


Article

An Efficient Two-Level-Partitioning-Based Double Array and Its Parallelization

Lianyin Jia ^{1,2}, Chongde Zhang ¹ , Mengjuan Li ³, Yinong Chen ⁴, Yong Liu ⁵ and Jiaman Ding ^{1,*}

¹ Faculty of Information Engineering and Automation, Kunming University of Science and Technology, Kunming 650500, China; jlianyin@163.com (L.J.); chongde1996@163.com (C.Z.)

² Yunnan Key Laboratory of Artificial Intelligence, Kunming University of Science and Technology, Kunming 650500, China

³ Library, Yunnan Normal University, Kunming 650500, China; lmjlykm@163.com

⁴ School of Computing, Informatics, and Decision Systems, Arizona State University, Tempe, AZ 85287, USA; yinong@asu.edu

⁵ School of Information Science and Engineering, Guangxi University for Nationalities, Nanning 530006, China; niu20060040@gmail.com

* Correspondence: tjom2008@126.com

Received: 2 June 2020; Accepted: 28 July 2020; Published: 30 July 2020



Abstract: Trie is one of the most common data structures for string storage and retrieval. As a fast and efficient implementation of trie, double array (DA) can effectively compress strings to reduce storage spaces. However, this method suffers from the problem of low index construction efficiency. To address this problem, we design a two-level partition (TLP) framework in this paper. We first divide the dataset into smaller lower-level partitions, and then we merge these partitions into bigger upper-level partitions using a min-heap based greedy merging algorithm (MH-GMerge). TLP has an excellent characteristic of load balancing and can be easily parallelized. We implemented two efficient parallel partitioned DAs based on TLP. Extensive experiments were carried out, and the results showed that the proposed methods can significantly improve the construction efficiency of DA and can achieve a better trade-off between construction and retrieval performance than the existing state-of-the-art methods.

Keywords: load balancing; MH-GMerge; parallel partitioned double array; TLP

1. Introduction

String storage and retrieval are fundamental operations in many fields, such as in search engine, natural language processing, and artificial intelligence applications. It is of great significance to improve the efficiency of these operations. Just as B⁺-Tree is the representative of database index for integer [1], trie is one of the most common structures for string storage and retrieval and is extensively used in artificial intelligence [2,3], natural language processing [4], data mining [5], IP address searching [6,7], string similarity joining [8,9], and many other fields. For a detailed description of trie and its applications, we refer the reader to [10,11] for more details. The storage and retrieval efficiency can be naturally improved by compressing common prefixes of strings into common paths.

The traditional trie can either be a matrix form or a linked form [12]. The matrix form is efficient in retrieval speed but with a higher space overhead. Meanwhile, the linked form is efficient in space overheads, but its retrieval efficiency is relatively slow. Both of them are difficult to balance between retrieval performance and storage overheads. Many efforts have been devoted to address this problem, and various tries and corresponding optimizations have been proposed [13–17]. Among them, Level-Ordered Unary Degree Sequence (LOUDS) [18,19] and double array (DA) [20] are the two most

eminent representatives. LOUDS is the most succinct trie to our best knowledge. It used bit strings to compress trie, which was extremely space-saving. However, retrieving in LOUDS was relatively time-consuming, because it requires expensive SELECT and RANK operations on generated bit strings. To improve the retrieval efficiency, the recent work, Zhang et al. [21], proposed Fast Succinct Trie (FST), which is a combination of LOUDS-SPARSE and LOUDS-DENSE. A fast bitmap-based LOUDS-DENSE is deployed in the upper-levels of a trie to enable fast and frequent access, and a space-efficient LOUDS-SPARSE is deployed in the lower-levels to decrease the space storage. Furthermore, FST introduces some other optimizations, e.g., lookup table and sampling, to speed up the SELECT and RANK operations, which makes FST both effective and succinct. Although FST has made a good trade-off between retrieval performance and storage overheads, its retrieval speed is still not fast because it still needs the SELECT and RANK operations to retrieve a child node of a specific node, which makes it not suit for frequently retrieval scenarios.

DA [18] proposed by Aoe, provides another way of making a good balance between retrieval performance and storage overheads. It compresses stored strings into two one-dimensional integer-arrays named BASE and CHECK. Retrieving a specific child node of a node is extremely fast, because only simple addition operations are needed.

Considering the high retrieval efficiency of DA, many research efforts are devoted to further optimize this structure. Yata et al. proposed a compact double array (CDA) [22], a statically constructed compressed double array. Unlike DA, CDA stores a character instead of an integer in CHECK to reduce the space overhead under the premise that each value in BASE is unique. Fuketa et al. [12] presented a single array with multiple code (SAMC) which sets $\text{BASE}[s]$ to s at first and then determines the content of CHECK in a depth manner. As $\text{BASE}[s]$ equals s , so BASE can be safely removed and thus storage spaces can be reduced. In double-array using linear functions (DALF) [23] proposed by Kanda et al., BASE is segmented and a linear function is designed for each segment to reduce the space overheads. Kanda et al. [24] introduced rearrangement methods to improve space efficiency as well as update time.

Most of the aforementioned methods are devoted to reducing space overheads. However, the index construction efficiency has not been well studied. Improving construction efficiency can increase the manageability of indexes, which is important for large datasets and for certain string join operations, because index construction time is always counted in total elapsed time for these operations. Morita et al. [25] presented a skip method and a link method by exploiting the empty elements in DA. Niu et al. [26] further represented the continuous available nodes using a bidirectional link. These studies obtained some improvements to a certain extent. Jia et al. [27] found that collision is the main reason for the low efficiency of DA construction. Based on this finding, a first-character-based partition framework was proposed and many multiple small DAs instead of a single DA were constructed. In this way, collisions were restricted within partitions and thus collisions and collision handling costs could be reduced.

In addition, most DA-based algorithms are serial algorithms and are not suitable or not efficient in parallel environments. Although the partitioned DA [27] can be used for parallelization, its efficiency is still limited because its partitions are not well balanced. With the popularity of multi-core processors, it is important to use parallelization to improve the construction and retrieval efficiency of DA.

