International Journal of Database Theory and Application
Vol.7, No.3 (2014), pp.13-22
http://dx.doi.org/10.14257/ijdta.2014.7.3.02

A Succinct String Dictionary Index in External Memory

Guoqing Zhang[1], Mei Rong[2*] and Guangquan Zhang[1,3]

[1]School of Computer Science & Technology,
Soochow University, Suzhou, 215006, China
[2]Shenzhen Tourism College, Jinan University, Shenzhen, 518053, China
[3]State Key Laboratory of Computer Science, Institute of Software,
Chinese Academy of Science, Beijing, 100190, China
**rongmei@sz.jnu.edu.cn*

Abstract

With the coming of the big data age, more and more string dictionaries need to be processed. The existing string dictionary indexes are either too space-consuming, or lack of locality of reference, making them inapplicable in the external memory environment. Targeted with these problems, first we design a new succinct representation of Patricia trie using LOUDS encoding. Then applying it to external memory indexing problem, we propose a new string dictionary index SB-trie, which is not only succinct on space, but also has good locality of reference, making it I/O efficient in external memory environment. Experiments show that SB-trie consumes less space and has greater searching performance in disk environment.

Keywords: *String Dictionary; Succinct Data Structure; Trie; Big Data Processing*

1. Introduction

Recently the rapid development of Internet and mobile technology leads us to the big data age. In the big data age more and more data need to be processed, especially text data. As the basis of text index, string dictionary index is ubiquitously applied in fields like RDF graph, IP datagram classification, search engine and bioinformatics computing *etc*.

Confronted with the challenge of large scale text data, there are 2 categories of solutions. The first is to design a more efficient external memory data structure, increasing the locality of reference. All data is stored in external memory and the needed data is fetched into main memory in little chunks on demand so that the I/O operations are efficient. The second is to compress the data so that under the same resources condition more data can be stored and processed.

On the external memory data structure category, the followings are the recent progresses. Ferragina etc., combining Patricia trie and B+tree, proposed String B-tree [1], which resolves the decrease of performance when the string keys are too long. Because of the independent storage of label strings, each search operation in the String B-tree node needs 2 I/O operations, which worsens the performance when string keys are shorter than 1000 bytes. And because of the use of implicit pointers when storing trie, the space consumption is still too large. Askitis *etc*. adapted the Burst trie to external memory and proposed the B-trie [2]. It adopts a 2 levels structure, the first or root level is an array based trie, the second or leaf level uses a simple mapping structure based on binary search algorithm. Normally the leaf nodes are not fully filled, the space efficiency is not as well as B+tree. A solution commonly used in industry is to compress the B+tree node using front coding to achieve the overall space reduction. The Cache Oblivious String B-tree [3] proposed by Ferragina *etc.* is theoretically analyzed and proved that it not only has good locality of reference characteristics, but also is compressed. But due to the complexity of the structure and the related algorithms, it has not been implemented and practically tested. The common problem of above indexes is the space consumption.

On the compressed index category, there are also some progresses recently. Klein *etc*. combined front coding and Huffman coding and proposed a new method to match string characters in compressed state directly[4]. Grossi *etc.* proposed path decomposed trie to achieve space compression [5]. Arz *etc.* proposed a method to compress string dictionary based on the classical LZ compression algorithm [6]. Brisaboa *etc.* surveyed 4 methods to compress the index of string collection and conducted a full experiments to compare the performance and trade-offs of these methods [7]. But all the methods are based on the main memory and weak on locality of reference, making them unable to adapt to external memory easily.

Targeted with the problems of these indexes, we first design a new method to succinctly represent a Patricia trie. Then applying it to the external memory index problem, we propose a new succinct string dictionary index in external memory, Succinct B-trie (SB-trie for short), which not only has great locality of reference, but also is succinct on space consumption. Experiments show that compared with existing indexes, this index consumes much less space and has good search performance.

