Available online at www.ijpe-online.com

International Journal of Performability Engineering

vol. 14, no. 4, April 2018, pp. 647-655
DOI: 10.23940/ijpe.18.04.p7.647655

Collision Analysis and an Efficient Double Array Construction Method

Lianyin Jia[a], Wenyan Chen[a], Jiaman Ding[a], Xiaohui Yuan[b], Binglin Shen[a], and Mengjuan Li[c,*]

[a]*Faculty of Information Engineering and Automation, Kunming University of Science and Technology, Kunming, 650500, China*
[b]*College of Engineering, University of North Texas, Denton, 76203, USA*
[c]*Yunnan Normal University, Kunming, 650500, China*

Abstract

Trie is fundamental to many applications, such as natural language processing, string similarity search and join, but it suffers from a high space overhead. Double array (DA) provides a new way to reduce the space overhead but suffers from low construction efficiency. There is an urgent demand to promote the construction efficiency of DA while maintaining a low memory overhead. To address this problem, we reveal that the collisions generated during DA construction process mainly contribute to the low construction efficiency. Based on this analysis, a partition double array (PDA) is proposed in this paper. PDA can reduce the number of collisions as well as the cost of handling collisions in DA, so higher construction efficiency is guaranteed. Experiments on real dataset indicates that PDAs have a construction efficiency 15x higher than DA. We also obtain a bonus 2.7x higher retrieval efficiency compared with DA.

Keywords: double array; partition double array; collision; construction efficiency

(Submitted on January 2, 2018; Revised on February 13, 2018; Accepted on March 26, 2018)

1. Introduction

Trie, originally proposed by Edward Fredkin [6], is an ordered structure with characters stored on the edges. Trie is widely used in many fields such as natural language processing [4,11], bibliographic search [1], language model implementation [15] , IP routing address lookup [8,12] , string or set similarity query and join [5,9,13] and so on. There are two common forms to express a trie [3]: the matrix form and the list form. The matrix form has faster retrieval efficiency but with a higher space overhead while the list form is contrary to the former.

In most cases, constructing a trie is memory consuming, especially when the dataset is sparse. To solve this problem, an efficient trie called Double Array (DA) [2] is proposed. DA uses two integer arrays named BASE and CHECK to compress a trie so it has a much smaller memory overhead. Retrieving a string on DA is also very fast since it only involves two operations: the access and the addition.

DA has the merits of both fast access speed of the matrix form and high compression ratio of the list form, so it has been widely used. However, it also suffers from some drawbacks such as inefficient construction efficiency and empty elements.

At present, many works have been carried out on DA and most of them try to further compress DA to make it more succinct. Wang *et al*. [16] proposed a heuristic optimization strategy that processes trie nodes with more children first. This strategy can improve space usage to a certain extent, but the comparison between different branches introduces additional overheads, thus reducing the construction efficiency of DA. Yata *et al*. [17] proposed compacted double array (CDA) which keeps characters rather than integers in CHECK, thus achieving a higher compression ratio. But, it also needs additional overheads to satisfy the uniqueness of the BASE value. Fuketa *et al*. [7] researched single array with multi code (SAMC)

* Corresponding author.
E-mail address: lmjlykm@163.com

which removes BASE array and turns DA into a single array, but it is only efficient for fixed length strings. Kanda *et al*. [10] presented double array using linear functions (DALF), which uses less bits to express an element in BASE but needs to be reconstructed when BASE value is far away from the corresponding linear function.

These aforementioned structures mainly focus on space usage, thus construction overhead is omitted. Moreover, most of these approaches are mainly static double arrays, which means they do not support dynamic update, which is commonly needed in many circumstances. From our standpoint, however, construction efficiency has great importance for many join based applications, such as trie-join [5], because in these applications the construction process is always included in the total elapsed time. Niu *et al*. [14] try to speed up this process, but finally get limited achievements. Experiments mentioned in this paper (described in section 4.2) show that the DA construction time increases dramatically with the increase of the number of strings. It urgently calls for a DA structure with a high construction efficiency.

