Efficient Grammar Generation for Inverted Indexes

Yan Fan[1], Xinyu Liu[1], Shuni Gao[1], Zhaohua Zhang[1], Xiaoguang Liu[1,⋆], and Gang Wang[1,⋆]

College of Computer and Control Engineering, Nankai University, China
{fanyan, liuxy, gaoshn, zhangzhaohua, liuxg, wgzwp}@nbjl.nankai.edu.cn

Abstract. Inverted indexes are commonly utilized in large-scale search engines to store lists of document identifies (docIDs) relevant to query terms, which are queried maybe thousands of times per second. Traditionally, optimized integer sequence encoding methods are applied to compress the inverted index while simultaneously maintaining reasonable query processing speeds. Recently, a context-free grammar-based method was introduced for inverted index compression, which is particularly useful for highly repetitive indexes.
Due to the high time and space cost of the traditional grammar generation (transform) algorithms designed for large inverted index collections with much redundancy, we propose a parallel generation algorithm for context-free grammar generation. We further propose a greedy dictionary pruning algorithm to reduce cache misses in query processing. We also implement encoding, list intersection, and WAND querying on the grammar index. Experimental results indicate that parallel grammar generation algorithm achieves a super-linear speedup with minor data overhead and nearly identical query efficiency compared to the single-threaded algorithm. For example, with 10 threads to process the data set, a speedup about 75 times faster is obtained with only 4.3% data overhead. Moreover, parallel grammar generation incurs negligible impact on query processing efficiency.

Keywords: inverted index compression, context-free grammar, parallel grammar generation algorithm

1 Introduction

In the compression of inverted indexes, both the compression ratio and decompression speed are important for performances of search engines, because they measure the space cost and retrieval efficiency, respectively. Previous work on index compression focused on encoding methods to compress document identifiers (docIDs, which are integers). Typically, docIDs are stored in increasing

⋆ Corresponding authors. This work is partially supported by NSF of China (grant number: 61602266), Science and Technology Development Plan of Tianjin (grant numbers: 17JCYBJC15300, 16JCYBJC41900).

order, so the differences between consecutive docIDs are often smaller than the docIDs themselves. As such, it is natural to replace the inverted lists by lists of differences before encoding; we refer to such lists as *differential lists* (or *Delta lists* for short).

A different approach to compressing inverted indexes, called *grammar-based compression*, is to remove the duplicate data among lists in the index through the use of context-free grammar. Common subsequences, called *patterns*, of docIDs (or differences between successive docIDs) are replaced by a reference to a single record in the *dictionary*. In this paper, the grammar generated on *Delta lists* is called *Delta grammar*.

There are two drawbacks to grammar-based compression: (a) The grammar generation process is time consuming and requires a large amount of memory, especially for large-scale collections (for example, in modern search engines). (b) The performance, in terms of compression and query response time, is dependent on the grammar dictionary. A larger dictionary allows more patterns to be identified, which results in better compression but slower query processing. In this paper, our contributions are:

- We propose a parallel grammar generation algorithm called *PIPara* (Pattern Identification in Parallel) to speed up grammar generation using *PISequential* in [11].
- We design a greedy dictionary pruning strategy which benefits query processing.
- We implement Delta grammar encoding, TAAT AND query and DAAT WAND query methods.

2 Preliminaries and Related Work

2.1 Inverted-Index Compression

Here, we study some basic arithmetic coding schemes, one of which we use throughout this paper.

Binary interpolative coding [5] encodes an increasing sequence by recursively splitting the sequence into two parts and encoding the middle integer, which achieves a high compression ratio, but poor decompression speed. Simple (along with its variants) is also designed to use different bits to compress integer sequences, but achieves a better trade-off between compression and decompression speed, like the widely used compression method Simple16 [10].

Another popular encoder is PForDelta [12] where the least significant b bits are used to represent most integers (usually around 90%) and the integers requiring more bits (exceptions) are stored and compressed separately. OptPForDelta [8], a block-wise version of PForDelta, instead selects an appropriate number of exceptions for each block, with a block size usually 128 or 256 [5]. In this paper, OptPForDelta and Simple16 are used as the Delta grammar encoder for testing.