In the following sections of this paper, we will explain the details of the new data structure and the relevant algorithm. Section 2 is the introduction of some basic data structures and algorithms used in SB-trie. Section 3 introduces the succinct representation of Patricia trie and the relevant algorithms. Combining things introduced in Section 2 and 3, the SB-trie and its algorithm is described in Section 4. Section 5 is the experiment and the analysis. Section 6 concludes this paper.

2. Basic Data Structures and Algorithms

2.1. Bit array

Suppose B[1, n] is a bit array of length n, there are following operations:

- B[i]: the i-th bit in the bit array, $1 \leqslant i \leqslant n$.
- Rank(B, i, b): the count of bit b in sub-array B[1, i], $1 \leq i \leq n$.
- Select(B, i, b): the position of the i-th bit b in the bit array B.

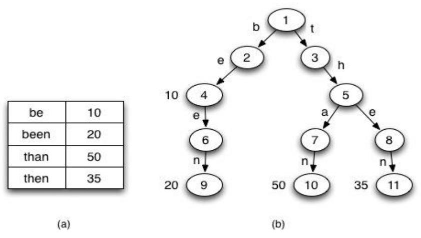
It is first proposed by Jacobson[8], then enhanced by Raman[9] etc. All the operations only cost o(n) extra space and O(1) time complexity.

2.2. Trie

Trie is an ordered multi-way tree, node of which consists of a value and multiple branches. Every branch corresponds to a distinct character label. String key is stored on the path from the leaf node to the root node and the corresponding value is stored in the leaf node.

Figure 1 is a typical trie. String keys sharing the same prefix cluster to share nodes. Node sharing lessen node counts and reduces the space consumption. The effect on space-reducing is really similar to the front coding, but the difference is that trie supports fast search while front coding not. Another key characteristic of trie is that trie supports searching pattern P in O(|P|) time complexity, namely it depends only on the length of the pattern to be searched and is independent of the key count stored in the trie.

Given a string key K_i, K_i can be a prefix of another string key K_j, so the corresponding value of a string key can occur at any node in the trie. In order to avoid ambiguity, we define tail node as follow:



be	10
been	20
than	50
then	35

Figure 1. (a) A Simple String Dictionary, (b) The Corresponding Trie.

Definition 1: the tail node of a string key in a trie is the node from which to the root node stored the string key. The value of the string key stored right in the tail node.

Algorithm 1 is the exact matching algorithm of trie. The invariant of the loop in step 3-7 is that the prefix *P[1, i]* is in the trie and *node* is the tail node of the prefix. Step 8 test the value of the node to determine whether *P* is truly in the trie, or *P* is just a prefix of some other string keys. Taking the trie in Figure 1 for example, given *P* = than, *node* will be node 1, 3, 5 ,7 10 in order, finally the value corresponding to *P* will be found in node 10.

Algorithm 1: Exact Match
Input: the trie rooted at *root*, the pattern to be searched *P*.
Output: the value corresponding to *P*, null if not exists.

```
1)  node := root;
2)  i := 1;
3)  WHILE i <= |P|
4)      node := find the branching node corresponding to character P[i] in node.
5)      IF node is null, RETURN null.
6)      i := i + 1
7)  END WHILE
8)  RETURN the value in node.
```

Each node in a trie corresponds to a distinct string, so given a node, we can compute the corresponding string. Algorithm 2 is it. The invariant of the loop in step 3-8 is that the concatenated string from node *tail* to node *node* is the string *key*. Let's see the example in Figure 1. Given *tail* is node 11, *node* will be node 11, 8, 5, 3, 1 in order. Prepending each character in *key*, the resulting *key* is 'then'. Same as Algorithm 1, this algorithm has time complexity O(|P|).

Algorithm 2: ComputeStringKey
Input: a trie rooted at *root,* a node in the trie *tail.*
Output: the string key *P* corresponding to node *tail*.

```
1)  node := tail
2)  key := “”
3)  WHILE node != root
4)      parent := Parent(node)
5)      ch :=  find the label character associated with node in parent.
6)      key := ch + key
7)      node := parent
8)  END WHILE
9)  RETURN key
```

2.3. The LOUDS Representation of Trie

The traditional pointers based representation of trie costs too much space, especially on 64 bit computers. The LOUDS (Level-Order Unary Degree Sequence) [11] representation is a new succinct representation of trie. Next is the details of the representation.

