Ginix: Generalized Inverted Index for Keyword Search

Hao Wu*, Guoliang Li, and Lizhu Zhou

Abstract: Keyword search has become a ubiquitous method for users to access text data in the face of information explosion. Inverted lists are usually used to index underlying documents to retrieve documents according to a set of keywords efficiently. Since inverted lists are usually large, many compression techniques have been proposed to reduce the storage space and disk I/O time. However, these techniques usually perform decompression operations on the fly, which increases the CPU time. This paper presents a more efficient index structure, the Generalized INverted IndeX (Ginix), which merges consecutive IDs in inverted lists into intervals to save storage space. With this index structure, more efficient algorithms can be devised to perform basic keyword search operations, i.e., the union and the intersection operations, by taking the advantage of intervals. Specifically, these algorithms do not require conversions from interval lists back to ID lists. As a result, keyword search using Ginix can be more efficient than those using traditional inverted indices. The performance of Ginix is also improved by reordering the documents in datasets using two scalable algorithms. Experiments on the performance and scalability of Ginix on real datasets show that Ginix not only requires less storage space, but also improves the keyword search performance, compared with traditional inverted indexes.

Key words: keyword search; index compression; document reordering

1 Introduction

With the huge amount of new information, keyword search is critical for users to access text datasets. These datasets include textual documents (web pages), XML documents, and relational tables (which can also be regarded as sets of documents). Users use keyword search to retrieve documents by simply typing in keywords as queries. Current keyword search systems usually use an inverted index, a data structure that maps each word in the dataset to a list of IDs of documents in which the word appears to efficiently retrieve documents.

The inverted index for a document collection consists of a set of so-called inverted lists, known as posting lists. Each inverted list corresponds to a word, which stores all the IDs of documents where this word appears in ascending order.

In practice, real world datasets are so large that keyword search systems usually use various compression techniques to reduce the space cost of storing inverted indexes. Compression of inverted index not only reduces the space cost, but also leads to less disk I/O time during query processing. As a result, compression techniques have been extensively studied in recent years. Since IDs in inverted lists are sorted in ascending order, many existing techniques, such as *Variable-Byte Encoding* (VBE)[1] and *PForDelta*[2], store the differences between IDs, called *d-gaps*, and then use various techniques to encode these *d*-gaps using shorter binary representations. Although a compressed inverted index is smaller than the original index, the system needs to decompress encoded lists during query processing, which leads to extra

• Hao Wu, Guoliang Li, and Lizhu Zhou are with the Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: micw.mm@qq.com.

$*$ To whom correspondence should be addressed.

Manuscript received: 2012-05-03; accepted: 2012-12-21

computational costs.

To address this problem, this paper presents the *Generalized INverted IndeX* (Ginix), which is an extension of the traditional inverted index (denoted by InvIndex), to support keyword search. Ginix encodes consecutive IDs in each inverted list of InvIndex into intervals, and adopts efficient algorithms to support keyword search using these interval lists. Ginix dramatically reduces the size of the inverted index, while supporting keyword search without list decompression. Ginix is also compatible with existing d-gap-based compression techniques. As a result, the index size can be further compressed using these methods. Technique of document reordering[3-7], which is to reorder the documents in a dataset and reassign IDs to them according to the new order to make the index achieve better performance, is also used in this paper. The contributions of this paper are:

- This paper presents an index structure for keyword search, Ginix, which converts inverted lists into interval lists to save storage space.
- Efficient algorithms are given to support basic operations on interval lists, such as union and intersection without decompression.
- The problem of enhancing the performance of Ginix by document reordering is investigated, and two scalable and effective algorithms based on signature sorting and greedy heuristic of Traveling Salesman Problem (TSP)[3] are proposed.
- Extensive experiments that evaluate the performance of Ginix are conducted. Results show that Ginix not only reduces the index size but also improves the search performance on real datasets.

2 Basic Concepts of Ginix

Let $D = \{d_1, d_2, \cdots, d_N\}$ be a set of documents. Each document in D includes a set of words, and the set of all distinct words in D is denoted by W. In the inverted index of D, each word $w \in W$ has an inverted list, denoted by I_w, which is an ordered list of IDs of documents that contain the word with all lists (ID lists and interval lists) sorted in ascending order. For example, Table 1a shows a collection of titles of 7 papers and Table 1b gives its inverted index.

The inverted index of this sample dataset consists of 18 inverted lists, each of which corresponds to a word. This example shows the lists of 4 most frequent words, i.e., “keyword”, “databases”, “searching”, and “search” (word stemming is not considered).

Table 1 A sample dataset of 7 paper titles.

(a) Dataset content

ID	Content
1	Keyword querying and ranking in databases
2	Keyword searching and browsing in databases
3	Keyword search in relational databases
4	Efficient fuzzy type-ahead search
5	Navigation system for product search
6	Keyword search on spatial databases
7	Searching for hidden-web databases

(b) InvIndex

Word	IDs
Keyword	1,2,3,6
···	···
Databases	1,2,3,6,7
Searching	2,7
Search	3,4,5,6
···	···

(c) Ginix

Word	Intervals
Keyword	[1,3],[6,6]
···	···
Databases	[1,3],[6,7]
Searching	[2,2],[7,7]
Search	[3,6]
···	···

Lists in inverted indexes can be very long for large datasets, and many existing approaches have paid much attention to how to compress them. An important observation is that there are many consecutive IDs on the inverted lists. The size of the whole inverted index can be reduced by merging these groups of consecutive IDs into intervals, since each interval, denoted by r, can be represented by only two numbers (the lower and upper bounds, denoted by $\texttt{lb}(r)$ and $\texttt{ub}(r)$). For example, the ID list of databases in the sample dataset can be converted into an interval list represented by $\langle[1, 3], [6, 7]\rangle$. We call the new index structure in which all the ID lists of a standard inverted index are converted into interval lists (called *equivalent interval lists*) the generalized inverted index (Ginix). Table 1c shows the generalized inverted index for the sample dataset.