2.2 Grammar-based Compression

A *context-free grammar* is a quadruple $G = (V, T, S, P)$ where T is a finite set with cardinality ≥ 2 called the *alphabet*, and elements in T are called *terminals*, V is another finite set (disjoint to T) whose elements are called *non-terminals* (NT), the unique non-terminal $S \in V$ is the *start symbol*. Any terminal or non-terminal is called a *symbol*. Set P is a finite set of *production rules* $A \to \alpha$, where $A \in V$ and α is a sequence of symbols called the *definition* or right-hand side of A. For any production like $A \to \alpha$, if we replace non-terminals in α recursively, we obtain a string only consisting of terminals, we call this operation *derivation*; We call the reverse *reduction*. For any NTs A and B, we say A equals B if the strings obtained by deriving A and B respectively are the same which we denote $A = B$. For the start symbol S, there is a production rule $S \to \beta$ in P, from which the original data can be recovered by replacing non-terminals recursively in β. In the context of grammar-based inverted index compression, however, the original data is a collection of posting lists rather than a single string, so we modify the notation for clarity. An index grammar is a quintuple $G = (V, T, P, R, S)$, where S becomes a set of non-terminals $\{l_1, l_2, \ldots, l_n\}$ corresponding to posting lists, so we call these non-terminals LNTs. Set R comprises of LNTs' production rules called *reduced lists*. We exclude S from V, so V contains only non-terminals that denote patterns called PNTs. Set P is the production set of PNTs, and we call it the *dictionary*. Because the maps between V and P, and between S and R are unique, in the following sections, we will denote G by P and R for simplicity. The size of an NT A is denoted $|A|$ which is the sum of the symbols in the definition of A and the size of the grammar G is denoted $|G|$ which is the total size of all unique NTs (including S) in G. The most basic unit of grammar reduction is a *digram*, which is defined as a pair of symbols next to each other appearing in an input string.

Grammar-based compression has been studied for several decades. Nevill-Manning and Witten [6] introduced a linear-time algorithm *Sequitur* that generates a context-free grammar to compress the input. Relevant to this paper is [9], where a greedy grammar generation algorithm *Sequential* is proposed which generates an irreducible grammar on any data sequence. In [9], an irreducible grammar G is defined which has the following three properties:

1) Except for the start symbol S, all other non-terminals in grammar G appear at least twice in the range of G.
2) There are no disjoint repeated substrings whose length greater than or equal to 2 in G.
3) Different non-terminals are not equal.

For the properties (we call constraints later) of an irreducible grammar, we see that these constraints indicate the effectiveness of a grammar transform algorithm in the view of compression. In this paper, we relax the three constraints to design PIPara.

In the setting of inverted index compression, *Re-Pair* [4] is an offline algorithm that recursively finds the most frequent pair of symbols and replaces

it with a new non-terminal, until no more replacements are valuable. Claude et al. [3] proposed *Repair-Skip*, which improves *Re-Pair* by compressing differential lists. They also add skip information for non-terminals to support fast skipping over compressed sequences during query processing.

Recently, Zhang et al. [11] proposed PISequential, a variant of Sequential to process posting lists of a general document collection directly, along with some optimizations. The present work takes PISequential as the baseline and use it as a calling process in PIPara.

3 Parallel Grammar Generation

The basic idea of the parallel grammar generation algorithm is to divide the original delta lists into c regions of almost the same size and assign each of them to a unique thread. Each thread will invoke PISequential [11] to generate a local irreducible grammar of the corresponding lists. After c local irreducible grammars have been generated, they are merged into a global grammar. This gives rise to an optimization problem: how to merge local irreducible grammars to obtain a small enough global grammar.

Irreducible grammar we describe above (in Section 2.2) gives the constraints of grammar reduction from the perspective of compression. However, considering the characteristic of parallel grammar, we weaken constraint 2 and get inferior properties a grammar have as follows:

Constraints 1 and 3 do not change because they are necessary for parallel grammar reduction.

We observed that if the input string only has duplicate digrams which appear only twice in the data without overlapping, then the generated irreducible grammar does not change the size of input string. Since two kinds of data are generated when the input string is transformed by PISequential, one is a grammatical form of the input string, which is the definition of S and the other is the dictionary containing all the definitions of NT except S. In practice, when the input string is long with many repetitive substrings, the size of S is much larger than the sum of the size of other NTs. Based on this we relax constraint 2 as follows:

Constraint 2: There are either no repeated substrings without overlapping or only substrings with length of 2 appearing twice in the definition of S (reduced lists).

The constraints above give a less strict form of an irreducible grammar which we call Sirreducible constraints.

3.1 Filtering and Rough Merging

After each thread generates a local irreducible grammar using PISequential, we use the main thread to merge all the local grammars generated to form a global grammar. Specifically, for each thread i ($i \in \{1, \ldots, c\}$), the initial local grammar is $G_i = (V_i, T_i, P_i, R_i, S_i)$ where V_i and P_i are both $\emptyset$, and $S_i = \{l_{i1}, l_{i2}, \cdots, l_{in_i}\}$

is a subset of LNTs and R_i contains their original posting lists. PISequential is used to identify patterns from R_i and transform it to an irreducible grammar G_i.

After this, each thread filters the irreducible grammar G_i by replacing all the PNTs occurring less than twice in R_i with their definitions and then removing these definitions. Then we relabel the remaining PNTs in V_i and update the relevant productions in D_i and R_i.