Given a trie node with 3 branches, the node can be represented by code 1000. The first 1 represent the node itself and the following 3 0s represent 3 branches of the node. Traversing the trie in level order and concatenating the code of each node, we get a bit array. For the convenience of computing, a super root will be added as the parent node of trie root node. So code 10 will be appended to the bit array. The resulting bit array is just the main part of the LOUDS representation of the trie.

The bit array only records the structure of the trie, we need more data structures to record the remaining information:

- Labels: an array of characters. Used to record the character labels of each trie node. Also concatenated in level order of the trie nodes.
- Values: an array of integers. Used to record the values of each trie node in level order.

Making use of the characteristic of LOUDS representation, the following operations can be supported at the cost of O(1) time complexity. Here we identify each trie node by the first bit 1.

- Branch(id, i), the i-th branching node of the node id: louds.select1(louds.rank0(id + i)).
- Parent(id), the parent node the node id: louds.select1(louds.rank1(louds.select0(louds.rank1(id) - 1))).
- Degree(id), the degree of node id: louds. select1(louds.rank1(id) + 1) - id - 1.

3. The Succinct Representation of Patricia Trie

This section first introduces the exact match and lower bound algorithm of Patricia trie, which will be used in the SB-trie introduced in the next section. We then introduce the succinct representation of Patricia trie and the branch matching algorithm in the succinct representation.

3.1. Patricia Trie

Patricia trie is a special trie, it enhances the traditional trie on the space consumption aspect. The traditional trie consists of many non-branching child nodes, which have no contribution to index when searching. When the non-branching nodes becoming a significant fraction, the overall space is wasted by the large portion of pointers. Patricia trie collapses the chain of non-branching nodes of traditional trie into a single branch node with all the character labels concatenated into a string label. Figure 2 is the Patricia trie transformed from the trie in Figure 1.

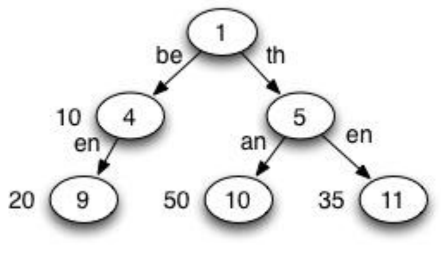


Figure 2. The Patricia Trie of the Trie in Figure 1

The exact match algorithm of Patricia trie is similar to trie's. The difference is that now the branch match is against string label instead of character label. Algorithm 3 is the algorithm. Started from the root node, the pattern is used to match the branch nodes continuously. The invariant of the loop is that the prefix P[1, i – 1] equals the string concatenated from the string labels from *node* to *root.* See the example in Figure 2. Given P = than, *node* will be node 1, 5, 10 in order. Finally the value is found in node 10. The time complexity is O(|P|) as well.

Algorithm 3: exact match in Patricia trie
Input: a Patricia trie rooted at *root*, the pattern to be searched *P*.
Output: the value corresponding to *P*, null if not exists.

```
1)  node := root
2)  i := 1
3)  WHILE i < |P|
4)      node, i := MatchChild (node, P, i)
5)      IF node is null, RETURN null
6)  END WHILE
7)  RETURN the value in node.
```

Besides the exact match algorithm, there will be a lower bound match algorithm needed.

Definition 2: Given a trie constructed from string key collection $S = \{s_1,\ s_2,\ ..,\ s_n\}$, the string keys are sorted in lexicographically ascending order. Given the pattern to be searched P, the lower bound match finds the first string key not less than P in S.

The lower bound match makes use of the following property:

Property 1: The string keys printed in depth first order of the trie are in lexicographically ascending order.