Ginix is more appropriate for those datasets whose documents are short or structured because relational tables usually have some attribute values that are shared by many records. As a result, inverted lists contain many consecutive IDs and the size of Ginix will be much smaller than a traditional inverted index. In addition, in such datasets, other information in the inverted lists such as the frequency information and position information do not significantly impact either the query processing or result ranking[8]. Thus, this paper only considers structured or short documents (DBLP and PubMed datasets) and does not consider the frequency and position information.

A straightforward way to store an interval in Ginix is to explicitly store both its lower and upper bounds, as is illustrated in Table 1c. However, if an interval $[l, u]$ is a single-element interval, i.e., $l = u$, two integers are still needed to represent the interval. Thus if there are many single-element intervals in the interval list, the space cost will be expensive. The extra overhead for storing the interval lists is reduced by splitting each original interval list into 3 ID lists with one for single-element intervals and the other two for the lower and upper bounds of multi-element intervals. These three lists are denoted as S, L, and U. For example, the interval list $\langle[1,1],[3,3],[6,7],[9,9],[12,15]\rangle$ can be split into 3 ID lists with $S = \langle 1,3,9\rangle$, $L = \langle 6,12\rangle$, and $U = \langle 7,15\rangle$. This reduces the number of integers from 10 to 7. Efficient sequential/sorted access is a basic requirement of keyword search based on the interval lists. Two position indicators, p and q, are used here to indicate the current positions in S and L/U. At the beginning, p and q are all set to 0, indicating that they are all pointing to the first elements in S and L/U. The current interval is found by comparing the two elements S_p and L_q. If S_p is smaller, we return the single-element interval $[S_p, S_p]$ and increment p by 1; if L_q is smaller, return the multi-element interval $[L_q, U_q]$ and increment q by 1.

Given an ID list S containing n IDs and its equivalent interval list R, the three lists, $R.S$, $S.L$, and $S.R$, used to store R will contain no more than n integers in total. This property of interval lists means that Ginix can be regarded as a compression technique, which is orthogonal to d-gap-based techniques. Moreover, d-gap-based compression algorithms, such as VBE and PForDelta, can still be applied to Ginix, since all the lists in Ginix are ordered lists of IDs. All of these are confirmed by our experiments presented in Section 5.

3 Search Algorithms

A keyword search system usually supports union and the intersection operations on inverted lists. The union operation is a core operation to support OR query semantics in which every document that contains at least one of the query keywords is returned as a result. The intersection operation is used to support AND query semantics, in which only those documents that contain all the query keywords are returned.

Traditional search algorithms are all based on ID lists. Specifically, a traditional keyword search system first retrieves the compressed inverted list for each keyword from the disk, then decompresses these lists into ID lists, and then calculates the intersections or unions of these lists in main memory. This method introduces extra computational costs for decompression, and ID list based search methods can be very expensive because ID lists are usually very long.

3.1 Union operation

As in set theory, the union (denoted by $\cup$) of a set of ID lists, denoted by $S = \{S_1, S_2, \cdots, S_n\}$, is another ID list, in which each ID is contained in at least one ID list in S. Thus the union of a set of interval lists can be defined as follows:

Definition 1 Union of Interval Lists Given a set of interval lists, $R = \{R_1, R_2, \cdots, R_n\}$, and their equivalent ID lists, $S = \{S_1, S_2, \cdots, S_n\}$, the union of R is the equivalent interval list of $\cup_{k=1}^{n} S_k$.

For example, consider the following three interval lists: $\langle[2,7],[11,13]\rangle$, $\langle[5,7],[12,14]\rangle$, and $\langle[1,3],[6,7],[9,9],[12,15]\rangle$. Their equivalent ID lists are $\langle 2,3,4,5,6,7,11,12,13\rangle$, $\langle 5,6,7,12,13,14\rangle$, and $\langle 1,2,3,6,7,9,12,13,14,15\rangle$. The union of these three ID lists is $\langle 1,2,3,4,5,6,7,9,11,12,13,14,15\rangle$; thus, the union of the three interval lists is the equivalent interval list of this ID list, i.e., $\langle[1,7],[9,9],[11,15]\rangle$.

In this algorithm, the interval lists are first converted into ID lists with the union calculated using the well-known multi-way merge algorithm and the result then converted back into an interval list. This method is called the NaïveUnion algorithm. Since the goal is to design an algorithm for calculating the union of interval lists without list conversion, this method will be used as a baseline for comparison.

3.1.1 Scan-line algorithm

A union algorithm without ID-interval conversion will only use the interval boundaries in the interval lists. Inspired by the scan-line rendering algorithm in computer graphics[9], the boundaries of all intervals in the interval lists are first sorted into ascending order, with a scan-line moves from the smallest boundary to the largest boundary to calculate the union list. The scan-line movement maintains a reference counter to count the number of intervals that the scan-line is currently hitting. The counter is incremented by 1 when the scan-line hits a lower bound and is decremented by 1 when it hits an upper bound. If the counter increases from 0 to 1 (which means that the scan-line is processing an interval), the current boundary is saved in variable a. When the counter decreases from 1 to 0 (which means that the scan-line will not hit any interval before it hits another lower-bound), the current boundary is saved in variable b and $[a, b]$ is returned as the resulting interval.

The heap-based merge is used on all the interval lists to enumerate all the lower bounds and upper bounds in ascending order. This algorithm is called the ScanLineUnion algorithm and illustrated in Algorithm 1.

3.1.2 Improved scan-line algorithm

The performance of the scan-line-based algorithm can be improved by maintaining an active interval to denote the current result interval. Similar to the ScanLineUnion algorithm, at the beginning, all pointers are pointing to the first intervals in the interval lists and the active interval is set to be empty. The difference is that only lower bounds are inserted into the heap. In each step, the algorithm first pops up the minimum lower bound in the heap, and then extends the active interval if the two intervals overlap. Finally, the lower bound of the next interval in the corresponding list is pushed into the heap. If the interval corresponding to the popped lower bound (denoted by r) and the active interval do not overlap, active interval is returned as a resulting interval and its lower and upper bounds are updated to $\texttt{lb}(r)$ and $\texttt{ub}(r)$. The details of this algorithm, called the ScanLineUnion+ algorithm, are illustrated in Algorithm 2.

