Bitlist: New Full-text Index for Low Space Cost and Efficient Keyword Search

Weixiong Rao[1,4] Lei Chen[2] Pan Hui[2,3] Sasu Tarkoma[4]

[1] School of Software Engineering
Tongji University, China
rweixiong@gmail.com

[2] Department of Comp. Sci. and Eng.
Hong Kong University of Sci.and Tech.
leichen@cse.ust.hk

[3] Telekom Innovation Laboratories
Berlin, Germany
pan.hui@telekom.de

[4] Department of Comp. Sci.
University of Helsinki, Finland
sasu.tarkoma@cs.helsinki.fi

ABSTRACT

Nowadays Web search engines are experiencing significant performance challenges caused by a huge amount of Web pages and increasingly larger number of Web users. The key issue for addressing these challenges is to design a compact structure which can index Web documents with low space and meanwhile process keyword search very fast. Unfortunately, the current solutions typically separate the space optimization from the search improvement. As a result, such solutions either save space yet with search inefficiency, or allow fast keyword search but with huge space requirement. In this paper, to address the challenges, we propose a novel structure `bitlist` with both low space requirement and supporting fast keyword search. Specifically, based on a simple and yet very efficient encoding scheme, `bitlist` uses a single number to encode a set of integer document IDs for low space, and adopts fast bitwise operations for very efficient boolean-based keyword search. Our extensive experimental results on real and synthetic data sets verify that `bitlist` outperforms the recent proposed solution, inverted list compression [23, 22] by spending 36.71% less space and 61.91% faster processing time, and achieves comparable running time as [8] but with significantly lower space.

1. INTRODUCTION

Nowadays Web search engines are experiencing significant performance challenges caused by a huge amount of Web pages and large number of Web users. For example, large search engines need to process thousands of keyword searches per second over tens of billions of Web pages [23].

The key to tackle the above challenges is to design a compact structure which can index Web documents with low space and fast keyword search. Practically, commercial Web engines use the *inverted list* index. The inverted list maintains a directory of terms, and each term in the directory refers to a *posting list* of document IDs (in short docIDs) and other items (e.g., term frequency).

Unfortunately, existing solutions typically separate space optimization from search improvement. Due to the huge amount of documents, the space cost of the associated inverted list is often large, with the size ranging from gigabytes to terabytes. The inverted list compression techniques [3, 9, 23, 24] greatly reduce the space cost, but compromise the search performance caused by decompression. On the other hand, the recent work [8] achieves fast keyword intersection search by hash bits, but introduces high space cost caused by the maintenance of reverse mapping from hash bits to docIDs.

In this paper, we propose a novel full-text indexing structure, namely `bitlist`, to achieve space optimization as well as search improvement. The key idea of `bitlist` is to design a simple and yet very efficient coding scheme, which uses an encoded number to represent a set of docIDs with low space cost. Meanwhile, `bitlist` adopts bitwise operations (such as `AND` and `OR`) over the encoded numbers for fast keyword search. As a result, `bitlist` achieves fast keyword search with low space cost. Specifically, `bitlist` first uses a binary 0/1 bit to indicate whether an indexed document contains a specific term. Then, `bitlist` encodes a base number (say B) of bits associated with consecutive docIDs into a single integer number (called *eid*). By maintaining only those non-zero *eid*s, `bitlist` achieves low space cost. Meanwhile, via the bitwise operations over the encoded *eid*s, `bitlist` can perform very fast keyword search. In this paper, other than the novel `bitlist` index itself, we further make the following contributions.

- First, the space of `bitlist` is determined by the number of non-zero *eid*s. To optimize the space cost of `bitlist`, we propose to re-assign new docIDs for the indexed documents, such that more consecutive docIDs are associated with the 0-bits and lead to more zero *eid*s (correspondingly fewer non-zero *eid*s are maintained). The key of the re-assignment is to leverage a new metric, *accumulation* similarity, instead of the traditional *pairwise* similarity. The new metric not only optimizes the space cost of `bitlist` by re-assigning new docIDs for the carefully selected documents, but also improves the bitwise operations for fast keyword search.

- Second, to enable fast boolean-based keyword search, most previous keyword search algorithms leverage the fact that the docIDs in posting lists are sorted (e.g., by ascending order), and then compare docIDs (with each other) in the visited posting lists. Instead, we adopt the bitwise operations (such as `AND` and `OR`) over the encoded 0/1 bits (i.e., the non-zero *eid*s) to allow very fast keyword search. Furthermore, we show that given multiple input terms, the keyword processing order is important to evaluate the keyword search. If some keywords are processed first, the associated posting lists are then visited earlier. Thus, we have chance to avoid visiting (part of) the posting lists of those terms with low priority, and thus decrease the processing time. Based on the

Proceedings of the VLDB Endowment, Vol. 6, No. 13

intuition, we define problems to design the optimal processing order such that we minimize the search processing time. After proving the ordering problems are NP-hard, we design the approximation algorithms.

- Finally, based on synthetic and real data sets (including input query logs and three real document sets), we conduct experiments to compare `bitlist` with the state of arts. The experimental results verify that `bitlist` outperforms the popular inverted list compression techniques [23, 22] and the recent work [8] in terms of space and searching time.

The rest of the paper is organized as follows. First we introduce the basic structure of `bitlist` in Section 2. Next, we give the details to optimize the space of `bitlist` in Section 3, and design the keyword search algorithms in Section 4. After that, we evaluate `bitlist` in Section 5, and review related works in Section 6. Finally Section 7 concludes the paper. Fig. 1 summarizes the main symbols and associated meanings in the paper. Due to the space limit, we skip the proofs of theorems and refer interested readers to our technical report [18] for a full version of the paper.

Symbol	Meaning
t_i, d_j	i-th term ($1 \le i \le T$), j-th document ($1 \le j \le D$)
θ_{ij}	a binary bit to indicate whether d_j contains t_i
$\mathcal{I}_i$, l_i; $\mathcal{L}_i$, L_i	Posting list of t_i, size of $\mathcal{I}_i$; Pair list of t_i, size of $\mathcal{L}_i$
$\langle did_{ik}, eid_{ik}\rangle$	k-th pair in $\mathcal{L}_i$ with docID num. and encoded num.
B, q, w_i	Base num., top-k query condition, weight of t_i in q
$GS(d_j)$, $GS_u(d_j)$, $AGS(d_j)$	goodness score, and its upper bound, avg. goodness.
D', $\mathcal{D}'$	# of docs reassigned with new docIDs, docs in tail cell.

Figure 1: Used Symbols and the meanings

2. OVERVIEW

Section 2.1 defines the design objective and Section 2.2 presents the data structure of `bitlist`.

2.1 Design Objective

Given a set of documents d_j with $1 \le j \le D$, we assume that each document d_j contains $|d_j|$ terms and the D documents contain totally T terms t_i with $1 \le i \le T$. Our task is to design a full-text indexing structure for such documents. The design should meet two requirements: (i) low space to index the documents, and (ii) fast running time to answer the boolean-based keyword search.

First, for a large number D of documents, the inverted list structure incurs high space. To reduce the space, various compression techniques are proposed, e.g., PForDelta coding [12, 25, 24, 23]. Such techniques typically compromise the search processing time caused by the decompression to restore original docIDs. Essentially the compression is independent of the keyword search and does not optimize search efficiency. Instead, `bitlist` integrates the compression and keyword search together for both lower space and faster keyword search.

Second, in terms of keyword search, the state-of-the-art query processing systems involve a number of phases such as query parsing, query rewriting, and ranking aggregation scores [23]. The aggregation scores frequently involve hundred of features, and need complex machine-learning techniques to compute the scores. However, at the lower layer, such query processing algorithms fundamentally rely on extremely fast access to the index, which helps finding the documents containing the input terms. Furthermore, keyword search frequently involves boolean-based intersection and union operations over the input terms. In order to achieve the required processing time, the key is to quickly find the expected documents based on the boolean operations of input terms.

As a summary, our objective of this paper is to design a compact indexing structure with low space to answer the boolean-based keyword search efficiently (i.e., the union and intersection queries).

2.2 Structure of bitlist

Before giving `bitlist`, we first introduce two alternative indexing schemes. For demonstration, these indexing schemes will index 12 documents $d_0 ... d_{11}$ shown in Table 1. For a specific document d_j, we assume that the docID of d_j is the integer j.

doc	terms	doc	terms	doc	terms
d_0	t_1, t_2, t_3	d_4	t_0, t_1	d_8	t_1, t_3
d_1	t_0, t_1, t_2, t_3	d_5	t_0	d_9	t_2, t_3
d_2	t_3	d_6	t_3	d_{10}	t_2
d_3	t_2	d_7	t_3	d_{11}	t_3

Table 1: 12 documents with the associated terms

First, the inverted list maintains a directory of terms. Each term t_i ($1 \le i \le T$) in the directory refers to a posting list $\mathcal{I}_i$. $\mathcal{I}_i$ contains docIDs of the documents containing t_i (each document is associated with a unique docID). We denote the size of $\mathcal{I}_i$ to be I_i.

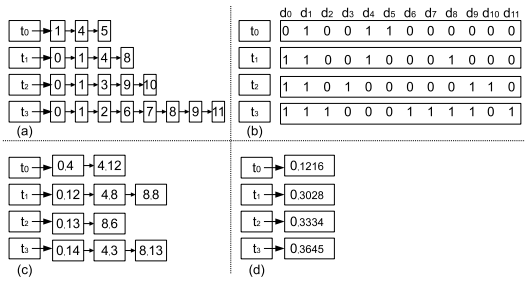

Figure 2: Three Indexing schemes: (*a*) Inverted list; (*b*) a 0/1 matrix; (*c*) `bitlist` with $B = 4$; and (*d*) `bitlist` with $B = 12$

Example 1 *Fig. 2(a) uses an inverted list to index the 12 documents. For example, the posting lists of the terms t_0 and t_3 respectively contain 3 docIDs (i.e., 1, 4 and 5) and 8 docIDs (i.e., 0, 1, 2, 6, 7, 8, 9 and 11) to represent the documents containing such terms.*

Second, we use a $T \times D$ matrix, consisting of binary bits θ_{ij} ($1 \le i \le T$ and $1 \le j \le D$), to indicate whether a document d_j contains a term t_i. If d_j contains t_i, the bit θ_{ij} is 1, and otherwise 0. We assume that the documents (i.e., the columns) in the matrix are associated with *consecutive* docIDs from 0 to $D-1$.

Example 2 *Fig. 2(b) uses a 4×12 matrix to indicate the occurrence of terms in documents. A problem of this scheme is the huge number of bits in the matrix. For example, given the terms t_0 and t_3, no matter how many documents contain the terms, the matrix always uses the exactly same 12 bits (i.e., the number D of the indexed documents). This obviously incurs high space.*

Third, we treat the proposed `bitlist` as a hybrid of the inverted list and binary matrix, illustrated by the following examples.