In this paper, we check the construction process of DA in detail. After that, we reveal the collisions generated during construction process mainly contribute to the low construction efficiency. On the basis of collision analysis, we propose a simple but rather effective partition double array (PDA) structure, which greatly reduces the number of collisions as well as the cost of handling collisions in DA, thus improving the construction efficiency. What's more, the retrieval efficiency also improved a great deal compared with DA.

2. Trie and double array

Trie, also noted as the word search tree, is a tree of shared prefixes. In a trie, each path from the root node to the leaf node represents a string stored in it. Figure 1 shows the trie of a string set S = {"abc#", "abhgc#", "abas#", "eak#"}. In order to distinguish two strings like "the" and "then", a special character '#' is added at the end of each string. Trie can be viewed as a finite state machine, where each node represents a state and an edge from a node to one of its child node represents a state transition.

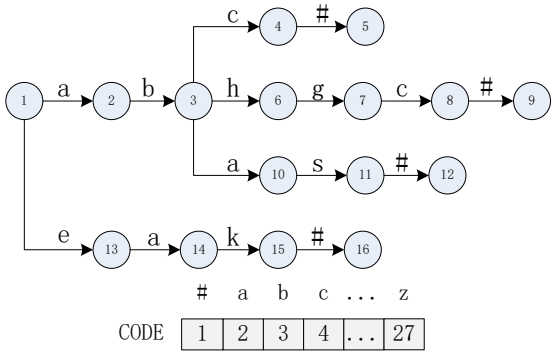

Figure 1. A trie for S

Commonly, a trie needs to store a large number of strings, which requires a large storage overhead. To alleviate this, DA was proposed which consists of 2 one-dimensional equal-length arrays named BASE and CHECK. DA only needs to store a reduced trie [3], which contains the prefix of each string that can be distinguished from all other strings. Another character array named TAIL is used to store the suffix of each string. BASE stores the state value, and CHECK stores the checksum that is used to check the status of state transition. In DA, a character c from state x to state y must satisfy Equation (1) and Equation (2):

$$BASE[x] + CODE[c] = y \tag{1}$$

$$CHECK[y] = x \tag{2}$$

where CODE[c] represents the numerical code of the character c.

BASE[i] and CHECK[i] all equaling to 0 indicates the i-th position is unused, whereas a negative BASE value indicates the position of the string suffix in TAIL. A reduced trie and its corresponding DA for S are shown in Figure 2. Note that the node number on a reduced trie equals to its position in DA. We use both the node number and the position interchangeably in this paper when there is no ambiguity. If there is a character c representing a transition from node x to node y on a

reduced trie, the position *x* in DA is called the parent position of *y*, position *y* is called a child position of *x*, and *c* is the transition character from *x* to *y*. All transition characters starting from x is denoted by O_x and all transition characters ending at y is denoted by I_y.

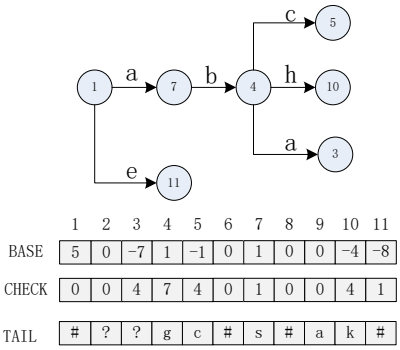

Figure 2. The reduced trie and DA for K

3. Partition double array

3.1. Collision analysis of DA

When inserting a string into DA, collisions may occur, thus degrading the construction efficiency. The main reason of collisions lies in that two transitions may reach a same position in DA. We name this collision as position competition collision (PCC). There are two possible circumstances resulting in a PCC as Equation (3) and Equation (4) show.

$$BASE[x]+CODE[c]=BASE[x']+CODE[c] \quad \text{for } x\neq x' \tag{3}$$

$$BASE[x]+CODE[c]=BASE[x']+CODE[c'] \quad \text{for } x\neq x'\wedge c\neq c' \tag{4}$$

Equation (3) indicates two different nodes, *x* and *x'*. The two may have the same BASE value, so when they met with the same transition character, a PCC occurs. Equation (4) indicates that even *x* and *x'* have different BASE values, when they met with different transition characters, a PCC may also occur. Figure 3 and Figure 4 give two examples of these two circumstances respectively.