Algorithm 1 ScanLineUnion ($\mathcal{R}$)

```
Input:  $\mathcal{R}$  A set of interval lists.
Output: $G$  The resulting interval list.
for all $k \in [1, n]$ do
    Let $r_k$ be the first interval of $R_k$
    Insert $\texttt{lb}(r_k)$ and $\texttt{ub}(r_k)$ to min-heap $\mathcal{H}$
$a \leftarrow 0, b \leftarrow 0, c \leftarrow 0$
while $\mathcal{H} \neq \emptyset$ do
    Let $t$ be the top element in $\mathcal{H}$
    Pop $t$ from $\mathcal{H}$
    if $t$ is a lower-bound then
        $c \leftarrow c + 1$
        if $c = 1$ then $a \leftarrow t$
    if $t$ is an upper-bound then
        $c \leftarrow c - 1$
        if $c = 0$ then $b \leftarrow t$ and append $[a, b]$ to $G$
        Let $r \in R_j$ be the corresponding interval of $t$
        Let $r'$ be the next interval (if any) of $r$ in $R_j$
        Insert $\texttt{lb}(r')$ and $\texttt{ub}(r')$ to $\mathcal{H}$
return $G$
```

3.2 Intersection operation

The intersection operation, denoted by $\cap$, calculates the intersection list of a set of ordered lists. As with the definitions of the union of interval lists, the intersection of interval lists can be defined as follows:

Definition 2 Intersection of Interval Lists Given a set of interval lists, $R = \{R_1, R_2, \cdots, R_n\}$, and their equivalent ID lists, $S = \{S_1, S_2, \cdots, S_n\}$, the intersection of R is the equivalent interval list of $\cap_{k=1}^{n} S_k$.

Consider the three interval lists that we have used previously: $\langle[2, 7], [11, 13]\rangle$, $\langle[5, 7], [12, 14]\rangle$, and $\langle[1, 3], [6, 7], [9, 9], [12, 15]\rangle$. Their equivalent ID lists are $\langle 2, 3, 4, 5, 6, 7, 11, 12, 13\rangle$, $\langle 5, 6, 7, 12, 13, 14\rangle$, and $\langle 1, 2, 3, 6, 7, 9, 12, 13, 14, 15\rangle$, respectively. The intersection list of these ID lists is $\langle 6, 7, 12, 13\rangle$, thus the intersection of the interval lists is the equivalent interval list of this ID list, i.e., $\langle[6, 7], [12, 13]\rangle$.

As with the union algorithms, a naïve intersection algorithm (called the NaïveIsect algorithm) and a scan-line algorithm (called the ScanLineIsect algorithm) can also be created. The NaïveIsect algorithm converts interval lists back into ID lists, calculates the intersection of these lists, and then

Algorithm 2 ScanLineUnion+ ($\mathcal{R}$)

```
Input:  $\mathcal{R}$  A set of interval lists.
Output: $G$  The resulting interval list.
for all $k \in [1, n]$ do
    Let $r_k$ be the first interval of $R_k$
    Insert $\texttt{lb}(r_k)$ to min-heap $\mathcal{H}$
$a \leftarrow 1, b \leftarrow 0$
while $\mathcal{H} \neq \emptyset$ do
    Let $l$ be the top (minimum) element in $\mathcal{H}$
    Let $r \in R_j$ be the corresponding interval of $l$
    if $b < l$ and $a \leqslant b$ then Add $[a, b]$ to $G$
    else $a \leftarrow l$
    if $b < \texttt{ub}(r)$ then $b \leftarrow \texttt{ub}(r)$
    Pop $l$ from $\mathcal{H}$
    Let $r'$ be the next interval (if any) of $r$ in $R_j$
    Insert $\texttt{lb}(r')$ to $\mathcal{H}$
if $a \leqslant b$ then Add $[a, b]$ to $G$
return $G$
```

converts back the result into an interval list. The ScanLineIsect enumerates the lower and upper bounds in ascending order and returns the intersected intervals based on a reference counter. The details of these two algorithms are omitted because both are straightforward.

3.2.1 Twin-heap algorithm

The performance of the basic scan-line algorithm can be improved by maintaining an active interval that indicates the interval currently being processed. However, a single heap is not sufficient because the lower and upper bounds must be maintained separately. The new TwinHeapIsect algorithm is illustrated in Algorithm 3.

The TwinHeapIsect algorithm manages the lower and upper bounds of the frontier intervals in two separate heaps instead of a single heap as in the basic scan-line algorithm. As a result, heap insertions are more efficient than in the basic scan-line algorithm since each heap is 50% smaller (so it takes less time to adjust the heap structures when inserting an element). Thus the TwinHeapIsect algorithm is more efficient than ScanLineIsect, as will be confirmed by the experiments in Section 5.

3.2.2 Probe-based algorithm

The probe-based intersection algorithm usually runs faster for ID lists than the merge-based intersection algorithm in real applications. A similar idea is used here to devise a probe-based algorithm to accelerate the interval list intersection process. Specifically, each interval in the shortest interval list is enumerated while the other interval lists are probed for intervals that overlap with it. The interval list probe is defined as follows:

Definition 3 Interval List Probe Given an interval r and an interval list R, probing r in R finds a list of all the intervals in R that overlap with r.

For example, the result of probing $r = [2, 6]$ in $R = \langle[1, 3], [4, 7], [9, 10]\rangle$ would be $\langle[1, 3], [4, 7]\rangle$, because both $[1, 3]$ and $[4, 7]$ in R overlap with $[2, 6]$. Since the 3 lists in the interval list are sorted in ascending order, a binary-search algorithm can be used for efficient probing. This algorithm has a time complexity of $\mathcal{O}(\log m)$ (m is the number of intervals in the list), which is much faster than the sequential scan, whose time complexity is $\mathcal{O}(m)$.

