Inverted Index Construction Algorithms For Large-Scale Data

He Wang, Chengying Chi, Xiumei Zhang, Yunyun Zhan

***Abstract*—With the exponential growth of the amount of information on Internet web pages,the search efficiency and accuracy of search engines become a serious challenge. Inverted index as the core index structure of the search engines, its organization and storage have a significant impact on the performance of the search engines. To address the problem that the classical fast inversion algorithm (FAST-INV) cannot build inverted indexes quickly in the face of large-scale document data. This paper proposes two new inverted index construction algorithms: an improvement upon FAST-INV algorithm named FASTER-INV and a new algorithm AC-INV based on Aho-Corasick (AC) automaton for inverted index construction. Firstly, aiming at the redundancy of four information documents in FAST-INV, FASTER-INV is proposed to reduce two unnecessary information documents to build an inverted index. FASTER-INV cuts down redundant information while optimizing the memory space cost. Then this paper further proposes AC-INV, which combines the process of constructing <Doc_ID, Term_ID> pairs and inverted indexing. AC-INV saves significant memory occupation while ensuring the integrity of information. In addition, it eliminates the time of constructing information documents and improves the algorithm's scalability. Finally, experiments have been conducted on the Chinese AI and Law challenge dataset. The experimental results show that FASTER-INV and AC-INV proposed in this paper are better than FAST-INV. FASTER-INV's speed has increased by 1.11~1.14 times, and the memory has saved about 10%. AC-INV's speed has increased by 1.33~1.43 times, and the memory saved about 35%.**

***Index Terms*—Search engine, Full-text index, Inverted index, Fast inverted, Aho-Corasick automaton**

I. Introduction

THE search engine is a general term for a class of system or software, and its functions are to retrieve documents that match information needs. Search engines can perform full-text search in two ways, one by sequentially scanning all documents and the other by indexing. Sequential scanning is generally only suitable for processing a small number of documents or cached documents. Therefore, it is essential for search engine systems to build full-text indexes. The data structure, construction, and compression algorithms used for full-text indexes affect the search engine retrieval efficiency [1-2]. Researchers have proposed different data structures to support the construction of full-text indexes, such as signature files [3], suffix trees [4], and inverted indexes [5]. Xiaozhu Liu et al. [6] studied the performance of these three index structures, they concluded that the inverted index is the fastest and most scalable data structure for building full-text indexes under the condition that the storage space is large enough. Combining resources such as time, storage space, and processor, most common search engine systems usually construct indexes in an inverted way [7-10]. Therefore, optimizing inverted index construction algorithms is a research hotspot, such as reducing the time of building inverted files and improving data storage performance.

Fig. 1 shows the structure of the inverted index. An inverted index consists of a dictionary (a collection of terms) and an inverted list (a collection of postings lists). The postings list contains postings, and the posting contains document ID, word frequency, and the position of the word term in the document. Because the inverted index stores the inverted list for each term, search engines can find the documents associated with each term in a query through direct access, and retrieve matching documents quickly [11].

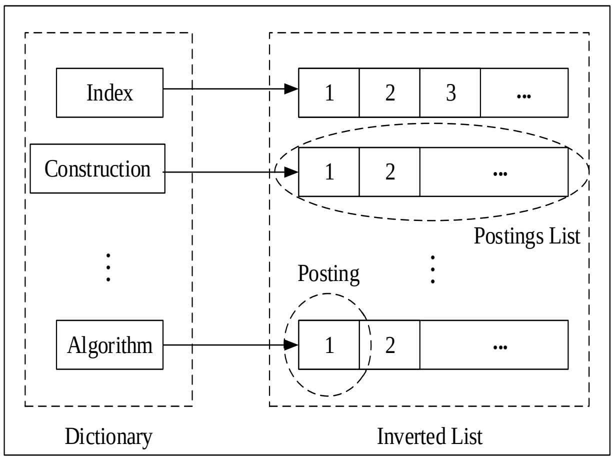

Fig. 1. Structure of the inverted index.

Fast inverted algorithm (FAST-INV) [12] is a classic inverted index construction algorithm, and it is suitable for building inverted indexes on some large-scale data. However, there is some redundancy between the information in the load

Manuscript received June 11, 2022; revised October 31, 2022. This work was supported in part by the University of Science and Technology Liaoning under Grant LKDYC201917

He Wang is a postgraduate student at the School of Computer Science and Software Engineering, University of Science and Technology Liaoning, Anshan, 114051, China(e-mail: ustlwanghe@163.com).

Chengying Chi is a professor at the School of Computer Science and Software Engineering, University of Science and Technology Liaoning, Anshan, 114051, China(e-mail: chichengying@ustl.edu.cn).

Xiumei Zhang is an associate professor at the School of Computer Science and Software Engineering, University of Science and Technology Liaoning, Anshan, 114051, China(corresponding author, phone: 86-186041281115; e-mail: aszxm2002@ustl.edu.cn).

Yunyun Zhan is an undergraduate student at the College of Science and Health, Technological University Dublin, Dublin, D08 X622, Ireland(e-mail: D16123420@mytudublin.ie).

tables, files of concept postings/pointers (CONPTR) and document vector loads files constructed by this algorithm. Building the above three data tables requires significant input and output (I/O) costs when facing large-scale data. Therefore, this paper optimizes the information redundancy of FAST-INV, new optimization algorithm is named faster inverted index algorithm (FASTER-INV). FASTER-INV eliminates the need to construct CONPTR and document vector loads files, the construction of inverted index only requires the input <Doc_ID, Term_ID> pairs and load tables to obtain enough information. FASTER-INV saves time and space compared to FAST-INV and eliminates the time overhead of reading in and out of CONPTR and document vector loads files.