The definition and property can be extended to Patricia trie naturally. Algorithm 4 is the lower bound match algorithm for Patricia trie. The first part of the algorithm is same as the exact match algorithm. After the loop in step 3-7, the string in *node* is the longest exact matched string in the Patricia trie. At this time, the Patricia trie is split into 3 parts. The first part is the sub-tree rooted at *node*. And the remaining part of Patricia trie was split into 2 parts by the path from *node* to *root*. All string keys in the left part are lexicographically less than *P*, while all string keys in the right part are greater than *P*. Then there are following cases:

- $i > |P|$, namely *P* is exactly matched, then the least string key in the sub-trie rooted at *node* is the result.
- $I <= |P|$, namely the prefix $P[1, i-1]$ is exactly matched. Then we need to find the branching node *next* whose string label is the first label not less than $P[i, |P|]$ via LowerBoundChild operation.
  - If *next* is found, then the least string key in the sub-trie rooted at *next* is the result.
  - If *next* is not found, namely all string keys in the sub-trie rooted in *node* are less than *P*. then the least string key of right part of the Patricia trie is the result.
    - If not found, namely the right part is empty. Matching failed.
    - If found, it is the result.

Algorithm 4: LowerBound

Input: a Patricia trie rooted at *root*, the pattern to be searched *P*.

Output: the first string key not less than *P* and the corresponding value. null if not found.

```
1)  node := root
2)  i := 1
3)  WHILE i <= |P|
4)      next, i := MatchChild (node, P, i)
5)      IF next is null, BREAK
6)      node := next
7)  END WHILE
8)  IF i <= |P|
9)      next := ChildLowerBound(node, P[i, |P|])
10)     IF next is null
11)             next := GeneralizedSibling(node)
12)     IF next is null, RETURN null
13)     node := next
14) node := LeftMostNode(node)
15) RETURN the string key and value in node
```

3.2. The Succinct Representation of Patricia Trie

Though Patricia trie tries to reduce space cost, it can't be represented succinctly and easily due to the label of each branch is variable length string. We decompose a Patricia trie to 2 tries, which can be represented by LOUDS encoding method.

Suppose the string label of a Patricia trie node is s = as’, a is a character and s’ is the remaining string of s excluding a. Temporarily ignore the s’ part of label, Patricia trie can be represented by LOUDS representation of trie. All the non-empty s’ forms another string set, which can be represented by another trie, which we call label trie, which can also be represented by LOUDS representation. Then the missing s’ information can be represented by a integer node identifier of the label trie. Just like to represent the value of each trie node, same method can be applied to the node identifier of label trie. This results in an array of integer called Links.

Because the string label of Patricia trie is split into 2 parts, the branch matching algorithm also needs 2 steps, the first step is same as traditional trie matching, to find the branch node corresponding to the first character. The second step is to compute the s’ of the branch node and verify that s’ is exactly matched as well. Algorithm 5 is the pseudo-code.

Algorithm 5: MatchChild
Input: the current node *id*, the whole pattern to be matched *P*, the current position of *P* to be matched *i*.
Output: the branching child node of node *id* corresponds to *P[i, |P|]*, and the new position.

```
1)  offset := louds.rank0(id)
2)  degree := Degree(id)
3)  rank := find position of P[i] in character array Labels[offset, offset + degree – 1]
4)  IF rank is null, RETURN null, i.
5)  child := Branch(id, rank)
6)  label := compute the s’ part of node child
7)  IF label is prefix of P[i + 1, …]
8)      RETURN child, i + length(label)
9)  RETURN null, i
```

Algorithm 6 is the ChildLowerBound algorithm. The first part is similar with Algorithm 5, the steps of 7-10 ensure that the string label is not less than *P*. If it is less than *P*, find the right sibling node of the child node.

