Decomposition of Inverted Lists and Word Labeling: A New Index Structure for Text Search

Y. Chen[1]**, and W. Shen**[2]
Dept. Applied Computer Science, University of Winnipeg, Winnipeg, Manitoba, Canada
[1]y.chen@uwinnipeg.ca, [2]wxshen1986@gmail.com

Abstract – *In a text database, a set of documents is maintained. To enquiry such a database, two kinds of queries are quite often used. One is the so-called conjunctive query, represented by a set of terms connected by conjunction (∧); and the other is the disjunctive query, which is also a set of terms, but connected by disjunction (∨). In this paper, we discuss an efficient and effective index mechanism to support the evaluation of both these two kinds of queries based on the inverted files. The main idea behind it is to associate each document word with an interval sequence based on a trie structure constructed over documents; and decompose an inverted list into a collection of disjoint sub-lists. In this way, both conjunctive and disjunctive queries can be conducted by performing a series of simple interval containment checkings. Experiments have been conducted, which shows that the new index is promising.*

Keywords: Search engine; Inverted files; queries

1 Introduction

Indexing the Web for fast keyword search is a key technology. In the past several decades, different indexing methods have been developed for this task, such as inverted files [1], signature files [5, 6] and signature trees [2] for indexing texts, and suffix trees and tries [7] for string matching. Especially, different variants of inverted files have been used by the Web search engines to find pages satisfying a query [8].

A text database can be roughly viewed as a collection of documents and each document is stored as a list of words. Over the documents, there are two kinds of Boolean queries, that is, queries that can be constructed from query terms by conjunction (∧) or disjunction (∨). A document D is an answer to a conjunctive query $w_1 \wedge w_2 \wedge \ldots \wedge w_k$ if it contains every w_i for $1 \le i \le k$ while D is an answer to a disjunctive query $w_1 \vee w_2 \vee \ldots \vee w_l$ if it contains any w_i for $1 \le i \le l$. Conjunction and disjunction can be nested to arbitrary depth, but can always be transformed to a conjunctive normal form:

$$(w_{11} \vee \ldots \vee w_{1l_1}) \wedge \ldots \wedge (w_{k1} \vee \ldots \vee w_{kl_k}).$$

In this paper, we discuss a new method to evaluate both conjunctive and disjunctive queries by decomposing an inverted list into a collection of disjoint sub-lists. The decomposition is based on the construction of a trie structure T over documents and then associating each document word with an interval sequence generated by labeling T by using a kind of tree encoding.

With this method, we can improve the efficiency of traditional methods by an order of magnitude or more.

2 New Index Structures

In this section, we mainly discuss our index structure, by which each word with high frequency will be assigned an interval sequence. We will then associate intervals, instead of words, with inverted sub-lists. To clarify this mechanism, we will first discuss interval sequences for words in 2.1. Then, in 2.2, how to associate inverted lists with intervals will be addressed.

2.1 Intervals assigned to words

Let $\boldsymbol{D} = \{D_1, ..., D_n\}$ be a set of documents. Let $W_i = \{w_{i1}, \ldots, w_{ij_1}\}$ $(i = 1,\ldots, n)$ be all of the words appearing in D_i, to be indexed. Denote $\boldsymbol{W} = \bigcup_{i=1}^{n} W_i$, called the *vocabulary*. We define the word appearance frequency by the following formula:

$$f(w) = \frac{num.of\ documents\ containing\ w}{num.\ of\ documents}, \qquad (w \in \boldsymbol{W}).$$

We then define a *frequency threshold* ζ. For any word w with $f(w) < \zeta$, we will associate it with an inverted list in a normal way, denoted as $\delta(w)$, exactly as in the method of inverted files. However, for all those with $f(w) \ge \zeta$, we will create a new index. For this, we will represent each D_i as a sequence containing all those words w with $f(w) \ge \zeta$, decreasingly sorted by $f(w)$. That is, in such a sequence, a word w precedes another w' if w is more frequent than w' in all documents. In addition, for any subset of words that have the same appearance frequency a *global ordering* is defined so that in each sorted word sequence this global ordering is followed. In addition, we maintain a hash table $\mathcal{H}$ that maps each word w to its inverted list $\delta(w)$ or to its new index.

Example 1 In Table 1, we show a set of four documents, their words w with $f(w) \ge \zeta = 0.4$, and the corresponding sorted word sequences, where we use a character to represent a word for simplicity.

Table 1: Documents and word sequences

DocID	words	sorted word sequence
1	*c, a, f, m, p*	*c, f, a, m, p*
2	*c, f, b, a*	*c, f, a, b*
3	*b, a, c, d*	*c, a, d, b*
4	*f, d, p, m*	*f, d, m, p*

Notice that the global order on {*f*, *a*, *c*} (with $f(w)$ = 0.75) is set to be $c \rightarrow f \rightarrow a$ while the global order on {*m*, *b*, *p*, *d*} (with $f(w)$ = 0.5) is $d \rightarrow b \rightarrow m \rightarrow p$.

For each document D_i (i = 1, …, n), we will use s_i to represent its sorted word sequence. Over all such sequences S = {s_1, …, s_n}, we will construct a digit tree, called a *trie*, as follows.

Assume that $\boldsymbol{W}$ = {w_1, …, w_m}. If $|S|$ = 0, the trie is, of course, empty. For $|S|$ = 1, *trie*(S) is a single node. If $|S| > 1$, S is split into m (possibly empty) subsets S_1, S_2, …, S_m so that a string is in S_j if its first word is w_j ($1 \le j \le m$). The tries *trie*(S_1), *trie*(S_2), …, *trie*(S_n) are constructed in the same way except that at the kth step, the splitting of sets is based on the kth words in the sequences. They are then connected from their respective roots to a single node to create *trie*(S). In Fig. 1, we show a trie T constructed over the sorted word sequences in Table 1.