To this end, after carefully analyzing the construction process of DA, this paper designs a two-level partitioning framework (TLP), which can achieve a better load balancing while keeping strings with the same prefixes in the same partition. Based on TLP, two parallel partitioned DAs are designed, which can effectively improve the construction and retrieval efficiency without causing too much storage overhead. One excellent characteristic of TLP is that it can be integrated with any other DAs and for further improving the construction efficiency. Extensive experiments show that our proposed indexes can significantly improve construction efficiency of DA and outperform some other state-of-the-art competitors in many aspects.

2. Preliminaries

Before we go on, we first give the nomenclature used in this paper, as shown in Table 1.

Table 1. Nomenclature.

Symbol	Description
D	A dataset
s	A string
c	A character
x, y	A node in trie or a position in DA
UP_i	The i -th upper-level partition
L	A collection of lower-level partitions
U	A collection of upper-level partitions
m	The number of lower-level partitions
n	The number of upper-level partitions
R_u	The partition range of U
H_t	The heap top
UPN	UP partition No
PQT	Partition query table

2.1. Double Array

For a string s in a dataset D , the character c is called a separated character if c is a sufficient character to distinguish s from all other strings in D . The substring from the beginning of s to its separate character c is called the reduced substring of s . Consequently, a trie built on reduced substrings of strings in D is a reduced trie.

DA uses two one-dimensional arrays called BASE and CHECK to store the reduced trie for a dataset D . For a node x , which is also a position x in the two arrays, BASE[x] stores the base value used for all children of x , whereas CHECK[x] stores the parent node index of x . An additional array, TAIL, is used to store the remaining suffixes.

The heart of DA lies in the following two important formulas:

$$\text{BASE}[x] + \text{CODE}[c] = y \quad (1)$$

$$\text{CHECK}[y] = x \quad (2)$$

From the above Formulas, child node y of node x can be easily obtained by executing $\text{BASE}[x] + \text{CODE}[c]$, and parent node x of node y can be found in $\text{CHECK}[y]$. Thus, accessing parent and child node is extremely fast by simple, add operations. CODE[c] is the numerical code of a character c . The codes corresponding to characters “#”, “a”, “b”, “c” ... “z” are 1, 2, 3, ..., 27, respectively.

Figure 1 shows the reduced trie and DA created on a dataset $D = \{\text{“abcd#”}, \text{“abdf#”}, \text{“abae#”}\}$. To distinguish strings like “ab” and “abc”, a special character “#” is added to the end of each string.

2.2. Collision Analysis

The construction efficiency of DA drops significantly with the increase of the number of strings for two characters may contend for a single position in DA, leading to position competition collisions. Two possible collisions are shown below.

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

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

A collision may occur when a character is inserted into two positions with the same base values, or when two different characters are inserted into two positions with different base values. When a

collision occurs, a function named X_CHECK used in [28] is executed to find a new base value for one of the old base values. X_CHECK increases the old base value incrementally to make sure it will not cause collisions anymore. Here we call the increased number the probe length. After that, the old base and check values are migrated to the new positions. The probe and migration costs will inevitably deteriorate the efficiency. For the complete construction process of DA, we refer reader to paper [28] for more details.

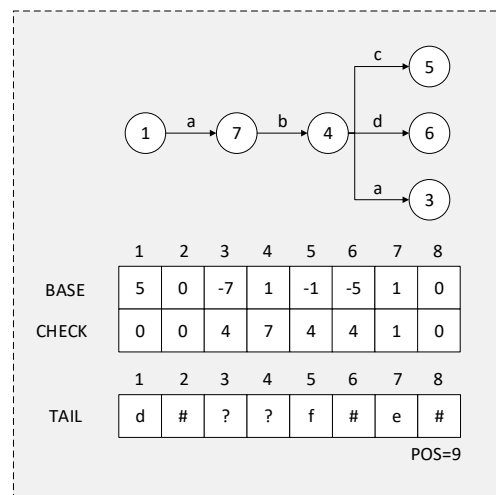


Figure 1. Reduced trie and DA for S.

2.3. Partitioning Method

Partitioning is an effective way to reduce collisions and collision handling costs, and thus it improves construction and retrieval speed. There are two common partitioning strategies available for parallel string processing: Balanced Partition (BP) and Balanced Partition with Partition Line Adjustment (BP_PLA).

2.3.1. BP

BP is the most intuitive partitioning method. It uses $n-1$ partition lines to evenly divide a data set into n partitions (We assume that the datasets are already sorted.). The first string in each partition can be viewed as a partition line (or a split string). A simple diagram of BP is shown in Figure 2. BP has the best load balancing. However, it does not guarantee that prefix integrity for strings with common prefixes are divided into the same partitions. Thus, expensive string comparisons are needed to determine the partition for a string, resulting in a decrease in retrieval efficiency.

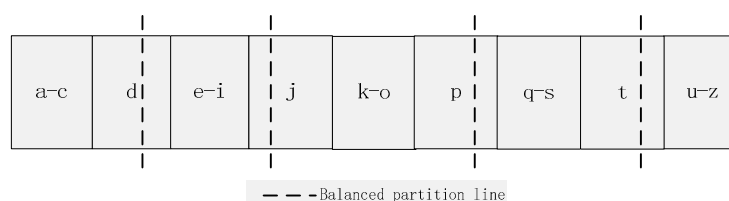


Figure 2. Balanced Partition.

2.3.2. BP_PLA

BP_PLA (In the original paper, it was simply called partition double array (PDA), but here it is referred to as BP_PLA to better distinguish it from BP.) [27] is a combination of BP and partition line adjustment (PLA), which uses BP to partition a dataset first; then, PLA is carried out to adjust each partition line to its nearest first-character border, as Figure 3 shows. In this way, strings with common

prefixes will be kept in a partition, thus prefix integrity can be guaranteed. The first character of the first string (instead of the string itself) in each partition can be chosen as a partition line, which makes partition detection very fast.