After this step, we perform a rough merging operation to form a global context-free grammar G. The local dictionaries P_i, for all $i \in \{1, \ldots, c\}$, are combined into P and likewise for V_i, R_i and S_i. We then reassign a global label to every PNT and update P and R accordingly. The following example illustrates this procedure.

Example 1. We invoke two threads T_1 and T_2 to process the inverted index $\{l_1, l_2, l_3, l_4, l_5\}$ shown below. We partition posting lists into two subsets $S_1 = \{l_1, l_2, l_3\}$ and $S_2 = \{l_4, l_5\}$ and assign them to T_1 and T_2 respectively.

$$l_1 = \langle 1, 2, 3, 14, 20, 21, 39, 40, 49, 57\rangle \qquad l_2 = \langle 1, 2, 3, 9, 14, 21, 39, 40, 49\rangle$$
$$l_3 = \langle 1, 14, 16, 21, 39\rangle \qquad l_4 = \langle 1, 2, 3, 14, 20, 37, 42, 57, 58\rangle$$
$$l_5 = \langle 1, 2, 3, 8, 15, 21, 39, 40, 49, 51\rangle$$

After performing the grammar transform action on R_1 and R_2 using PISequential in T_1 and T_2, we obtain the irreducible grammars

$$G_1 = \{\{A_1 \to \langle 1, 2, 3\rangle, B_1 \to \langle C_1, 40, 49\rangle, C_1 \to \langle 21, 39\rangle\},$$
$$\{l_1 \to \langle A_1, 14, 20, B_1, 57\rangle, l_2 \to \langle A_1, 9, 14, B_1\rangle, l_3 \to \langle 1, 14, 16, C_1\rangle\}\}$$
$$G_2 = \{\{A_2 \to \langle 1, 2, 3\rangle\},$$
$$\{l_4 \to \langle A_2, 14, 20, 37, 42, 57, 58\rangle, l_5 \to \langle A_2, 8, 15, 21, 39, 40, 49, 51\rangle\}\}$$

By filtering G_1 and G_2, we remove the PNT C_1 which appears once in R_1 and therefore the grammars become

$$G_1 = \{\{A_1 \to \langle 1, 2, 3\rangle, B_1 \to \langle 21, 39, 40, 49\rangle\},$$
$$\{l_1 \to \langle A_1, 14, 20, B_1, 57\rangle, l_2 \to \langle A_1, 9, 14, B_1\rangle, l_3 \to \langle 1, 14, 16, 21, 39\rangle\}\}$$
$$G_2 = \{\{A_2 \to \langle 1, 2, 3\rangle\},$$
$$\{l_4 \to \langle A_2, 14, 20, 37, 42, 57, 58\rangle, l_5 \to \langle A_2, 8, 15, 21, 39, 40, 49, 51\rangle\}\}$$

We obtain a global grammar G after a rough merging operation, where

$$G = \{\{A \to \langle 1, 2, 3\rangle, B \to \langle 21, 39, 40, 49\rangle, C \to \langle 1, 2, 3\rangle\},$$
$$\{l_1 \to \langle A, 14, 20, B, 57\rangle, l_2 \to \langle A, 9, 14, B\rangle, l_3 \to \langle 1, 14, 16, 21, 39\rangle,$$
$$l_4 \to \langle C, 14, 20, 37, 42, 57, 58\rangle, l_5 \to \langle C, 8, 15, 21, 39, 40, 49, 51\rangle\}\}$$

In this example, we see that after this first step, no PNT appears less than twice in local grammars G_1 and G_2. However, the rough merged global grammar G has two equivalent PNTs, namely $A = C$, so it doesn't satisfy all three Sirreducible constraints.

3.2 Dictionary Merging

We remove duplicate PNTs in the rough global grammar $G = (V, T, P, R, S)$. Since this kind of duplication occurs only in the dictionary, in this section we tackle this problem by proposing a dictionary merging algorithm. We first derive all the PNTs, and then transform the resulting terminal strings to a grammar using PISequential, and finally eliminate duplication. More specifically, we perform the following five steps:

1) We recursively derive all the PNTs according to their production rules in the dictionary.
2) We use the resulting terminal strings as PISequential input, thereby generating a grammar $G_p = \{V_p, T, P_p, \emptyset, \emptyset\}$.
3) We merge G_p into G by finding the correspondence between V_p and V.
4) We find the PNTs of size 1 and replace any occurrence of them in R by their definition.
5) We remove useless PNTs and their productions, and relabel the remaining PNTs and update R accordingly.