Example 3 *We divide the 12 columns of Fig. 2(b) by a base number $B = 4$, and have 3 cells in each of the 4 rows. For example, for the term t_0, the associated bits 010011000000 are divided into three cells 0100, 1100 and 0000. The first cell 0100 is associated with the docIDs 0, 1, 2 and 3. By treating the bits 0100 as the binary form of the integer number 4, we represent the four docIDs of the whole cell by a pair $\langle 0, 4\rangle$, where the first number 0 is the leftmost docID in the cell and 4 is just the encoded number. Similarly, the second cell 1100 and third one 0000 are represented by $\langle 4, 12\rangle$ and $\langle 8, 0\rangle$, respectively. Next, for the term t_3, the associated bits 111000111101 are divided into three cells 1110, 0011 and 1101. We use 3 pairs $\langle 0, 14\rangle$, $\langle 4, 3\rangle$ and $\langle 8, 13\rangle$ to represent the docIDs of*

the three cells, respectively. As shown in Fig. 2(c), each term refers to a list of the pairs. Note that if the encoded number in a pair is zero (e.g., the pair $\langle 8, 0\rangle$), we will not maintain the pair in order to reduce the space cost.

Given the above pairs, we can restore the associated docIDs. For example, we consider the pair $\langle 4, 12\rangle$ in row t_0. The binary number of 12 is 1100*. Then, we infer that the two documents, having the docIDs 4 and 5 respectively, contain t_0 (due to the consecutive docIDs from 0 to 11 in the columns of the matrix).*

Example 4 *To reduce the space of* `bitlist`*, we use a larger base $B = 12$ to divide the matrix of Fig. 2(b). In Fig. 2(d), each term refers to only 1 pair. Instead, the terms t_0 and t_3 in Fig. 2(c) with $B = 4$ refer to 2 and 3 pairs, respectively.*

Based on the above examples, we define the `bitlist` structure as follows. Specifically, `bitlist` maintains a directory of all T document terms t_i. Each term t_i in the directory refers to a list of the $\langle did_{ik}, eid_{ik}\rangle$ pairs. Formally, we denote the list of pairs (or equally the pair list) of t_i by $\mathcal{L}_i$ as follows.

Definition 1 *$\mathcal{L}_i$ consists of a list of $\langle did_{ik}, eid_{ik}\rangle$ pairs sorted by ascending order of did_{ik}, where did_{ik} $\%B = 0$ and $eid_{ik} > 0$. If the l-th leftmost bit ($0 \le l \le B-1$) in the binary number of eid_{ik} is 1, the document having the docID $l + did_{ik}$ must contain t_i.*

In the above definition, did_{ik} is the minimal (or leftmost) docID encoded by the $\langle did_{ik}, eid_{ik}\rangle$ pair, and did_{ik} $\%B = 0$ indicates that we use the pair to encode the bits associated with B consecutive docIDs. In addition, for lower space, `bitlist` does not maintain the pairs $\langle did_{ik}, eid_{ik}\rangle$ with $eid_{ik} = 0$. As shown above, we use a simple but yet efficient coding scheme by treating the bits of the B consecutive docIDs as the *binary* form of eid_{ik}.

Based on the definition, we can decode the pair $\langle did_{ik}, eid_{ik}\rangle$ back to the original docIDs. Given the binary form of eid_{ik}, we determine whether a bit is either 1 or 0. If the l-th leftmost bit ($0 \le l \le B-1$) is 1, the document having the docID $did_{ik} + l$ must contain t_i (since the docIDs of the matrix columns are consecutive), and otherwise it does not contain t_i.

We are interested in the size L_i of a pair list $\mathcal{L}_i$, and particularly, how L_i is comparable with the size I_i of the posting list $\mathcal{I}_i$.

Theorem 1 *For any term t_i, $\lceil I_i/B\rceil \le L_i \le I_i$ holds.*

We have has introduced the basic structure of `bitlist`. Nevertheless, the basic structure is far beyond the optimal space cost and search efficiency. In rest of the paper, we present solutions to further optimize `bitlist` in terms of the space cost and search efficiency in Section 3 and Section 4, respectively.

3. SPACE OPTIMIZATION

In this section, to optimize the space cost of `bitlist`, we first highlight the basic idea (Section 3.1), and define the optimization problem and show the complexity (Section 3.2). Next, we present the algorithm details (Section 3.3), give a practical design (Section 3.4), and finally report the maintenance of `bitlist` (Section 3.5).

3.1 Basic Idea

In this section, we give the overview of two techniques to save the space of `bitlist`.

(*i*) *DocID re-assignment*: It is easy to find that the space of `bitlist` depends on the non-zero *eid*s and thus the key is to reduce the number of non-zero *eid*s. To this end, we propose to re-assign a new docID for every document d_j, illustrated by Fig. 3. Before the re-assignment, row t_0 needs 2 pairs $\langle 0, 4\rangle$ and $\langle 4, 12\rangle$ (with $B = 4$).

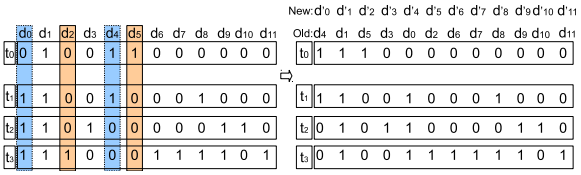

Figure 3: Re-assignment of docIDs

To reduce the number of non-zero *eid*s in this row, we exchange the docIDs between d_0 and d_4 and the docIDs between d_2 and d_5 for the re-assignment of new docIDs. After the re-assignment, the row t_0 maintains all 0-bits for the consecutive new docIDs 4...11 (and all 1-bits for the consecutive new docIDs 0,1 and 2). The pair list of t_0 maintains only one pair $\langle 0, 14\rangle$.

The re-assignment for a single row is simple. However, given totally T terms and D documents, the global re-assignment is rather challenging. Fig. 3 does optimize the space regarding to row t_0. Yet for row t_1, the re-assignment does not reduce the associated space cost (still using 3 pairs). Similar situation occurs for row t_2 and row t_3. Consequently, the re-assignment for the total T pair lists is challenging. Our purpose is to optimize the overall space of `bitlist` by minimizing the total number of non-zero *eid*s.

(*ii*) *Setting the base B*: Example 4 indicates that a lager base B might save more space. However, as shown above, the local re-assignment is difficult to optimize the global space of all pair lists. Thus, a larger B unnecessarily means smaller space. Moreover, there are limitations of the bit length (e.g., an integer *eid* typically allows 32 bits). Finally, there are options to set either fixed a B for all pair lists or a various B adaptively decided by each pair list.

In view of the above issues, we will give a practical design to set B for the tradeoff between the space and search processing time.

3.2 Problem Definition and Complexity

In this section, we assume a fixed base $B = 32$, depending on the limit of machine word length. With the assumption, we first formally define the docID re-assignment problem to minimize the overall space, and then prove that it is NP-hard. Before giving the formal problem definition, we note that the key is to explicitly measure the overall space in terms of the number of non-zero *eid*s. To this end, we define three binary parameters.

Doc Membership: Given the D documents d_j ($1 \le j \le D$) and associated T terms t_i ($1 \le i \le T$), we define a binary coefficient α_{ij}:

$$\alpha_{ij} = \begin{cases} 1 & \text{if } d_j \text{ contains } t_i; \\ 0 & \text{otherwise;} \end{cases}$$

Group Membership: Recall that based on a fixed base B, we divide the total D columns (i.e., documents) into $R = \lceil D/B\rceil$ groups (denoted by $\mathcal{R}_r$ with $1 \le r \le R$). Each group $\mathcal{R}_r$ has the $\lceil B\rceil$ member docIDs. Note that in case that the $(D\%B) \ne 0$, where % indicates the modulus operator, we have one group with the cardinality equal to $D\%B$. Given the membership of $\mathcal{R}_r$, we define the second binary parameter β_{jr} as follows.

$$\beta_{jr} = \begin{cases} 1 & \text{if the docID of } d_j \text{ is inside } \mathcal{R}_r; \\ 0 & \text{otherwise;} \end{cases}$$

Based on the above α_{ij} and β_{jr}, we define the third binary parameter γ_{ir} as follows:

$$\gamma_{ir} = \begin{cases} 1 & \text{if there exists docs } d_j \text{ satisfying both } \alpha_{ij} = 1 \text{ and } \beta_{jr} = 1; \\ 0 & \text{otherwise;} \end{cases}$$

The parameter $\gamma_{ir} = 1$ indicates the existence of documents d_j satisfying two requirements: (i) the docIDs of d_j are inside $\mathcal{R}_r$, i.e., $\beta_{jr} = 1$, and (ii) the documents d_j meanwhile contain the term t_i,

i.e., $\alpha_{ij} = 1$. Thus, $\gamma_{ir} = 1$ means that $\mathcal{R}_r$ contains at least one member docID, such that the associated document contains t_i.

Based on the parameters, we compute the overall space of `bitlist` implicitly by the total number of non-zero *eid*s. First, for a pair list of t_i, if the group $\mathcal{R}_r$ has associated with $\gamma_{ir} = 1$, the cell regarding to the group $\mathcal{R}_r$ contains at least one 1-bit. We then have a non-zero number *eid*. For the pair list of t_i having R cells, we have the number $L_i = \sum_{r=1}^{R} \gamma_{ir}$ of non-zero *eid*s. Given T pair lists, we have totally $\sum_{i=1}^{T} (\sum_{r=1}^{R} \gamma_{ir})$ non-zero *eid*s.

Now we define the following re-assignment problem.

Problem 1. Given the D documents d_j, the associated T terms t_i, and the binary coefficient α_{ij} with $1 \le j \le D$ and $1 \le i \le T$, we want to configure the parameter β_{jr} (with $1 \le r \le R$), such that the overall space $\sum_{i=1}^{T} (\sum_{r=1}^{R} \gamma_{ir})$ is minimized.

Problem 1, reducible from the SET BASIS problem [11], is NP-hard (the proof refers to the report [18]). Following the previous work [21, 15], the SET BASIS problem is equivalent to the minimal biclique cover (MBC) problem and hard to approximate. For example, results of Simon [20] and of Lund and Yannikakis [16] have shown that there is no polynomial time approximation to solve MBC (and SET BASIS) with factor n^{δ} for $\delta > 0$ unless P=NP.

3.3 Heuristics

Since Problem 1 is NP-hard, we propose a heuristic algorithm by the following basic idea. By re-assigning a new docID for a carefully selected document, we repeat the re-assignment by D iterations. In each iteration, we select a best document d_j among the remaining unprocessed documents. The selected d_j, together with those already selected documents, causes the least space. For the selected d_j, we re-assign an incrementally larger new docID (e.g., starting from the integer zero). The re-assignment is terminated until all documents are re-assigned with new docIDs.