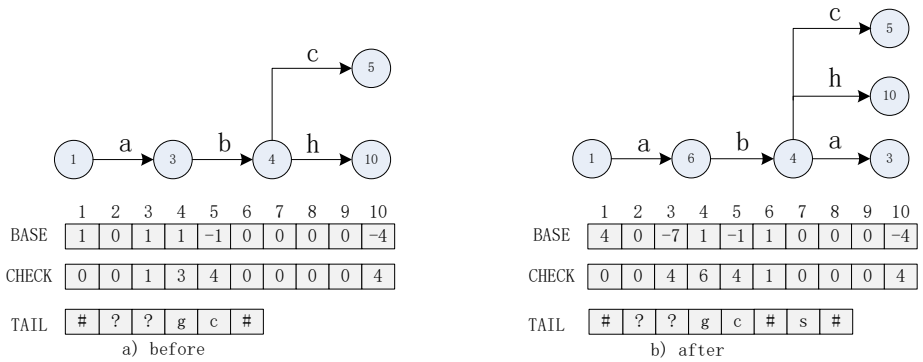

Figure 3. DA before and after inserting "abas#"

Example 1. In Figure 3, when inserting "abas#" into DA, the third character 'a' corresponds to a state transition from position 4 to position 3, but the state transition from position 1 to position 3 corresponding to the first character 'a' of "abc#" already exists in DA. So, position 1 and position 4 compete for position 3, resulting in a PCC.

Example 2. In Figure 4, when inserting "eak#" into DA, the first character 'e' corresponds to a state transition from position 1 to position 10, but the state transition from position 4 to position 10 corresponding to the third character 'h' of "abhgc#" already exists in DA. So, position 1 and position 4 compete for position 10, resulting in a PCC.

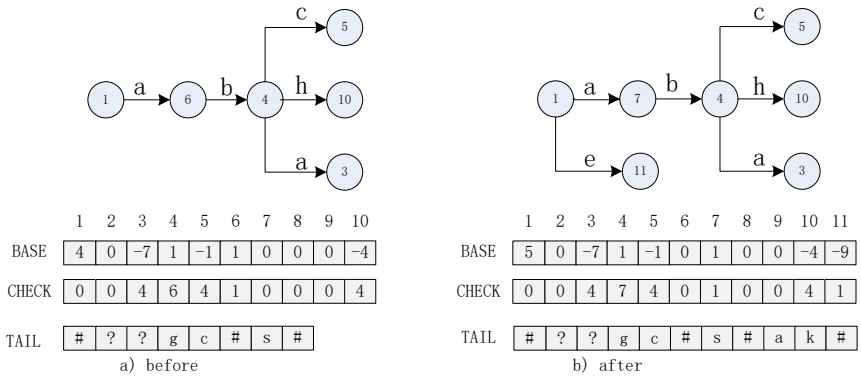

Figure 4. DA before and after inserting "eak#"

When a PCC appears, the following operations should be performed to tackle this problem: 1) select the position p with fewer children as the position to be sacrificed 2) execute *X_CHECK* function for p to reselect a new BASE value q. The possible values are incrementally probed from current BASE value (the initial value is 1) and until we find a minimum q satisfying CHECK [q + CODE[c]] = 0 for all $c \in O_p$. For ease of expression, q is also called the probe length in this paper 3) move the BASE and CHECK values for each child position p' of p to its new position q +CODE[c] where c is the transition character from p to p'.

Example 3. In Figure 3, when resolving PCC caused by inserting " abas #" into DA, the position 1 is sacrificed (only 1 child position 3); perform *X_CHECK* function and we get q= 4 (q+CODE['a']=4+2=6 and position 6 is unused). Position 6 is the new available position. Then, we move the BASE and CHECK values of position 3 to position 6. Here, we need 4 probes and 1 movement. Similarly, we need additional 1 probe (BASE[1] from 4 to 5) and 1 movement (from position 6 to 7) when inserting "eak#" into DA. There are 2 PCCs in total.

It can be seen from the above analysis that when a PCC occurs, a large amount of probes are needed for executing *X_CHECK* function, and the probe length increases sharply with the increase in the number of strings. Moreover, handling PCC needs a large number of data movements, thus deteriorating the construction efficiency.