As the number of dictionary terms increases continuously, the cost for I/O of the texts also increases gradually. Therefore, this paper proposes a novel AC automaton-based inverted index construction algorithm (AC-INV).The AC automaton algorithm is a string search algorithm proposed by Alfred V. Aho and Margaret J. Corasick [13]. It is used to match substrings in a finite dictionary in an input string and is an exact matching algorithm. AC automaton is built based on the trie structure combined with the ideas of KMP [14]. The algorithm provides solutions to many real-world problems and is one of the most fruitful algorithms in computer science [15-16]. AC-INV uses a trie for preprocessing to construct terms of the dictionary to achieve compression of the dictionary. Then, build inverted indexes and fail pointers based on the trie tree, thus constructing the AC automaton. Finally, based on the characteristics of the AC automaton, AC-INV can count all the terms in the dictionary by reading the document once. At the same time, using the insertion property of hash, the construction of the inverted index is completed in O(logn) time complexity. AC-INV reduces the time overhead in I/O compared with FASTER-INV. In the construction of the lexicon and intermediate process, not only dictionary term compression is performed, but also the building of the load tables is canceled, so that the inverted file can be constructed with less space overhead.

The contributions of the paper can be summarized as follows:

- FASTER-INV reduces the information redundancy of FAST-INV.
- AC-INV merges the process of constructing <Doc_ID, Term_ID> pairs and inverted index, reducing substantial time and space overhead.
- For large-scale data, this paper compares FASTER-INV and AC-INV with the classical FAST-INV to verify the performance of the algorithm.

II. Related Work

The inverted index can be realized by structures such as sorted arrays, B-trees, and Hashes [17]. The sorted array structure is the simplest and easiest to implement. Manning et al. [18] used sorted arrays to build the inverted index, and input texts had to be parsed into a list of terms and the position of the terms in the text. Sorted arrays need to be sorted continuously during the index building process. For large-scale document sets, there is not enough storage space to keep both sorted and unsorted versions of this list of terms. Gupta et al. [19] used B-tree to construct inverted indexes, which are easier to update, faster to retrieve, and more suitable for storing text indexes compared to sorted arrays. However, implementing inverted documents using B-tree is more complex than sorted arrays. Tan et al. [20] designed and combined hash functions to implement an upgraded inverted index to make the corresponding queries more accurate and memory efficient. The construction of the fundamental inverted index is improved, Heinz et al. [21] proposed the single-pass in-memory indexing(SPIMI). SPIMI eliminates the <term , document> pair mapping and sorting operations,it uses the term instead of the term_ID, writes the dictionary for each block to disk, and then starts a new dictionary for the next block. SPIMI can index collections of any size but requires sufficient available disk space. Fox et al. [12] proposed FAST-INV, which uses multiple memory loads to invert the file to use the disk optimally. FAST-INV provides time and space optimizations over sorted arrays for building inverted indexes, but there are still areas that can be optimized. The overall scheme of the FAST-INV algorithm is shown in Fig. 2.

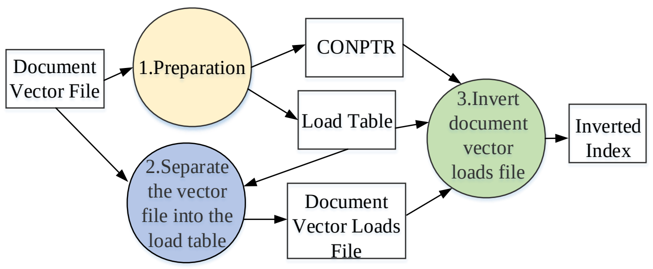
Document Vector File
1.Preparation
CONPTR
Load Table
2.Separate the vector file into the load table
Document Vector Loads File
3.Invert document vector loads file
Inverted Index

Fig. 2. Overall scheme of the FAST_INV algorithm.

FAST-INV is a fast inverted algorithm based on sorted arrays. The algorithm uses mem ory in a near-optimal way and processes the data through three operations. The input to FAST-INV is a document vector file. This document vector file contains the term_ID of each document, and each term value points to the same term-document association table. Notably, the document vector file is an ordered file, this file is sorted based on the term_ID and then sorted based on the doc_ ID, which is the key to the correct use of FAST-INV.To better explain FAST-INV, the definitions of the relevant terms used in the algorithm are listed below.

Definition 1. HCN is the total number of types of term ids in the dictionary.

Definition 2. L is the length of the document vector file, which is the length of the input.

Definition 3. M is the assumed memory size, which is the available memory space. The actual index file to be processed may require multiple disks for storage. Memory does not fit, so a flexible conversion between memory and external memory is required.

Definition 4. Assuming that $M >> HCN$, CONPTR and load table can be created in the main memory.

Definition 5. Assuming that $M << L$, multiple main memory load spaces are needed to process the document vector data.

Definition 6. LL is the current load length, equal to the number of <Doc_ID, Term_ ID> pairs.

Definition 7. S is the distribution of the number of terms in the current load, equal to the difference between the ending term_ID and the starting term_ID plus one.

Definition 8. The following constraint is imposed on the load space: 8 bytes is the space needed for each term_ID / weight pair. 4 bytes is the number of pointers to store for each term. Therefore, to fit the loaded data into memory, adding a term to the current load space must satisfy the requirement of $8 * LL + 4 * S < M$.

The pseudo code of the FAST-INV algorithm is shown in Table I.

TABLE I
PSEUDO CODE OF FAST-INV

```
Algorithm 1: FAST-INV
Input: Compression matrix formed by <Doc_ID, Term_ ID> pairs
Output: Inverted_index
1. read_vector←Ø
2. Init( $HCN$ , $L$ , $M$ )
3. for i in range(read_vector)
4.   con_entries.push_back(i)
5. for i in con_entries
6.   if ( $8 * LL + 4 * S < M$ )
7.     post/points++
8.   else post/point.push_back(i)
9. build_load(post_points)
10. for i in Load table , j in post/points
11.   if (con < Load_table[i].EndCon )
12.     D_V_L_F.push(Doc, Con)
13.   else
14.     D_V_L_F++
15. for i in Load table , j in post/points , k in D_V_L_F
16.   if(con < Load_table[i].EndCon)
17.     Inverted_index.push(Build(i, j, k))
18.   else
19.     Inverted_index.add;
20. return Inverted index
```

Algorithm 1 has five steps:

Step 1. [Algorithm initialization] (lines 1-4): The initialization function stores the read <Doc_ID, Term_ID> pairs and updates global variables such as HCN, L, and M.

Step 2. [Generate CONPTR] (lines 5-8): When looping through <Doc_ID, Term_ID> pairs, if the number of terms exceeds the constraint defined by the current load space, new load space is created in the CONPTR. If the current bound is satisfied, then add operation is performed in the load space of the current CONPTR.