The details of the probe-based intersection algorithm (called the ProbeIsect algorithm) are illustrated in Algorithm 4. The time complexity of ProbeIsect is $C_{\mathrm{P}} = \mathcal{O}(\min\{|R_j| \cdot \sum_{k\neq j} \log |R_k|\})$, while that of TwinHeapIsect is $C_{\mathrm{H}} = \mathcal{O}(\log n \cdot \sum_k |R_k|)$.

Algorithm 3 **TwinHeapIsect** ($\mathcal{R}$)

```
Input:   $\mathcal{R}$   A set of interval lists.
Output:  $G$   The resulting interval list.
Let $\mathcal{L}$ be a max-heap and $\mathcal{U}$ be a min-heap
for all $k \in [1, n]$ do
    Let $r_k$ be the frontier interval of $R_k$
    Insert lb($r_k$) and ub($r_k$) to $\mathcal{L}$ and $\mathcal{U}$ respectively
while $\mathcal{U} \neq \emptyset$ do
    Let $l$ be the top (maximum) element in $\mathcal{L}$
    Let $u$ be the top (minimum) element in $\mathcal{U}$
    if $l \leq u$ then Add $[l, u]$ to $G$
    Let $r \in R_j$ be the corresponding interval of $u$
    Remove lb($r$) from $\mathcal{L}$ and pop $u$ from $\mathcal{U}$
    Let $r'$ be the next interval (if any) of $r$ in $R_j$
    Insert lb($r'$) and ub($r'$) to $\mathcal{L}$ and $\mathcal{U}$ respectively
return $G$
```

Algorithm 4 **ProbeIsect** ($\mathcal{R}$)

```
Input:   $\mathcal{R}$   A set of interval lists.
Output:  $G$   The resulting interval list.
Sort $\mathcal{R}$ in ascending order of list lengths
for all $r \in R_1$ do
    $R_1^* \leftarrow \langle r \rangle$
    for $k = 2, 3, \ldots, n$ do $R_k^* \leftarrow$ Probe($r$, $R_k$)
    Add TwinHeapIsect($\{R_1^*, \ldots, R_n^*\}$) to $G$
return $G$

procedure Probe($r$,$R$)
    Input:   $r$     An interval.
             $R$     An interval list.
    Output:  $R^*$   The list of all the intervals in $R$
                     that overlap with $r$.
    $p_1 \leftarrow$ BinarySearch($r.l$, $R.S$)
    $p_2 \leftarrow$ BinarySearch($r.u$, $R.S$)
    $q_1 \leftarrow$ BinarySearch($r.l$, $R.U$)
    $q_2 \leftarrow$ BinarySearch($r.u$, $R.L$)
    for $p \in [p_1, p_2]$ do Add $[R.S_p, R.S_p]$ to $R^*$
    for $q \in [q_1, q_2]$ do Add $[R.L_q, R.U_q]$ to $R^*$
    Sort $R^*$ in ascending order of lower-bounds
    return $R^*$
end procedure
```

Specifically, if $|R_1| = |R_2| = \cdots = |R_n| = m$, then $C_{\mathrm{P}} = \mathcal{O}(m \cdot n \cdot \log m)$ and $C_{\mathrm{H}} = \mathcal{O}(m \cdot n \cdot \log n)$. Thus ProbeIsect will be more costly than the heap-based algorithm if $m > n$. However, in most cases, ProbeIsect will run faster because the inverted lists of query keywords usually have very different lengths.

3.2.3 Early termination

ProbeIsect probes all the lists w.r.t. an interval and uses TwinHeapIsect to perform in-function intersections. Some probings are unnecessary because they never lead to final results. For example, consider 3 lists: $\langle[2,5],[11,13]\rangle$, $\langle[6,7],[12,14]\rangle$, and $\langle[1,3],[6,7],[9,9],[12,15]\rangle$. Probing $[2,5]$ in the third list is unpromising because this interval has no overlaps with the second interval list. A mechanism based on this observation is then used to terminate unpromising probes early. The basic idea is to probe the lists sequentially and ignore the intervals that result in empty probed intervals. The Probe function is called recursively so that unpromising probes are naturally avoided. We call this algorithm the ProbeIsect+ algorithm, and its details are illustrated in Algorithm 5.

4 Document Reordering

Document reordering also improves the performance of Ginix. This section first explains the necessity of document reordering. Then, since finding the best order of documents is NP-hard, a sorting-based method and a sorting-TSP hybrid method are used to find near-optimal solutions.

Algorithm 5 **ProbeIsect+** ($\mathcal{R}$)

```
Input:  $\mathcal{R}$  A set of interval lists.
Output: $G$  The resulting interval list.
Sort $\mathcal{R}$ in ascending order of list lengths
for all $r \in R_1$ do Add CasProbe($r, \mathcal{R} - R_1$) to $G$
return $G$

procedure CasProbe($r$,$R$)
    Input:  $r_0$  A non-empty interval.
            $\mathcal{R}$  An set of interval lists.
    Output: $G$  The resulting interval list.
    $R_1^* \leftarrow$ CasProbe($r_0, R_1$)
    for all $r \in R_1^*$ do
        $r^* \leftarrow [\max(\mathtt{lb}(r_0), \mathtt{lb}(r)), \min(\mathtt{ub}(r_0), \mathtt{ub}(r))]$
        if $n > 1$ then Add CasProbe($r^*, \mathcal{R} - R_1$) to $G$
        else Add $r^*$ to $G$
    return $G$
end procedure
```

4.1 Necessity of document reordering

The time complexities of the search algorithms given in the previous section all depend on the number of intervals in the interval lists instead of the numbers of IDs. For example, the time complexity of the ProbeIsect algorithm is $\mathcal{O}(m \cdot n \cdot \log m)$ where n denotes the number of interval lists and m denotes the number of intervals in each interval list. Thus, if the interval lists in Ginix contain fewer intervals, the search algorithms will be faster. On the other hand, interval lists containing fewer intervals will require less storage space. Therefore, the search speed and the space cost are both improved by reducing the number of intervals in Ginix.