In the rest of this section, we first define a new metric, the *accumulation* similarity of documents, to measure the goodness of the selected document d_j (Section 3.3.1), next we formulate the problem to find the best document as a classic top-k (= 1) query problem [10], and finally propose an efficient algorithm to solve the top-k problem (Section 3.3.2). Please note that the defined accumulation similarity *differs* from the traditional pairwise document similarity [19, 5, 6], because the former one defines the similarity of ($\ge$ 2) documents and the latter one involves the similarity of a pair of documents.

3.3.1 Measuring the goodness of a document d_j

The goodness of a candidate document d_j depends on the increased space incurred by d_j. For the document d_j, we have a bit $\theta_{ij} = 1$ for every $t_i \in d_j$ and otherwise $\theta_{ij} = 0$ for $t_i \notin d_j$. When d_j is selected, we append the 0/1 bits to the *tail cell* of each pair list (suppose we have re-assigned new docIDs for the number D' of documents, and the tail cell is the $\lfloor D'/B \rfloor$-th one, having the new docIDs from $B \cdot \lfloor D'/B \rfloor$ to D'). Next, when the selected d_j is re-assigned with a new docID, a 1-bit is appended to the tail cell in the pair list of $t_i \in d_j$ and a 0-bit is appended to the tail cells in the pair lists of all other terms $t_i \notin d_j$.

Example 5 *In Fig. 4, we need to select one of the 4 remaining documents d_j...d_{j+3} to re-assign a new docID. First if d_j is selected, the 1-bit is appended to the two pair lists of $t_0 \in d_j$ and $t_1 \in d_j$, respectively. Because the tail cell in each of the two pair lists consists of one 1-bit and one 0-bit, the appending of 1-bit does not incur an extra non-zero eid. Next, the appending of 0-bit to the pair lists of t_2, t_3 and t_4 does not affect their original eids.*

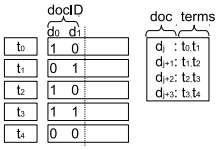

Figure 4: Goodness of selecting documents. Base number $B = 4$, two documents d_0 and d_1 have been re-assigned with consecutive docIDs, and still four documents d_j...d_{j+3} remain to re-assign new docIDs.

Second, if d_{j+1} is selected, the above situation occurs for the appending of 1-bit to the pair lists of t_1 and t_2, without incurring an extra non-zero eid.

Next, if d_{j+2} is selected, the appending of 1-bit to the pair lists of t_2 and t_3 similarly does not incur an extra non-zero eid. Note that in the pair list of t_3, the tail cell consists of two 1-bits. We consider that the appending of 1-bit to the pair list of t_3 saves more space. Generally, if a tail cell contains more 1-bits, the appending of 1-bit to the cell could save more space.

Finally, if d_{j+3} is selected,1-bit is appended to the pair lists of t_3 and t_4, and 0-bit to the other three pair lists. As before, the appending of 1-bit to the pair list of t_3 does not incur an extra eid. However, in the pair list of t_4, the appending of 1-bit incurs an extra non-zero eid. It is because the current cell in the pair list of t_4 contains two 0-bits. Thus, the selection of d_{j+3} incurs one extra non-zero eid.

Until now, the goodness of d_j depends on the two following benefits. (i) *Column benefit:* among the $|d_j|$ tail cells associated with $t_i \in d_j$, some of the appended 1-bits incur extra non-zero *eid*s (e.g., the 1-bit appended to the pair list of $t_4 \in d_{j+3}$) and others not (e.g., the 1-bits appended to the pair list of $t_0 \in d_j$ and to the pair list of $t_3 \in d_{j+3}$). Thus, we are interested in the number of extra non-zero *eid*s caused by the appended 1-bits. (ii) *Row benefit*: for the 1-bits appended to the pair list of $t_0 \in d_j$ and to the pair list of $t_3 \in d_{j+3}$, it is obvious that the latter case saves more space than the former case. Intuitively, when the tail cell contains more 1-bits, we have chance to save more space. Based on the two benefits, we define the following formula to quantify the goodness score of d_j:

$$GS(d_j) = \sum_{i=1}^{|d_j|} \vartheta_{t_i} \cdot bs(t_i) \quad (1)$$

In the above formula, ϑ_{t_i} is a binary coefficient decided by whether the tail cell of $t_i \in d_j$ contains at least one 1-bit. It is related to the column benefit. $bs(t_i)$ is the number of 1-bits inside the tail cell of t_i. It is related to the row benefit.

Based on the above equation, in Example 5, d_{j+2} is associated with the largest goodness score 3. We thus select the document d_{j+2} as the best document. Note that if the widely used pairwise similarity [19, 5, 6] is adopted, we compute the similarity of a pair of documents. Following Fig. 4, the pairwise similarity between the document d_1 and any of four remaining documents is 1, and any remaining document can be selected as the best document. By referring back to Example 5, we easily verify that the selection of d_{j+2}, instead of any other documents, offers more benefits in terms of the space cost of `bitlist`.

Essentially, the above $GS(d_j)$ accumulates the pairwise similarity and indicates more advantages than the pairwise similarity. In detail, suppose that the number D' of documents have already been re-assigned with new docIDs. Then, the number $D'\%B$ of documents (i.e., those assigned with the new docIDs ranging from

$B \cdot \lfloor D'/B \rfloor$ to D') are inside the tail cell. We denote such $D'\%B$ documents to be $\mathcal{D}'$. The $GS(d_j)$ is just equal to the sum of the pairwise similarity between d_j and each of the documents $\mathcal{D}'$. Thus, we call $GS(d_j)$ the *accumulation similarity* of d_j. By the incorporation of both row benefit and column benefit, the above $GS(d_j)$ intuitively ensures that all 1-bits are densely encoded by a small number of *eid* and leads to low space cost and fast keyword search.

Due to the above difference between the pairwise similarity and accumulation similarity, the previous re-assignment schemes [19, 5, 6], based on pairwise similarity, are inapplicable for our problem (the reason refers to the related work section). We propose an accumulation similarity-based algorithm as follows.

3.3.2 Algorithm Details

Top-k query formulation: Intuitively, the accumulation similarity $GS(d_j)$ is an aggregation score function between a document d_j and the documents $\mathcal{D}'$. We treat the selection of the best document d_j as a top-k (=1) query problem, illustrated by the following example.

Example 6 *In Fig. 4, documents $\mathcal{D}'$ (i.e., d_0 and d_1) contain four terms t_0, t_1, t_2 and t_3, respectively having the non-zero weights* 1, 1, 1 *and* 2 *(Note that a term t_i's weight w_i is equal to $bs(t_i)$ in Eq. 1, and $bs(t_i)$ is the number of documents inside $\mathcal{D}'$ containing t_i). Such terms and weights are together as the top-k query condition q. Then the top-*1 *query problem is to find a document d_j among the remaining documents, such that the goodness score between q and d_j is largest. Since the goodness score between q and d_{j+2} is 3 and all other scores are 2, we select d_{j+2} as the best document.*

We might follow the classic algorithm TA/NRA [10] to find the top-k (=1) document. To enable the TA/NRA algorithm, we assume that the remaining $(D - D')$ documents d_j are indexed as the traditional inverted list with the space of $O(\sum_{j=1}^{D-D'} |d_j|)$ where $|d_j|$ is the number of terms inside d_j. In the inverted list, each posting list maintains sorted docIDs in ascending order (note that such docIDs are the old ones before the re-assignment of new docIDs, and we denote the old docIDs to be *oid*s). Next, the TA/NRA continuously scans the posting lists associated with the query terms $t_i \in q$ to select candidates of the final result, until the TA/NRA stopping condition is met. Once the final top-1 document d_j is selected, we then append the bits of the terms $t_i \in d_j$ to the tail cell for the construction of `bitlist`.

When more documents are appended to the tail cell, $\mathcal{D}'$ contains more documents. Then the number $|q|$ of terms in q becomes larger, and the TA/NRA algorithm correspondingly scans more posting lists, leading to higher overhead.

Basic Idea: We extend the TA/NRA algorithm for less processing overhead. The basic idea is as follows. We denote d_c to be a current candidate having the largest goodness score $GS(d_c)$ among the already known candidates. If the score $GS(d_c)$ is larger than the goodness upper bound $GS_u(d_j)$ of any remaining document d_j, then d_c must be the final top-k (=1) document. Thus, the key is to measure the goodness upper bound $GS_u(d_j)$.

To this end, we sort the query terms $t_i \in q$ by descending order of weights w_i with $w_1 \geq ... \geq w_{|q|}$. After that, we process the associated posting lists by the order of $t_1 \rightarrow ... \rightarrow t_{|q|}$. That is, the posting lists $\mathcal{I}_i$ with higher weights w_i are processed first. If all the documents inside the posting list of a term t_i (associated with the weight w_i) are processed, we then mark the term t_i to be *processed*.

Now we leverage the above processing order and derive the following theorem in terms of the upper bound $GS_u(d_j)$.

Theorem 2 *Assume we begin to scan the posting list of a term t_i associated with the weight w_i, for any document d_j containing only those unprocessed terms (i.e., from t_i to $t_{|q|}$), the associated goodness score upper bound is at more $\sum_{i'=i}^{i+|d|-1} w_{i'}$.*

In the above theorem, the parameter $|d|$ denotes the number of terms inside a document, and we use the average value of the test data in our experiment. Once the above upper bound $GS_u(d_j)$ is even smaller than the goodness $GS(d_c)$ of the current top-k (=1) candidate d_c, we then return d_c as the final result.

Improvement: We improve Theorem 2 by setting a tighter upper bound $GS_u(d_j)$ as follows. We note that Theorem 2 implicitly assumes the terms from t_i to $t_{i+|d|-1}$ all appear inside the same documents. In case that no document contains all such terms, we correspondingly reduce GS_{ub} to a smaller value. Thus, we propose the following technique to determine whether there exists a document containing such terms $t_i ... t_{i+|d|-1}$.

Inside a posting list $\mathcal{I}_i$ of t_i, we denote oid_i^L and oid_i^U to be the *oid*s of the currently processed document and the final to-be-processed document, respectively. Since the member *oid*s in $\mathcal{I}_i$ are sorted in ascending order, the unprocessed docIDs in $\mathcal{I}_i$ must be inside the interval $[oid_i^L, oid_i^U]$. For a term $t_{i'}$ with $i+1 \leq i' \leq i+|d|-1$, we similarly have the interval $[oid_{i'}^L, oid_{i'}^U]$. If the overlapping result $[oid_i^L, oid_i^U] \sqcap [oid_{i'}^L, oid_{i'}^U]$ is empty, then it is obvious that no documents contain the two terms t_i and $t_{i'}$.