Step 3. [Generate load table] (line 9): Generate the starting term_ID, ending term_ID, kind of term_ID, and number of term_ID of load tables by the term_ID, starting address number, number of existing documents, and load table number of the CONPTR.

Step 4. [Generate document vector loads file] (line 10-14): Loop through the generated load tables and the CONPTR. Determine if the constraint of the current document vector loads file is exceeded, and if it is in the current constraint, push the current <Doc_ID, Term_ID> pairs into the document vector loads file. Instead, apply a new document vector loads file.

Step 5. [Generate inverted index] (line 15-20): Loop through the generated CONPTR, load tables, and document vector loads files. If the current term_ID does not exceed the end term_ID of the load table, press all the generated data into the inverted index. If not, request a new inverted index. Finally, return the generated inverted index.

It can be seen that Algorithm 1 generates four intermediate table documents and performs three times I/O. Let the number of lexical items be *W*, the number of documents is *D*, and the average document length is *DL*. The time complexity of Algorithm 1 is *O*(*W***D***DL*), and the additional space complexity of Algorithm 1 is *O*(*W***D*). The optimization of Algorithm 1 will be given in the FASTER-INV algorithm section in Section 3.

III. FASTER-INV ALGORITHM

A. Algorithm Description

The FASTER-INV is an optimized and improved algorithm based on FAST-INV, which suffers from information redundancy.FASTER-INV achieves temporal and spatial optimization of the FAST-INV algorithm by trimming down the CONPTR and the document vector loads file in the process of building the inverted index. The pseudo code of the FASTER-INV algorithm is shown in Table II.

TABLE II
PSEUDO CODE OF FASTER-INV

```
Algorithm 2: FASTER-INV
Input: Compression matrix formed by <Doc_ID, Term_ ID> pairs
Output: Inverted_index
1. read_vector←Ø
2. Init( $HCN$ , $L$ , $M$ )
3. for i in range(read_vector)
4.   con_entries.push_back(i)
5. for i in con_entries
6.   if ( $8 * LL + 4 * S < M$ )
7.     Load_table++
8.   else
9.     Load_table.push_back(i)
10. for i in con_entries j in Load_table
11.   if (cies[i].con_entron < Load_table[j].EndCon)
12.     Inverted_index.push_back(i, j)
13.   else
14.     Inverted_index.add
15. return Inverted index
```

Algorithm 2 has three steps:

Step 1. [Algorithm initialization] (lines 1-4): The implemented function is the same as Algorithm 1.

Step 2. [Generate load table] (line 5-9): Loop through the con_entries array to count document_ID and term_ID. Determine whether the lexical items that join the current load table conform to the space constraints, and update the information of the load table when it meets the constraint. If the constraint is not satisfied, apply a new load table.

Step 3. [Generate inverted index] (line 10-15): Loop through the con_entries array and the load table. When the term_ID is greater than the end term_ID of the current load table, a new inverted index is applied. If it is smaller than the end term_ID of the current load table, the read information is added to the current inverted index. Finally, return the generated inverted index.

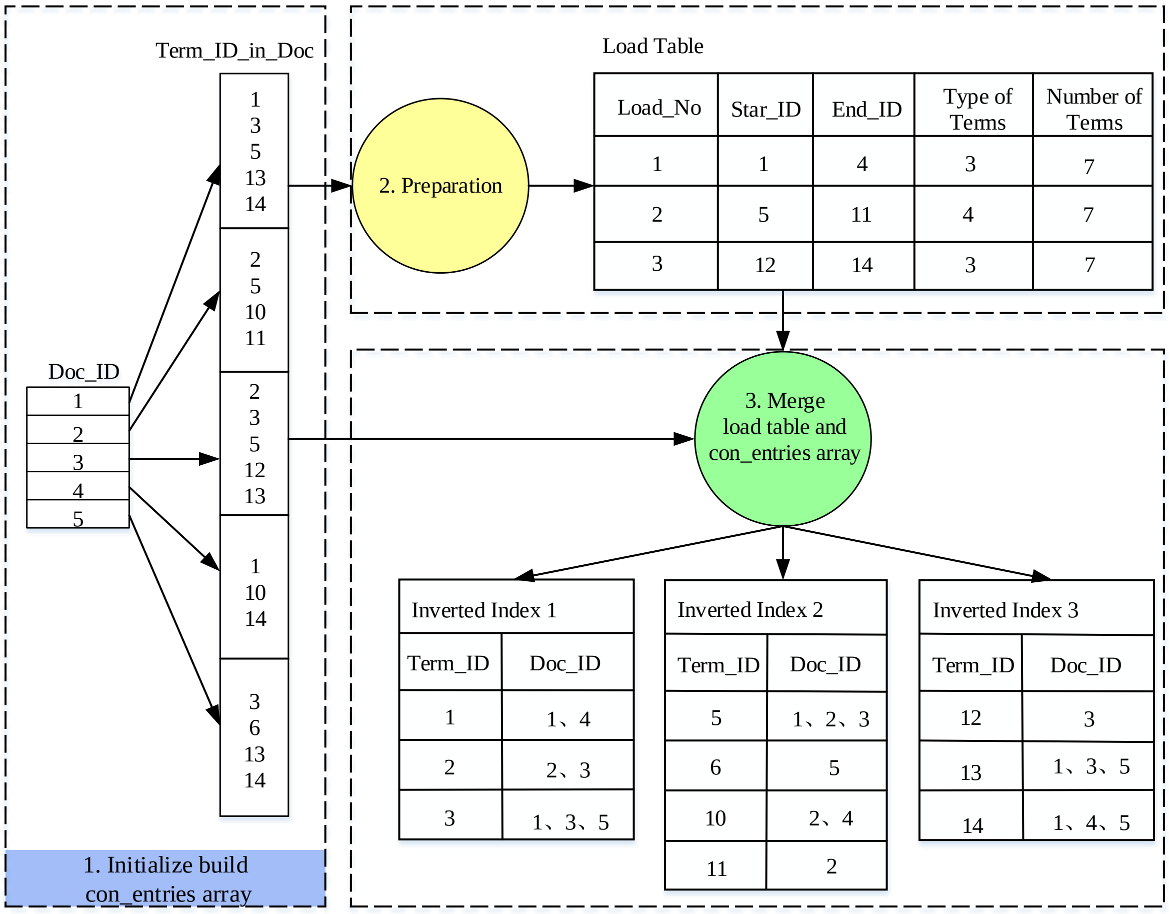

Fig. 3. FASTER-INV algorithm to construct inverted index.