Example 2. For the rough global grammar G generated in Example 1, we obtain the grammar

$$G_p = \{\{A \to \langle D\rangle, B \to \langle 21, 39, 40, 49\rangle, C \to \langle D\rangle, D \to \langle 1, 2, 3\rangle\}, \{\}\}$$

We see that A and C have size 1, and therefore we replace all of their occurrences (in l_1, l_2, l_4, and l_5) by their definition D. The remaining PNTs B and D are then relabeled as A and B respectively and R is updated accordingly, we obtain

$$\begin{aligned} G = \{&\{A \to \langle 21, 39, 40, 49\rangle, B \to \langle 1, 2, 3\rangle\}, \\ &\{l_1 \to \langle B, 14, 20, A, 57\rangle, l_2 \to \langle B, 9, 14, A\rangle, l_3 \to \langle 1, 14, 16, 21, 39\rangle, \\ &\ l_4 \to \langle B, 14, 20, 37, 42, 57, 58\rangle, l_5 \to \langle B, 8, 15, 21, 39, 40, 49, 51\rangle\}\} \end{aligned}$$

Since PISequential removes duplicate digrams in the input data, after the grammar transform on the dictionary, the definitions of equal PNTs are reduced to a single newly created PNT. Since any PNT appears at least twice in R, even after step 5, we still guarantee that the remaining PNTs appear at least twice in R. In Example 2, we replace A and C in l_1, l_2, l_3, l_4, and l_5, after which B and D appear at least twice in these reduced lists. Therefore, the first two Sirreducible constrains are satisfied.

3.3 Grammatical Iteration

From Example 2, we see that reduced lists generated by different threads may have common digrams after dictionary merging. In this section, we propose a grammatical iteration method to remove all such duplications. In brief, we perform a grammar transform on the reduced lists and the dictionary, which we see as a secondary reduction on the current grammar G. The main steps are as follows:

1) Take P and R as the input and execute PISequential to generate a new version of G. Once the new version is generated, one of the following four cases holds for any PNT A, whose definition is $A \to \alpha$:
   (a) PNT A appears only once in the right-hand side of the productions in P and R.
   (b) Size of A equals 1.
   (c) Cases (a) and (b) hold simultaneously.
   (d) Both (a) and (b) do not hold.
2) For any PNT A in V, if (d) doesn't hold, we replace every occurrence of A with its definition.
3) Once all the PNTs have been processed, we remove useless PNTs and their productions. Finally, we relabel the remaining PNTs and update P and R accordingly.

Example 3. For the resulting grammar $G = (V, T, P, R, S)$ in Example 2, we take P and R as input and perform the secondary reduction. The new version is

$$\begin{aligned} G = \{\{&A \to \langle D\rangle, B \to \langle 1,2,3\rangle, C \to \langle B, 14\rangle, D \to \langle 21,39,40,49\rangle\},\\ &\{l_1 \to \langle C, 20, A, 57\rangle, l_2 \to \langle B, 9, 14, A\rangle, l_3 \to \langle 1, 14, 16, 21, 39\rangle,\\ &\ l_4 \to \langle C, 20, 37, 42, 57, 58\rangle, l_5 \to \langle B, 8, 15, D, 51\rangle\}\} \end{aligned}$$

We see that $|A| = 1$, so its occurrences are replaced by D and it is removed. The final version of G is thus

$$\begin{aligned} G = \{\{&A \to \langle 1,2,3\rangle, B \to \langle A, 14\rangle, C \to \langle 21,39,40,49\rangle\},\\ &\{l_1 \to \langle B, 20, C, 57\rangle, l_2 \to \langle A, 9, 14, C\rangle, l_3 \to \langle 1, 14, 16, 21, 39\rangle,\\ &\ l_4 \to \langle B, 20, 37, 42, 57, 58\rangle, l_5 \to \langle A, 8, 15, C, 51\rangle\}\} \end{aligned}$$

After grammatical iteration, we have three conclusions below.

1: G satisfies all three Sirreducible constraints.
   After the first step of grammatical iteration, the repeated digrams in the reduced lists have been removed so as to satisfy Sirreducible constraint 2. However, constraints 1 and 3 do not hold when (a), (b), or (c) hold. In the second step we identify PNTs that do not satisfy (d) and replace all the occurrences of these PNTs with their definitions. So after the end of grammatical iteration, G satisfies all the three Sirreducible constraints.
2: The size of the grammar G obtained by the parallel grammar generation algorithm is $\mathcal{O}\left(\text{size}/\log(\text{size}/\text{c})\right)$, where size is the size of the original data. Yang and Kieffer [9] prove that the grammar size obtained by Sequential is $\mathcal{O}(\text{size}/\log \text{size})$ when size $\to \infty$. When we process c local grammars with c threads, the total size of grammars obtained by these c threads is $\mathcal{O}(c \times ((\text{size}/\text{c})/(\log(\text{size}/c)))) = \mathcal{O}\left(\text{size}/\log(\text{size}/\text{c})\right)$. Because after grammatical iteration, G satisfies the Sirreducible constraints and Sirreducible constraints force the size of the global grammar to be smaller than the total size of c local grammars, so conclusion 2 holds.