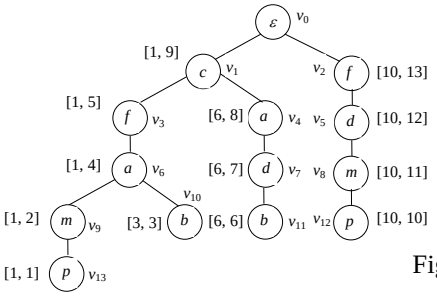

Figure 1. A trie

In the trie, v_0 is a *virtual* root, labeled with an *empty* word ε while any other node is labeled with a *real* word. Therefore, all the words on a path from the root to a leaf spell a sorted word sequence for a certain document. For instance, the path from v_0 to v_{13} corresponds to the sequence: *c*, *f*, *a*, *m*, *p*. Then, to check whether two words w_1 and w_2 are in the same document, we need only to check whether there exist two nodes v_1 and v_2 such that v_1 is labeled with w_1, v_2 with w_2, and v_1 and v_2 are on the same path. This shows that the *reachability* needs to be checked for this task, by which we ask whether a node v can reach another node u through a path. If it is the case, we denote it as $v \Rightarrow u$; otherwise, we denote it as $v \not\Rightarrow u$. The reachability problem on tries can be solved very efficiently by using a kind of tree encoding [3], which labels each node v in a trie with an interval $I_v = [\alpha_v, \beta_v]$, where β_v denotes the rank of v in a *post-order* traversal of the trie. Here the ranks are assumed to begin with 1, and all the children of a node are assumed to be ordered and fixed during the traversal. Furthermore, α_v denotes the lowest rank for any node u in $T[v]$ (the subtree rooted at v, including v). Thus, for any node u in $T[v]$, we have $I_u \subseteq I_v$ since the post-order traversal enters a node before all of its children, and leaves after having visited all of its children. In Fig. 1, we also show such a tree encoding on the trie, assuming that the children are ordered from left to right. It is easy to see that by interval containment we can check whether two nodes are on a same path. For example, $v_3 \Rightarrow v_{10}$, since $I_{v_3} = [1, 5]$, $I_{v_{10}} = [3, 3]$, and $[3, 3] \subset [1, 6]$; but $v_2 \not\Rightarrow v_9$, since $I_{v_2} = [10, 13]$, $I_{v_9} = [1, 2]$, and $[1, 2] \not\subset [10, 13]$.

Let $I = [\alpha, \beta]$ be an interval. We will refer to α and β as $I[1]$ and $I[2]$, respectively.

Lemma 1 For any two intervals I and I' generated for two nodes in a trie, one of four relations holds: $I \subset I'$, $I' \subset I$, $I[2] < I'[1]$, or $I'[2] < I[1]$. □

However, more than one node may be labeled with the same word, such as nodes v_9, and v_8 in Fig. 1. Both are labeled with word *m*. Therefore, a word may be associated with more than one node (or say, more than one node's interval). Thus, to know whether two words are in the same document, multiple checkings may be needed. For example, to check whether *p* and *d* are in the same document, we need to check v_{13} and v_{12} each against both v_7 and v_5, by using the node's intervals.

In order to minimize such checkings, we associate each word w with a word sequence of the form: $L_w = I_w^1, I_w^2, \ldots, I_w^k$, where k is the number of all those nodes labeled with w and each $I_w^i = [I_w^i[1], I_w^i[2]]$ ($1 \le i \le k$) is an interval associated with a certain node labeled with w. In addition, we can sort L_w by the interval's first value such that for $1 \le i < j \le k$ we have $L_w^i[1] < L_w^j[1]$, which will greatly reduce the time for the reachability checking. We illustrate this in Fig. 2, in which each word in Table 1 is associated with an interval sequence. From this figure, we can see that for any two intervals I and I' in L_w we must have $I \not\subset I'$, and $I' \not\subset I$ since in any trie no two nodes on a path are labeled with the same word.

c:	[1, 9]
f:	[1, 5][10, 13]
a:	[1, 4][6, 8]
d:	[6, 7][10, 12]
b:	[3, 3][6, 6]
m:	[1, 2][10, 11]
p:	[1, 1][10, 10]

Figure 2. Sorted interval sequences

As will be seen below, using such interval sequences, the checking of whether two words are in the same document can be done in a very efficient way.

Definition 1 (*word topological order*) Let S = {s_1, s_2, ..., s_n} be a set of n sorted word sequences. A word topological order over S is a sequence $\vartheta = w_1, w_2, \ldots, w_m$, which contains all the words appearing in S such that for any two words w and w' if w appears before w' in some s_j ($1 \le j \le n$) then w appears before w' in ϑ, denoted as $w \prec w'$. □

In Fig. 2, the words are also listed (from top to bottom) in a word topological order with respect to the sorted word sequences given in Table 1. To find a word topological order over $S = \{s_1, s_2, ..., s_n\}$ with $\boldsymbol{W} = \{w_1, \ldots, w_m\}$, we will transform the corresponding trie T to an *acyclic directed graph* (DAG) G by splitting the node set of T (except for the virtual root) into m groups such that all the nodes in a group are labeled with the same word, and then collapsing each group g to a single node u. There is an edge in G from u (standing for a group g) to u' (for another group g') if T contains (x, y) with $x \in g$ and $y \in g'$. For example, the trie shown in Fig. 1 will be transformed to a DAG shown in Fig. 3(a). Using a hash function H on the words in $\boldsymbol{W}$, the transformation can be done in $O(|\boldsymbol{W}|)$ time, by which all those nodes labeled with the same word w will be mapped to a single node identified by $H(w)$.