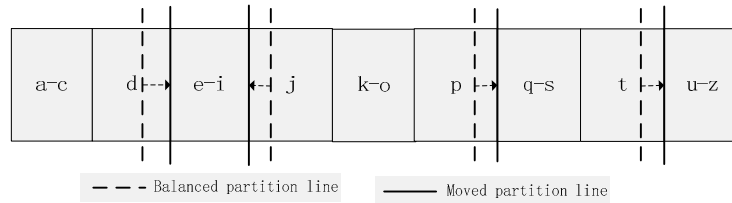


Figure 3. Balanced Partition with Partition Line Adjustment (BP_PLA).

However, BP_PLA may lead to a large deviation of strings in each partition, which may affect the load balancing performance (in Figure 3, partition *e-i* and partition *j-p* partition may have a large deviation).

3. TLP-Based Partitioned DAs

3.1. TLP

Two strings may lead to collisions when they are inserted into a DA, so it is not practical inserting them into a DA simultaneously. To construct DA in parallel, we need to divide a dataset into partitions and build multiple independent small DAs in parallel. As mentioned above, BP and BP_PLA are not suitable or less efficient in parallel environments due to the difficulties in keeping prefix integrity and load balancing together. To this end, this paper proposes a bottom-up two-level partitioning (TLP) framework to achieve a better balance between these two characteristics.

TLP firstly divides a dataset D into m discrete lower-level partitions (LP) ($L = \{LP_1, LP_2, \dots, LP_m\}$) based on the first character of strings in D . Each LP is a first-character partition (all strings with the same first character in D constitute a first-character partition). $|LP_j|$ is used to represent the number of strings in LP_j . This can ensure the prefix integrity within each LP. Secondly, TLP uniquely assigns each LP to an upper-level partition (UP), eventually merging m LPs into n UPs ($U = \{UP_1, UP_2, \dots, UP_n\}$). Similarly, $|UP_i| = \sum_{LP_j \in UP_i} |LP_j|$ is used to represent the number of strings in UP_i . A brief structure of TLP is shown in Figure 4.

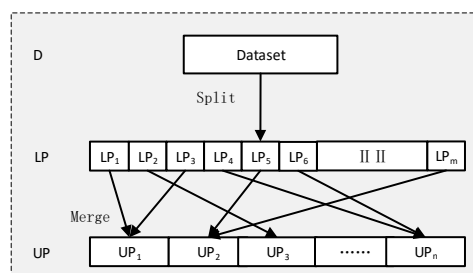


Figure 4. A brief structure of two-level partition (TLP).

Merging m LPs into n UPs is non-trivial. Intuitively, there are n^m different merging strategies. Different merging strategies lead to different load balancing performances. Thus, choosing the best merging strategy to obtain the best load balancing is a key to TLP. For easy explanation, we give the definition of Partition Range (PR) as follows:

Definition 1. Given $U = \{UP_1, UP_2, \dots, UP_n\}$, the PR of U is defined as $R_u = \max(U) - \min(U)$, where $\max(U)$ and $\min(U)$ represent the maximum and minimum number of strings of each UP in U , respectively.

PR can be used to measure load balancing of a partition strategy. To obtain the optimal load balancing performance, the optimal LP merging is defined as follows:

Definition 2. *Optimal LP Merging (OLM): Given $L = \{LP_1, LP_2, \dots, LP_m\}$, each LP in L is uniquely assigned to an UP, thus merging UP into n disjoint Ups $U = \{UP_1, UP_2, \dots, UP_n\}$ while keeping R_u to be minimum.*

OLM is essentially a multi-way number partitioning problem [29,30], which is proved to be an NPC problem, solving it in limited time is key to TLP. Based on the analysis above, this paper proposes a Min-Heap based Greedy Merging algorithm (MH_GMerge) to quickly find an approximate optimal merging strategy. The basic idea of MH_GMerge is as follows:

- (1) sort m LPs to be merged in descending order of the number of strings;
- (2) given the partition parameter n , build a min-heap for the first n LPs in L , and set the corresponding UP partition No. (UPN) of the n LPs to 1, 2, 3, ..., n , accordingly;
- (3) process the subsequent $m-n$ LPs in order until all LPs have been processed. For LP_j , assign its UPN to UPN of the heap top H_t and execute $|H_t| = |H_t| + |LP_j|$, where $|H_t|$ means the number of strings in all LPs related to H_t . At last, we readjust the heap to min-heap.

The number of elements in the created min-heap is always n and there is no need to actually load the subsequent $m-n$ LPs into the heap, which makes MH_GMerge very fast. The completed MH-GMerge algorithm is shown in Algorithm 1.

Algorithm 1. Min-Heap based Greedy Merging (MH-GMerge) algorithm.

Algorithm 1 MHG-Merge

```
//Input: L, a set of m LPs
      n, number of partitions
//Output: PQT, partition query table
1. L ← sort L by number of strings in descending order
2. PQT ← empty map from LP to UP
3. for j from 1 to n
4.   PQT[j] ← j
5. H ← minimum heap constructed according to the first n elements of L.
6. Ht ← top element of H
7. for j from n + 1 to m
8.   |Ht| ← |Ht| + |LPj|
9.   PQT[j] ← Ht.upNo
10.  adjust Heap to minimum
```

Complexity Analysis: The time complexity of sorting m LPs is $O(m \log m)$, the time complexity of constructing and adjusting heap is $O(m \log n)$. As $n < m$, so the overall time complexity of MH-GMerge algorithm is $O(m \log m)$. The major space overheads of MH-GMerge are the overheads of partition query table (PQT) and the heap, which are $m * \text{sizeof}(\text{integer})$ and $n * \text{sizeof}(\text{integer})$ respectively, where $\text{sizeof}(\text{integer})$ represents the overhead of an integer. As a result, the total space overheads of MH-GMerge is $(m + n) * \text{sizeof}(\text{integer})$.

After MH-GMerge finished, we can obtain a UPN for each LP, which means a mapping between LP and UP is established. This mapping is expressed as a partition query table (PQT) in this paper. As all strings in a LP have the same first character c , we can use an array to implement PQT and store UPN of each UP directly into PQT[c]. In the subsequent index construction and string retrieval processes, the UPN can be retrieved directly by the first character of a string, thus avoiding the expensive overheads of string comparison.