B. *FASTER-INV Algorithm Example*

The document ID set Doc_ID = {1, 2, 3, 4, 5} and these five documents correspond to five lexical item ID vector files Term_No_in_Doc = {{1, 3, 5, 13, 14}, {3, 5, 10, 11}, {2, 3, 5, 12, 13}, {1, 10, 13, 14}, {3, 6, 13, 14}} as an example to illustrate the FASTER-INV to construct inverted index process, the detailed process is shown in Fig. 3.

In the first stage, the FASTER-INV algorithm reads the compression matrix constructed by the <Doc_ID, Term_ID> pairs and initializes the variables according to the read data. Then the FASTER-INV algorithm transforms the compression matrix into an array of con_entries through the data's relationship. Documents 1, 2, 3, 4, and 5 in Doc_ID point to the 1st, 2nd, 3rd, 4th, and 5th vectors in Term_ID_in_Doc. These documents' data denote the information about the term_ID that appears in the documents.

In the next stage, the load table is constructed based on the data of the con_entries array. The load table generates the corresponding starting term_ID, ending term_ID, kind of term_ID, and number of term_ID in each of the three load tables based on the data of the con_entries array and the constraints of the load table.

In the last stage, match the information in the con_entries array and load table to generate the inverted indexes. The information in load tables {1, 2, 3} is used to construct the corresponding three inverted indexes.

C. *Algorithm Complexity Analysis*

FASTER-INV is optimized in constant space compared to the FAST-INV algorithm. The FASTER-INV algorithm performs I/O operations two times in constructing the con_entries array and the load table. Thus the time complexity of FASTER-INV is $O(W*D*DL)$. Compared to the FAST-INV algorithm, the FASTER-INV algorithm is optimized in constant time. The additional space complexity is $O(W*D)$.

IV. AC-INV Algorithm

A. *Algorithm Description*

AC-INV is an algorithm for constructing an inverted index based on the multi-pattern matching characteristics of AC automaton. After the AC-INV algorithm uses the AC automaton to count the information of terms, the AC-INV algorithm uses the hash mapping to construct the corresponding inverted index. AC-INV algorithm compares with FASTER-INV algorithm, AC-INV algorithm saves the process of building the con_entries array and load table. It further reduces the waste of information redundancy. Thus, it achieves better improvements in time and space than the FASTER-INV algorithm. Before the description of the AC-INV algorithm, definitions of relevant terms used in the algorithm are given as follows:

Definition 9. P = {p_1, p_2, ..., p_n} represents the set of pattern strings. p_1, p_2, ..., p_n denotes the encoded pattern string of the term, where n denotes the number of pattern strings entered.

Definition 10. Node = {S_0, S_1, ... S_m} represents the state set of the AC automaton, where m denotes the number of nodes of an AC automaton. EndNode = {S_0, S_1, ... S_i} represents the set of end states. EndNode is a subset of the state set of the AC automaton.

Definition 11. Each node of the trie has ten subnodes, and each node corresponds to a pointer that represents the integer [0, 9] in the number.

The pseudo code of the AC-INV algorithm is shown in Table III.

TABLE III
PSEUDO CODE OF AC-INV

```
Algorithm 3: AC-INV
Input: Terms to count and Documents to consult
Output: Inverted_index
1. read←Ø
2. for i in read:
3.   s = make_word_to_figure(i)
4.   insert(s)
5. while(!queue<Node*> P.empty( ) )
6.   cur = P.front( ),P.pop( )
7.   for (int i = 0 ; i < 10 ; i++)
8.     if (cur->Next[i])
9.       cur->Next[i]->fail = root
10.      cfail = cur -> fail
11.      While (fail)
12.        if (fail->Next[i])
13.          cur->Next[i]->fail = cfail->Next[i]
14.          break
15.    P.push(cur->Next[i])
16. read_article←Ø
17. for i in read_article:
18.   if ($8 * LL + 4 * S < M$)
19.     inverted_index.push(Map<con,vector<Doc>>)
20.   else
21.     inverted_index.add
22. return inverted index
```

Algorithm 3 has three steps:

Step 1. [Construct terms into trie] (lines 1-4): Read terms into memory sequentially, then iterate over the terms. Encode terms into ASCII codes and push them codes into the trie.

Step 2. [Construct fail pointers] (lines 5-15): Determine whether the node in the queue is empty, if the node is not empty, perform the queue exit operation. Iterate through the children nodes of the current node and determine whether the ith pointer of the current node is empty or not. If it is not empty, point the fail of the ith pointer of the current node to the root node. And also, point the cfail pointer to the fail pointer of the current node. When fail points to non-empty, determine whether the ith pointer of the fail pointer is empty. If not empty, point the ith pointer of the current node's fail pointer to the ith pointer of the cfail pointer, and then jump out of the loop. Finally, the ith pointer of the current pointer is pushed into the queue.

Step 3. [Read documents and Construct inverted indexes] (lines 16-22): Read documents to be queried in sequence. Iterate through the read documents and constrain the memory size occupied by the current inverted index. If the current inverted index satisfies the constraint, a series of statistics of the word items are stored in the inverted index ,the form is mapping. If not, a new inverted index is requested. Finally, return the generated inverted index.

B. AC-INV Algorithm Example

The following is an example of {"法律(law)", "律师(lawyer)", "大法官(grand justice)"} to illustrate the process of constructing the inverted index by the AC-INV algorithm.

In the first stage, Encode {"法律(law)", "律师(lawyer)", "大 法 官 (grand justice)"} as {"24781", "781620", "14152436"}. A trie is constructed based on the encoding of each term. The constructed trie is shown in Fig. 4.

