HAT-trie: A Cache-Conscious Trie-based Data Structure for Strings," *in Proceedings of the 30th Australasian Conference on Computer science*, pp. 97-105, 2007.

[2] Askitis N. and Zobel J., "Redesigning the String Hash Table, Burst Trie, and BST to Exploit Cache," *Journal of Experimental Algorithmics*, vol . 15, no. 1, 2011.

[3] Bando M. and Chao J., "FlashTrie: Hash-based Prefix-Compressed Trie for IP Route Lookup Beyond 100Gbps," *in Proceedings of IEEE Communications Society Subject Matter Experts for publication in the IEEE INFOCOM*, San Diego, pp. 1-9, 2010.

[4] Behdadfar M. and Saidi H., "The CPBT: A Method for Searching the Prefixes using Coded Prefixes in B-tree," *in Proceedings of the 7th International IFIP-TC6 Networking Conference on AdHoc and Sensor Networks, Wireless Networks, Next Generation Internet*, Singapore, pp. 562-573, 2008.

[5] Bodon F. and Ronyal L., "Trie: An Alternative Data Structure for Data Mining Algorithms," *Mathematical and Computer Modelling*, vol. 38, no. 7-9, pp. 739-751, 2003.

[6] Ferragina P. and Grossi R., "the String B-Tree: A New Data Structure for String Search in External Memory and its Applications," *Journal of the ACM*, vol. 46, no. 2, pp. 236-280, 1999.

[7] Fredkin E., "Trie Memory," *Communications of the ACM*, vol. 3, no. 9, pp. 490-499, 1960.

[8] Fu J., Hagsand O., and Karlsson G., "Improving and Analyzing LC-Trie Performance for IP-Address Lookup," *Journal of Networks*, vol. 2, no. 3, pp. 1-10, 2007.

[9] Hanandeh F., Alsmadi I., and Kwafha M., "Evaluating Alternative Structures for Prefix Trees," *in Proceedings of World Congress on Engineering and Computer Science*, San Francisco, pp. 109-114, 2014.

[10] Heinz S., Zobel J., and Williams H., "Burst Tries: a Fast, Efficient Data Structure for String Keys," available at: http://goanna.cs.rmit.edu.au/~jz/fulltext/acmtois02.pdf, last visited 2002.

[11] Knuth D., *The Art of Computer Programming Sortiand Searching*, Redwood City, USA, 1998.

[12] Leis V., Kemper A., and Neumann T., "The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases," available at: http://db.in.tum.de/~leis/papers/ART.pdf, last visited 2013.

[13] Mihăescu M. and Burdescu D., "Using M Tree Data Structure as Unsupervised Classification Method," *Informatics*, vol. 36, pp. 153-160, 2012.

[14] Morrison D., "PATRICIA-Practical Algorithm To Retrieve Information Coded in Alphanumeric," *Journal of the ACM*, vol. 15, no. 4, pp. 514-534, 1968.

[15] Mount D. and Park E., "A Dynamic Data Structure for Approximat Range Searching," *in Proceedings of the 26th Annual Symposium on Computational Geometry*, Snowbird, pp. 247-256, 2010.

[16] Nilsson S. and Tikkanen M., "An Experimental Study of Compression Methods for Dynamic Tries," *Algorithmica*, vol. 33, no. 1, pp.19-33, 2002.

[17] Park G., Hwang H., Nicod`eme P., and Szpankowski W., "Profiles of Tries," *Society for Industrial and Applied Mathematics*, vol. 38, no. 5, pp. 1821-1880, 2009.

[18] Patel P. and Garg D., "Comparison of Advance Tree Data Structures," *International Journal of Computer Applications*, vol. 41, no. 2, pp. 11-21, 2012.

[19] Reznik Y., "Some Results on Tries with Adaptive Branching," *Theoretical Computer Science*, vol. 289, no. 2, pp. 1009-1026, 2002.

[20] Shafiei N., "Non-blocking Patricia Tries with Replace Operations," *in Proceedings of the 33rd International Conference on Distributed Computing Systems*, Toronto, pp. 216-225, 2013.

[21] Yan K., Zhu H., Lu K., "VParC: A Compression Scheme for Numeric Data in Column-oriented

Databases," *The International Arab Journal of Information Technology*, vol. 13, no. 1, published online, 2016.

Feras Al-Hanandeh is currently an Associate Professor at the Prince Al Hussein Bin Abdullah II, Faculty of Information Technology, Al-Hashemite University, Jordan. He obtained his PhD degree in Computer Science from the University Putra Malaysia, Malaysia, in 2006. His current research interests include distributed databases, parallel databases focusing on issues related to integrity maintenance, transaction processing and query processing.

Izzat Alsmadi is a PhD in Software Engineering from NDSU, 2008. Currently; he is an Associate Professor in CIS Department at Boise State University, USA.

Mohammed Akour is an Assistant Professor in the Department of Computer Information System at Yarmouk University. He got his Bachelor (2006) and Master (2008) degree from Yarmouk University in Computer Information System with Honor. He joined YU again in April 2012 after graduating with his PhD in Software Engineering from NDSU with Honor.

Essam Al Daoud received his BSc degree from Mu'tah University, MSc degree from Al al-bayt University and his PhD degree in Computer Science from University Putra Malaysia in 2002. Currently, he is an associate professor and the chairman of Computer Science Department at Zarka University. His research interests include machine learning, soft-computing, cryptography, bioinformatics, quantum cryptography, quantum computation, DNA computing, nano-technology.