Example 1: Given $L = \{LP_1, LP_2, \dots, LP_m\}$, $m = 7$ and $n = 3$, the number of strings corresponding to each LP is 100, 80, 65, 60, 55, 20, and 10, respectively. The initial min-heap constructed on the first

3 LPs is shown in Figure 5, the number on the left part of a heap node is the *UP* partition number, and the right is the total number of strings in the *UP*. For LP_4 , we set its UPN to 3 the UPN of the heap top, 3, then we execute $|H_t| = |H_t| + |LP_j| = 65 + 60 = 125$. At last, we adjust the heap to min-heap. Following this way, after all *LPs* have been processed, we can obtain the final result status as Figure 6 shows. The final PR of this example is 10.

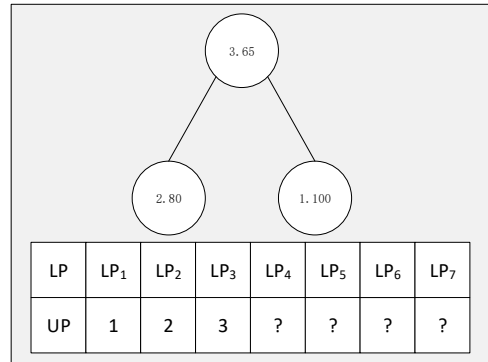


Figure 5. Initialize min-heap.

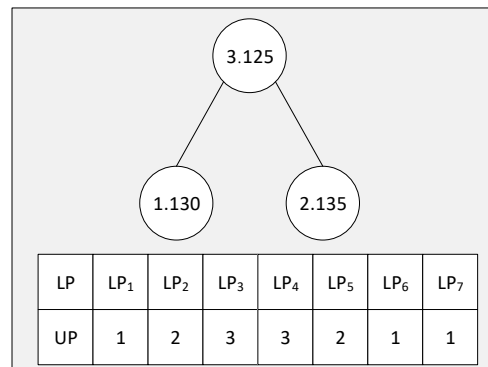


Figure 6. Final min-heap and partition query table (PQT).

It is worth noting that TLP is independent of any specific DA and can be seamlessly integrated with other DAs and further improves the construction efficiency.

From the analysis above, it is easy to see that TLP combined with MH-GMerge has a better load balancing performance while keeping prefix integrity. The final TLP algorithm is shown in Algorithm 2.

Algorithm 2. TLP algorithm.

Algorithm 2 TLP

//Input: D , a string dataset

n , number of partitions

//Output: PQT , partition query table

1. $L \leftarrow$ divide D into partitions by initial characters

2. $PQT \leftarrow$ MHG-Merge(L, n)

3.2. TLP-Based Partitioned DA

Based on TLP, we can easily implement TLP-based partitioned DA (TPDA) both in serial and in parallel. For ease of description, we only discuss constructing TPDA in parallel. To do so, this paper employs a popular parallel library, OpenMP, to construct the final TPDA in parallel. Using OpenMP, n threads are started, each thread is responsible for the DAs corresponding to a *UP*. Based on the levels the DAs created on, two kinds of TPDAs are designed as Figure 7 shows.

- (1) DAs are created on *UP*(UP-DA), one DA for each *UP*, n DAs in total;
- (2) DAs are created on *LP*(LP-DA), one DA for each *LP*, m DAs in total. A *UP* corresponds to multiple independent DAs.

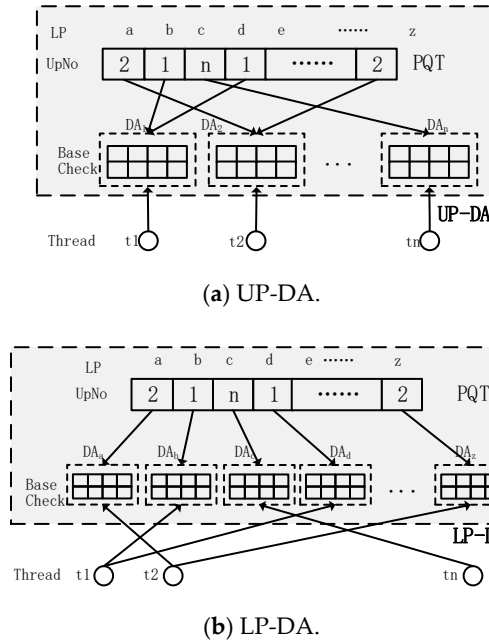


Figure 7. UP-DA and LP-DA.

Both UP-DA and LP-DA consist of PQT and DA. PQT is responsible for determining the UPN for a string, while DA is for indexing and retrieving strings. Despite the different number of DAs created for the two structures, for each thread, the loads are still relatively balanced because threads are *UP* oriented. Generally speaking, LP-DA is expected to have a much better performance for it creates multiple smaller DAs, thus reducing the collisions and the collision-handling costs.

Taking UP-DA as an example, the algorithm for constructing UP-DA is shown in Algorithm 3.

Algorithm 3. Constructing algorithm for UP-DA.

Algorithm 3 CreateUPDA

//Input: D , a string dataset
 n , number of partitions
 //Output: PQT , partition query table
 1. $L \leftarrow$ divide D into partitions by initial characters
 2. $PQT \leftarrow$ MHG-Merge(L, n)

3.3. TPDA-Based Parallel String Retrieving

Implementing parallel string retrieving on UP-DA and LP-DA is rather straightforward. For a string s currently read by a thread, we first obtain the corresponding partition p according to the first character of s . (For UP-DA, the corresponding UPN can be obtained by PQT, for LP-DA, the corresponding DA can be obtained directly by the first character). Then, s is searched on the DA corresponding to p . For UP-DA, the parallel string retrieving algorithm is shown in Algorithm 4.

Algorithm 4. Retrieving algorithm for UP-DA.