3: The time complexities of PIPara are $\mathcal{O}(\tau)$ and $\omega\left(\tau/c^2\right)$ where c is the number of threads and τ is the time complexity of PISequential.
In practice, the execution time of PISequential is mainly consumed in digram searching and production matching. Both operations are based on the hashing operations in hash tables. The hash buckets are organized as linked lists to resolve collisions. In PISequential, assuming that a hash bucket B has length k, the average time for accessing it is $\mathcal{O}(k/2)$. In the parallel grammar generation algorithm PIPara, the number of access operations on B issued by each thread becomes $1/c$ times that of PISequential. Moreover, the number of values inserted into B becomes $1/c$ times that of PISequential on average, so the number of comparison operations on B of each thread in PIPara is $\Omega\left(\frac{k}{2c^2}\right)$ and time consumed in hashing is $\Omega\left(\tau/(2c^2)\right)$. As for dictionary merging and grammatical iteration, for the same hash bucket B, the average length of a linked list is $\sum_{1\leq i\leq c} k_i$, where k_i is the length of the linked list in the corresponding bucket in thread i and $\sum_{1\leq i\leq c} k_i$ must be less than k because the input comprises of reduced lists and/or the dictionary. Similarly, there are fewer access operations on B than in PISequential. So the total time consumed by comparison operations in dictionary merging and grammatical iteration is $\mathcal{O}(\tau)$ and conclusion 3 holds.

4 Greedy Dictionary Pruning

In the process of list intersection and WAND query processing, the dictionary is frequently accessed to recover associated lists. To reduce cache misses in the derivation of PNT, we consider how to expand dictionary properly by fully deriving every PNT while keeping the grammar as small as possible.

The main idea is a greedy strategy; we optimize the Delta grammar by greedy iteration pruning, namely each round we prune some PNTs in the current grammar to obtain a smaller grammar under the premise that all PNTs in V have been derived in the current round.

We define the *top PNT* as follows: for a Delta grammar $G = (V, T, P, R, S)$ and a non-empty subset V_s of V, we call symbol $A \in V_s$ the top PNT in set V_s if A satisfies $C \to \alpha \in P$ for all $C \in V_s$, A does not appear in α. The proposed greedy pruning algorithm proceeds as follows:

1. We recursively remove all top PNTs that appear fewer than 2 times in R and update G accordingly.
2. For any top PNT A in V, if A satisfies

$$\mathit{freq}(A) + \|A\| + \mathit{offset} \geq \mathit{freq}(A) \times |A|$$

   where $\mathit{freq}(A)$ is the frequency of which A appears in D, and $|A|$ and $\|A\|$ are the length of the right-hand side of the production of A before and after deriving A, respectively, and *offset* is an external parameter, then we remove A by replacing the occurrences of A with its definition.
3. We update grammar G by relabeling the remaining PNTs and derive all PNTs for the complete dictionary.

Since the final to-be-compressed index is a Delta grammar G with a fully expanded dictionary, if a PNT A appears at least twice in G while it appears less than twice in R, we consider A and its production to be redundant. In this case, the position of A appearing in R is replaced by the right-hand side of its production, which saves time when accessing the dictionary and also saves one element of space. The reason for choosing the top PNTs to take into account is that it does not introduce repeated PNTs. To illustrate, assuming there are two productions $A \to \langle 1, 2, 3\rangle$ and $B \to \langle A, 7, 8\rangle$ which both appear once in R, PNT A is not a top PNT according to our previous definition. If we first replace A and then replace B with $\langle A, 7, 8\rangle$, then A is reintroduced into R. Instead, if we replace top PNT B first, then A will appear twice in R and is not removed, which is beneficial to storage cost. From this example we see that the pruning order has an influence on the size of the grammar after dictionary expansion.

For step 2, the left- and right-hand sides of the inequality represent the cost of storage related to A before and after A is replaced, respectively. If there is a decrease in size after replacing A, then we remove A and replace it by its definition.

5 Data Layout and Query Implementations on Delta Grammar Index

5.1 Data Structure of Delta Grammar Index

In the dictionary, for each PNT, instead of storing its corresponding d-gap list, we store the prefix sum results. Since the dictionary is frequently accessed during query process, such a format reduces the time spent on recovering lists, and moreover, we encode the dictionary via the interpolative scheme at a small storage cost.

To support fast query operations, we make use of a skip-table data structure and a block-based encoding method. For each reduced list, which are compressed using context-free grammar on delta lists, we split it into 128-element blocks, except possibly for the last block which may contain anywhere between 1 and 127 elements. The sample data layout for a reduced list is depicted in Figure 1, and it is similar to that in [11].