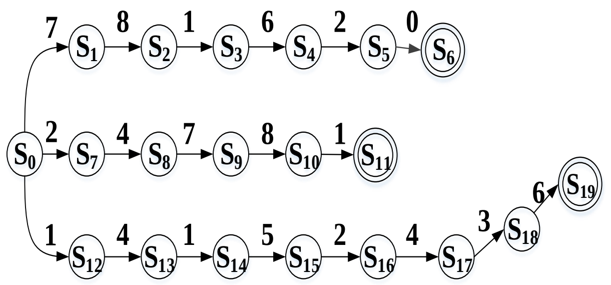

Fig. 4. Trie constructed from {"24781", "781620", "14152436"}.

In the next stage, the AC-INV algorithm constructs fail pointers to generate an AC automaton. Fig. 5 shows the process of constructing fail pointers. This description of the algorithm is ordered by dictionary.As shown in Fig. 5, if the fail pointer of the parent node of the current node points to a node with the same number as the current node， the curent node's fail pointer of the current node poiont to the node of the fail pointer of the parent node of the current node, otherwise the trie-tree make the point the fail pointer of the current node to the root node. Follow this rule, the trie-tree builds fail pointers to "14152436", "24781", and "781620" in turn.

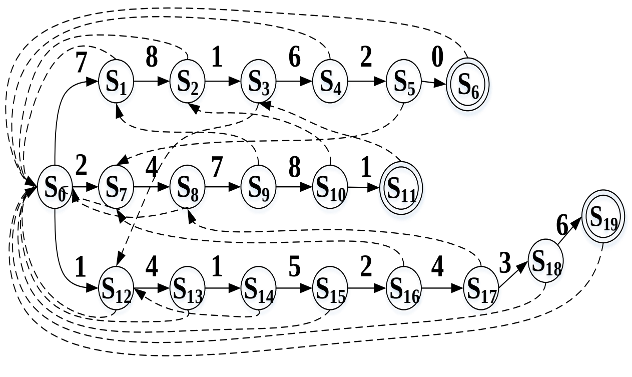

Fig. 5. Construct fail pointers to generate AC automaton.

In the last stage, Fig. 6 shows the procedure for reading the document and performing inverted index construction by the AC-INV algorithm. The rule of multi-mode matching of AC automata: According to the matching current node, if there is a matching node, it goes to the next node. Otherwise, traverse to the fail pointer pointing to the current node's parent node. Repeat the rule until the match is successful, or iterate through the fail pointers until the root node. As shown in Fig. 6(a) and Fig. 6(b), "24781624781" and "7814152436" are patterns matching the fragments in the document in turn. Finally, the inverted index on the right side of the figure are formed in turn.

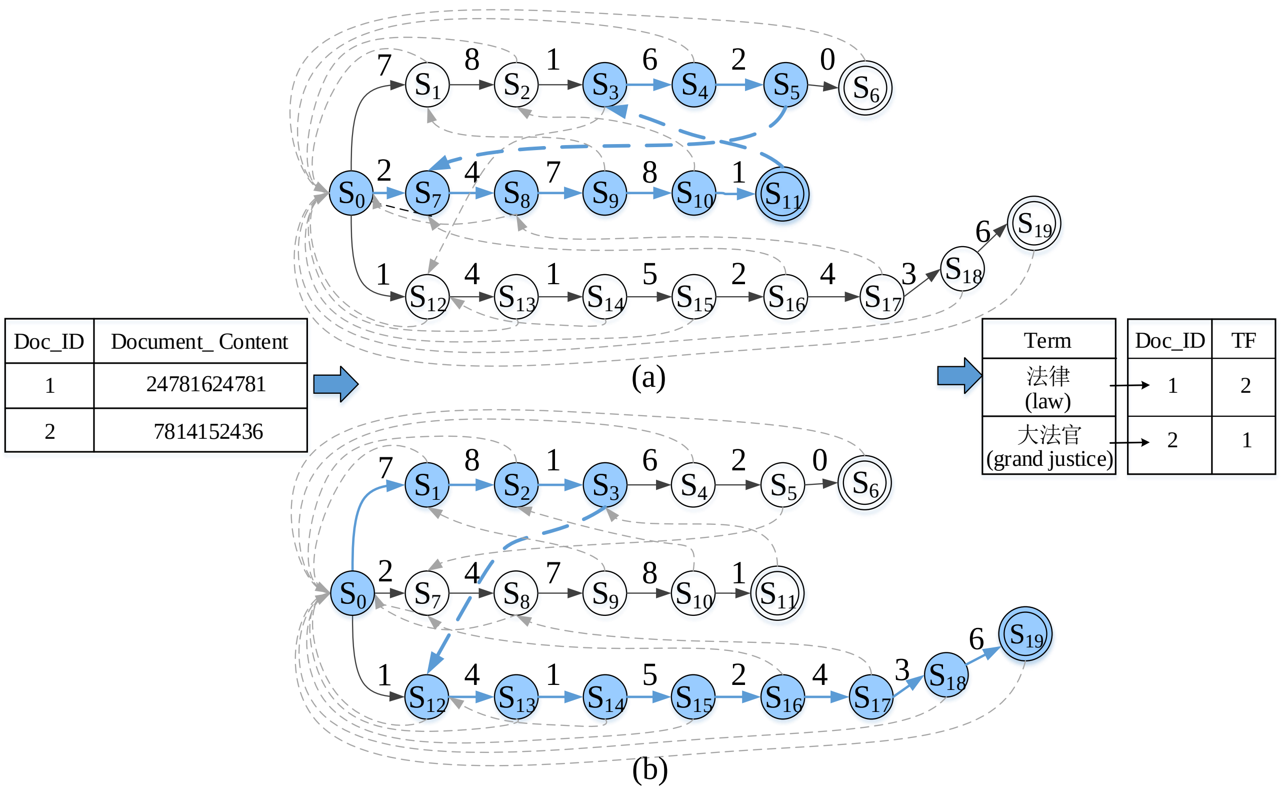

Fig. 6. AC-INV algorithm to construct inverted index.

C. *Algorithm Complexity Analysis*

Let *N* be the number of nodes of the AC automaton, then the time complexity of the AC-INV algorithm is $O(3*W+10*N+3*D+W*log(W))$. Because $W*log(W)$ is the smallest, its time complexity is negligible. Therefore the final time complexity of the AC-INV algorithm is $O(W+N+D)$. The additional space complexity of AC-INV is $O(W+D)$.