Algorithm 4 QueryUPDA

```

//Input:  $Q$ , a set of query strings
//Output:  $F$ , a flag array to indicate whether the corresponding string of  $s$  is in UP-DA
1. #pragma omp parallel for
2. for the  $i$ -th string  $s$  in  $Q$ 
3.    $c \leftarrow$  the first character of
4.    $upNo \leftarrow PQT[c]$ 
5.   query  $s$  directly in DA corresponding to  $upNo$ , if  $s$  in DA, set  $F[i]$  to true, false otherwise

```

4. Experiments*4.1. Datasets for the Experiments*

All experiments were carried out on an Intel i7-7700 CPU @ 3.60GHz (four cores and eight hyper threads) with 16 GB memory running Ubuntu 11. CodeBlocks is used as the programming platform. Gcc-4.2 and OpenMp-3.1 are used as the default compiler and parallel library, respectively.

We use the following three datasets in our experiments and some statistical information is shown in Table 2:

- DBLP [31]: DBLP is a library in computer science. We extracted all the independent words in title field of DBLP. After the duplicated strings were removed, there were 211,517 independent strings in total.
- EN-WIKI [32]: We extracted title field of Wikipedia and got 347,961 independent words after the duplicated strings were removed.
- SynthDS: To verify the effects of our proposed structures and algorithms on large dataset, we also randomly generated 2 million unique strings with lengths varied from 3 to 20 as a synthetic dataset.

Table 2. Statistical information of the three datasets.

Dataset	EN_WIKI	DBLP	SynthDS
Size (MB)	3.25	2.13	26
String number	347,961	211,517	2,000,000
Minimum length	1	1	3
Maximum length	62	46	20
Average length	8	8	11

*4.2. Serial Algorithms**4.2.1. Construction Performances***Effects of Different Number of Partitions**

In order to evaluate the construction efficiency of different partitioning algorithms under different number of partitions, construction algorithms of BP_PLA-based DA (BP_PLA-DA) (We do not compare BP here, for it has a much lower retrieval performance.), UP-DA and LP-DA are compared under different number of partitions. The results of DBLP and EN-WIKI are shown in Figure 8a–c (We assume the datasets are already sorted, because the sorting times are the same for all these algorithms.). It can be seen from the figures that the curves of both BP_PLA-DA and UP-DA decrease sharply with the increase of the number of partitions at first, and then tend to be flat. This is due to the fact that as the number of partitions increases, more small DAs are created, which reduces collisions and the collision-handling costs (e.g., the total probe lengths of BASE for the three datasets are shown in Figure 9a–c). LP-DA has the highest performance in the three methods, especially when partition number is small. For example, when partition number is 4, the construction times of LP-DA for the three datasets are 2.58 s, 4.21 s, and 108.43 s, respectively, 4x faster than BP_PLA-DA. The curve

of LP-DA is almost flat because it always creates the same number of DAs under different number of partitions.

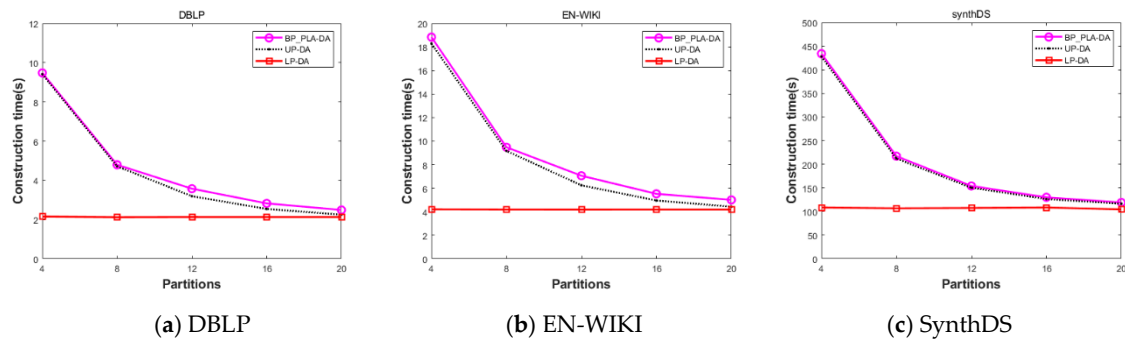


Figure 8. Construction times for different partition numbers.

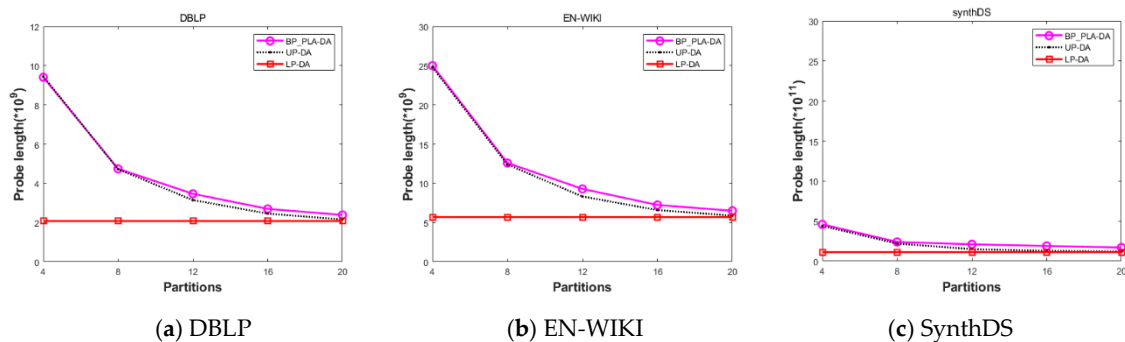


Figure 9. Total probe lengths for different partition numbers.

Comparisons with the Other Algorithms

To investigate the efficiency of different algorithms under different number of strings, the above three partitioned methods and the other two non-partitioned methods (DA and FST) are compared. For partitioned methods, we hereafter fix the number of partitions to 8 unless otherwise stated. The experimental results of the three datasets are shown in Figure 10a–c, respectively. It can be seen from the figures that FST has a higher construction efficient than DA-based algorithms because there are no collisions introduced in FST. Compared to DA, LP-DA is 7x–15x faster which shows that our partition can effectively boost the construction speed of DA.