3.2. The design of partition double array

It can be seen from the above analysis that DA generates a large number of collisions and costs during index construction. As the amount of strings increases, the number of collisions goes up rapidly so we need to find new and efficient measures to address this issue.

3.2.1. Motivation

Based on collision analysis mentioned above, we have the following observation.

Observation: Dividing a string set into multiple partitions reduces the number of collisions and the costs of handling.

Example 4. Assume the string set S = {"abc#", "abhgc#", "abas#", "eak#"} is divided into S_1 = {"abc#", "abhgc#", "abas#"} and S_2 = {"eak#"} and for each partition DA is created as Figure 5 shows. Inserting "eak#" into the second partition does not cause any collisions, let alone the cost to handle the collisions.

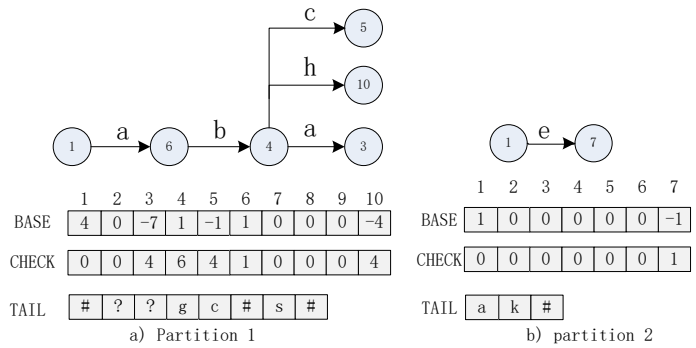

Figure 5. Dividing K into two Partitions

Based on the observation above, partition can reduce collisions because the collisions are constrained within their partition. Besides, the costs of handling collisions can also be reduced since we use much shorter arrays. So, the construction efficiency can be boosted.

3.2.2. Partition strategy

To partition strings, a good partition strategy is of prime importance. When designing a partition strategy, two principles should be considered: 1) Intuitively, for a string set *S* and a given partition number *k*, if *S* is evenly divided into *k* partitions, it will have the lowest collision probability. 2) Strings containing public prefixes should be grouped into the same partition to minimize the space overhead.

Based on the aforementioned principles, a balanced partition strategy considering the common prefixes is proposed. Given partition number *k*, the strategy is implemented as follows:

1) Determine *k*-1 division lines to divide the dataset equally into *k* partitions.

2) Adjust the division lines according to the common prefix. If some strings having common prefixes (for example, strings with the first letter 'b') is divided into two different partitions by a certain division line, the division line is moved to the nearest edge.

These strategies allow each partition to be as balanced as possible, and strings with a common prefix are always in the same partition. For ease of description and implementation, in this paper, only initial letters are considered to split strings and extend to a much larger partition parameter *k* to support large scale parallel. A sketch map dividing a string set into 5 partitions is shown in Figure 6.

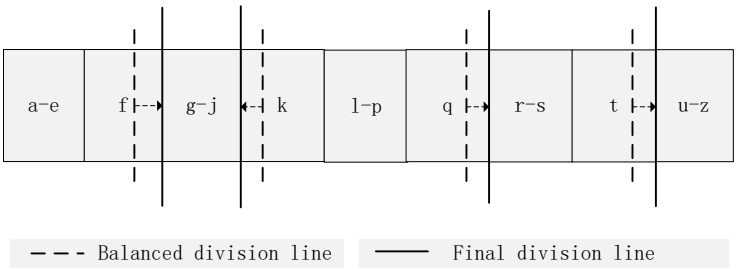

Figure 6. The partition strategy

3.2.3. Partition mapping

After the partition strategy is designed, the partition to be inserted for each string should be determined. To do this, an efficient partition map table (PMT) is designed in which a partition entry is created for each independent character. The designed PMT is efficient since the subscript of an initial character in PMT can be computed directly. Both the construction and retrieval process can benefit a lot from PMT. The PMT and the final PDA are shown in Figure 7.