Suppose that A and B are two ID lists with the same number of IDs. A's equivalent interval list will have less intervals than that of B's if A contains more consecutive IDs than B. Thus the order of the documents should be rearranged (or the IDs to the documents should be reassigned) so that the inverted lists contain as many consecutive IDs as possible. For example, if the 4th and 6th records in the dataset in Table 1a are switched, the interval lists for "keyword" and "databases" will become $\langle[1,4]\rangle$ and $\langle[1,4],[7,7]\rangle$. This will save two integers storage space for the interval lists.

There have been many efforts on finding the optimal document ordering that maximizes the frequencies of d-gaps in inverted lists to enhance the performance of existing inverted list compression techniques[3-7]. The current problem is a special case of this problem (i.e., to maximize the frequencies of 1-gap). Previous studies of document reordering have all been designed for unstructured long documents (e.g., news and web pages), so methods are needed for structured or short documents, which are the focus of this study.

4.2 SigSort: Signature sorting method

The problem of document reordering is equivalent to making similar documents stay near to each other. Silvestri[5] proposed a simple method that sorts web pages in lexicographical order based on their URLs as an acceptable solution to the problem. This method is reasonable because the URLs are usually good indicates of the web page content. However, this method is not applicable to datasets whose URLs do not represent meaningful content (e.g., Wikipedia pages), or even do not have a URL field.

Other fields can also be used to represent the documents. For example, the reordering can use the Conf field (i.e., conference name) in the DBLP dataset. Sorting the documents by this field can also give acceptable results as well. However, a more flexible method is to generate a summary for each document and then sort the documents according to these summaries. Summaries can be generated as follows. First, all the words are sorted in descending order of their frequencies. Then, the top n (e.g., $n = 1000$) most frequent words are chosen as signature vocabulary. For each document, a string, called a *signature*, is generated by choosing those words belong to the signature vocabulary and sorting them in descending order of their frequencies. The document sorting compares each pair of signatures word-wise instead of comparing them letter-wise. This sort-based algorithm is called the SIGSORT algorithm.

Sorting documents by their signatures is effective because more frequent words are more likely to have consecutive IDs in its inverted list. In addition, since SIGSORT is very simple, it can easily handle large datasets.

SIGSORT is more effective for structured and short text data. Such data has more representative words since more records share the same words than general text data such as long web pages. As a result, each word in the signature vocabulary has a higher clustering power and the signatures are more effective. For general text data, a more effective method should consider more sophisticated summaries based on features other than words, such as categories and statistical topics.

4.3 Scale TSP-based method using SIGSORT

Shieh et al.[3] transformed the problem of finding the optimal ordering to the Traveling Salesman Problem (TSP). They built an undirected graph based on the underlying dataset by considering each document as a vertex and the number of words shared by the two documents as the weight of each edge. Finding an optimal ordering of documents is equivalent to solving the traveling salesman problem on this graph (i.e., to find a cycle on this graph that maximizes the sum of the weights of involved edges).

Finding an optimal cycle for TSP is NP-hard. Shieh et al.[3] used the Greedy-Nearest-Neighbor (GNN) heuristic, which expands the path by adding a vertex that is closest to the current path, to find near-optimal solutions. The TSP-based method can provide good results for document reordering, but it can not scale to large datasets since solving the TSP using GNN heuristic on a complete graph with n vertexes has a time complexity of $\mathcal{O}(n^2)$.

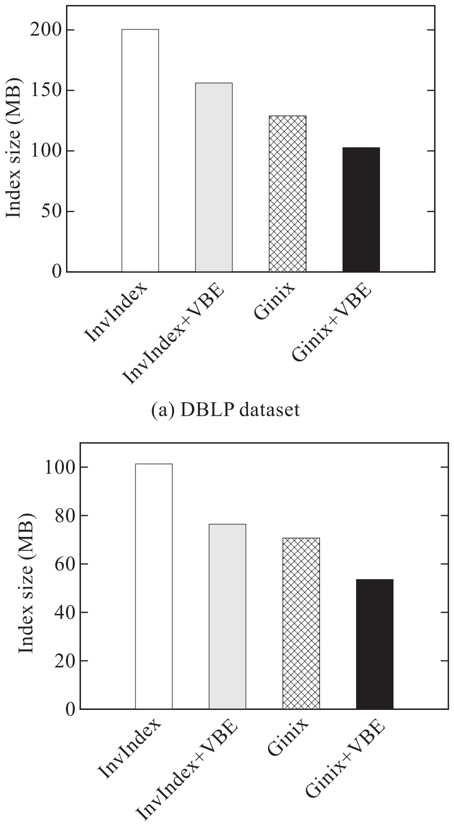
SIGSORT can be used to scale the TSP-based method to larger datasets, such as DBLP and PubMed datasets. First, all the documents are sorted according to their signatures using SIGSORT. Then, when the current path is expanded, the nearest vertex (document) is found within only a small set of candidates. Instead of the entire datasets, the candidate set for each document is the k consequent documents in the order obtained by SIGSORT. This method is called the SIGSORTTSP algorithm, which is more efficient than traditional TSP methods and which can be slightly better than pure SIGSORT for finding near-optimal solutions for the document reordering problem.

5 Experiments

The performance and scalability of Ginix was evaluated by experiments on a Linux server with an Intel Xeon 2.50 GHz CPU and 16 GB RAM. Two datasets were used in the experiments, DBLP[10] and PubMed[11]. The DBLP dataset is a bibliography database on computer science that contains more than 1.4 million publications. The Title, Authors, Year, Conf (i.e., conference name), and URL of each publication were concatenated as a document with indexes built for these documents. PubMed is an extension of the MEDLINE database that contains citations, abstracts, and some full text articles on life sciences and biomedical topics. This study used 1.4 million articles with the Title, JournalIssue, and JournalTitle attributes as the dataset. Ginix was implemented in C++ using the `gcc` compiler and `/O3` flag.