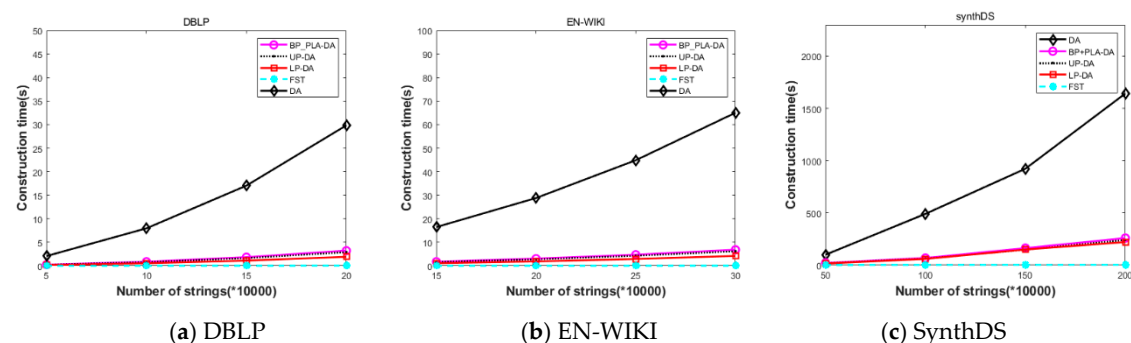


Figure 10. Construction times for different number of strings.

4.2.2. Retrieval Performances

To evaluate retrieval performance of different algorithms, EN_WIKI and DBLP were used as query sets interchangeably (indexes created on one dataset we use partial of another dataset as a query

set). The results under different numbers of strings are shown in Figure 11a–c, respectively. DA-based algorithms have almost the same retrieval time and is nearly 8–14x faster than FST-based algorithms. The reason for this is because DA only needs simple addition operations to find a child node, while FST still needs costly SELECT and RANK operations. Furthermore, note that all DA-based algorithms execute the same addition operations, so their retrieval times are almost the same. As retrieval efficiency is emphasized in general cases, we prefer LP-DA to FST.

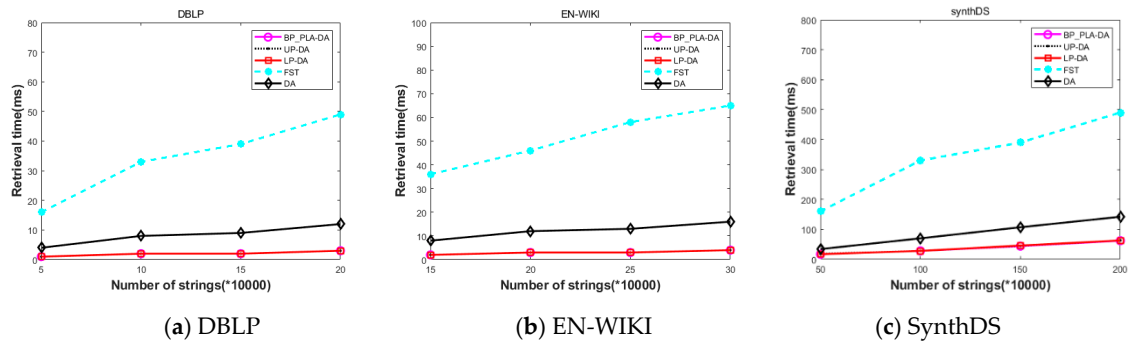


Figure 11. Retrieval times for different number of strings.

4.2.3. Comparisons of Storage Spaces

For the above indexes, the storage spaces are calculated as shown in Tables 3–5.

Table 3. DBLP storage spaces (MB).

String Numbers	50 k	100 k	150 k	200 k
DA	0.82	1.60	2.36	3.16
BP_PLA-DA	0.82	1.60	2.36	3.16
UP-DA	0.82	1.60	2.36	3.16
LP-DA	0.83	1.62	2.37	3.17
FST	0.53	1.10	1.64	2.14

Table 4. EN-WIKI storage spaces (MB).

String Numbers	150 k	200 k	250 k	300 k
DA	2.34	3.11	3.89	4.69
BP_PLA-DA	2.34	3.11	3.89	4.69
UP-DA	2.34	3.11	3.89	4.69
LP-DA	2.35	3.12	3.90	4.70
FST	1.61	2.12	2.63	3.19

Table 5. SynthDS storage spaces (MB).

String Numbers	500 k	1000 k	1500 k	2000 k
DA	8.6	16.8	25.7	33.8
BP_PLA-DA	8.6	16.9	25.7	33.8
UP-DA	8.6	16.8	25.7	33.8
LP-DA	8.7	17	25.8	34
FST	5.32	10.68	16.34	21.78

Compared with DA-based structures, FST has a higher compression rate than its DA counterparts as the underlying of FST is LOUDS based. It is also important to note that although LP-DA and UP-DA contain multiple DAs, they do not bring too much storage overheads compared with DA, because they do not generate many more empty elements than DA.

4.3. Parallel Algorithms

4.3.1. Parallel Construction

To evaluate the parallel construction efficiency of different partition methods, the three partition-based methods are compared under different partition numbers. The results in Figure 12a–c show that both UP_DA and LP-DA outperform BP_PLA-DA, as TLP has a much smaller PR than BP-PLA, as shown in Figure 13a–c.

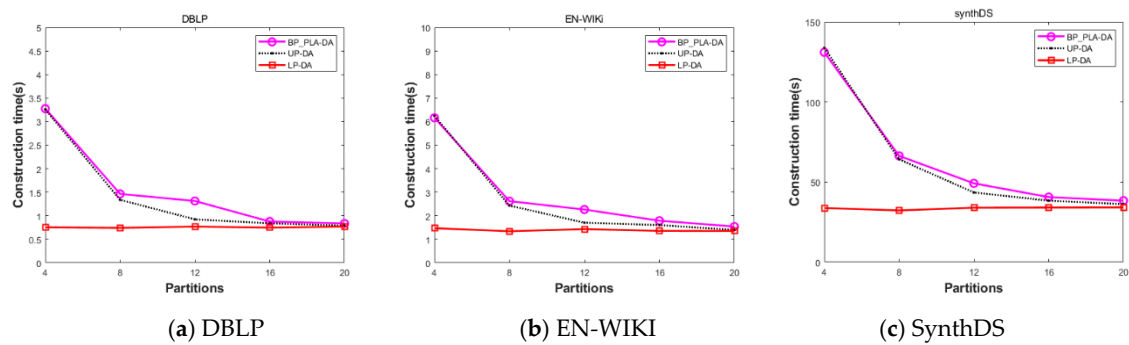


Figure 12. Parallel construction times for different partition numbers.