Now, if the terms t_i and $t_{i'}$ do not appear in the same documents, we then consider the following scenarios to set a smaller GS_{ub}:

- We assume that the $|d|$ terms $t_{i+1}...t_{i'}...t_{i+|d|}$ could appear inside the same document, and then have a new upper bound $\sum_{l=i+1}^{|d+i|} w_l$. It intuitively indicates that we slide down the original terms $t_i...t_{i'}...t_{i+|d|-1}$ to the terms $t_{i+1}...t_{i'}...t_{i+|d|}$.
- We alternatively assume that the $(i' - i)$ terms $t_i...t_{i'-1}$ and the $(|d| + i - i')$ terms $t_{i'+1}...t_{i+|d|}$ could appear inside the same document and have the upper bound $\sum_{l=i}^{|d+i|} w_l - w_i + w_{t_{i+|d|}}$.

Given the above two bounds, we use the larger one as $GS_u(d_j)$ without introducing false negative.

Algorithm 1: Select best candidate (input `bitlist` and inverted list)

```
1  build the document selection query q consisting of |q| pairs ⟨t_i, w_i⟩ with w_i > 0;
2  sort the pairs ⟨t_i, w_i⟩ with w_1 ≥ ... ≥ w_|q|;
3  denote GS_ub to be the goodness score upper bound of an unprocessed doc;
4  initiate a variable d_c to be the candidate result;
5  for i=1...|q| do
6      GS_ub = Σ_{i'=i+1}^{|d|} w_i', i.e., sum of top-|d| weights of unprocessed terms;
7      foreach unprocessed doc d_j in the inverted list of t_i do
8          if GS(d_j) > GS(d_c) then d_c = d_j;
9          foreach term t_i' from t_{i+1} to t_{i+|d|} do
10             if [oid_i^L, oid_i^U] ⊓ [oid_i'^L, oid_i'^U] == null then
11                 GS_ub = larger one btw. Σ_{i'=i}^{|d|} w_i' − w_i' + w_{i+|d|} and
                   Σ_{i'=i+1}^{|d+i|} w_i';
12         if GS(d_c) > GS_ub then break;
13 post-process the input bitlist and inverted lists, and return d_c;
```

Pseudocode: Based on the above improvement, we report the pseudocode in Alg. 1. First, we build the query condition q based on the tail cells. Since only the terms having weights $w_i > 0$ contribute to the goodness score, the input query q contain only the terms having $w_i > 0$. Next we sort the weights w_i by descending order. After that, we define two variables GS_{ub} and d_c to be the goodness score upper bound of any unprocessed document and a candidate of the final chosen document, respectively.

Next, for each document d_j inside the currently processed $\mathcal{I}_i$, we determine whether $GS(d_j) > GS(d_c)$ holds. If true, we then let $d_c = d_j$. After that, among the terms $t_{i+1}...t_{i+|d|-1}$ which contribute to the GS_{ub}, we verify whether there exist a document containing all such terms from $d_i...d_{i+|d|-1}$ by the **for** loop in lines 9-11. In the loop, we follow the aforementioned idea to determine whether the

overlapping result $[oid_j, oid_i^U] \sqcap [oid_{i'}^L, oid_{i'}^U]$ is null or not. If yes, we verify that the terms t_i and $t_{i'}$ do not appear inside the same documents, and line 11 sets a tighter bound GS_{ub} as described before.

After the above steps, we post-process the `bitlist` and inverted list as follows. For the `bitlist`, given the selected document d_c, we append the 1-bit associated with $t_i \in d_c$ to the tail cell in $\mathcal{I}_i$. For the inverted list, we remove the *oid* regarding to the selected d_c.

In addition, recall that Alg. 1 selects only one document. We actually need to select B documents to encode the associated docIDs into the $\langle did, eid \rangle$ pair. Thus, during the selection of the leftmost document encoded by the pair, we practically select top-k (> 1) best documents. The top-1 document is as the leftmost document. The other $(k-1)$ selected documents help setting goodness score lower bounds to prune unnecessary documents for the selection of the remaining $(B-1)$ documents.

3.4 Practical Design

In this section, we implement two `bitlist` versions: `bitlist-fix` and `bitlist-dyn`.

3.4.1 *Design of* `bitlist-fix`

After Alg. 1 re-assigns new docIDs for documents, we follow the idea of `bitlist` (see Section 2.2) to build the lists of bits, which are then encoded into lists of pairs $\langle did, eid \rangle$. For such pairs, the `bitlist` associates each *did* with a corresponding *eid*. It essentially adopts a fixed base number B to encode the bits in `bitlist`. We call this approach `bitlist-fix`.

We show the organization of `bitlist-fix` as follows. The organization is fairly similar to the previous inverted list and the compressed one [24]. The overall structure of `bitlist-fix` is shown in Fig. 5 (a). The index is partitioned into data blocks (e.g., with the size of 64 KB). Due to the large size of a pair list, the pair list could span multiple blocks (e.g., starting from some position of a block and ending at some position of another block). Inside each block, the metadata information of the pairs maintained inside the block (including the number of the maintained *did*s, the number of *eid*s, and positions of the pairs, etc.) are located at the beginning; after that, hundreds of chucks are maintained. Inside the chucks, besides the maintained *did*s and *eid*s, we also need to maintain other information (such as term frequency and term positions inside the documents). Here, we do not organize the pairs (e.g., 128 pairs) of *did* and *eid* belonging to a document together as a unit. Instead, the chunk maintains 128 *did*s, followed by 128 *eid*s and 128 items of other information. The purpose of our organization is to further leverage previous compression techniques to reduce the space of `bitlist`. For example, due to the sorted *did*s, we follow PForDelta and the improved version [23] to compress the 64 sorted *did*s, Rice coding to compress the 64 *eid*s (note that the *eid*s are not necessarily sorted as the *did*s do), and [22] to compress the term frequency.

3.4.2 *Design of* `bitlist-dyn`

`bitlist-dyn` optimizes the space of `bitlist-fix` by a variable base number B for each pair list of t_i. That is, given a pair list $\mathcal{L}_i$ in `bitlist`-fix, the following case could occur. For ℓ adjacent pairs $\langle did_{i,k}, eid_{i,k} \rangle$, $\langle did_{i,k+1}, eid_{i,k+1} \rangle$ ... $\langle did_{i,k+\ell-1}, eid_{i,k+\ell-1} \rangle$ in $\mathcal{L}_i$, the equation

$$did_{i,k} + \ell * B = did_{i,k+1} + (\ell - 1) * B = ... = did_{i,k+\ell-1}$$

holds. `bitlist-dyn` then optimizes `bitlist-fix` by using only one *did* and ℓ *eid*s, i.e., $\langle did_{i,k}, eid_{i,k} ... eid_{i,k+\ell-1} \rangle$, to replace the original ℓ pairs. Thus `bitlist-dyn` uses less space than `bitlist-fix` does. Intuitively, the base number in `bitlist-dyn` adaptively varies by the value $\ell \cdot B$, instead of the fixed B in `bitlist-fix`. For example, in Fig. 5 (b), we assume that the `bitlist-fix` has the fixed $B = 32$, and the three pairs $\langle 0, 8 \rangle$, $\langle 32, 17 \rangle$ and $\langle 64, 41 \rangle$ follow the above case $0 + 32 * 2 = 32 + 32 * 1 = 64$. Thus, we use the quadruple $\langle 0, 8, 17, 41 \rangle$ to replace the original three pairs.

By the above idea, we implement `bitlist-dyn` based on an input `bitlist-fix`. First, for each pair list $\mathcal{L}_i$ in `bitlist-fix`, `bitlist-dyn` creates a list of L_i bit flags to indicate the association between the *did* and *eid*s. The number of such flags is equal to the size L_i of the pair list $\mathcal{L}_i$ in `bitlist-fix`. After that, we set the such bit flags f_l ($1 \le l \le L_i$) as follows. First for $k = 1$, `bitlist-dyn` always sets the first bit flag f_1 to be 0. Next, for $l > 1$, given the number L_i of *did*s in `bitlist-fix`, if $did_{i,l} = did_{i,l-1} + B$, `bitlist-dyn` does not maintain the $did_{i,l}$ and instead set the l-th bit flag f_l be to 1. If and only if $did_{i,l} \neq did_{i,l-1} + B$, `bitlist-dyn` maintains the $did_{i,j}$ and meanwhile sets $f_l = 0$. Finally, for such B_i bit flags, we again encode it into a list of $\lceil B_i/B \rceil$ numbers. In Fig. 5 (b), `bitlist-dyn` maintains the bit flags 0110....

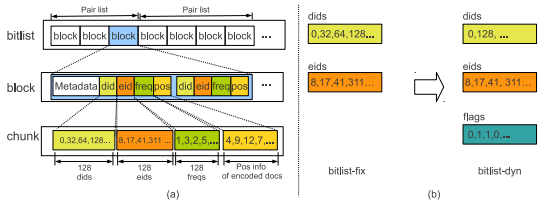

Figure 5: (a) `bitlist-fix` organization, (b) Transform from `bitlist-fix` to `bitlist-dyn`

Our experiments in Section 5 will show that `bitlist-dyn` saves around 23% space when compared with `bitlist-fix`. Nevertheless, `bitlist-dyn` meanwhile introduces more overhead for the search algorithm. For example, to restore the docID, we need extra overhead to check the bit flags. Thus, it involves the tradeoff between the reduced space and increased query processing time.

In the rest of the paper, for convenience of presentation, the term `bitlist` by default indicates the implementation of `bitlist-fix`, consisting of the lists of pairs $\langle did, eid \rangle$.

3.5 Maintenance

When a set $\mathcal{D}$ of documents are indexed by a `bitlist` structure, we need to maintain the `bitlist` structure for newly coming documents d_j. We might simply use the solution in Section 3.3 to periodically index such documents. However, before that, the un-indexed documents cannot benefit from the proposed `bitlist` scheme. Thus, we propose a solution to update the existing `bitlist` structure for newly added documents d_j.

The basic idea is as follows. Among all pairs of $\langle did, eid \rangle$ in the existing `bitlist` structure, we first find those not-well-compressed pairs (i.e., the pairs with the smallest $AGS(did)$ that will be defined soon). Next, we update such not-well-compressed pairs by the new documents d_j. In this way, we avoid processing the whole documents in $\mathcal{D}$, and meanwhile achieve low space cost.

Based on the above idea, we need to consider (i) how to efficiently find those not-well-compressed pairs? and (ii) how to update such pairs by the new documents d_j? The two subproblems are answered as follows.

First, for each *did* in a `bitlist` index, we define an *average accumulation similarity* $AGS(did)$ to measure the compression goodness, and select the smallest one as the not-well-compressed. We illustrate this using the following example.

Example 7 *In Fig. 2(c), for a did value, say 0, we first find all four pairs with the $did_i = 0$, i.e., $\langle 0,4\rangle$,$\langle 0,12\rangle$,$\langle 0,13\rangle$ and $\langle 0,14\rangle$. Next, we count the 1-bits in the four pairs, and have the average $AGS(0) = 9/4$. For the $did_i = 4$ and $did_i = 8$, we have $AGS(4) = 5/3$ and $AGS(8) = 6/3$, respectively. Given the smallest $AGS(4) = 5/3$, we consider the three pairs with the $did_i = 4$, i.e., $\langle 4,12\rangle$, $\langle 4,8\rangle$ and $\langle 4,3\rangle$, as those not-well-compressed.*

Next, among the documents encoded by the not-well-compressed pairs, we find the document having the fewest 1-bits, and swap it with the new document d_j, if the swap leads to a larger $AGS(did)$.