Let $G(V, E)$ be such a DAG. It is well known that only $O(|V| + |E|)$ time is required to find a *topological order* of G, which is a linear ordering of all its nodes such that if $u \rightarrow v \in E$, then u appears before v in the ordering. Replacing each node in the ordering with the corresponding word, we will obtain a word topological sequence, as illustrated in Fig. 3(b).

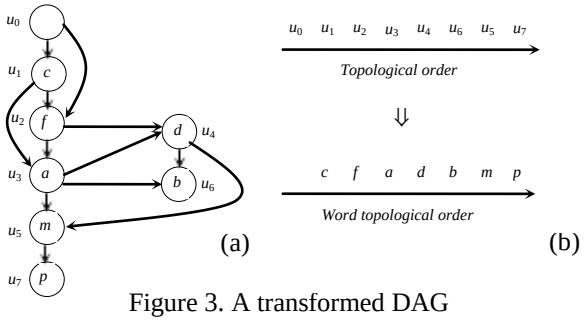

Figure 3. A transformed DAG

Now we consider two words w, w' with $w \prec w'$. It is easy to see that any interval in L_w cannot be contained in any interval in $L_{w'}$. Thus, to check whether w and w' are in the same document, we need only to check whether there exist $I \in L_w$ and $I' \in L_{w'}$ such that $I \supset I'$. This checking can be efficiently conducted as follows.

- Assume that $w \prec w'$. Let $L_w = I_w^1, I_w^2, \ldots, I_w^k$. Let $L_{w'} = I_{w'}^1, I_{w'}^2, \ldots, I_{w'}^{k'}$.
- Step through L_w and $L_{w'}$ from left to right. Let I_w^p and $I_{w'}^q$ be the intervals currently encountered. We will do one of the following operations:
  (1) If $I_w^p \supset I_{w'}^q$, report that w and w' are in the same document. Stop.
  (2) If $I_w^p[2] < I_{w'}^q[1]$, move to I_w^{p+1} if $p < k$ (then, in a next step, we will check I_w^{p+1} against $I_{w'}^q$.)
  (3) If $I_w^p[1] > I_{w'}^q[2]$, move to $I_{w'}^{q+1}$ if $q < k'$ (then, in a next step, we will check I_w^p against $I_{w'}^{q+1}$).
  (4) If $I_w^p \not\subset I_{w'}^q$ and $p = k$ or $q = k'$, report that w and w' are not in the same document. Stop.

The above process is referred to as a *two-word checking*, in which each interval in L_w and $L_{w'}$ is accessed only once. So only $O(|L_w| + |L_{w'}|)$ time is required. In Fig. 4, we illustrate the working process to check whether two words *d* and *m* are in a same document shown in Table 1.

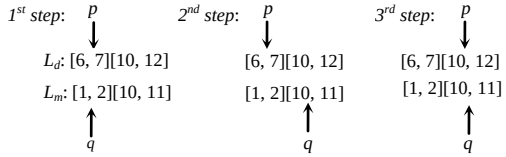

Figure 4. Illustration of two-word checking

In Fig. 4, we first notice that L_d = [6, 7][10, 12] and L_m = [1, 2][10, 11]. In the 1st step, we will check L_d^1 = [6, 7] against L_m^1 = [1, 2]. Since $L_d^1[1] = 6 > L_m^1[2] = 2$, we will check L_d^1 against L_m^2 = [10, 11] in a next step, and find $L_d^1[2] = 7 < L_m^2[1]$. So we will have to do the third step, in which we will check L_d^2 = [10, 12] against L_m^2. Since $L_d^2 \supset L_m^2$, we get to know that *d* and *m* are in the same document.

What we want is to extend this process to check whether a set of words are in the same document, based on which an efficient evaluation of conjunctive queries can be achieved. We will address this issue in Section 3.

2.2 Assignment of DocIds to Intervals

Another important component of our index is to assign document identifiers to intervals. An interval I can be considered as a representative of some words, i.e., all those words appearing on a *prefix* in the trie, which is a path P from the root to a certain node that is labeled with I. Then, the document identifiers assigned to I should be those containing all the words on P. For example, the words appearing on the prefix: $v_1 \rightarrow v_3 \rightarrow v_6$ in the trie shown in Fig. 1 are words *c*, *f*, and *a*, represented by the interval [1, 4] associated with v_6. So, the document identifiers assigned to [1, 4] should be {1, 2}, indicating that both documents D_1 and D_2 contain those three words. See the trie shown in Fig. 5 for illustration, in which each node v is assigned a set of document identifiers that is also considered to be the set assigned to the interval associated with v.

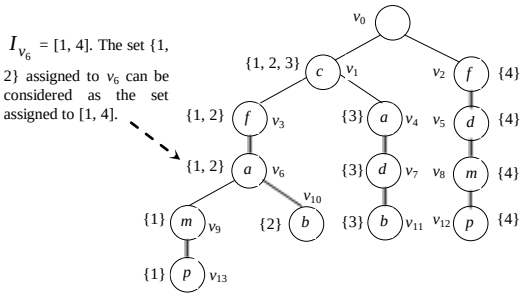

Figure 5. Illustration for assignment of document identifiers

Let v be the ending node of a prefix P, labeled with I. We will use $\delta(I)$, interchangeably $\delta(v)$, to represent the set of document

identifiers containing the words appearing on P. Thus, we have $\delta(v_6) = \delta([1, 4]) = \{1, 2\}$.