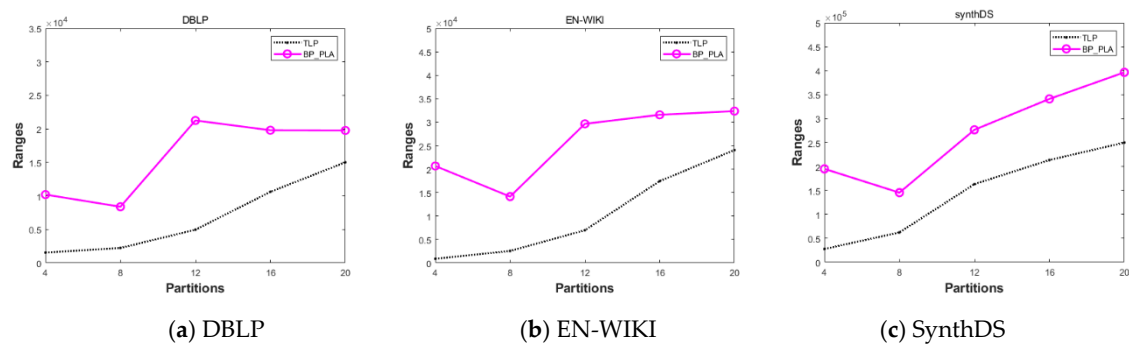


Figure 13. Partition ranges for different partition numbers.

4.3.2. Parallel Retrieval

Similar to its serial counterparts, DA-based parallel retrieval algorithms are much faster than FST-based parallel retrieval algorithms, as shown in Figure 14a–c. Compared with serial algorithms, parallel LP-DA-based algorithms have a speed up about 3x over serial LP-DA-based algorithms.

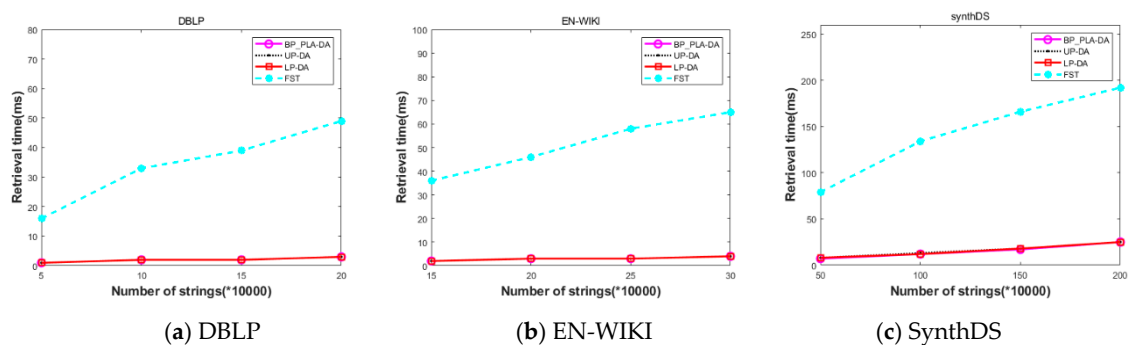


Figure 14. Parallel retrieval times for different number of strings.

5. Conclusions

To improve the construction efficiency of DA, this study developed an efficient two-level partitioning framework TLP, which merges smaller lower-level partitions to obtain much more balanced upper-level partitions. TLP had a good characteristic of load balancing while keeping prefix integrity. Based on TLP, two efficient indexes, LP-DA and UP-DA, were designed. LP-DA is 7–15x higher in construction efficiency than DA and is 8–14x higher in retrieval performance than FST, thus having a good overall performance. LP-DA has a much smaller PR than BP-PLA, thus having a much better load balancing and parallel construction and retrieval efficiency.

As this paper aims to exploit the efficient partitioning framework, we use original DA as a black box in this manuscript. However, we believe integrating some optimized DA, e.g., SAMC [12] into TLP, can further improve the construction efficiency or decrease storage overheads. Besides, developing more application fields for this structure, e.g., spatial queries [33], will also be an interesting research branch.

Author Contributions: Conceptualization, L.J. and C.Z.; methodology, L.J. and J.D.; software, L.J. and C.Z.; validation, L.J., C.Z. and Y.C.; writing, L.J. and C.Z.; resources, M.L., J.D. and Y.L.; review, L.J. and Y.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Natural Science Foundation of China under Grant no. 61562054, China Scholarship Council under Grant no. 201908530036, the National Key Research and Development Program of China under Grant no. 2018YFB1003904, the Talents Introduction Project of Guangxi University for Nationalities, under Grant no. 2014MDQD020.

Conflicts of Interest: There is no conflict of interest regarding the publication of this paper.