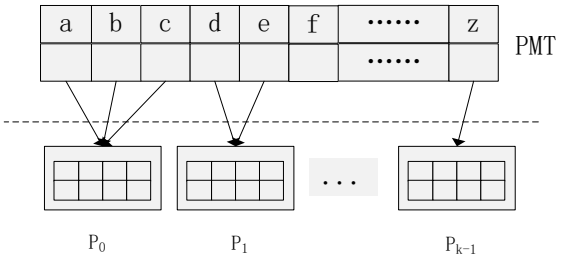

Figure 7. PDA structure

3.2.4. Construction algorithm

Based on the discussions above, a simple but rather efficient partition double array (PDA) is constructed. PDA divides the strings into multiple partitions and then constructs DA independently for each partition. By partitioning, the collision can be limited within the partition, so the number of collisions and the cost of handling collisions can be greatly reduced.

Based on the discussions above, the construction algorithm of PDA is given in Algorithm 1.

PDA Construction Algorithm

ConstructPDA(S，k)
//Input：S, a string set
k，numbers of partitions
//Output：a PDA index
1. $S_1,S_2,\cdots,S_k\leftarrow$ divide R into k relative even subsets using our partition strategy
2. create a PMT for S
3. for each subsets S_i
4. DA(S_i)
5. add pointers in corresponding entries of PMT to point to the i-th partition

Algorithm 1. PDA construction algorithm

The algorithm divides S into k subsets according to the partition strategy proposed in this paper, and then creates a PMT for S. For each subset, we create a DA using traditional DA construction algorithm that sets the mapping between the created partition and PMT.

3.2.5. Retrieval algorithm

Retrieving on PDA is rather intuitive and efficient. Given a string s, we obtain its partition by retrieving its initial letter in PMT, and then execute the traditional DA retrieval algorithm in this partition. The retrieval algorithm of PDA is shown in Algorithm 2.

PDA Retrieval Algorithm

RetrievalPDA(s)
//Input：s, a query string
//Output：f, a flag denoting whether s is indexed
1. $c\leftarrow$the initial character of s
2. $p\leftarrow$get the partition of c using PMT
3. $f\leftarrow$DA(p,s)

Algorithm 2. Query algorithm based on PDA

Note that DA can be viewed as a special case of PDA with $k = 1$. More importantly, although PDA contains multiple DAs, it does not lead to an apparent increase in space usage compared with DA because when |S| is large, PDA do not generate too much empty elements than DA. The following experiments help illustrate this point. So, PDA can greatly improve the construction efficiency and retrieval efficiency under the premise of not significantly increasing the space overhead.

4. Experiment evaluation

To evaluate the performance of PDA, we have performed extensive experiments on real dataset to look into and compare with other competitors. All the experiments were run on a windows 10 64-bit machine with an Intel (R) Core 4 i5-6500 CPU @ 3.20GHz processor and 8GB memory. The algorithms were implemented in Microsoft Visual Studio C++ 2010.

The dataset used here is DBLP[1] and we extract 183361 unique strings of title field in DBLP. The minimum string length is 1, the maximum length is 49 and the average length is 8.6. We create PDA using this dataset and also use it as queries to examine the construction and retrieval efficiency.

4.1. Impact on partition number

Firstly, the impact of partition number on PDA construction efficiency is researched. We set partition number k to be 1, 5, 10, 15 and 20, respectively to examine their construction times. The result is shown in Figure 8. It can be observed from

[1] http://www.informatik.uni-trier.de/~ley/db/

Figure 8 that the construction time is significantly reduced as the number of partitions increases. In extreme cases, PDA with k=1 takes about 105s, whereas the time is only 6.7s for k = 20, 15 times faster than the former. The curve tends to be smooth when k=10. In the experiments below, when not pointed out explicitly, we use k = 10 as the default setting.

As previously analyzed, the higher efficiency with a larger k lies in that the total number of collisions (sum of collisions of all partitions) and the total probe length (sum of all probe lengths) are reduced with the increase of the number of partitions, which undoubtedly boosts efficiency. The total number of collisions and the total probe length are shown in Figure 9 and Figure 10, respectively.