Lemma 2 Let u and v be two nodes in a trie T. If u and v are not on the same path in T, then $\delta(u)$ and $\delta(v)$ are disjoint, i.e., $\delta(u) \cap \delta(v) = \Phi$. □

Proposition 1 Assume that $v_1, v_2, \ldots, v_j$ be all the nodes labeled with the same word w in T. Then, $\delta(w)$, the inverted list of w (i.e., the list of all the documents identifiers containing w) is equal to $\delta(v_1) \uplus \delta(v_2) \uplus \ldots \uplus \delta(v_j)$, where $\uplus$ represents *disjoint union* over disjoint sets that have no elements in common.

Proof. Obviously, $\delta(w)$ is equal to $\delta(v_1) \cup \delta(v_2) \cup \ldots \cup \delta(v_j)$. Since $v_1, v_2, \ldots, v_j$ are labeled with the same word, they definitely appear on different paths as no nodes on a path are labeled with the same word. According to Lemma 2, $\delta(v_1) \cup \delta(v_2) \cup \ldots \cup \delta(v_j)$ is equal to $\delta(v_1) \uplus \delta(v_2) \uplus \ldots \uplus \delta(v_j)$. □

As an example, see the nodes v_2 and v_3 in Fig. 5. Both are labeled with word *f*. So the inverted list of *f* is $\delta(v_2) \uplus \delta(v_3) = \{4\} \uplus \{1, 2\} = \{1, 2, 4\}$.

3 Query Evaluation

Based on the interval sequences associated with words and the lists of document identifiers with intervals, we design our algorithm for evaluating queries.

3.1 Containment checking

Let $Q = \{w_1, w_2, \ldots, w_l\}$ be a set of words. Without loss of generality, assume that $w_1 \prec w_2 \prec \ldots \prec w_l$. We will check whether $w_1, w_2, \ldots, w_l$ are in the same document. For this purpose, we need to check whether there exists an interval sequence $\boldsymbol{I} = I_1, I_2, \ldots, I_l$ such that $I_j \in L_{w_j}$ and $I_j \supset I_{j+1}$ $(1 \le j \le l)$, where $I_{l+1} = \phi$, representing an empty interval. We call $\boldsymbol{I}$ a *containment sequence*.

Lemma 3 Let $Q = \{w_1, w_2, \ldots, w_l\}$ with $w_1 \prec w_2 \prec \ldots \prec w_l$. Denote by I_j an interval in L_{w_j} $(1 \le j \le l)$. If for some $1 \le i < j \le l$ we have $I_i \supset I_l$ and $I_j \supset I_l$, then $I_i \supset I_j$. □

As an example, consider $Q = \{f, a, p\}$ with $f \prec a \prec p$. From Fig. 2, we can see that $L_f = [1, 5][10, 13]$, $L_a = [1, 4][6, 8]$, and $L_p = [1, 1][10, 10]$. Obviously, $I_f^1 = [1, 5] \supset I_p^1 = [1, 1]$, and $I_a^1 = [1, 4] \supset I_p^1 = [1, 1]$. Then, we must have $I_f^1 \supset I_a^1$.

According to the above lemma, the checking of $I_{j+1} \subset I_j$ can be replaced by checking whether we have $I_{j+1} \supset I_l$ if we know $I_j \supset I_l$. Thus, the task to find a containment sequence can be done by slightly modifying step (1) in the two-word checking discussed in 2.1. That is, each time we find p, q $(1 \le p \le |L_{w_{l-1}}|, 1 \le q \le |L_{w_l}|)$ such that $I_{w_{l-1}}^p \supset I_{w_l}^q$, we need only to further check whether there exist $l - 2$ other intervals $I_1, I_2, \ldots, I_{l-2}$ such that each I_j is in L_{w_j} and $I_j \subset I_{w_l}^q$ for $1 \le j \le l - 2$. This will greatly simplify the process for finding a containment sequence.

For this purpose, we define an operation *con*(w, I, j) to check whether an interval I is contained in some interval between jth and the last interval in L_w. If I is contained in an ith interval in L_w, the return value of *con*(w, I, j) is a pair (*true*, i); otherwise, the return value is (*false*, i'), where i' is the least number such that $I_w^{i'}[1] > I[2]$. In addition, to simplify the control process, we place a *sentinel* at the end of L_w, whose value is set to be $[\infty, \infty]$ so that whenever we reach the sentinel of L_w, *con*(w, I, j) returns (*false*, $|L_w| + 1$).

This operation will be used in the following algorithm, by which we will check, for a set $Q = \{w_1, w_2, \ldots, w_l\}$ with $w_1 \prec w_2 \prec \ldots \prec w_l$, whether each L_{w_j} $(1 \le j \le l)$ possesses an interval which contains a given interval I.

The input of this algorithm is a triplet (Q, I, b), where b is an integer array of length $|Q|$ with each $b[j]$ indicating the starting position to check L_{w_j} $(1 \le j \le l)$. For example, if $b[i] = 2$ for some i, we will check L_{w_i} starting from $I_{w_i}^2$. Initially, each entry in b is set to be 1. We also store Q as an array. Then, $Q[i]$ refers to w_i for $1 \le i \le l$.