Example 8 *Following the above example, we assume the document d_j containing two terms t_0 and t_1 in Fig. 4 is newly added. Among the documents $d_4...d_7$ encoded by the pairs with $did_i = 4$, d_4 contributes to two 1-bits, and each of the three documents $d_5...d_7$ contributes to only one 1-bit. If any of the documents $d_5...d_7$, say d_5, is swaped with the new document d_j, we have a larger $AGS(4) = 6/3$, and update the original three pairs by the new ones $\langle 4,12\rangle$, $\langle 4,12\rangle$ and $\langle 4,3\rangle$. Finally, for the replaced document d_5, we re-assign it with a new docID 12, and encode it with a new pair $\langle 12,8\rangle$.*

In terms of running time of the maintenance, $AGS(did)$ can be pre-computed, and the not-well-compressed pairs can be found by the solution of Section 3.3 in builds a `bitlist` structure. Suppose that only the *did* values of such not-well-compressed pairs are maintained for low maintenance overhead, we then have the complexity $O(\sum_{t_i \in d_j} \log L_i + |d_j| \cdot B)$. Here, $O(\sum_{t_i \in d_j} \log L_i)$ is caused by finding the pairs having the maintained *did* over each of the sorted pair lists $\mathcal{L}_i$ (see Definition 1) involving the terms $t_i \in d_j$; and $O(|d_j| \cdot B)$ is the cost of finding the to-be-replaced document (say the above d_5) and the swap between such a document and a new document d_j.

4. KEYWORD SEARCH ALGORITHMS

In this section, we design the keyword search algorithms, including the union (Section 4.1) and intersection (Section 4.2).

4.1 Union

Given a set of input terms $t_1...t_n$, the union $docs(t_1 \sqcup ... \sqcup t_n)$ returns the docIDs of those documents containing any of the input terms t_i (with $1 \le i \le n$). To answer the union, we scan the number n of associated pair lists $\mathcal{L}_1...\mathcal{L}_n$, and conduct the union operation over the pairs in such pair lists. The process involves two subproblems: (i) for any $1 \le i \ne i' \le n$, how to conduct the union over any two pairs $\langle did_i, eid_i\rangle$ and $\langle did_{i'}, eid_{i'}\rangle$ which are inside the pair lists $\mathcal{L}_i$ and $\mathcal{L}_{i'}$, respectively (Section 4.1.1); (ii) given the n pair lists and associated pairs, which of them are loaded and processed first and which are later, i.e., the processing order of the pairs. Based on the two subproblems, we give the final algorithm detail (Section 4.1.2).

4.1.1 Union of two pairs $\langle did_i, eid_i\rangle \sqcup \langle did_{i'}, eid_{i'}\rangle$

To conduct the union of two pairs, we use the intervals of docIDs to model the encoded docIDs. Given the two pairs $\langle did_i, eid_i\rangle$ and $\langle did_{i'}, eid_{i'}\rangle$, we denote the associated intervals to be $[a_i, b_i]$ and $[a_{i'}, b_{i'}]$, respectively. For `bitlist-fix`, the interval $[a_i, b_i]$ is calculated by $[did_i, did_i + B - 1]$; for `bitlist-dyn`, the interval $[a_i, b_i]$ is calculated by $[did_i, did_i + \ell * B - 1]$ where ℓ is the number of *eid*s associated with the *did*. We consider the following cases.

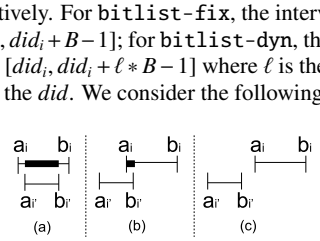

Figure 6: Three cases of $[a_i, b_i]$ and $[a_{i'}, b_{i'}]$

(i) *Covering*: In Fig. 6(a), the interval $[a_i, b_i]$ covers the interval $[a_{i'}, b_{i'}]$. Denote $\mathcal{L}_i(a_{i'}, b_{i'})$ to be the subset of docIDs in $\mathcal{L}_i$ inside the range $[a_{i'}, b_{i'}]$, highlighted by the bold segment in Fig. 6(a). If $\mathcal{L}_i(a_{i'}, b_{i'})$ is associated with all 1-bits, we directly return $\langle did_i, eid_i\rangle$ as the union result of $\langle did_i, eid_i\rangle \sqcup \langle did_{i'}, eid_{i'}\rangle$, without efforts to load the pair $\langle did_{i'}, eid_{i'}\rangle$. Otherwise, if $\mathcal{L}_i(a_{i'}, b_{i'})$ is not associated with all 1-bits, we still need to load $\langle did_{i'}, eid_{i'}\rangle$ and use the machine bitwise `OR` operation to answer $\langle did_i, eid_i\rangle \sqcup \langle did_{i'}, eid_{i'}\rangle$. The detail of the bitwise operation will be given by Alg. 2.
(ii) *Overlapping*: In Fig. 6(b), for the interval $[a_i, b_i]$, its left part, denoted by $\mathcal{L}_i(a_i, b_{i'})$ and highlighted by the bold segment in the figure, overlaps the interval of $\mathcal{L}_{i'}$. If the subinterval $\mathcal{L}_i(a_i, b_{i'})$ consists of all 1-bits, we only need to scan the remaining subinterval $\mathcal{L}_{i'}(a_{i'}, a_i)$, thus leading to less overhead. Otherwise, if the $\mathcal{L}_i(a_i, b_{i'})$ does not consist of all 1-bits, we still need to load the pair $\langle did_{i'}, eid_{i'}\rangle$, and then adopt the machine bitwise `OR` operation to answer $\langle did_i, eid_i\rangle \sqcup \langle did_{i'}, eid_{i'}\rangle$.
(iii) *No overlapping*: In Fig. 6(c), the two intervals do not overlap. Thus, we need to load the two pairs and decode their associated docIDs as the union result.

Note that for `bitlist-fix`, the intervals $[a_i, b_i]$ and $[a_{i'}, b_{i'}]$ are associated with the equal width B. Thus, we have only two cases for the intervals associated with the pairs: exact coverage (a special case of the coverage) and non-overlapping. The overlapping case is useful for the `bitlist-dyn` when the subinterval $\mathcal{L}_i(a_i, b_{i'})$ consists of pairs of all 1-bits.

4.1.2 Algorithm Detail

Given n terms $t_1...t_n$, we have n pair lists $\mathcal{L}_1...\mathcal{L}_n$ and associated pairs. Given such pairs, the order to load the pairs and conduct the union over the loaded pairs is the key to optimize the union, which is formally defined as follows.

Problem 2. To answer $docs(t_1 \sqcup ... \sqcup t_n)$, our goal is to minimize the overhead of retrieving the pair lists of the `bitlist`.

In the above problem, we implicitly measure the overhead by the number of the processed pairs of `bitlist`. Unfortunately, Problem 2, reducible from the known set cover problem, is NP-hard.

Since Problem 2 is NP-hard, we might use the document-at-a-time (DAAT) query processing style [23], widely adopted by the literature of inverted list-based keyword search. By the DAAT scheme, the posting lists associated with input keywords are all opened and read in an interleaved fashion.

Beyond the DAAT scheme, for less overhead, our solution can reduce the number of processed pairs in the opened pair lists. The basic idea of our solution is as follows. For the currently visited pairs $\langle did_{ik}, eid_{ik}\rangle$ in all opened pair lists, we sort such pairs by descending order of the associated did_{ik} values. Next, with a higher priority, we process the pairs having smaller did_{ik} values. In this way, all pairs $\langle did_{ik}, eid_{ik}\rangle$ are processed and loaded by descending order of the associated did_{ik} values. Such an order ensures that (i) each pair is processed and loaded by only one time, and (ii) the pairs involving the covering or overlapping cases (See Fig. 6) are aligned for further processing. The alignment will reduce the number of loaded and then processed pairs for less overhead. Consequently, the algorithm focuses on how to load and process the $\langle did_{ik}, eid_{ik}\rangle$ pairs by ascending order of did_{ik}, and how to reduce the processing overhead for the the covering or overlapping pairs.

Alg. 2 gives the pseudo-code of the union query. First, the minimal heap $\mathcal{H}$ (implemented by Fibonacci heap) in line 1 and the inner **while** loop (line 5) work together to ensure that the pairs are loaded by ascending order of the did_i values, and pair lists are traversed by the interleaved style. Second, the outer **while** loop (line 3) ensures that Alg. 2 reaches the end positions of all pair lists.

Algorithm 2: Answer $docs(t_1 \sqcup ... \sqcup t_n)$

```
1  initiate a minimal heap H and sets U and S, initiate two variables did and eid;
2  for all terms t_1...t_n do add ⟨t_i, did_i⟩ to the heap H;
3  while at least a pair list L_i does not reach the end position do
4      did ← −1; eid ← −1;
5      while H is not empty do
6          //pairs are loaded by ascending order of the did_i values
7          pop the item ⟨t_i, did_i⟩ from H ;
8          if did == −1 and eid == −1 then did ← did_i; eid ← eid_i;
9          else if did == did_i and eid < 2^B − 1
               //align equal dids and eid contains at least one 0 − bit then
               eid ← eid ∨ eid_i;
10         else if did < did_i then break the inner while loop;
11     if did ≥ 0 and eid > 0 then add ⟨did, eid⟩ to U;
12     foreach term t_i popped by line 7 do
13         if t_i is processed by line 8 or 9 then
14             add ⟨t_i, did_{i+1}⟩ to H //move forward pair lists, load new pairs;
15         else add the ⟨t_i, did_i⟩ to H;
16 foreach pair ⟨did, eid⟩ in U //decoding do
17     for {n = 0; n < B; n + +} do
18         if {eid ∧ (1 << n) == (1 << n)} then {add (did + n) to S};
19 return S;
```

Next, by the first condition $did == did_i$ in line 9, we ensure that the covering or overlapping pairs are aligned for the union operation $eid \leftarrow eid \vee eid_i$, if and only if the current eid is associated with at least one 0-bit (i.e., $eid < 2^B - 1$). It implicitly means that if $eid == 2^B - 1$ holds (i.e., the eid is associated with all B 1-bits), the result $eid \vee eid_i$ is always equal to eid. Thus, it is unnecessary to load the eid_i, saving the overhead of loading eid_i.