Each reduced list is made up of two parts, namely a *list head* and the *block info*. For a *list head*, if the number of elements in a list exceeds 128, we store *len*, the length of the reduced list, and store an array of pairs, each pair of which corresponds to one block in the reduced list. The first value in each pair is the offset to the next block in bytes, and the other value gives the maximum docID in the block. For each block in *block info*, the first value nT, is the number of terminals in the block, and the "i-th pos" is the offset of the i-th terminal in the block. The last part of each block, *compressed data*, contains the encoded symbols of the reduced block. For the second encoding, we choose OptPForDelta and Simple16 to encode *compressed data*.

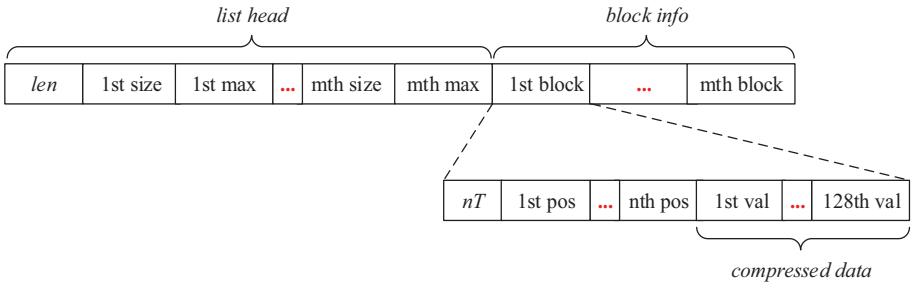

Fig. 1. Data layout for a reduced index

5.2 List Intersection and WAND Query Processing

For list intersection, motivated by the layout of reduced indexes, we design a TAAT intersection method which mainly involves searching docID in a block B_i consisting of PNTs and terminals. For any docID e, after assuring that it is smaller than the max docID in B_i, it is compared with the symbols in B_i sequentially. If the current symbol is a terminal, say t, and a prefix sum ss is computed from the first element in B_i to the element right before t, and $ss + t$ is compared with e and e is kept in the temporary result of this intersection operation if $e = ss+t$. Otherwise if the symbol is a PNT (whose maximum value is smaller than e), say A, we fetch the definition of A and compare e with the sums of values between each value in A and ss sequentially, and e is kept if it equals one of these sums.

The WAND [2] algorithm is an early stop technique using DAAT query processing. For WAND query, we only need to re-implement the routine Next_GEQ (d) in the WAND procedure, which returns the next docID greater than or equal to d in the current reduced list. We achieve this using a method similar to that described in the preceding paragraph.

6 Experiments

We design experiments to focus on the impact of parallel grammars with varying number of threads on the compression ratio and query response time in addition to the effectiveness of greedy dictionary pruning on the grammar size. The following experiments test the efficiency of the parallel algorithm by comparing the single-threaded and multi-threaded grammar transform in terms of grammar generation time, grammar size, Delta grammar compression ratio, and response time for both AND and WAND queries and the grammar size with/without greedy dictionary pruning on the dictionary. The empirical parameter *offset* (described in Section 4) is set to 10 on all three indexes.

6.1 Experimental Setup

All experiments are carried out on a PC server; its statistics are listed in table 1. Experiments are performed on three indexes of the GOV2 collection reordered by URL, IBDA [1] and TRM [7] with corresponding term frequency information, denoted GOV2URL, GOV2IBDA and GOV2TRM, respectively.

Table 1. The experimental platform details

OS Version	Red Hat 4.4.7-4
GCC version	gcc-4.8.4
CPU Clock	2.20 GHz
Logical CPU Cores	48
Cache size	30720 KB
Memory size	504 GB

6.2 Effectiveness and Efficiency of Grammar Generation and Dictionary Pruning

The first step of grammar compression, namely grammar generation, is a time consuming process which, along with the size of the generated grammar, varies according to the data size and the redundancy in the collection. In this paper, to relieve hash collisions among string matching in PISequential, the first three symbols of a string to be matched are combined into one lookup key to compute a hash value, instead of one symbol used in the original implementation and the updated version of PISequential is used as the call function in PIPara. In the `C++` implementation of PIPara, CPU threads are employed in parallel and there is not any communication between threads. To generate the Delta grammar, first we compute d-gaps of the three indexes of GOV2 data sets. Then we compare two kinds of PISequential algorithms (the single-threaded original, and the updated one) with the proposed PIPara. Figures 2(a) and 2(b) respectively plot experimental results for time consumption and space usage as the number of threads varies.

In Figure 2(a), the highest speedup ratio 75.4 of PIPara over PISequential (original) is achieved on GOV2URL with 10 threads and we see that the proposed PIPara significantly outperforms PISequential (both the original and updated versions) in terms of time consumption and achieves super-linear speedup as the number of threads increases, which we prove in Section 3.3.