ALGORITHM *interval-check*(Q, I, b)

begin

1. *mark* := *true*; j := $|Q|$; assume that $Q[1] \prec Q[2] \prec \ldots \prec Q[l]$;
2. **while** (*mark* = *true* and $j \ge 1$) **do** {
3. (x, y) := *con*($Q[j]$, I, $b[j]$); $b[j]$:= y; /* $Q[j] = w_j$ */
4. **if** (x = *true*) **then** j := $j - 1$
5. **else** {*mark* := *false*;}
6. }
7. **if** (*mark* = *true*) **then** return (*true*, b)
8. **else** return (*false*, b);

end

The output of the algorithm is a pair (t, b'). If in each L_{w_j} $(1 \le j \le l)$ we can find an interval that contains I, t is *true*; otherwise, t is *false*. b' is an array satisfying the following properties:

(i) If t is *true*, each $b'[j]$ is an integer i showing that it is the ith interval in L_{w_j} that contains I.

(ii) If t is *false*, there exists j dividing b into three parts: $b'[1 .. j - 1]$, $b'[j]$, and $b'[j + 1 .. l]$ such that for any index k,

1. If $j + 1 \le k \le l$, then $b'[k]$ is an integer i such that ith interval in L_{w_k} contains I.
2. If $k = j$, then in L_{w_k} no interval is able to contain I and $b'[k]$ is $|L_{w_k}| + 1$ or a least number i such that $I_{w_k}^i[1] > I[2]$.
3. If $1 \le k \le j - 1$, then $b'[k]$ is the same as $b[k]$ (see line 5; the execution of this line will enforce the control to get out of the **while**-loop, and leave $b[1 .. j - 1]$ not updated.)

Lemma 4 Let (t, b') be the return value of *interval-check*(Q, I, b). Then, if t is *true*, b' satisfies property (i). Otherwise, b' satisfies (ii). □

The two properties (i) and (ii) described above are very important to the efficiency and correctness of our main algorithm to check whether $Q = \{w_1, w_2, \ldots, w_l\}$ is in the same document. Assume that $w_1 \prec w_2 \prec \ldots \prec w_l$. Its main idea is to find p, q such that $I^p_{w_{l-1}} \supset I^q_{w_l}$, and then use the above algorithm to check whether for each $w \in R = \{w_1, \ldots, w_{l-2}\}$ L_w has an interval containing $I^q_{w_l}$.

ALGORITHM *containment*(Q, b)

```
begin
2.  let |Q| = l; assume that Q[1] ≺ Q[2] ≺ … ≺ Q[l];
3.  R := {Q[1], …, Q[l - 2]};
3.  p := b[l – 1]; q := b[l];
4.  while (p ≤ |L_Q[l-1]|) and q ≤ |L_Q[l]|) do {
5.    if L^p_Q[l–1] ⊃ L^q_Q[l] then {
6.      (x, b) := interval-check(R, L^q_w[l], b);
7.      if (x = true) then {return (true, b);}
8.      else {q := q + 1; b[l] := q;}
9.    }
10.   else {
11.     if ( L^p_Q[l–1][2] < L^q_Q[l][1]) then {p := p + 1; b[l - 1] := p;}
12.     else {q := q + 1; b[l] := q;}
13.   }
14. }
15. return (false, b);
end
```

The **while**-loop in the above algorithm is almost the same as the two-words checking (see 2.1). The only difference consists in lines 5 – 9. In the case of $L^p_{Q[l-1]} \supset L^q_{Q[l]}$, we will continually check whether there is an interval in each $L_{Q[j]}$ $(1 \le j \le l - 2)$ which contains $L^q_{Q[l]}$; but this is done simply by calling the algorithm *interval-check*( ) (see line 6.)

In addition, special attention should be paid to array b, whose values can also be utilized to indicate the checked intervals in every interval sequence. This enables us to avoid any redundancy when we want to find all the possible containment sequences by using this algorithm, which is required to evaluate conjunctive queries.

Example 2 Continued with Example 1. We will check two sets of words: $Q = \{f, a, p\}$ and $Q' = \{c, d, m, p\}$ to see whether each of them is in the same document.

For Q, we have $Q[1] = f \prec Q[2] = a \prec Q[3] = p$. Initially $b = \{1, 1, 1\}$ (i.e., b is an array containing three entries $b[1] = b[2] = b[3] = 1$). From Fig. 2, we see that $L_{Q[1]} = L_f = [1, 5][10, 13]$; $L_{Q[2]} = L_a = [1, 4][6, 8]$; and $L_{Q[3]} = L_p = [1, 1][10, 10]$.

In the 1st iteration of the **while**-loop, we will check $L^1_{Q[2]}$ against $L^1_{Q[3]}$. Since $L^1_{Q[2]} = [1, 4] \supset L^1_{Q[3]} = [1, 1]$, we will call *interval-check*(R, I, b), where $R = \{f\}$, $I = [1, 1]$, and $b = \{1, 1, 1\}$ (note that $b[2]$ and $b[3]$ will not be used in the execution of *interval-check*(R, I, b)). This call returns (*true*, $\{1, 1, 1\}$), which is used as the return value of *containment*(Q, b) (see line 7).

Now we consider $Q' = \{c, d, m, p\}$ with $c \prec d \prec m \prec p$. Again, initially $b = \{1, 1, 1, 1\}$; $L_{Q[1]} = L_c = [1, 9]$; $L_{Q[2]} = L_d = [6, 7][10, 12]$; $L_{Q[3]} = L_m = [1, 2][10, 11]$; and $L_{Q[4]} = L_p = [1, 1][10, 10]$. We will have the following working process.