After that, in the **for** loop (lines 12-15), it moves the current positions of the pair lists (processed by lines 8 or 9) forward and then adds the did_{i+1} at the new position to the heap. For the pair lists processed by line 10, we have to push the already poped did_i values back to the heap (line 13) without falsely missing the pairs for future alignment. Finally, for the set $\mathcal{U}$ containing the pairs $\langle did, eid \rangle$, we restore them to the result docIDs.

The running time of Alg. 2 mainly depends on three parts: (i) the traversal of the pairs in all pair lists $\mathcal{L}_1 ... \mathcal{L}_n$ (operated by the outer **while** loop), (ii) the `pop`/`add` operations of the heap $\mathcal{H}$ (operated by the inner **while** loop), and (iii) the final decoding (lines 16-18). Since the $\mathcal{H}$ maintains at most the number n of dids and the decoding involves at most $N \cdot B$ operations (where N is the total number of pairs in the n pair lists), the complexity of Alg. 2 is $O(N \cdot (\log n + B))$.

4.2 Intersection

Given a set of input terms $t_1 ... t_n$, the intersection $docs(t_1 \sqcap ... \sqcap t_n)$ returns the docIDs of those documents containing all of the input terms t_i (with $1 \leq i \leq n$). Following Section 4.1, we are interested in (i) the intersection over any two loaded pairs respectively from the lists $\mathcal{L}_i$ and $\mathcal{L}_{i'}$ (with $1 \leq i \neq j \leq n$), and (ii) the optimal order of processing the pairs.

First, in terms of the intersection over two pairs $\langle did_i, eid_i \rangle$ and $\langle did_{i'}, eid_{i'} \rangle$, we note that if and only if the associated intervals $[a_i, b_i]$ and $a_{i'}, b_{i'}$ involve the overlapping or covering relations, it is necessary to conduct the intersection $\langle did_i, eid_i \rangle \sqcap \langle did_{i'}, eid_{i'} \rangle$ with the help of the machine bitwise AND operation.

Second, in terms of the optimal order to load the pairs from pair lists, we define the following problem.

Problem 3. To answer $docs(t_1 \sqcap ... \sqcap t_n)$, we minimize the overhead of retrieving the docIDs of the `bitlist`.

Unfortunately, we prove that Problem 3 is still NP-hard by showing that it is reducible from the known NP-complete problem Traveling Salesman Problem (TSP).

Since Problem 3 is NP-hard, we propose a heuristic algorithm (Alg. 3). Similar to Alg. 2, we adopt the interleaved fashion of the DAAT scheme. Nevertheless, due to the difference between intersection and union, the order to process the pairs differs from the one in Alg. 2, and the minimal heap $\mathcal{H}$ in Alg. 3 maintains the number U_i of currently unprocessed pairs in $\mathcal{L}_i$ (note that the heap in Alg. 2 maintains the did_i values). Thus, the pair list $\mathcal{L}_i$ having the smallest number U_i is processed first, such that Alg. 3 reaches the end position of the pair list $\mathcal{L}_i$ with the fewest U_i as early as possible and the outer **while** loop is then terminated.

Algorithm 3: Answer $docs(t_1 \sqcap ... \sqcap t_n)$

```
1  initiate a minimal heap H and a set U, initiate two variables did and eid;
2  for all terms t_1...t_n do add ⟨t_i, U_i⟩ to the heap H;
3  while all pair lists L_i do not reach the end position do
4      did ← −1; eid ← −1;
5      while H is not empty do
6          pop the item ⟨t_i, U_i⟩ from H;
7          if did == −1 then did ← did_i; eid ← eid_i;
8          else if advance(did, L_i) == 1 and eid > 0 then eid ← eid ∧ eid_i;
9          else eid ← −1;
10         if eid ≤ 0 then beak the inner while loop;
11     if did ≥ 0 and eid > 0 then add ⟨did, eid⟩ to U;
12     foreach term t_i popped by line 6 do
13         if did ≠ −1 and eid == −1 then advance(did, L_i);
14         add the upt ⟨t_i, U_i⟩, if any, to H;
15 same to lines 14-16 of Alg. 2;
```

Next, with help of the `advance`() function (we will introduce the function soon), the inner **while** loop aligns the dids of the current position of the n pair lists. In line 8, we conduct the insertion $eid \wedge eid_i$ if and only if the alignment is successful (i.e., `advance`(did, $\mathcal{L}_i$)==1) and $eid > 0$. Otherwise, if $eid == 0$, it is unnecessary to load the eid_i, avoiding the loading overhead. By the function `advance`(did, $\mathcal{L}_i$), the current position of $\mathcal{L}_i$ moves forward to a new position, such that the did_i value at the new position of $\mathcal{L}_i$ is at most did, i.e., $did_i \leq did$. If such a did_i is found, the function returns 1 and otherwise 0.

Similar to Alg. 2, Alg. 3 has the complexity $O(N \cdot (\log n + B))$, where N is the total number of pairs of the pair lists associated with the n input terms.

5. EVALUATION

Based on the experimental setting (Section 5.1), we conduct experiments on real and synthetic data sets to study the performance of `bitlist` respectively in Sections 5.2 and 5.3.

5.1 Experimental Setting

We use both real and synthetic data sets including indexed documents and query terms.

5.1.1 Real Data Set

Documents: We use two standard Text Retrieval Conference (TREC) data sets and one social blog data set. (i) The TRACE AP data set is composed of only 1,050 articles but with a large number of terms, on average 6,054.9 per article. (ii) We use an available subset of TREC WT10G web corpus, a large test set widely used in web retrieval research. The subset contains around 159,339 web page documents. Different from the TREC AP dataset with large articles, the average size of each document is only 5.91KB. (iii) For the social blog data set (http://socialcomputing.asu.edu/datasets/BlogCatalog), we preprocess the entries of $< snippet > ... < /snippet >$ in the set

	TREC AP			TREC WT			Blog data		
	Space	Intersection	Union	Space	Intersection	Union	Space	Intersection	Union
Inv. List	23.96 MB	0.0099 ms	0.00962 ms	39.03 MB	0.0231 ms	0.0194 ms	188.96 MB	0.511 ms	0.225 ms
IntGroup	119.78 MB	0.0553 ms		172.23 MB	0.0937 ms		370.37 MB	0.612 ms	
RanGroupScan	102.25 MB	0.00792 ms		155.49 MB	0.0195 ms		358.89 MB	0.255 ms	
Kamikaze	13.13 MB	0.0168 ms	0.0269 ms	10.66 MB	0.0512 ms	0.1957 ms	139.27MB	0.626 ms	0.325 ms
Zip	4.61 MB	160.147 ms	160.241 ms	5.56 MB	182.0512 ms	182.1957 ms	38.80 MB	2258.52 ms	2252.17 ms
`bitlist-fix`	6.25 MB	0.00816 ms	0.00836 ms	7.25 MB	0.0105 ms	0.0116 ms	106.59 MB	0.405 ms	0.169 ms
`bitlist-dyn`	4.82 MB	0.0104 ms	0.0112 ms	5.95 MB	0.0452 ms	0.0348 ms	95.95 MB	0.593 ms	0.308 ms

Table 2: Space and query processing time on three real data sets

and generate 111,772 files with on average 99.28 unique terms per file. The above three data sets were stemmed with the Porter algorithm and common stop words e.g.,"the" and "and" were removed.

Query log: We conduct keyword search based on a query log file (with 80,000 queries) of a search engine. For each query, we conduct intersection and union over the above three data sets. In the query log, the average number of terms per query is 2.085. The largest number of terms per query is 11. We note that though around 38.13% queries contain only 1 term, the remaining queries contain at least 2 terms. As a result, the optimal order of processing input terms is still useful to improve the efficiency of those queries containing multiple terms.

5.1.2 Synthetic Data Set

Documents: We generate documents to measure how `bitlist` performs well on three key parameters (similarities s of documents, number $|d_j|$ of terms in documents d_j, and number D of documents to be indexed). Specifically, to generate a document, we first follow the Zipf distribution to generate the number $|d_j|$ with a given maximal number of terms. Next, depending on the similarity s, we select the document term by probability s among 1,000 available subject words, and the remaining terms among a very huge amount $(2^{32} - 1,000)$ of words. In this way, a larger s indicates that these generated documents more similarly contain the subject words.

Queries: Depending on the query length, we randomly select the average three query terms per query among the 1,000 subject words.

5.1.3 Metrics and Counterparts

We use the following counterparts (See Section 6 for the introduction of [23, 22, 8]). (i) Kamikaze: We use the LinkedIn's open source implementation of [23, 22], namely Kamikaze version 3.0.6 [1, 2] released on Jan. 4th, 2012. Kamikaze supports the optimized PForDelta compression [23, 22] over sorted arrays of docIDs and docID set operations (intersection and union). (ii) Inverted list: We use Kamikaze's Integer arraylist to implement the uncompressed inverted list and then conduct the boolean-based keyword search (intersection and union). (iii) Zip: For comparison, we use the Zip compression package provided by JDK 1.6.0. to compress the sorted arrays of docIDs (offered by Kamikaze) and measure the space. Next, we then decompress the compressed docIDs before we conduct the docID set operations. (iv) IntGroup and RanGroupScan [8]: Following the previous work [8], we implement the the fixed-width partition algorithm (called by IntGroup) and the efficient and simple algorithm based on randomized partitions (called RanGroupScan). Since the work [8] only supports intersection searches, we ignore the evaluation of IntGroup and RanGroupScan for union searches.

Since Kamikaze is implemented by Java, for fairness, we implement IntGroup and RanGroupScan and the proposed `bitlist-fix` and `bitlist-dyn` also by JDK 1.6.0 (64 bits).

We measure the space and average query time of the above schemes, and are particularly interested in (i) how the two `bitlist` versions are comparable with Kamikaze and Zip in terms of the space, and (ii) how `bitlist-fix` and `bitlist-dyn` are comparable with IntGroup and RanGroupScan in terms of the query processing time.

5.2 Experiments on Real Data

In this section, the baseline test first compares the `bitlist-fix` and `bitlist-dyn` (setting $B = 32$) with the four counterparts. Next, the sensitivity test varies the parameters (such as B and the term processing order) to study how the performance of `bitlist` responds to the changes of such parameters. All results in this section is based on the three real document sets.

Baseline test: We report the results of the baseline test in Table 2.

First for the TREC AP data, the Zip approach uses the least space and RanGroupScan has the fastest running time. Compared with the Zip approach, `bitlist-dyn` uses the slightly more space, but achieves a significantly smaller running time than Zip. It is because the running time of Zip is dominated by the decompression time (around 159 ms). Next compared with the Kamikaze approach, `bitlist-fix` uses 47.60% of Kamikaze's space, and meanwhile achieves 51.43% and 68.91% less running time for the intersection and union queries than Kamikaze. Meanwhile `bitlist-dyn` uses 22.89% less space than `bitlist-fix`, but with 21.53% and 25.36% more running time for the intersection and union queries than `bitlist-fix`.