V. Experiments

A. *Experimental Setup and Dataset*

The experiments in this paper focus on comparing the space/time efficiency between our proposed algorithms and the FAST-INV. Since the final construction of inverted indexes by the three algorithms is the same, the evaluation criteria of the experiments are as follows：

- Time for the algorithm to construct the inverted index
- Memory occupation of the algorithm

The basic configuration of the machine used for the experiments is 11th Gen Intel(R) Core(TM) i7-1160G7 @ 1.20GHz 2.11 GHz, 16.0 GB RAM, 0.5 TB hard disk, and Window 10 operating system. All algorithms are implemented in C++ and compiled by Dev-C++ 5.10 compiler.

The experimental dataset is the Chinese AI and Law challenge dataset (CAIL2018), a large-scale Chinese legal dataset constructed by the "China Law Research Cup" Judicial AI Challenge [22]. The dataset consists of public case documents from the China Judicial Documents website. Each sample case in the dataset contains information such as factual descriptions, applicable laws, crimes, and sentences. These experiments will construct inverted indexes for the CAIL2018-Small and CAIL2018-Large datasets released by the competition. The basic information of the datasets used in the experiments is given in Table IV.

TABLE IV
Datasets used in the experiments

Dataset	CAIL2018-Small	CAIL2018-Large
No. of documents	200000	1500000
Average length	693	660
Dataset size（MB）	131.56	941.37
Type	Long text	Long text

B. *Experiment Results and Analysis*

1) Construction Time

Table V shows the comparison experiment results of the time required to construct inverted indexes on different datasets by the three algorithms. It is clear from Table V that the inverted index construction for the CAIL2018-Small dataset takes 127.077s for FASTER-INV and 106.231s for AC-INV. The inverted index construction for the CAIL2018-Large dataset takes 969.254s for FASTER-INV and 776.331s for AC-INV. The FASTER-INV and AC-INV proposed in this paper outperform FAST-INV and improve the speed of inverted index construction.

TABLE V
Experiment results of inverted index construction time

Algorithm	Algorithm Execution Time(s)	
	CAIL2018-Small	CAIL2018-Large
FAST-INV	141.416	1101.347
FASTER-INV	127.077	969.254
AC-INV	106.231	776.331

Table VI shows the speedup (the ratio of FAST-INV to the execution time of the algorithm proposed in this paper) of FASTER-INV and AC-INV. On the CAIL2018-Small dataset, the speedup of FASTER-INV and AC-INV amount to 1.11 and 1.33, respectively. On the CAIL2018-Large dataset, the speedup of FASTER-INV and AC-INV amount to 1.14 and 1.43, respectively. It is worth noting that the AC-INV algorithm accelerates significantly, obtaining a maximum of 41.9% acceleration.

TABLE VI
EXPERIMENT RESULTS OF SPEEDUP

Algorithm	Speedup	
	CAIL2018-Small	CAIL2018-Large
FASTER-INV	1.11	1.14
AC-INV	1.33	1.43

To better analyze the algorithm performance of FAST-INV, FASTER-INV, and AC-INV. This paper conducted experiments on the time and speedup ratio cases of constructing inverted indexes at different data sizes, and the experimental results as shown in Fig. 7 and Fig. 8.

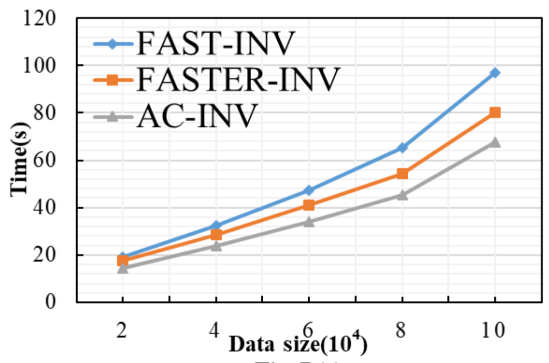

Fig. 7 (a)

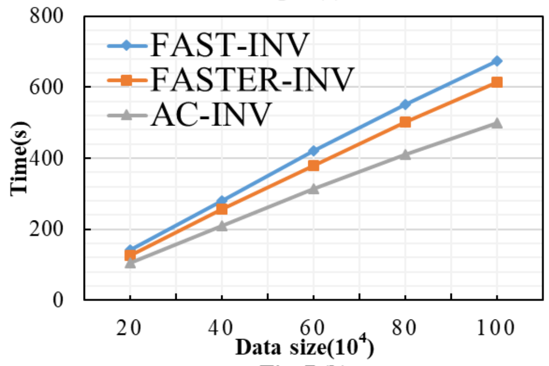

Fig. 7 (b)

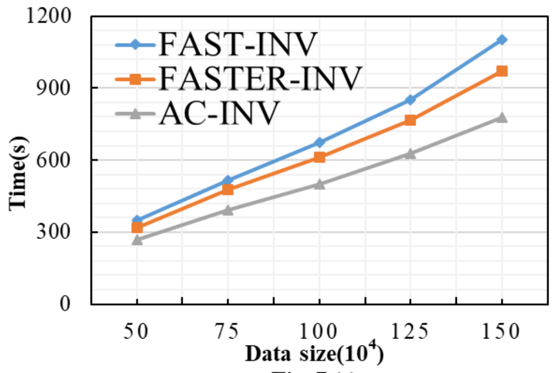

Fig. 7 (c)

Fig. 7. Inverted index construction time at different data size.

Fig. 7 shows the execution time variation of these three algorithms at different data sizes. The execution time of the three algorithms grows approximately linearly as the data size increases. The FASTER-INV and AC-INV algorithms take less time to construct the inverted index than FAST-INV, and AC-INV takes the least time. The reason is that the FASTER-INV algorithm eliminates the need to construct the CONPTR and the document vector loads file compared to the FAST-INV algorithm. The AC-INV algorithm replaces the CONPTR, loading table, and document vector loads file in the FAST-INV algorithm by the algorithmic feature of multi-pattern matching of AC automaton.it combined with the advantage of the hash function to find matching string information quickly. Thus AC-INV eliminates the time for constructing redundant information and achieving a time advantage over FAST-INV and FASTER-INV algorithms.