For each data set, as the number of threads increases, the generated grammar size slightly increases and the trend is relatively flat, which we see in Figure 2(b). The maximum observed storage overhead is 9% for GOV2URL with 10 threads while the minimum overhead is 1.4% for GOV2IBDA with 2 threads. This indicates that the impact of PIPara on grammar size is modest when we require satisfying Sirreducible constraints in Section 3.

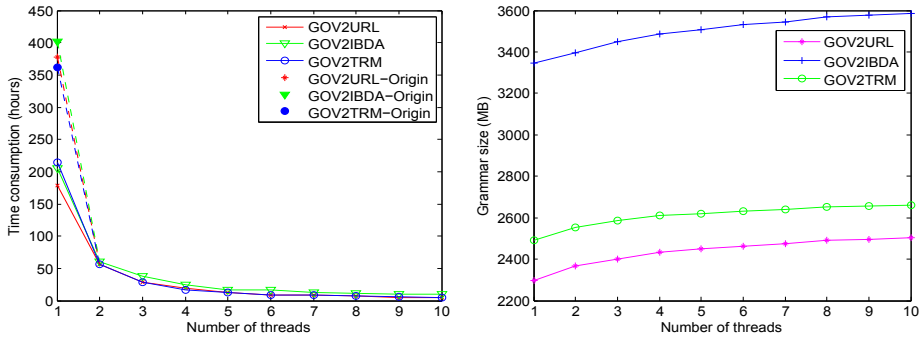

Fig. 2. Time consumption (left) and grammar size space (right) for single-threaded PISequential (original and updated versions) and PIPara as the number of threads varies from 2 to 10.In Figure 2(a), the time consumptions of the original PISequential on GOV2URL, GOV2IBDA, and GOV2TRM are denoted by GOV2URL-Origin, GOV2IBDA-Origin, and GOV2TRM-Origin, respectively. For thread 1 of GOV2URL, GOV2IBDA, and GOV2TRM , drawn using solid lines, the updated PISequential is used. For Figure 2(b), the space usage of grammar index includes the dictionary and reduced lists without any pattern pruning strategies or arithmetic encoding

Table 2 tabulates the observed grammar sizes of the Delta grammar on three datasets obtained from PISequential before and after the greedy dictionary pruning, in addition to directly deriving $PNTs$, and are refered as Origin, Greedy, and Direct. For GOV2URL, GOV2IBDA, and GOV2TRM, the greedy pruing strategy reduce the space usage of the grammars by 31%, 19%, and 17%, respectively, compared to directly deriving all $PNTs$ assuming the definition of every PNT contains only terminals, which indicates the effectiveness of the proposed greedy dictionary pruing strategy.

Table 2. Grammar size (MB) after directly deriving and applying greedy dictionary pruning.

Type	GOV2URL	GOV2IBDA	GOV2TRM
Origin	2297	3346	2492
Direct	3493	4357	3079
Greedy	2408	3529	2564

6.3 Compression and Query Processing

In Table 3 we list the final compression ratio of the compressed index. Here OptPForDelta is used for the second encoding.

Table 3. Space usage (bits/docID) of compressed indexes as the number of threads varies.

Threads	1	2	3	4	5	6	7	8	9	10
GOV2URL	4.509	4.592	4.620	4.632	4.656	4.665	4.681	4.677	4.687	**4.702**
GOV2IBDA	6.968	**7.021**	7.094	7.137	7.156	7.191	7.204	7.224	7.238	7.236
GOV2TRM	4.932	5.013	5.037	5.047	5.056	5.069	5.077	5.080	5.088	5.087

From Table 3, we make the following observations. The biggest difference between PISequential and PIPara is 4.3% (on GOV2URL with 10 threads) and the smallest difference is 0.7% (on GOV2IBDA with 2 threads). Compared with the range 9% to 1.4% of grammar storage overhead we see in Figure 2(b), we observe that second encoding further reduces the overhead. For each data set, the compression ratio difference between PISequential and PIPara varies slowly as the number of threads increases, which indicates the effectiveness of the proposed parallel grammar generation method.

To verify the query processing performance of indexes generated using PIPara and PISequential, we test two different query operations that are widely used, TAAT AND and DAAT WAND.

The experimental results are listed in Tables 4 and 5, where we compare the average response time (ms) for query processing on the indexes compressed by PISequential and PIPara.

Table 4. Average response time (ms) for TAAT AND query as the number of threads varies.