In terms of IntGroup and RanGroupScan, the associated high space is caused by the reverse mapping from hash bits to the original docIDs. Moreover, due to the existence of hash collusion, one hash bit could be associated with multiple docIDs, which harm the running time of IntGroup and RanGroupScan. When the number of documents ia larger, the accumulated chance of such collusion becomes higher, incurring higher running time. This is verified by the results using the large data set TREC WT and blog data sets, and `bitlist-fix` uses less running time than RanGroupScan.

Next, for the TREC WT data, the two `bitlist` implementations consistently follow the similar results as the TREC AP data. Note that though TREC WT contains much more documents than TREC AP (with more than 100 folds of the number of documents), the used space of uncompressed inverted list is only 1.62 folds. It is because the average number of terms per document in TREC WT is much smaller than the one in TREC AP and particularly the overall number of (distinct) terms in TREC WT is even smaller than the one in TREC AP. In addition, for all approaches, the query processing time of TREC WT is larger than the one of TREC AP. It is due to the average size of a posting list (i.e., the average number of docIDs per posting list) in TREC WT is much larger than the one in TREC AP.

Third, in terms of the social blog data, the associated result is roughly consistent with the results of the two TREC data. Nevertheless, the compression rates of Kamikaze and two `bitlist` versions are relatively smaller than those of the two TREC data. By carefully analyzing the original inverted lists of three data sets, we find that the average size per posting list in the social blog set, only 8.47, is much smaller than the average size of the TREC AP and WT data sets with 130.31 and 422.69 respectively, and meanwhile the number of the unique terms in the social blog set, 1,310,015,

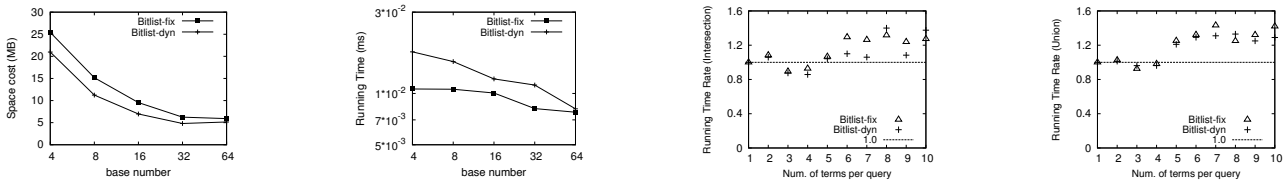

Figure 7: **From left to right: effect of base number (a) space cost and (b) intersection, and effect of the processing order of input terms (c) intersection and (d) union**

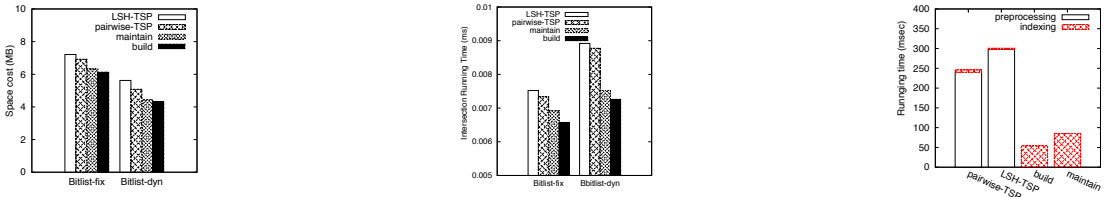

Figure 8: **From left to right: Effect of document re-assignment algorithms (a) Space cost, (b) Intersection, and (c) Indexing efficiency**

is significantly larger than the numbers in two other sets, 48,788 and 14,877 respectively. Such numbers indicate that the short articles have low similarities in the blog data set, consequently incurring lower compression rate of the Kamikaze and two `bitlist` versions. This result will be further verified by the results of our synthetic data.

Study of base num. B: Next, we vary the base number B from 4 to 64 and build `bitlist-fix` and `bitlist-dyn`. Fig. 7 (a-b) show the space and average query processing time (we use intersection as the example) associated with the varied base number. The base number $B = 64$ indicates the least space and query processing time for both `bitlist-fix` and `bitlist-dyn`. Nevertheless, we note that the used space of both `bitlist-fix` and `bitlist-dyn` has no significant decrease when B varies from 32 to 64. It is consistent with our claim in Section 3.1 that a larger B unnecessarily leads to less space. In addition, for $B = 64$, we use a long data type and the number of used bytes by a long number is still two folds of the ones used by two integers. As a result, the number of data bytes used by the *eid*s is not changed when B varies from 32 to 64.

Study of query processing alg: We measure the effect of the processing order of input terms. Besides, we are interested in how the number of terms per query affects the query processing time. Thus, based on the documents, we particularly generate 10 groups of queries (each group has 1000 queries). For the i-th group ($1 \leq i \leq 10$), each query contains the number i of terms. For all queries, besides the processing order of our approaches (i.e., Alg. 3 for intersection and Alg. 4 for union), we adopt the random order as the counterpart. Next, for each group of queries (including both intersection and union operations), we measure the query processing time by the different term processing order.

Fig. 7 (c-d) plots the rate between the query time by the random approach and the time by our approaches. The line 1.0 indicates that the time by the random approach is just equal to the one by our approach. Except two outline points (i.e., the groups with 3 and 4 terms), other groups of queries benefit from the proposed processing order. When the groups are associated with a larger number of input terms per query, the rate values become higher. This indicate that the proposed approach is helpful for those queries having more input terms. For example, for the groups of queries having 10 terms, the rate of union on the `bitlist-fix` and `bitlist-dyn` is 1.375 and 1.29, respectively. It is because given more terms, Alg. 3 has more options to find the best term processing order.

Study of building and maintaining `bitlist`: In this experiment, we study how our re-assignment algorithm is comparable with the previous works [19, 9] in terms of space cost, intersection time and running time to build and maintain the associated indexes.

[19] re-assigns new docIDs based on the pairwise document similarity and are reducible from the travelling salesman problem (TSP). [9] adopts the Local Sensitive Hashing (LSH) scheme (we use an open source implementation TarsosLSH in http://tarsos.0110.be/) to reduce the dimensionality of documents by the KNN approach (we choose K to be the base number B) to achieve a sparse graph. After that, we still follow the classic TSP approach to re-assign docIDs as [19] does. With the LSH-based dimensionality reduction, [9] then improves the running time of the TSP approach. Based on the re-assigned docIDs by [19, 9], we then follow the idea of `bitlist-fix` and `bitlist-dyn` to encode such new docIDs, and measure the associated space and average intersection query time.

In Fig. 8, we respectively label the results of [19] by "pairwise TSP" and [9] by "LSH TSP", and label the results of Section 3.3.2 to build the `bitlist-fix` by"build" and the results of Section 3.5 to maintain the `bitlist-fix` by "maintain". During the maintenance, we first build the `bitlist-fix` to index half of TREC AP files, and then update the built `bitlist-fix` index by another half of TREC AP files.

In Fig. 8 (a-b), the proposed "build" and "maintain" schemes achieve less space cost and less running time than the two pairwise similarity-based TSP schemes ("pairwise TSP" and "LSH TSP"). The results verify the advantages of `bitlist` over the TSP schemes. Second, the "LSH TSP" incurs higher space cost and larger running time than the "pairwise TSP". It is because the sparse graph achieved by dimensionality reduction could miss the terms contributing to the document similarity. Lastly, recall that the maintenance only updates an existing `bitlist` index, without optimizing the space cost of new documents. Thus, the "maintain" scheme uses higher space cost and more keyword query time than the "build" scheme which is based on the optimization of the whole data set.

Next we proceed to evaluating the indexing efficiency in terms of average running time to build and maintain the associated indexes per document. We respectively measure (i) the running time of "pairwise TSP" [5] to compute pairwise similarity, re-assign docIDs and then build the `bitlist-fix`, (ii) the running time of "LSH TSP" [9] to adopt the LSH, compute the document similarity of the spare graph, re-assign docIDs and finally build the `bitlist-fix`, (iii) the running time of building the `bitlist-fix`, and (iv) finally the running time of maintaining the `bitlist-fix`.

We plot the above running time in Fig. 8 (c). First, the running time of two TSP approaches is dominated by the preprocessing time (i.e., the time to compute pairwise similarity in "pairwise TSP" and time to adopt the LSH and compute the document similarity of the spare graph in "LSH TSP"). Consequently, the overall running time of such approaches is much higher than the time of

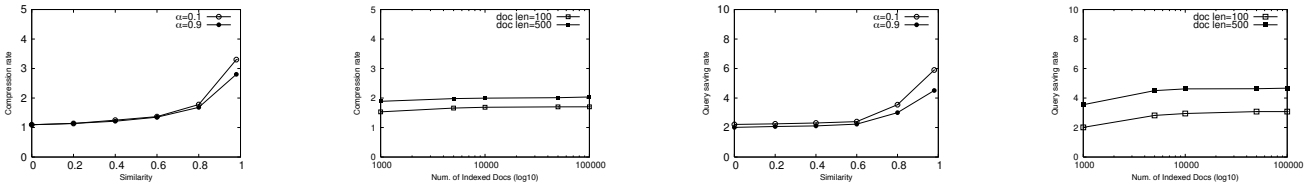

Figure 9: **From left to right: Compression rate (a) similarity of docs and (b) num. of docs; Query saving rate (c) similarity of docs and (d) num. of docs.**

the "build" scheme. The "build" scheme can leverage the existing inverted list instead of any preprocessing efforts needed by the two TSP schemes. For example, "pairwise TSP" requires high overhead to compute and maintain the pairwise similarity at the scale of $O(D^2)$, where D is the number of documents. We note that the overall time of our approach, even if the efforts to build an inverted list are considered, is still only 41.34% of the time used by "pairwise TSP". Finally, the "maintain" scheme uses slightly higher average time than the "build" scheme. Nevertheless, the benefit is that new documents can be immediately incorporated to the `bitlist` index. Since the "maintain" scheme uses higher space cost and intersection time than the "build" scheme, it makes sense to periodically use Alg. 1 in Section 3.3.2 to optimize the `bitlist` index updated by the "maintain" scheme.

5.3 Experiments on Synthetic Data

We next use the synthetic data to measure the space cost and running time of `bitlist`. Due to the space limit, we mainly report the results of `bitlist-fix`. In terms of the space cost, we define the relative compression rate to be the rate between the space size of inverted list and the size of `bitlist-fix`. A higher compression rate indicates less space cost of `bitlist-fix`. Next, we define the query saving rate to be the rate between the running time of Kamikaze on the inverted list and the time of the proposed intersection algorithm on `bitlist-fix`. A higher query saving rate indicates a faster running time of the proposed algorithm.