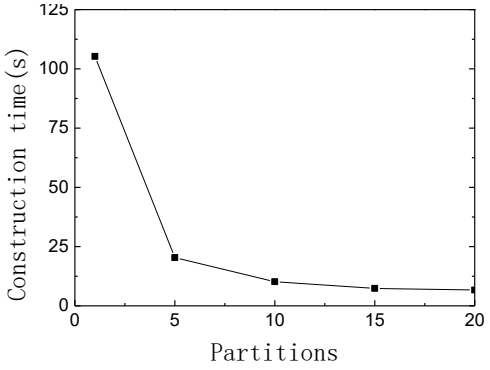

Figure 8. Construction time

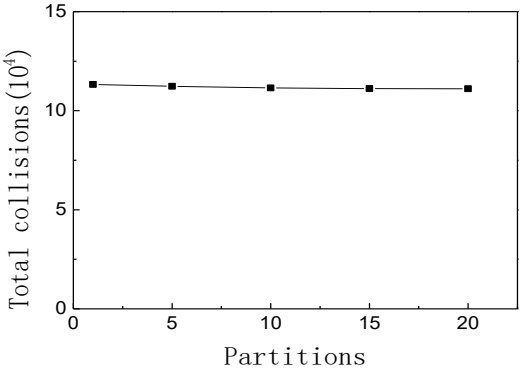

Figure 9. Number of Collisions

Then, the impact on space usage for different k is evaluated. For each DA of PDA, the space usage is calculated based on Equation (5).

$$(|BASE|+|CHECK|)*sizeof(\text{int})+|TAIL| \tag{5}$$

where sizeof (int) represents the number of bytes in an integer. For PDA, we accumulate the space usage of all DAs and report the results in Figure 11. As can be seen from Figure 11, as the number of partitions increases, there is no obvious increase in space usage for PDA, which is consistent with the above analysis.

Finally, the impact of partition number on retrieval time of PDA is analyzed and the results are shown in Figure 12. As the number of partitions increases, the retrieval time shows a descending trend. The retrieval time for 20 partitions is about 2.7 times higher than the 1 partition counterpart. The main reason may lie in that PDA uses much smaller arrays, and thus benefits the cache hit ratio.

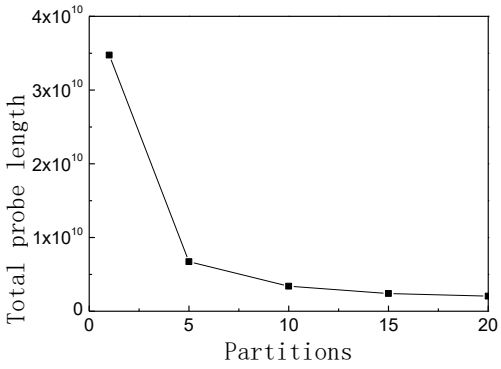

Figure 10. Total probe length

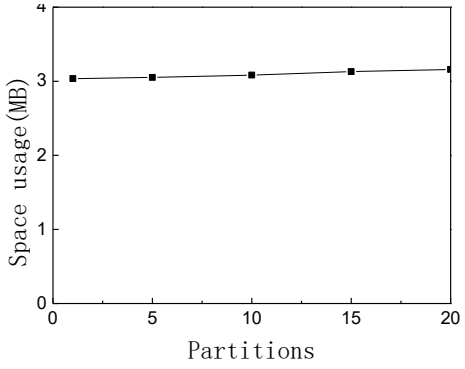

Figure 11. Space usage for different number of partitions

4.2. Comparison of different algorithms

We compare PDA with the other two competitors, DA and CDA, from construction times, retrieval times and space usages to verify the efficiency of the PDA. In the following experiments, we select the first 50,000, 100,000, 150,000 and all strings in DBLP respectively and then construct its corresponding PDA. The partition parameter k used here is 10.

The comparison of construction times is shown in Figure 13, which shows that PDA is much faster than the other 2 structures. More importantly, with the increase of string number, the advantage of PDA is much more obvious which turns out PDA is more suitable to be used in large datasets.