Algorithm 6: ChildLowerBound
Input: current node *id*, the pattern to be searched *P*.
Output: the first child node whose string label is not less than *P*.

```
1)  offset := louds.rank0(id)
2)  degree := Degree(id)
3)  rank, ch := find the first character not less than P[1] in character array Labels[offset, offset + degree – 1],
    return the position and the character.
4)  IF rank is null, RETURN null
5)  child := Branch(id, rank)
6)  label := compute the s’ part of string label of node child
7)  label := ch + label
8)  IF P <= label
9)      RETURN child
10) RETURN Sibling(child)
```

4. SB-trie

This section introduces how to integrate the succinct representation of Patricia trie and the relevant algorithms to become the whole SB-trie succinct index in external memory.

4.1. The Construction of SB-trie

SB-trie’s overall structure is similar to B-trie. There is one central root node and a number of leaf nodes. Each node is logically a Patricia trie decomposed into a index trie and a label trie. Then all the label tries are merged together into a unifying label trie. So SB-trie is composed of a root index trie, a number of leaf index trie and a unifying label trie. And all tries are represented by the LOUDS succinct representation.

The details of construction procedure are as follow. Suppose the input string key collection is $K = \{K_1, .., K_n\}$, the keys are sorted in lexicographically ascending order. First partition K into several groups, each of which contains consecutive keys. The string keys of each group are used to construct a Patricia trie which is transformed into a index trie and a label trie. And each group becomes a leaf node of SB-trie. Then we take the greatest string key of each group as the representative of the node, insert it into a new string dictionary along with the identifier of the responding leaf node. The new string dictionary is used to construct another Patricia trie and then transformed into a index trie and label trie. The index trie becomes the root of the SB-trie. Finally we merge all the label tries into a unifying label trie and stored in LOUDS representation. Figure 3 is a simple diagram of the overall structure of SB-trie.

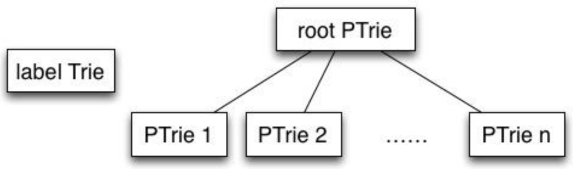


Figure 3. The Layout of SB-trie

4.2. Search in SB-trie

Algorithm 7 is the pattern searching algorithm of SB-trie. First in step 1, a lower bound match introduced in Section 3.1 is performed to find the leaf node in which *P* possibly occur. If the leaf node is not found, it means *P* is greater than all the string keys stored in the SB-trie, search failed and null is returned. Otherwise the leaf node is found, then the leaf node is fetched into main memory and the exact match introduced in Section 3.1 is performed to find whether *P* really occurs in it. If we cache the root node in main memory, the search algorithm only incurs one I/O operation, so the time complexity is O(1) in external memory model.

Algorithm 7: Find
Input: the root of SB-trie *root*, the pattern to be searched *P*
Output: the value corresponding to *P* if exists, null otherwise.

```
1)    offset := LowerBound(root, P)
2)    IF offset is null, RETURN null
3)    leafNode := fetch the leaf node at offset into main memory.
4)    RETURN ExactMatch(leafNode , P)
```

5. Experiments

We implemented SB-trie and conducted experiments to compare the time and space performance with other existing string dictionary indexes. We adopt the layout used in paper[4] to implement a simple and efficient B+tree for string keys. For front coded B+tree, we just compress each B+tree node and we adopt the one for front coded data structure in paper[6] as search algorithm. For String B-tree we use an open source static implementation[11]. At last, Leveldb is a lightweight key/value store developed by Google. The SB-trie implementation makes use of the bit array with rank/select operation support from sdsl and DAC code is adapted and optimized from the one in paper.

5.1. Experiment Environment

The experiment platform is as follow: Intel Core 2 Duo 2.4Ghz CPU, 2G RAM, 160G 5400rpm hard drive, Ubuntu 12.04 64bit LTS OS, FAT32 file system. And all code is written in C/C++ and compiled by g++ 4.6.3 with –O3 optimization.

The data used is the titles of all English Wikipedia entries on 2012/12/01, totally 9864458 items and 201.7MB. it is referred as 10000 kilo items approximately in the following figures. The other smaller scale data are extracted from the beginning part of it.