1st iteration of the **while**-loop:
check $L^1_{Q[3]}$ against $L^1_{Q[4]}$. Since $L^1_{Q[3]} = [1, 2] \supset L^1_{Q[4]} = [1, 1]$, we will call *interval-check*$(R = \{c, d\}, I = [1, 1], b = \{1, 1, 1, 1\})$, which returns (*false*, $b = \{1, 1, 1, 1\}$). In this case, line 8 will be conducted (by which index q – index to scan $L_{Q[4]}$, will be increased by 1), and then in a next iteration we will check $L^2_{Q[4]}$.

2nd iteration of the **while**-loop:
check $L^1_{Q[3]}$ against $L^2_{Q[4]}$. Since $L^1_{Q[3]}[2] = 2 < L^2_{Q[4]}[1] = 10$, line 11 will be conducted (by which index p, - index to scan $L_{Q[3]}$, will be increased by 1), and in a next iteration we will check $L^2_{Q[3]}$.

3rd iteration of the **while**-loop:
check $L^2_{Q[3]}$ against $L^2_{Q[4]}$. Since $L^2_{Q[3]} = [10, 11] \supset L^2_{Q[4]} = [10, 10]$, we will call *interval-check*$(R = \{c, d\}, I = [10, 10], b = \{1, 1, 2, 2\})$, which returns (*false*, $b = \{3, 2, 2, 2\}$). In this case, line 8 will be conducted (by which index q will be increased by 1), which will get the execution out of the **while**-loop and *containment*(Q, b) returns (*false*, $\{3, 2, 2, 3\}$). □

Proposition 2 Algorithm *containment*( ) is correct. □

Proof. We only need to prove that values for b are correctly changed, since it guarantees that the return value of each call *interval-check*( ) is correct. We prove this by induction of the number k of *interval-check*( ) calls.

When $k = 1$, it is obviously correct since each entry $b[j]$ is set to 1.

Assume that when k it is correct, we will prove that by the $(k + 1)$th call b is also correctly changed. We first notice that if the return value of the kth call is (*true*, b) the $(k + 1)$th call will not be invoked. So we consider only the case that the return value of the kth call is (*false*, b). Assume that the kth call is of the form *interval-check*$(R, L^q_{Q[l]}, b)$. Then, the $(k + 1)$th call is of the form *interval-check*$(R, L^{q+1}_{Q[l]}, b')$, where b' is an array changed by the execution of *interval-check*$(R, L^q_{Q[l]}, b)$. In terms of the induction hypothesis, it is correct. Also, b' can be divided into three parts according to property (ii) shown above. From this, we can see that $L^{q+1}_{Q[l]}$ cannot be contained in the $(b'[j] - 1)$th interval in any $L_{Q[j]}$ $(1 \le j \le l - 2)$. From Lemma 3, we know that b' will be correctly changed by the execution of *interval-check*$(R, L^{q+1}_{Q[l]}, b')$. □

The above algorithm can be greatly improved as follows.

- *By checking sentinels*. Once the return value of a call *con*($R[j]$, $L^q_{w[l]}$, $b[j]$) is of the form (*false*, *y*) with *y* pointing to a sentinel, we can stop the whole process immediately as in this case, $w_1, w_2, \ldots, w_l$ cannot be in the same document.
- *By marking successful checkings*. Each time we find a containment sequence $I_1, I_2, \ldots, I_{l-1}, I_l$ such that $I_j \in L_{Q[j]}$ ($1 \le j \le l$) and $I_j \supset I_{j+1}$ ($1 \le j \le l - 1$), we mark I_{l-1}. Then, we can find a next containment sequence $I_1, I_2, \ldots, I_{l-1}, I$ immediately, where I is an interval directly next to I_l in L_{w_l}, if $I_{l-1} \supset I$ and I_{l-1} is marked. In this way, each interval in all $L_{Q[j]}$'s can be visited at most two times by using the algorithm to find all the possible containment sequences.

We refer to the modified algorithm as *containment**(*Q*, *b*). However, due to space limitation, its formal description is omitted.

Proposition 3 The time complexity of *containment**(*Q*, *b*) is bounded by $O(\sum_{w \in Q} |L_w|)$. □

Finally, we notice that each L_w is sorted, and then we can use the binary or galloping search [5] to scan it. In this way, the average time complexity can be improved to $O(|L_{w_l}| + \sum_{w \in Q \setminus \{w_l\}} \log^2 |L_w|)$. We can also use the interpolation method to probe position in an interval sequence.

3.2 Evaluation of conjunctive queries

The containment-checking algorithm discussed in 3.1 can easily be adapted to evaluate conjunctive queries of the form $Q = w_1 \wedge w_2 \wedge \ldots \wedge w_l$ with $w_1 \prec w_2 \prec \ldots \prec w_l$. What needs to change is to find all the possible containment sequences for $\{w_1, w_2, \ldots, w_l\}$. This can simply be done by repeatedly calling the algorithm *containment**( ). Let $\boldsymbol{I}_1, \boldsymbol{I}_2, \ldots, \boldsymbol{I}_m$ be all the found containment sequences. Let $\boldsymbol{I}_i = I_{i1}, I_{i2}, \ldots, I_{il_i}$ ($i = 1, \ldots, m$). Then, the answer to *Q* should be $\delta(I_{1l_1}) \uplus \ldots \uplus \delta(I_{ml_m})$. Based on this analysis, we give the following algorithm for evaluating conjunctive queries.

ALGORITHM *con-evaluation*(*Q*)

```
begin
4.  let |Q| = l; assume that Q[1] ≺ Q[2] ≺ … ≺ Q[l];
5.  for (j = 1 to l) do b[j] := 1;
6.  R := Φ; i := 1;
4.  while (i ≤ |L_w[l]|) do { (t, b) := containment*(Q, b);
5.    if t = true then {
6.          R := R ⊎ δ(I^j_w[l]); b[l] := b[l] + 1;
7.    }
8.    i := b[l] ;
9.  }
10. return R;
end
```

In the main **while**-loop (see line 4) of the above algorithm, we repeatedly call the algorithm *containment**( ) to find all the possible containment sequences. For each of them, a set of document identifiers can be determined and the disjoint union of all such sets makes up the result.