5.1 Index size

Figure 1 shows the index sizes using different compression techniques. The widely-adopted VBE is used to evaluate the present technique of converting consecutive IDs to intervals in Ginix. Figure 1 compares the original inverted index (denoted by InvIndex), the inverted index compressed by VBE (denoted by InvIndex+VBE), the present inverted index (denoted by Ginix), and the present inverted index compressed by VBE (denoted by Ginix+VBE) for both the DBLP and PubMed datasets. The results show that the Ginix



(a) DBLP dataset

(b) PubMed dataset

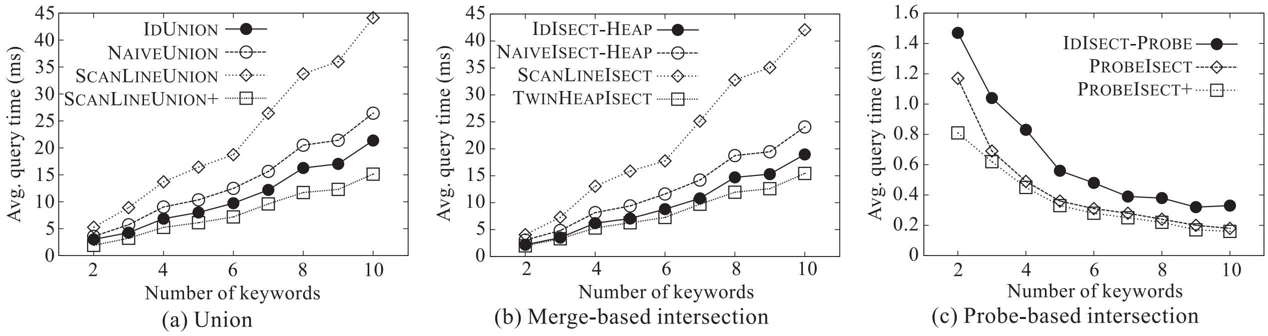
Fig. 1 Comparison of sizes of indexes using different compression techniques.

compression is much better than that of VBE. The Ginix+VBE result has the smallest index size.

5.2 Search performance

The performance of keyword search algorithms was compared using synthetic queries. Each dataset had 9 query workloads, each containing 1000 k-word queries, where $k = 2, 3, \cdots, 10$. The keywords in each query were drawn according to their frequencies, in other words, if a keyword appears more frequently in the dataset, it is more likely to be drawn as a query keyword. The memory-based algorithms have their indexes in main memory without VBE compression. Figures 2 and 3 compare the union and intersection algorithms applied on the DBLP and PubMed datasets. In these figures, IdUnion, IdIsect-Heap, and IdIsect-Probe denote the three algorithms for union and intersection operations on InvIndex. The results show that:

- ScanLineUnion+ and TwinHeapIsect, the two merge-based algorithms, are 30% and 20% faster than IdUnion and IdHeapIsect.
- ProbeIsect runs 2 times faster than IdProbeIsect, so interval list intersection is more efficient than ID list intersection.
- ProbeIsect+, the improved probe-based interval list intersection algorithm, runs faster than

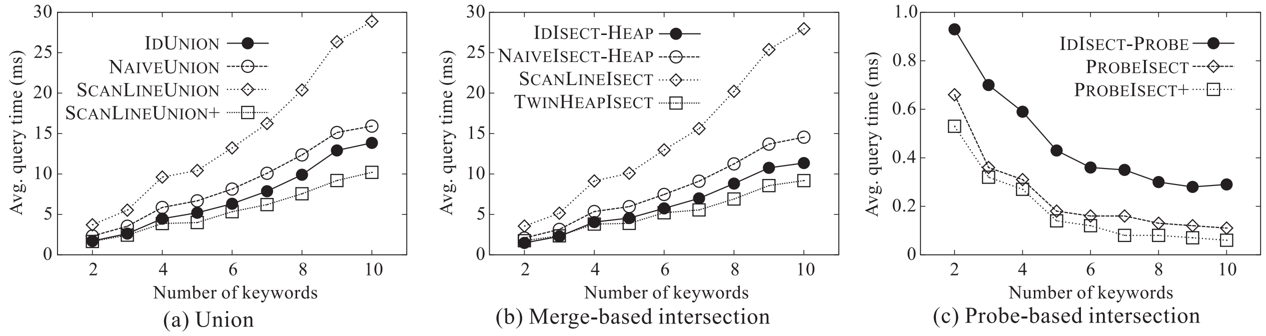


(a) Union

(b) Merge-based intersection

(c) Probe-based intersection

Fig. 2 Performance of keyword search algorithms in DBLP dataset.



(a) Union

(b) Merge-based intersection

(c) Probe-based intersection

Fig. 3 Performance of keyword search algorithms in PubMed dataset.

ProbeIsect since many unnecessary probe operations are avoided. However, when there are many keywords in the query, the computation savings are not significant since the result list is already short.

Note that the naïve probe-based intersection algorithm is very inefficient compared with the other probe-based intersection methods. As a result, it was omitted in these two figures for clarity.

Disk-based search algorithms introduce additional time to load the lists of inverted index (or Ginix) into main memory during query processing. In addition, if VBE is used on indexes, additional de-compression operations must be performed, thus the overall query time gets longer compared with memory-based algorithms. Figure 4 shows the query processing times for probe-based intersections of InvIndex, InvIndex+VBE, Ginix, and Ginix+VBE on the DBLP and PubMed datasets. These four indexes are denoted as “I”, “IV”, “G”, and “GV” in the figure. The results show that:

- **IO time**: The IO time of Ginix is approximately 30% shorter than that of InvIndex because the interval lists are shorter than the ID lists.
- **Decompression time**: Since the computational cost of VBE is proportional to the list length, the decompression time of Ginix+VBE is also approximately 30% shorter than that of InvIndex+VBE.
- **Search time**: Since the current algorithms take advantage of the intervals, the search time of Ginix is nearly 2x faster than that of InvIndex.

In summary, the overall performance of Ginix is much higher than that of InvIndex, with or without VBE compression.

5.3 Impact of document reordering

The impact of document reordering was evaluated for the DBLP dataset. The experiments considered four reordering methods: (1) Rand, which randomly shuffles the dataset; (2) Conf, which sorts the records according to the values of the Conf attribute; (3) SigSort, which uses the top 1000 most frequently occurring words as signature words; and (4) SigSortTSP, which uses 100 consequent records in the sorted list obtained by SigSort as the candidate set for each record ($k = 100$) and uses GNN heuristics to solve the TSP. The original InvIndex is used as a baseline. The method in Shieh et al.[3] was not evaluated because it can not scale to handle large datasets like DBLP. The index sizes and average query times are illustrated in Table 2.