Threads	1	2	3	4	5	6	7	8	9	10
GOV2URL	3.941	3.924	3.957	3.894	3.944	3.931	3.872	**3.749**	3.981	3.927
GOV2IBDA	7.272	7.242	7.103	6.981	7.061	7.106	7.125	6.991	6.983	7.128
GOV2TRM	4.598	4.632	4.609	4.625	4.621	**4.636**	4.502	4.461	4.493	4.467

In Table 3 and 4, we find that a higher compression ratio leads to higher efficiency of list intersection. The reason for this is that a smaller grammar implies fewer blocks in reduced lists, which accelerates the docID locate speed. For each data set, the average response time varies only slightly as the number of threads varies. The maximum difference between PIPara and PISequential is from −4.90% on GOV2URL with 8 threads and the smallest is 0.83% on GOV2TRM with 6 threads, which shows that PIPara has little additional impact on TAAT AND query performance (compared with PISequential). For DAAT AND query, the results are similar and are not discussed.

Table 5. Average response time (ms) for DAAT WAND query with the number of threads varies

Threads	1	2	3	4	5	6	7	8	9	10
GOV2URL	10.431	10.279	9.938	9.782	9.901	9.873	9.919	9.782	**9.761**	9.777
GOV2IBDA	16.046	**16.273**	16.229	16.183	16.183	16.171	16.153	16.135	16.067	16.078
GOV2TRM	7.951	7.904	7.954	7.982	7.843	7.782	7.797	7.631	7.607	7.704

For WAND query, in Table 5, we arrive at the same conclusions as that for TAAT AND query, there is little difference in terms of time efficiency as the number of threads increases. On all these data sets, the difference between PIPara and PISequential varies from -6.42% to 1.41%. We see that the impact of the parallel grammar generation method on WAND query processing is minor, which also indicates the high efficiency of the Next_GEQ(d) function we rewrite.

When Simple16 is used as the second encoder of Delta grammar, similar results are obtained and not discussed.

7 Conclusions and Future Work

The proposed parallel grammar generation algorithm PIPara, along with the proposed greedy pruning strategy, reinforces and improves upon previous related work [9, 11]. Compared with single-threaded Sequential [9] and PISequential [11], we obtain 75.4 speedup in the grammar generation process with minor overhead (almost 9% and 4.3% before and after the secondary encoding). For list intersection and WAND query, PIPara increases the response time by up to 0.83% and 1.41% respectively compared to PISequential.

Since Sirreducible constrains are weaker than irreducible constraints, in the future we intend to consider other constraints which are more rigorous than Sirreducible constrains to obtain better parallel algorithms. In PIPara, each thread performs linear scanning on lists, so the lists' order will affect the grammar transform's performance. Therefore, it would be worthwhile studying how best to rearrange the original index lists.

In our experiments, we design a parallel algorithm by multi-threading. However, in order to make efficient use of memory, it is natural to extend PIPara for multiprocessing in distributed systems. In this case, PIPara could extend to the large data to generate a context-free grammar. In addition, it maybe beneficial to rewrite our algorithm for GPU to further speed up the grammar generation process.

Context-free grammar is widely used in data compression, natural language processing, biological DNA matching and pattern recognition, so studying how to use PIPara in these fields would open this research to a wide range of applications.

8 Acknowledgement

Thank Rebecca J. Stones for her guidance of writing in this paper.

References

1. D. Arroyuelo, S. González, M. Oyarzún, and V. Sepulveda. Document identifier reassignment and run-length-compressed inverted indexes for improved search performance. In *Proc. SIGIR*, pages 173–182, 2013.
2. A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Y. Zien. Efficient query evaluation using a two-level retrieval process. In *Proc. CIKM*, pages 426–434, 2003.
3. F. Claude, A. Fariña, M. A. Martínez-Prieto, and G. Navarro. Indexes for highly repetitive document collections. In *Proc. CIKM*, pages 463–468, 2011.
4. N. J. Larsson and A. Moffat. Off-line dictionary-based compression. In *Proc. DCC*, pages 296–306, 1999.
5. A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Inf. Retr.*, 3(1):25–47, 2000.
6. C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammar. *Comput. J.*, 40:103–116, 1997.
7. L. Shi and B. Wang. Yet another sorting-based solution to the reassignment of document identifiers. In *Information Retrieval Technology*, volume 7675 of *Lecture Notes in Computer Science*, pages 238–249, 2012.
8. H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. WWW*, pages 401–410, 2009.
9. E.-H. Yang and J. C. Kieffer. Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform-part one: Without context models. *IEEE Trans. Inf. Th.*, 46:755–777, 2000.
10. J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proc. WWW*, pages 387–396, 2008.
11. Z. Zhang, J. Tong, H. Huang, J. Liang, T. Li, R. J. Stones, G. Wang, and X. Liu. Leveraging context-free grammar for efficient inverted index compression. In *Proc. SIGIR*, pages 275–284, 2016.
12. M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *Proc. ICDE*, pages 59–70, 2006.