5.2. Experiment Results and Analysis

A comparison experiment is performed with the 5 indexes run on 5 different scale of data. In order to reduce the disturbance of cache, before each run of program, the experiment partition is remounted and the RAM is shuffled by running an auxiliary program. We run the search operation with string keys from the original data set as the search patterns. Every 1000th string key of a set of string keys is used to search on the index of that set of string keys. The total run time is averaged to the search time cost per operation. Figure 4(a) shows the search time results, and Figure 4(b) shows the space ratio of each index on each data set. Next is the analysis of each index.

B+tree: because string keys are stored directly in B+tree nodes without any compression, the space consumption is roughly 140% of the size of original data set, regardless of the scale of original data. On the search time aspect, due to multi-level structure, with the increase of the scale of data, the search time cost becomes longer.

Front coded B+tree: due to the adoption of front coding method to compress B+tree nodes, the space consumption ratio decreases with the increase of the scale of data set. On the search time aspect, though the front coding method increase the computation complexity, the decreased space consumption reduced the need to perform I/O operations. The experiment results show that the reduced I/O time complements the increased CPU time, so the total performance is increased.

String B-tree: similar to B+tree, it includes multi-level index structure. The use of implicit pointer based trie makes it too costly on space, even worse than B+tree. Increased space consumption make it need to perform more I/O operation to fulfill search operation. Considering it is designed for long length string keys, and the string keys used in the experiments are mostly short, shorter than 1000 bytes, so the results are expected.

Leveldb: it use the Snappy general purpose compression library to reduce the index space. The effect is similar to front coded B+tree. On search time aspect, it is slower than SB-trie because it supports dynamic operations and other functionalities.

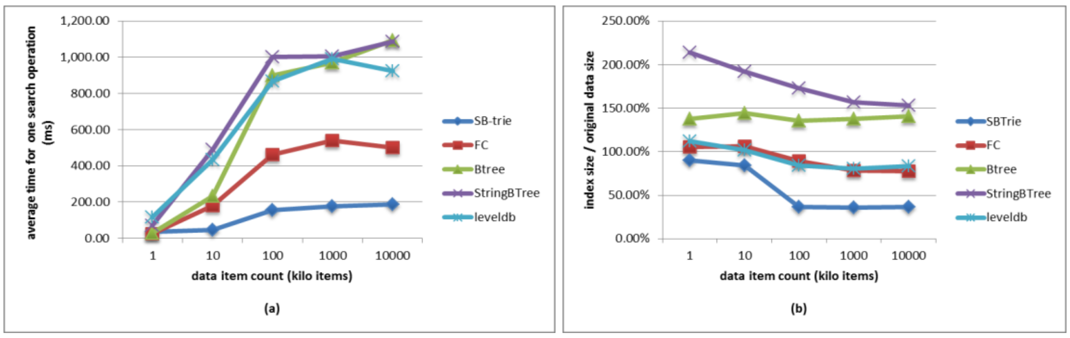


Figure 4. Time(a) and Space(b) Performance on Different Scale of Data Set

SB-trie: the experiments results show that the SB-trie outperforms all other indexes on the space aspect. When the scale of data set increases, the space ratio is close to 36%. The index of 201.7MB of original data set only costs 73.1MB. There are 2 main aspects achieving this result. The first is the use of succinct representation of Patricia trie in SB-trie nodes. Patricia trie reduces space cost by sharing same prefix of string keys, and the LOUDS use bit array to record the structure of trie and avoid using implicit pointers. The combination of these 2 methods greatly reduce the cost of the index. Lastly we merge all the label trie into a unifying label trie to reduce redundancy of storing the same string labels more than once. This method reduce the space cost even more.

On search time aspect, though the succinct representation of Patricia trie has greater search time complexity, the greatly reduced space cost reduce the node count, making it possible to use only one root node. So each search operation only needs one I/O operation. The reduced I/O time significantly complements the increased CPU time, so when the scale of data set become greater than 10000 items, the search efficiency outperforms other indexes as well.