Space compression rate: When the similarity of generated documents grows from 0.0 to 0.98, we note that the real space sizes of both inverted list and `bitlist-fix` decrease. Specifically, when we fix the number of terms of generated documents, the high similarity means longer post lists but with a smaller number of posting lists. In addition, we use the Zipf distribution to generate the number of terms of synthetic documents (the maximal number of terms per document is 100 by default). When the Zipf parameter α becomes larger (e.g., from 0.1 to 0.9), more synthetic documents contain a small number of terms, leading to smaller size of the both data structure. Besides the real space cost, we are interested in the relative compression rate. As shown in Fig. 9(a), we find that a larger similarity (from 0.0 to 0.98) leads to a higher compression rate (from 1.09 to 2.81 for $\alpha = 0.9$ and from 1.09 to 3.31 for $\alpha = 0.1$). This result show that `bitlist-fix` achieves the higher compression benefit when the indexed documents are similar.

When we change the number of indexed documents from 10^3 to 10^5, the real space sizes of both inverted list and `bitlist-fix` obviously increase too. In addition, when we change the maximal number of terms per document (used by the Zipf distribution to generate the number of terms per document) from 100 to 500, the synthetic documents contain more terms and the real sizes of the both indexing structures also increase. Now, in terms of the relative compression rate in Fig. 9(b), a higher number of indexed documents (i.e., the x-axis) leads to a slightly larger compression rate (from 1.53 to 1.71 for the maximal number of 100 terms). In addition, this figure indicates that when the maximal number of terms per document becomes larger (e.g., 500), we have a higher compression rate. For example, given 10^5 documents, we have the compression rate 2.04 for the 500 maximal terms per document. That is because given the fixed document similarity, a larger maximal number of document terms helps select more subject terms and the synthetic documents become more similar.

Query saving rate: By using the intersection as an example, we measure the running time of `bitlist-fix` and inverted list based on the previous experiment, and plot the query saving rate in Fig. 9 (c-d). First, a larger similarity leads to the increased intersection time for both `bitlist-fix` and inverted list. It is because given higher similarity, the posting lists contain more member items of docIDs inside posting lists and thus more traversal time is needed. Nevertheless, in terms of the query saving rate of the y-axis in Fig. 9(c), the larger similarity still benefits `bitlist-fix` with a higher query saving rate. This result indicates the benefit of `bitlist-fix` to save more query time when documents are more similar. Second, when more documents are indexed, the real running time of both inverted list and `bitlist-fix` obviously becomes higher. In terms of the query time rate, as shown in Fig. 9(d), `bitlist-fix` saves more running time when the `bitlist-fix` indexes a larger number of documents.

Finally, as for `bitlist-dyn`, it achieves a larger compression rate but smaller query saving rate than `bitlist-fix`. This is consistent with the previous baseline test of real data set. Nevertheless, `bitlist-dyn` exhibits the similar curve as Fig. 9. For example, given 10^5 documents and 500 maximal terms per document, `bitlist-dyn` has the compression rate 2.41, with 21.2% growth than `bitlist-dyn`.

6. RELATED WORK

Inverted List Compression: To save the space of inverted lists, some inverted list compression techniques leverage the following intuition. Given the inverted list consisting of a sequence of docIDs (e.g., integers) in ascending order, it is also considered as a sequence of *gaps* between docIDs. If the keyword search starts from the beginning of the list, the original docIDs are recomputed by sums of the gaps. Simple9 [3], Simple16 [24] and PForDelta [12, 25] adopt this idea for compression, and [24, 23] further improves the PForDelta. Instead, Rice coding and Interpolative Coding (IPC) [17] compress standard unordered docIDs.

The above approaches essentially separate the space optimization from the search improvement. For example, the encoded number in the above approaches is only used for compression but not helpful for the search. Moreover, the search needs overhead to restore the original docIDs. Taking the compression solutions using gaps of docIDs as an example. The decompression requires the full scan of the posting lists from the beginning. It incurs high overhead, particularly for the intersection search which may result in a very few number of search results. The improvement by dividing the posting lists into pieces of gap-encoding and using two-levels of hierarchy could avoid the full scan. However, the gap-encoded numbers inside each piece still need the full scan of such a piece.

DocID Reassignment: Re-assigning docIDs is frequently used to compress inverted lists. The general intuition is as follows. Encod-

ing a small integer docID requires fewer bits than a large integer docID. Thus, consecutive docIDs mean small gaps, which then need a small number of bits. Based on the intuition, the previous works [19, 5, 9] connect the pairs of documents into the paths, such that the whole paths are associated with the largest weight sum (e.g., formulated as a traveling salesman problem TSP). [9] enhances the works [19, 5] by LSH dimensionality reduction. In addition, [14], focusing on the general topic of sparse matrix compression, shares the similar idea to the docID reordering solutions (e.g., [19, 5]). It is treated as a parallel line of the docID reordering problem.

We cannot simply re-use the above TSP schemes. The TSP schemes model documents as vertexes in a graph, and use pairwise similarity as weights of edges. During each iteration of the TSP schemes, only an edge (involving only two vertexes) is connected. Because the proposed accumulation similarity involves (≥ 2) documents (and the vertexes), an edge obviously cannot cover (≥ 2) documents, and the TSP schemes are inapplicable to our case.

Boolean-based Keyword Search: Based on sorted docIDs inside posting lists, [13, 7] use the number of comparisons (instead of running time) to measure the search cost, and minimize the cost by binary merging and binary search algorithms. In case that the posting lists significantly differ in terms of the associated size, the hash-based implementation [4] can greatly speedup the search. The recent work [8] studies the problem of fast set intersection for main memory resident data. The basic idea of [8] is to partition input sets into smaller subsets, and leverage hash functions to map the subsets into a small universe. Next, the small universe is encoded as single machine words. The bitwise-`AND` operations over the machine words then achieve very fast set intersection.

We discuss our work with the above work [8] as follows. First, [8] is not optimized for less space cost. It maintains the reverse mapping from each universe item to the associated set element, incurring high space cost. Second, in term of intersection search, given multiple input sets, [8] does not specify the processing order. Instead, we design algorithms to optimize the processing orders which then save the running time.

7. CONCLUSION

In this paper, we present a novel full-text index `bitlist`. It is a hybrid structure of the traditional inverted list and the $T \times D$ binary matrix. By using a simple yet efficient bit encoding technique and machine bitwise operations, `bitlist` achieves both low space cost and fast keyword search. Based on real and synthetic data sets, our experimental results show significant benefits of `bitlist` over the previous works in terms of low space and fast keyword search.

Our work could motivate several open questions. First, the design of a goodness score function is the key for better tradeoff between the space cost and query processing time. For example, an optimization might be tuning a weight of each term t_i based on t_i's popularity in query logs and t_i's frequency in the indexed documents. The selected best document is helpful for lower space cost and faster keyword search. Moreover, we note that most users are interested in the most relevant documents of input terms. Many previous works assume that the docIDs in inverted lists are sorted by descending order of term weights. To this end, we optionally plug-in the term weights to the goodness score function in Eq. 1. Then the documents with higher term weights have chance to be the ones having larger goodness score. Therefore, the meaningful goodness score function motivates our continuing work to enable the effectiveness and efficiency of `bitlist`.

Second, the performance result of `bitlist` depends on the similarity and number of documents to be indexed. For example, our experiments indicate that higher similarity and larger number of documents help achieving better compression result. How to further optimize `bitlist` for live and short documents (e.g., social blogs, Twitter Tweets) could be one of our future works.

Finally, we are interested in how a main memory-based `bitlist` can work together with the disk-based `bitlist` studied in this paper. For example, the design of an efficient `bitlist` caching scheme can speedup the access to the disk-based `bitlist`.

Acknowledgment: This work is supported in part by the Hong Kong RGC Project *M-HKUST602/12*, National Grand Fundamental Research 973 Program of China under Grant *2012-CB316200*, Microsoft Research Asia Grant, Huawei Noahs ark lab project *HWLB06-15C03212/13PN*, Google Faculty Award 2013, China NSFC Grant *61103006*, Shanghai Committee of Science and Technology Grant *12510706200*, the Fundamental Research Funds for the Central Universities (Tongji University), and the Academy of Finland (No. *139144*). Part of this work was done when the first author was at the Department of Computer Science, University of Helsinki, Finland.

8. REFERENCES

[1] http://data.linkedin.com/opensource/kamikaze.
[2] http://sna-projects.com/kamikaze/.
[3] V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Inf. Retr.*, 8(1):151–166, 2005.
[4] P. Bille, A. Pagh, and R. Pagh. Fast evaluation of union-intersection expressions. In *ISAAC*, pages 739–750, 2007.
[5] R. Blanco and A. Barreiro. Tsp and cluster-based solutions to the reassignment of document identifiers. *Inf. Retr.*, 9(4):499–517, 2006.
[6] D. K. Blandford and G. E. Blelloch. Index compression through document reordering. In *DCC*, pages 342–351, 2002.
[7] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *SODA*, pages 743–752, 2000.
[8] B. Ding and A. C. König. Fast set intersection in memory. *PVLDB*, 4(4):255–266, 2011.
[9] S. Ding, J. Attenberg, and T. Suel. Scalable techniques for document identifier assignment ininverted indexes. In *WWW*, pages 311–320, 2010.
[10] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.
[11] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
[12] S. Héman. Super-scalar database compression between ram and cpu-cache. In *MS Thesis, Centrum voor Wiskunde en Informatica, Amsterdam, Netherlands*, July 2005.
[13] F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly-ordered sets. *SIAM J. Comput.*, 1(1):31–39, 1972.
[14] D. S. Johnson, S. Krishnan, J. Chhugani, S. Kumar, and S. Venkatasubramanian. Compressing large boolean matrices using reordering techniques. In *VLDB*, pages 13–23, 2004.
[15] L. T. Kou and C. K. Wong. A note on the set basis problem related to the compaction of character sets. *Commun. ACM*, 18(11):656–557, 1975.
[16] C. Lund and M. Yannakakis. On the hardness of approximating minimization problems. *J. ACM*, 41(5):960–981, 1994.
[17] A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Inf. Retr.*, 3(1):25–47, 2000.
[18] W. Rao, L. Chen, P. Hui, and S. Tarkoma. Bitlist: New full-text index for low space cost and efficient keyword search. *Technical Report, Department of Computer Science, University of Helsinki, Finland, 2013 June*.
[19] W.-Y. Shieh, T.-F. Chen, J. J.-J. Shann, and C.-P. Chung. Inverted file compression through document identifier reassignment. *Inf. Process. Manage.*, 39(1):117–131, 2003.
[20] H.-U. Simon. On approximate solutions for combinatorial optimization problems. *SIAM J. Discrete Math.*, 3(2):294–310, 1990.
[21] L. J. Stockmeyer. The set basis problem is np-complete. In *Technical Report RC-5431, IBM*, 1975.
[22] H. Yan, S. Ding, and T. Suel. Compressing term positions in web indexes. In *SIGIR*, pages 147–154, 2009.
[23] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *WWW*, pages 401–410, 2009.
[24] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *WWW*, pages 387–396, 2008.
[25] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Super-scalar ram-cpu cache compression. In *ICDE*, page 59, 2006.