Obviously, the time complexity of the algorithm is bounded by $O(\sum_{w \in Q} |L_w|)$, but can be further improved by using the binary, or galloping search [5], as well as the interpolation probing [17].

Example 3 Continued with Example 1. Let $Q = f \wedge m \wedge p$. Then, the execution of *containment**( ) will find two containment sequences: $\boldsymbol{I}_1$ = [1, 5], [1, 2], [1, 1] and $\boldsymbol{I}_2$ = [10, 13], [10, 11], [10, 10]. The results is then $R = \delta([1, 1]) \uplus \delta([10, 10]) = \{1\} \uplus \{4\} = \{1, 4\}$. □

3.3 Evaluation of disjunctive queries

Based on the interval sequences associated with words, the disjunctive queries can also be evaluated efficiently and even more interesting. For ease of explanation, we first show how to evaluate a query of the form: $w \vee w'$. Then, the general case will be discussed.

Again, we assume that $w \prec w'$. Then, any interval in L_w cannot be contained in any interval in $L_{w'}$. However, some intervals in $L_{w'}$ may fall in some intervals in L_w. To find all the documents each containing either *w* or *w*′, we need to merge any interval in $L_{w'}$ into L_w if it does not fall in any interval in L_w. As with the containment-checking algorithm, we will scan both L_w and $L_{w'}$ from left to right, but with some intervals in $L_{w'}$ possibly merged into L_w:

- Let $L_w = I^1_w, I^2_w, \ldots, I^k_w$. Let $L_{w'} = I^1_{w'}, I^2_{w'}, \ldots, I^{k'}_{w'}$.
- Step through L_w and $L_{w'}$ from left to right. Let I^p_w and $I^q_{w'}$ be the intervals currently encountered. We will do the following checkings:
  (1) If $I^p_w \supset I^q_{w'}$, move to $I^{q+1}_{w'}$ if $q < k'$. If $q = k'$, go to (4).
  (2) If $I^q_{w'}[2] < I^p_w[1]$, insert $L^q_{w'}$ into L_w just before I^p_w. If $q < k'$, move to $L^{q+1}_{w'}$; otherwise ($q = k'$), go to (4).
  (3) If $I^p_w[2] < I^q_{w'}[1]$, move to I^{p+1}_w if $p < k$. If $p = k$, append $I^q_{w'}, \ldots, I^{k'}_{w'}$ to the end of L_w and then go to (4).
  (4) Let $I_1, \ldots, I_{k''}$ be all the intervals in the changed L_w. Return $\delta(I_1) \uplus \ldots \uplus \delta(I_{k''})$.

  We denote this procedure as $L = merge(L_w, L_{w'})$.

Example 4 Continued with Example 1. Let $Q = d \vee m$. We have $d \prec m$. By using the above procedure to merge L_m = [1, 2][10, 11] into L_d = [6, 7][10, 12], we will get a new sequence: [1, 2][6, 7][10, 11]. So, the result is $\delta([1, 2]) \uplus \delta([6, 7]) \uplus \delta([10, 12]) = \{1\} \uplus \{3\} \uplus \{4\} = \{1, 3, 4\}$. In the first step, we compare I^1_d = [6, 7] and I^1_m = [1, 2]. Since $I^1_d[1] = 6 > I^1_m[2] = 2$, I^1_m will be inserted into L_d just before I^1_d. Then, in the second step, we will compare I^1_d and I^2_m. Since $I^1_d[2] = 7 < I^2_m[1] = 10$, we will move to I^2_d. Next, in the third step, we compare I^2_d and I^2_m, and find $I^2_d \supset$

I_m^2. Since I_m^2 is the last interval in L_m, we terminate the merging process and return the result. □

Fig. 6 shows the entire merging process.

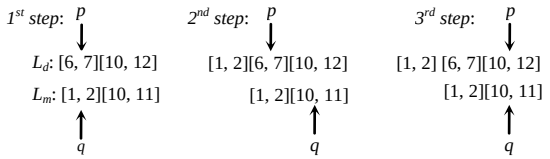

Figure 6: A merging process

This merging process can easily be extended to a general algorithm to evaluate disjunctive queries of the form $Q = w_1 \vee w_2 \vee \ldots \vee w_l$ with $w_1 \prec w_2 \prec \ldots \prec w_l$, as shown below.

ALGORITHM *dis-evaluation*(Q)

begin

1. let $|Q| = l$; assume that $Q[1] \prec Q[2] \prec \ldots \prec Q[l]$;
7. $L := L_{Q[1]}$;
8. **for** (i = 2 to l) **do** {
9. $L := merge(L, L_{Q[i]})$;
5. }
6. let $L = I_1, \ldots, I_k$;
7. return $\delta(I_1) \uplus \ldots \uplus \delta(I_{k''})$;

end

In the above algorithm, we use *merge*( ) to merge $L_{Q[i]}$ for i = 2, …, l into $L_{Q[1]}$ one by one. The running time is obviously bounded by $O(l \cdot r)$, where r is the largest number of intervals in all $L_{Q[i]}$'s which are not contained in each other. Again, the time requirement can be improved by using the binary search, the galloping search, and the interpolation probing.