6. Conclusion

In the big data age, large scale of data has been a challenge of string dictionary index problem. The existing string dictionary indexes are either too space-consuming, or inefficient on I/O operation when working in external memory, which makes it difficult to rise to the challenge. Targeted with these problems, first we design a new succinct representation of Patricia trie. Then applying it to external indexing algorithm, we propose a new string dictionary index data structure SB-trie, which is not only succinct on space, but also has good locality of access, making it I/O efficient in external memory environment. Experiments show that SB-trie consumes less space and has greater searching performance in disk environment. But SB-trie is only a static index, it does not support dynamic operations like insertion or deletion. So how to adapt it to support dynamic operation will be a more challenging problem.

Acknowledgements

We are grateful to the authors of the open source libraries for kindly sharing them. The work of this paper is supported by the Natural Science Foundation of Jiangsu, China (Grant No. BK2011281) and the Applied Fundamental Research Program of Suzhou, China (Grant No. SYG201241).

References

[1] P. Ferragina and R. Grossi, "The string B-tree: a new data structure for string search in external memory and its applications", Journal of the ACM (JACM), vol. 46, no. 2, **(1999)**, pp. 236-280.
[2] N. Askitis and J. Zobel, "B-tries for disk-based string management", The VLDB Journal, vol. 18, no. 1, **(2009)**, pp. 157-179.
[3] P. Ferragina, R. Grossi, A. Gupta, R. Shah and J. S. Vitter, "On searching compressed string collections cache-obliviously" in Proceedings of the twenty-seventh ACM SIGMOD-SIGACT- SIGART symposium on Principles of database systems, ACM, **(2008)**, pp. 181-190
[4] S. T. Klein and D. Shapira, "Compressed Matching in Dictionaries", Algorithms, vol. 4, no. 4, **(2011)**, pp. 61-74.
[5] R. Grossi and G. Ottaviano, "Fast compressed tries through path decompositions" in Proceedings of the 14th Meeting on Algorithm Engineering & Experiments, ALENEX, Kyoto, Japan: **(2012)**, pp. 65-74
[6] J. Arz and J. Fischer, "LZ-Compressed String Dictionaries", CoRR, **(2013)**.
[7] N. Brisaboa, R. Cánovas, F. Claude, M. Martínez-Prieto and G. Navarro, "Compressed string dictionaries", Experimental Algorithms, **(2011)**, pp. 136-147.
[8] G. Jacobson, "Space-efficient static trees and graphs", Foundations of Computer Science, **(1989)**, pp. 549-554.
[9] R. Raman, V. Raman and S. R. Satti, "Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets", ACM Trans. Algorithms, vol. 3, no. 4, **(2007)**, pp. 43-68.
[10] D. Arroyuelo, R. Cánovas, G. Navarro and K. Sadakane, "Succinct trees in Practice", Proceedings of the Twelfth Workshop on Algorithm Engineering and Experiments, ALENEX, Austin, Texas, USA, **(2010)**, pp. 84-97.
[11] J. Lemoine, "Text Mining Software - C++ string b-tree Library", http://wikipedia-clustering.speed, blue.org/strBTree.php, **(2013)**, pp. 7-20.

Authors

Guoqing Zhang, he received his Bachelor of Science degree from Huaiyin Normal University, Jiangsu, P.R. China in 2010. Now he is a graduate student majoring in Computer Software and Theory at Soochow University in P.R. China. His current researches focus on data compression and data index algorithms.

Mei Rong, she received the PhD degrees in Computer Science from Chongqing University, in 1998, respectively. She is currently an associate professor in the Shenzhen Tourism College, Jinan University, China, and is the member of CCF. Her research interests include software engineering, formal methods and Cyber Physical Systems.

Guangquan Zhang, he received his Ph.D. of Computer Software and Theory from Chongqing University, P.R. China in 1999. He is a professor at School of Computer Science and Technology, Soochow University, China and a senior member of China Computer Federation. His research interests include software engineering, cloud computing, CPS and formal methods.