Fig. 8 shows the variation of the algorithm speedup of FASTER-INV and AC-INV with the size of the data. The AC-INV and FASTER-INV speedup gradually increase with the increase of the data set, and the speedup of AC-INV is significantly more effective than that of FASTER-INV. Because FASTER-INV needs to traverse the data twice to construct the load table and con_entries array, the speedup is not as effective as the AC-INV algorithm. Therefore, the AC-INV algorithm proposed in this paper can guarantee good construction efficiency despite the large data size.

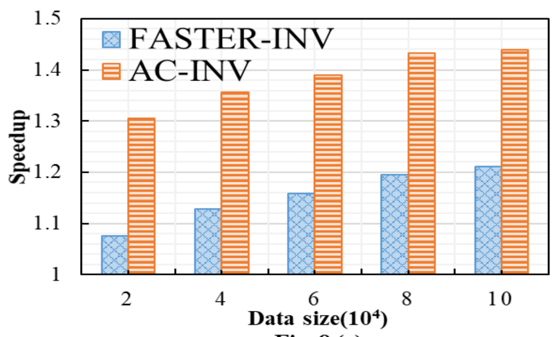

Fig. 8 (a)

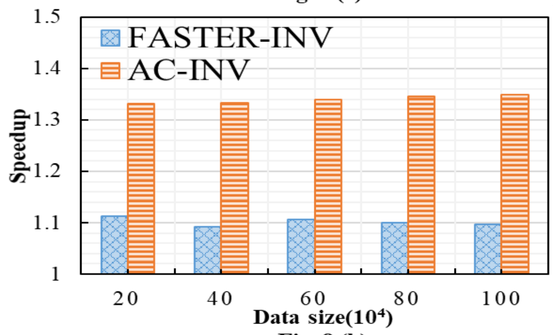

Fig. 8 (b)

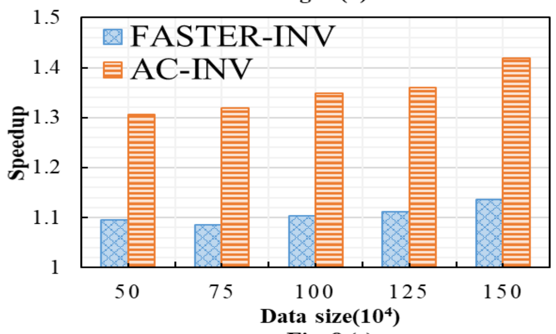

Fig. 8 (c)

Fig. 8. Speedup for algorithms at different data size.

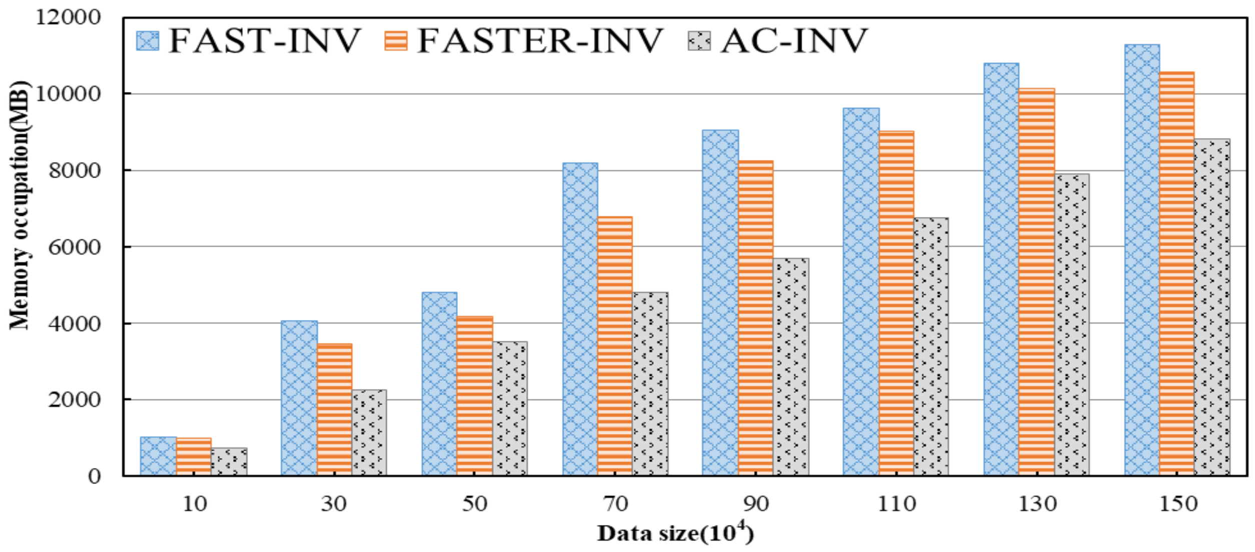

Fig. 9. Comparison of memory occupation for constructing inverted index.

2) Memory Occupation

The algorithm calls memory during execution, and the less memory called for equal data indicates that the algorithm has better space complexity. The less memory increase during the increase of data set size suggests that the data has less impact on memory, and the algorithm scales better. Table VII shows the experimental results of the computer memory usage when the three algorithms construct inverted indexes on different data sizes. Fig. 9 visualizes the experimental results to enable a clearer analysis and comparison of the memory usage of the algorithm execution.

From Table VII and Fig. 9, it is clear that the memory occupation of the FASTER-INV algorithm is smaller than that of the FAST-INV algorithm, reducing it by about 10%. Because the FAST-INV algorithm needs to generate four information files for mapping relationships to build inverted index files, this process consumes lots of memory. The FASTER-INV algorithm only needs to produce two information files, reducing the memory occupation. The memory occupation of the AC-INV algorithm is smaller than that of the FASTER-INV algorithm. The memory occupation of the AC-INV algorithm is reduced by about 35% compared to the FASTER-INV algorithm. Because the AC-INV algorithm reduces the memory occupation by eliminating the construction of two information files compared to the FASTER-INV algorithm. It shows that the AC-INV algorithm has the lowest space complexity, the FASTER-INV algorithm has better space complexity than the FAST-INV algorithm, and the FAST-INV algorithm has the worst space complexity.