References

1. Yinong, C. *Service-Oriented Computing and System Integration: Software, IoT, Big Data, and AI as Services*; Kendall/Hunt Publishing Co.: Dubuque, IA, USA, 2017.
2. Bharti, H.; Saxena, R.K.; Sukhija, S. *Using Trie Structures to Efficiently Identify Similarities among Topical Subjects*; Google Patents: Mountain View, CA, USA, 2019.
3. Bhatia, M.A. Artificial Intelligence—Making an Intelligent personal assistant. *Indian J. Comput. Sci. Eng.* **2016**, *6*, 208–214.
4. Yang, L.; Xu, L.; Shi, Z. An enhanced dynamic hash TRIE algorithm for lexicon search. *Enterp. Inf. Syst.* **2012**, *6*, 419–432. [[CrossRef](#)]
5. Ao, X.; Luo, P.; Wang, J.; Zhuang, F.; He, Q. Mining precise-positioning episode rules from event sequences. *IEEE Trans. Knowl. Data Eng.* **2017**, *30*, 530–543. [[CrossRef](#)]
6. Huang, J.-Y.; Wang, P.-C. TCAM-based IP address lookup using longest suffix split. *IEEE/ACM Trans. Netw.* **2018**, *26*, 976–989. [[CrossRef](#)]
7. Lee, J.; Lim, H. Multi-Stride Decision Trie for IP Address Lookup. *IEIE Trans. Smart Process. Comput.* **2016**, *5*, 331–336. [[CrossRef](#)]
8. Li, G.; He, J.; Deng, D.; Li, J. Efficient Similarity Join and Search on Multi-Attribute Data. In Proceedings of the ACM SIGMOD International Conference on Management of Data, Victoria, Australia, 31 May–4 June 2015; pp. 1137–1151.
9. Jia, L.; Zhang, L.; Yu, G.; You, J.; Ding, J.; Li, M. A Survey on Set Similarity Search and Join. *Int. J. Perform. Eng.* **2018**, *14*, 245–258. [[CrossRef](#)]
10. Crochemore, M.; Hancart, C.; Lecroq, T. *Algorithms on Strings*; Cambridge University Press: Cambridge, MA, USA, 2007.
11. Aho, A.V.; Hopcroft, J.E. *The Design and Analysis of Computer Algorithms*; Pearson Education India: Chennai, India, 1974.
12. Fuketa, M.; Kitagawa, H.; Ogawa, T.; Morita, K.; Aoe, J.I. Compression of double array structures for fixed length keywords. *Inf. Process. Manag. Int. J.* **2014**, *50*, 796–806. [[CrossRef](#)]
13. Binna, R.; Zangerle, E.; Pichl, M.; Specht, G.; Leis, V. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In Proceedings of the International Conference on Management of Data, Houston, TX, USA, 11–15 June 2018; pp. 521–534.

14. Navarro, G.; Sadakane, K. Fully functional static and dynamic succinct trees. *ACM Trans. Algorithms (TALG)* **2014**, *10*, 1–39. [[CrossRef](#)]
15. Arroyuelo, D.; Cánovas, R.; Navarro, G.; Sadakane, K. Succinct trees in practice. In Proceedings of the Twelfth Workshop on Algorithm Engineering and Experiments (ALENEX), Austin, TX, USA, 16 January 2010; pp. 84–97.
16. Grossi, R.; Ottaviano, G. Fast compressed tries through path decompositions. *J. Exp. Algorithmics (JEA)* **2015**, *19*, 1–20. [[CrossRef](#)]
17. Kanda, S.; Morita, K.; Fuketa, M. Practical string dictionary compression using string dictionary encoding. In Proceedings of the 2017 International Conference on Big Data Innovations and Applications (Innovate-Data), Prague, Czech Republic, 21–23 August 2017; pp. 1–8.
18. Jacobson, G. Space-efficient static trees and graphs. In Proceedings of the 30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, NC, USA, 30 October–1 November 1989; pp. 549–554.
19. Rahman, N.; Raman, R. Engineering the LOUDS succinct tree representation. In Proceedings of the International Workshop on Experimental and Efficient Algorithms, Menorca, Spain, 24–27 May 2006; pp. 134–145.
20. Aoe, J.I. An Efficient Digital Search Algorithm by Using a Double-Array Structure. *IEEE Trans. Softw. Eng.* **1989**, *15*, 1066–1077. [[CrossRef](#)]
21. Zhang, H.; Lim, H.; Leis, V.; Andersen, D.G.; Kaminsky, M.; Keeton, K.; Pavlo, A. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In Proceedings of the International Conference on Management of Data, Houston, TX, USA, 10–15 June 2018; pp. 323–336.
22. Yata, S.; Oono, M.; Morita, K.; Fuketa, M.; Sumitomo, T.; Aoe, J.I. A compact static double-array keeping character codes. *Inf. Process. Manag.* **2007**, *43*, 237–247. [[CrossRef](#)]
23. Kanda, S.; Fuketa, M.; Morita, K.; Aoe, J.I. A compression method of double-array structures using linear functions. *Knowl. Inf. Syst.* **2016**, *48*, 55–80. [[CrossRef](#)]
24. Kanda, S.; Fujita, Y.; Morita, K.; Fuketa, M. Practical rearrangement methods for dynamic double-array dictionaries. *Softw. Pract. Exp.* **2018**, *48*, 65–83. [[CrossRef](#)]
25. Morita, K.; Fuketa, M.; Yamakawa, Y.; Aoe, J.I. Fast insertion methods of a double-array structure. *Softw. Pract. Exp.* **2001**, *31*, 43–65. [[CrossRef](#)]
26. Niu, S.; Liu, Y.; Song, X. Speeding Up Double-Array Trie Construction for String Matching. *Commun. Comput. Inf. Sci.* **2013**, *320*, 572–579.
27. Jia, L.; Chen, W.; Ding, J.; Yuan, X.; Shen, B.; Li, M. Collision Analysis and an Efficient Double Array Construction Method. *Int. J. Perform. Eng.* **2018**, *14*, 647–655. [[CrossRef](#)]
28. Aoe, J.I.; Morimoto, K.; Sato, T. An efficient implementation of trie structures. *Softw. Pract. Exp.* **1992**, *22*, 695–721. [[CrossRef](#)]
29. Korf, R.E. Multi-way number partitioning. In Proceedings of the IJCAI, Pasadena, CA, USA, 11–17 July 2009; pp. 538–543.
30. Korf, R.E. A hybrid recursive multi-way number partitioning algorithm. In Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence, Catalonia, Spain, 16–22 July 2011.
31. DBLP. Available online: <https://dblp.uni-trier.de/xml/> (accessed on 28 July 2019).
32. English Wikipedia. Available online: <https://dumps.wikimedia.org/enwiki/> (accessed on 1 September 2019).
33. Ganti, R.; Srivatsa, M.; Agrawal, D.; Zerkos, P.; Ortiz, J. MP-trie: Fast Spatial Queries on Moving Objects. In Proceedings of the Industrial Track of the International MIDDLEWARE Conference, Trento, Italy, 12–16 December 2016; p. 1.