The results in Table 2 show that:

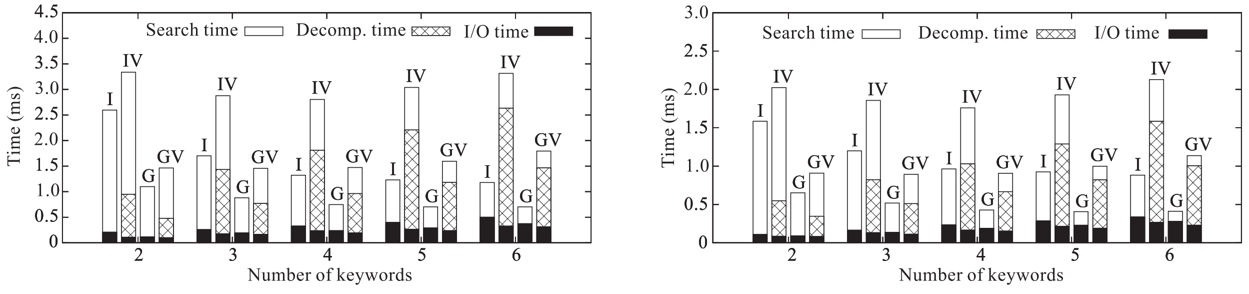
- The size of Ginix is smaller than that of InvIndex, even when the records are ordered randomly.
- Sorting records according to their Conf values provides a good ordering, with which the index size is 128.9 MB and the average query time is 0.88 ms.
- Ginix can achieve the best performance in terms of both the index size and the average query time when reordering the records using SigSortTSP. Similar results were found for the PubMed dataset.

5.4 Scalability

The scalability of Ginix was evaluated using different numbers of reocrds in the DBLP dataset. The index sizes for InvIndex and Ginix without VBE and the search speeds of the ScanLineUnion+, TwinHeapIsect, ProbeIsect, and ProbeIsect+

Table 2 Impact of document reordering (DBLP).

	Size (MB)	Time (ms)
Baseline	200.30	1.47
Rand	172.90	1.80
Conf	128.90	0.88
SigSort	130.30	0.68
SigSortTSP	**124.80**	**0.62**



(a) DBLP dataset

(b) PubMed dataset

Fig. 4 Overall query processing time of performing probe-based intersections.

algorithms (tested by 1000 2-word queries) are illustrated in Fig. 5.

Both the Ginix index size and the average query time grow almost linearly with the data size, which indicates that Ginix has good scalability.

6 Related Work

Keyword search is widely used by users to access text data with many studies in recent years. Keyword search is not only convenient for document collections but also for accessing structured or semi-structured data, such as relational databases and XML documents[12-19].

Inverted indexes are widely used to efficiently answer keyword queries in most modern keyword search systems, with techniques designed to compress the inverted indexes[20]. Most techniques first convert each ID in an inverted list to the difference between it and the preceding ID, called the d-gaps, and then encode the list using integer compression algorithms[1,20-24]. Variable-Byte Encoding is widely used in systems since it is simple and provides fast decoding[1].

Other studies have focused on how to improve the compression ratio of inverted index using *document reordering*[4,6,7]. Here, if the document IDs are reassigned so that similar documents are close to each other, then there are more small d-gaps in the converted lists and the overall compression ratio is improved.

The *interval tree*[25] is widely used to directly calculate the unions and intersections of sets of intervals. However, interval trees are not good for keyword search because: (1) an interval tree is needed for each word, which increases the index size; (2) interval trees can not be easily compressed; and (3) interval trees can not support multi-way merging and probing, which are important for accelerating calculations.

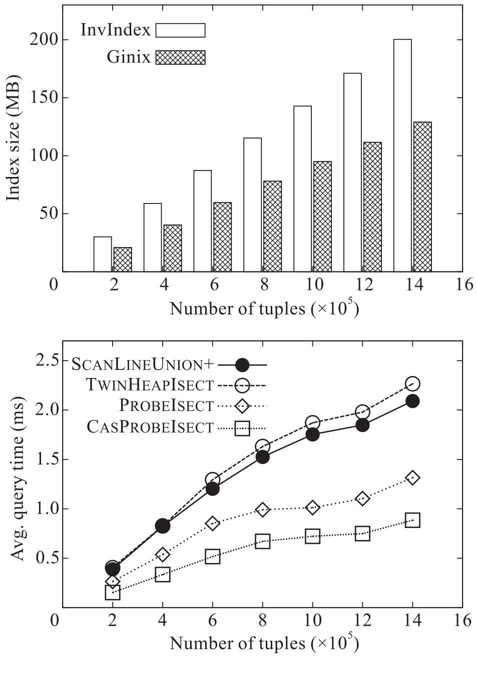


Fig. 5 Scalability for DBLP.

7 Conclusions

This paper describes a generalized inverted index for keyword search in text databases. Ginix has an effective index structure and efficient algorithms to support keyword search. Fast scalable methods enhance the search speed of Ginix by reordering documents in the datasets. Experiments show that Ginix not only requires smaller storage size than the traditional inverted index, but also has a higher keyword search speed. Moreover, Ginix is compatible with existing d-gap-based list compression techniques and can improve their performance.

Acknowledgements

This work was supported by the National Natural Science Foundation of China (No. 60833003).

References

[1] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel, Compression of inverted indexes for fast query evaluation, in *Proc. of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Tammpere, Finland, 2002, pp. 222-229.

[2] M. Zukowski, S. Hman, N. Nes, and P. A. Boncz, Super-scalar RAM-CPU cache compression, in *Proc. of the 22nd International Conference on Data Engineering*, Atlanta, Georgia, USA, 2006, pp. 59.

[3] W. Shieh, T. Chen, J. J. Shann, and C. Chung, Inverted file compression through document identifier reassignment, *Information Processing and Management*, vol. 39, no. 1, pp. 117-131, 2003.