TABLE VII
EXPERIMENT RESULTS OF MEMORY OCCUPATION

Data Size(10^4)	Algorithm Memory Occupation(MB)		
	FAST-INV	FASTER-INV	AC-INV
10	1017	994	735
30	4061	3464	2259
50	4812	4188	3527
70	8198	6789	4820
90	9049	8255	5688
110	9634	9023	6770
130	10811	10147	7894
150	11293	10571	8823

VI. CONCLUSION

In this paper, we designed two new algorithms to construct inverted indexes for large-scale data: FASTER-INV and AC-INV. The FASTER-INV algorithm is an improved and optimized algorithm for the classical inverted index construction algorithm FAST-INV, which avoids unnecessary information documents and improves the construction time and space consumption. AC-INV is an algorithm for constructing an inverted index based on an AC automaton. The algorithm first builds a trie and fail pointers from the terms in the dictionary to form an AC automaton. Then, the encoded documents are pushed into the AC automaton one by one, and the information of the terms contained in each document is counted. At the same time, inverted index is constructed based on relevant constraints. Compared with the FASTER_INV algorithm, the AC-INV algorithm reduces the cost of significant I/O and merging overhead time.

Theoretical analysis and experiments on real datasets demonstrated that the time and memory consumption of FASTER-INV algorithm constructing inverted indexes are smaller than FAST-INV. The speedup process of large-scale data improves by 1.11-1.14 times. However, when the number of documents is large, the memory consumption optimization of the FASTER-INV algorithm will be weakened. The AC-INV algorithm can save time and reduce space overhead. It can construct a much larger inverted index while consuming the same memory. The AC-INV algorithm is better than the FAST-INV and FASTER_INV algorithms in terms of performance because it is less affected by the length of the document and the number of them. Compared with the first two algorithms, AC-INV speedup of large-scale data processing improves by 1.33-1.43 times.

REFERENCES

[1] A. Moffat, and J. Zobel, “Self-indexing inverted files for fast text retrieval,” *ACM Transactions on Information System*, vol. 14, no. 4, pp. 349-379, 1996.

[2] E. Naufal and J. R. Tom, “IIU: Specialized architecture for inverted index search,” in *Proc. ASPLOS*, New York, NY, USA, pp. 1233–1245, 2020.

[3] B. Carterette, and F. Can, “Comparing inverted files and signature files for searching a large lexicon,” *Information Processing & Management*, vol. 41, no. 3, pp. 613-633, 2005.

[4] E. M. McCreight, “A space-economical suffix tree construction algorithm,” *Journal of the ACM*, vol. 23, no. 2, pp. 262-272, 1976.
[5] J. Zobel, A. Moffat, ans K. Ramamohanarao, “Inverted files versus signature files for text indexing,” *ACM Transactions on Database Systems*, vol. 23, no. 4, pp. 453-490, 1998.
[6] Z. Zhang, P. Q. Jin, and X. K. Xie, “Learned Indexes: Current Situations and Research Prospects,” *Ruan Jian Xue Bao/Journal of Software*, vol. 32, no. 4, pp. 1129-1150, 2021.
[7] L. Wang, T. D. Zhou, and Z. F. Wang, “Search on encrypted electronic medical records using inverted index based on bloom filter and B+tree,” *Computer Applications and Software*, vol. 38, no. 4, pp. 276-280, 2021.
[8] N. Sousa, N. Oliveira, and I. Praça, “Machine Reading at Scale: A Search Engine for Scientific and Academic Research,” *Systems*, vol. 10, no. 2, pp. 43, 2022.
[9] K. Figueroa, A. Camarena-Ibarrola, and N. Reyes, “Shortening the Candidate List for Similarity Searching Using Inverted Index,” in *Proc. MCPR*, Mexico City, Mexico, pp. 89-97, 2021.
[10] J. Lin, “A proposed conceptual framework for a representational approach to information retrieval,” in *Proc. SIGIR*, New York, NY, USA, pp. 1-29, 2022.
[11] X. Yu, “Construction and Application on Parallel Corpus for College Japanese Translation Teaching,” in *Proc. ICISCAE.* Dalian, LN, China, pp. 1706-1710, 2021.
[12] E. Fox, and W. Lee, “FAST-INV: A fast algorithm for building large inverted files,” Tech. rep. TR 91-10, Virginia Polytechnic Institute and State University, Blacksburg, VA, 1991.
[13] A. V. Aho, and M. J. Corasick, “Efficient string matching: an aid to bibliographic search,” *Communications of the ACM*, vol. 18, no. 6, pp. 333-340, 1975.
[14] D. E. Knuth, J. H. Morris, and V. R. Pratt, “Fast pattern matching in strings,” *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323-350, 1977.
[15] S. Hasib, M. Motwani, and A. Saxena, “Importance of aho-corasick string matching algorithm in real world applications,” *Journal of Computer Science and Iinformation Technologies*, vol. 4, no. 3, pp. 467-469, 2013.
[16] A. R. Chayapathi, “Survey and Comparison of String Matching Algorithms,” *Turkish Journal of Computer and Mathematics Education* , vol. 12, no. 12, pp. 1471-1491, 2021.
[17] A. Mohamed, M. Abdel-Fattah, and A. Khedr, “Challenges and recommendationsin big data indexing strategies,” *International Journal of e-Collaboration*, vol. 17, no. 2, pp. 22–39, 2021.
[18] C. Manning, P. Raghavan, and H. Schütze, “Introduction to information retrieval,” *Natural Language Engineering*, vol. 16, no. 1, pp. 100-103, 2010.
[19] A. Gupta, D.Yadav, “A novel approach to perform context-based automatic spoken document retrieval of political speeches based on wavelet tree indexing,” *Multimedia Tools and Applications*, vol. 80, no. 14, pp. 22209-22229, 2021.
[20] C. C. Tan, B. Sheng, H. Wang and Q. Li, “Microsearch: A search engine for embedded devices used in pervasive computing,” *ACM Transactions on Embedded Computing Systems*, vol. 9, no. 4, pp. 1-29, 2010.
[21] S. Heinz, J. Zobel, “Efficient single-pass index construction for text databases,” *Journal of the American Society for Information Science and Technology*, vol. 54, no. 8, pp. 713-729, 2003.
[22] C. Xiao, H. Zhong, Z. Guo, C. Tu, Z. Liu, M. Sun, et al. , “CAIL2018: A large-scale legal dataset for judgment prediction,” arXiv:1807.02478, 2018. Available: https://arxiv.org/abs/1807.02478.