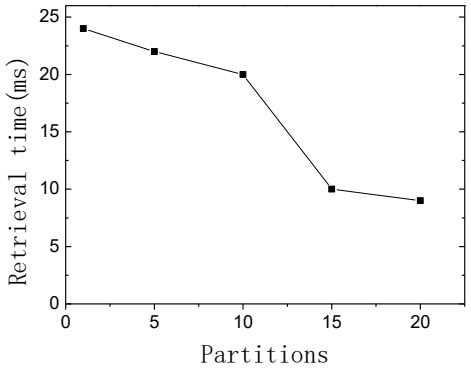

Figure 12. Retrieval times for different number of partitions

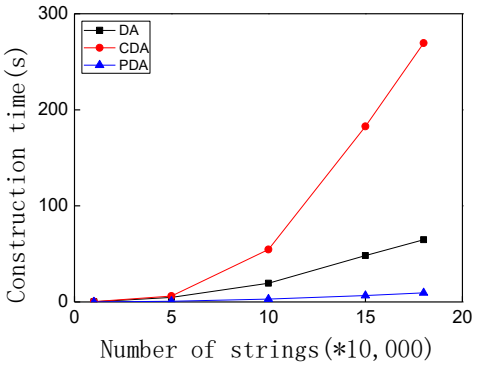

Figure 13. Construction times for the 3 structures

The retrieval times for the 3 structures are shown in Figure 14. PDA is also the most efficient among them. The retrieval time of CDA has no great differences with DA since CDA does not change the retrieval algorithm of DA.

The space usages for the 3 structures are shown in Figure 15. As described before, the space usage of PDA is comparable to DA. CDA has the least space usage since it uses unique BASE value for each position. As a result, it can use characters (smaller than integer) to represent CHECK values.

5. Conclusions

DA is a succinct string index and can be used in many fields, but suffers from low construction efficiency. This paper reveals that it is the collision that degrades the construction efficiency. In view of this, PDA is proposed. PDA can effectively reduce the number of collisions and the cost of handling collisions, thus greatly improving the construction and retrieval efficiency while maintaining a small space overhead. Next, we will try to decrease the space usage of PDA by developing some compression methods suiting for dynamic double array.

Acknowledgements

The research is supported by Grants from the National Natural Science Foundation of China (No. 61562054,51467007,61462050), and the Personnel Training Project of Yunnan Province(No.KKSY201603016).

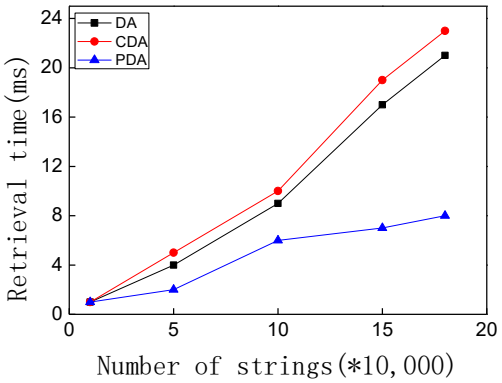

Figure 14. Retrieval times for the 3 structures

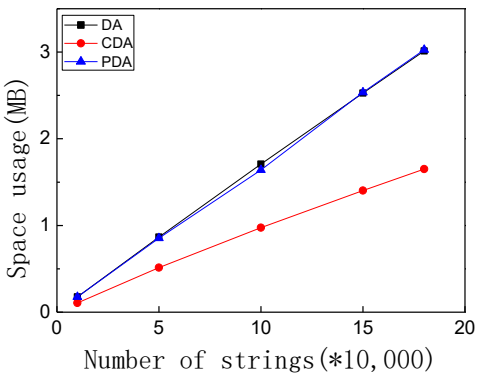

Figure 15. Memory overheads for the 3 structures

References

1. A. V. Aho and M. J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," in *Comm. ACM*, 2010.
2. J. I. Aoe, "An Efficient Digital Search Algorithm by Using a Double-Array Structure," *IEEE Transactions on Software Engineering*, vol. 15,no. 9, pp. 1066-1077, 1989.
3. J. I. Aoe, K. Morimoto, and T. Sato, *An Efficient Implementation of Trie Structures*: John Wiley & Sons, Inc., 1992.