[4] R. Blanco and A. Barreiro, TSP and cluster-based solutions to the reassignment of document identifiers, *Information Retrieval*, vol. 9, no. 4, pp. 499-517, 2006.

[5] F. Silvestri, Sorting out the document identifier assignment problem, in *Proc. of the 29th European Conference on IR Research*, Rome, Italy, 2007, pp. 101-112.

[6] H. Yan, S. Ding, and T. Suel, Inverted index compression and query processing with optimized document ordering, in *Proc. of the 18th International Conference on World Wide Web*, Madrid, Spain, 2009, pp. 401-410.

[7] S. Ding, J. Attenberg, and T. Suel, Scalable techniques for document identifier assignment ininverted indexes. in *Proc. of the 19th International Conference on World Wide Web*, Raleigh, North Carolina, USA, 2010, pp. 311-320.

[8] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava, Fast indexes and algorithms for set similarity selection queries, in *Proc. of the 24th International Conference on Data Engineering*, Cancun, Mexico, 2008, pp. 267-276.

[9] W. J. Bouknight, A procedure for generation of three-dimensional half-toned computer graphics presentations, *Communications of the ACM*, vol. 13, no. 9, pp.527-536, September 1970.

[10] Home page of DBLP bibliography, http://www.informatik.uni-trier.de/ ley/db, 2012.

[11] Home page of PubMed, http://www.ncbi.nlm.nih.gov/pubmed, 2012.

[12] S. Agrawal, S. Chaudhuri, and G. Das, DBXplorer: A system for keyword-based search over relational databases, in *Proc. of the 18th International Conference on Data Engineering*, San Jose, California, USA, 2002, pp. 5-16.

[13] V. Hristidis and Y. Papakonstantinou, DISCOVER: Keyword search in relational databases, in *Proc. of the 28th International Conference on Very Large Databases*, Hong Kong, China, 2002, pp. 670-681.

[14] V. Hristidis, L. Gravano, and Y. Papakonstantinou, Efficient IR-style keyword search over relational databases, in *Proc. of the 29th International Conference on Very Large Databases*, Berlin, Germany, 2003, pp. 850-861.

[15] H. He, H. Wang, J. Yang, and P. S. Yu, BLINKS: ranked keyword searches on graphs, in *Proc. of the ACM SIGMOD International Conference on Management of Data*, Beijing, China, 2007, pp. 305-316.

[16] Y. Luo, X. Lin, W. Wang, and X. Zhou, SPARK: Top-k keyword query in relational databases, in *Proc. of the ACM SIGMOD International Conference on Management of Data*, Beijing, China, 2007, pp. 115-126.

[17] F. Liu, C. T. Yu, W. Meng, and A. Chowdhury, Effective keyword search in relational databases, in *Proc. of the ACM SIGMOD International Conference on Management of Data*, Chicago, Illinois, USA, 2006, pp. 563-574.

[18] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou, EASE: An effective 3-in-1 keyword search method for unstructured, semi-structured and structured data, in *Proc. of the ACM SIGMOD International Conference on Management of Data*, Vancouver, BC, Canada, 2008, pp. 903-914.

[19] G. Li, J. Feng, and L. Zhou, Interactive search in XML data, in *Proc. of the 18th International Conference on World Wide Web*, Madrid, Spain, 2009, pp. 1063-1064.

[20] J. Zobel and A. Moffat, Inverted files for text search engines, *ACM Computing Surveys*, vol. 38, no. 2, pp. 6, 2006.

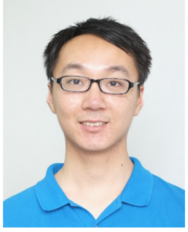
[21] A. Moffat and L. Stuiver, Binary interpolative coding for effective index compression, *Information Retrieval*, vol. 3, no. 1, pp. 25-47, 2000.

[22] V. N. Anh and A. Moffat, Inverted index compression using word-aligned binary codes, *Information Retrieval*, vol. 8, no. 1, pp. 151-166, 2005.

[23] V. N. Anh and A. Moffat, Improved word-aligned binary compression for text indexing, *IEEE Transactions on Knowledge and Data Engineering*, vol. 18, no. 6, pp. 857-861, 2006.

[24] J. Zhang, X. Long, and T. Suel, Performance of compressed inverted list caching in search engines, in *Proc. of the 17th International Conference on World Wide Web*, Beijing, China, 2008, pp. 387-396.

[25] H. Edelsbrunner, A new approach to rectangle intersections, *International Journal of Computer Mathematics*, vol. 13, no. 3/4, pp. 209-219/221-229, 1983.



Hao Wu is currently a PhD candidate in the Department of Computer Science and Technology, Tsinghua University, Beijing, China. He received his BEng degree in computer science from Tsinghua University in 2006. His research interest includes keyword search in structured database. He has published several research papers in main database conferences such as VLDB, WAIM, and NDBC. HP: http://dbgroup.cs.tsinghua.edu.cn/wuhao.

Guoliang Li is an assistant professor of Department of Computer Science and Technology, Tsinghua University, Beijing, China. He received his PhD degree in Computer Science from Tsinghua University in 2009, and his BEng degree in Computer Science from Harbin Institute of Technology in 2004. His research interests include data cleaning and integration, spatial databases and crowdsourcing. He has published more than 30 papers in premier conferences and journals, such as SIGMOD, VLDB, ICDE, IEEE TKDE, and VLDB Journal. He has served on the program committees of many database conferences, such as VLDB'12, VLDB'13, ICDE'12, ICDE'13, KDD'12, CIKM'10, and CIKM'12. His papers have been cited more than 1000 times, with two of them receiving more than 100 citations each. HP: http://dbgroup.cs.tsinghua.edu.cn/ligl.

Lizhu Zhou is a full professor of Department of Computer Science and Technology at Tsinghua University, China, and the President of Database Technology Committee of China Computer Federation. He received his MS degree in Computer Science from University of Toronto in 1983. His major research interests include database systems, digital resource management, web data processing, and information systems.