4. B. Ding, H. Wang, R. Jin, J. Han, and Z. Wang, "Optimizing Index for Taxonomy Keyword Search," in *ACM SIGMOD International Conference on Management of Data*, 2012, pp. 493-504.
5. J. Feng, J. Wang, and G. Li, "Trie-Join: A Trie-Based Method for Efficient String Similarity Joins," *The VLDB Journal*, vol. 21,no. 4, pp. 437-461, 2012.
6. E. Fredkin, "Trie Memory," *Communications of the Acm*, vol. 3,no. 9, pp. 490-499, 1960.
7. M. Fuketa, H. Kitagawa, T. Ogawa, K. Morita, and J. I. Aoe, "Compression of Double Array Structures for Fixed Length Keywords," *Information Processing & Management An International Journal*, vol. 50,no. 5, pp. 796-806, 2014.
8. K. Huang, G. Xie, Y. Li, and A. X. Liu, "Offset Addressing Approach to Memory-Efficient Ip Address Lookup," *Proceedings - IEEE INFOCOM*, vol. 42,no. 4, pp. 306-310, 2011.
9. L. Y. Jia, J. Q. Xi, M. J. Li, Y. Liu, and D. C. Miao, "Eti: An Efficient Index for Set Similarity Queries," *Frontiers Of Computer Science*, vol. 6,no. 6, pp. 700-712, 2012.
10. S. Kanda, M. Fuketa, K. Morita, and J. I. Aoe, "A Compression Method of Double-Array Structures Using Linear Functions," *Knowledge & Information Systems*, vol. 48,no. 1, pp. 55-80, 2016.
11. Lai Yang, Lida Xu, and Zhongzhi Shi, "An Enhanced Dynamic Hash Trie Algorithm for Lexicon Search," *Enterprise Information Systems*, vol. 6,no. 4, pp. 419-432, 2012.
12. J. Lee and H. Lim, "Multi-Stride Decision Trie for Ip Address Lookup," *Ieie Transactions on Smart Processing & Computing*, vol. 5,no. 5, pp. 331-336, 2016.
13. G. Li, J. He, D. Deng, and J. Li, "Efficient Similarity Join and Search on Multi-Attribute Data," presented at the Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, 2015.
14. S. Niu, Y. Liu, and X. Song, "Speeding up Double-Array Trie Construction for String Matching," *Communications in Computer & Information Science*, vol. 320,no. pp. 572-579, 2013.
15. J. Y. Norimatsu, M. Yasuhara, T. Tanaka, and M. Yamamoto, "A Fast and Compact Language Model Implementation Using Double-Array Structures," *ACM Transactions on Asian and Low-Resource Language Information Processing*, vol. 15,no. 4, pp. 1-27, 2016.
16. S. L. Wang, H. P. Zhang, and B. Wang, "Research of Optimization on Double-Array Trie and Its Application," *Journal of Chinese Information Processing*, vol. 20,no. 5, pp. 24-30, 2006.
17. S. Yata, M. Oono, K. Morita, M. Fuketa, T. Sumitomo, and J. I. Aoe, "A Compact Static Double-Array Keeping Character Codes," *Information Processing & Management*, vol. 43,no. 1, pp. 237-247, 2007.

Lianyin Jia is an Associate Professor in the Faculty of Information Engineering and Automation, Kunming University of Science and Technology, Kunming, China. He received his Ph.D. degree in Computer Science from South China University of Technology, Guangzhou, China in 2013. His current research interests include database, data mining, information retrieval and parallel computing.

Wenyan Chen is an MPhil student in the Faculty of Information Engineering and Automation, Kunming University of Science and Technology, Kunming, China. His current research interests include database, information retrieval, parallel computing.

Jiaman Ding is an Associate Professor in the Faculty of Information Engineering and Automation, Kunming University of Science and Technology, Kunming, China. He now is also a Ph.D. candidate in Kunming University of Science and Technology. His current research interests include datamining, cloud computing.

Xiaohui Yuan is an Associate Professor in the College of Engineering, University of North Texas, Denton, America. He received his Ph.D. degree in Computer Science from Tulane University in 2004. His current research interests include computer vision, data mining, machine learning, and artificial intelligence.

Binglin Shen is an MPhil student in the Faculty of Information Engineering and Automation, Kunming University of Science and Technology, Kunming, China. Her current research interests include database, information retrieval, parallel computing.

Mengjuan Li is a Librarian in the Department of Technology, Library, Yunnan Normal University, Kunming, China. She received her master degree in Computer Science from Kunming University of Science and Technology, Kunming, China in 2008. Her main research interests include information retrieval, parallel computing.