4 Experiments

In the experiments, we have tested four methods:
Signature trees [2] (*ST* for short),
Inverted files [1] (*IF* for short),
Set intersection [4] (*SI* for short),
Interval based method (discussed in the paper; *IbM* for short).

All our experiments are performed on a 32-bit Windows operating system. The processor is Intel Core 2 Duo CPU with 4GB RAM. All index techniques are implemented by C++ and compiled by Microsoft Visual Studio 2010. We use the function *QueryPerformanceCounter()* from the *Kernel32.lib* library to measure the *CPU time*, which provides a high-precision timing (microsecond precision) on the Windows Platform.

- *Data sets*

To test the effectiveness of our index, we use a sample Web corpus, which contains one million text documents. We numbered the documents as they were stored, by assigning them a sequential number indicating their order in the indexing process. The characteristics of this collection are shown in Table 2.

Table 2: Characteristics of Web

	Web
Documents	1,000,000
Size (gigabytes)	7.5
Word occurrences (without markup)	3,603,556
Distinct words (after stemming)	285,344

- *Index construction time and sizes*

In Table 3, we show the time for constructing different indexes and their sizes. For this test, each document identifier and each interval occupy 4 bytes. For our method, the threshold ζ is set to be 1/1000. That is, only for those words w appearing in more than 100 documents an interval sequence will be established.

Table 3: Index construction time and size

	IF	*SI*	*ST*	*IbM*
Time (ms)	8,755	8,755	153,847	52,861
Size (MB)	14	14	20	14.4

From this table, we can see that the inverted file has the best time and space requirement than the other two methods. However, the space requirement by our method is just a little bit worse than the inverted method. For *SI*, they are exactly the same as *IF*.

- *Time of conjunctive queries*

In Fig. 6, we show the number of page access and the elapsed times for evaluating conjunctive queries containing different number of words. For this test, all the words are chosen randomly, but appear in more than 100 documents since only for such words the interval sequences are created. In addition, the page size is set to be 4KB. For the inverted file, a *melding algorithm* [5, 6] is used for doing the set intersection, which intersects the inverted lists two at a time in increasing order by size, starting with the two smallest. Also, it performs a binary search to determine whether a document identifier in the first list appears in the second list.

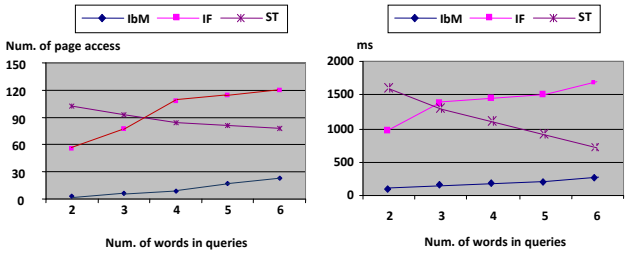

Figure 6. Test of conjunctive queries with page size 4KB

For each query, we average the running time over 20 executions.

From Fig. 6, we can see that our method is much better than both the inverted file and the signature file. Even the signature tree beats the inverted file. Especially, as the number of words in queries increases, both the number of

page access and the time of the signature tree decease. It is because a query signature is formed by superimposing (bit-wise OR) all the signatures of the words in a query. So, the more words in a query, the more 1's in a query signature, which will lead to less nodes to be explored in a signature tree. *SI* is an in-memory algorithm, not run for this test.

In Fig. 7, we show the results when the page size is set to be 12KB. From this, we can see that although the number of page access has been reduced, the time used is almost for all the three tested methods.

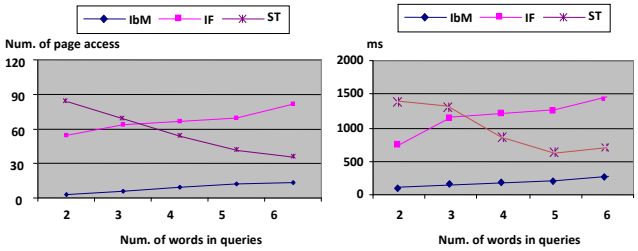

Figure 7. Test of conjunctive queries with page size 12KB

In Fig. 8, we show the test results when the whole index structure is accommodated in main memory for all four different methods.

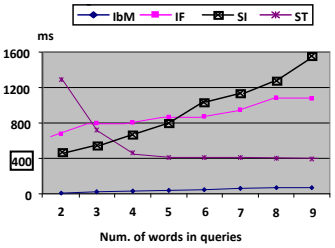

Figure 8. Conjunctive queries with whole index in main memory

- *Time of disjunctive queries*

In Fig. 9, 10, and 11, we show the test results for disjunctive queries, for which the signature file is not tested since it is totally not suitable for this task.

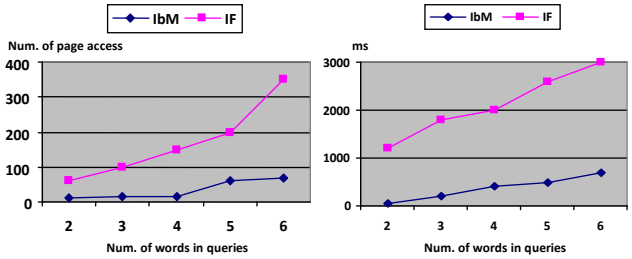

Figure 9. Test of disjunctive queries with page size 4KB

From these figures, we can see that more time is needed to evaluate a disjunctive query than a conjunctive for both the inverted file and ours. However, the discrepancy between these two kinds of queries for the inverted file is larger than for ours. It is because by the inverted file the normal set union is used with not much optimality being made. In the opposite, by ours the interval containment checking still works quite well even though the binary or galloping search has not been utilized.

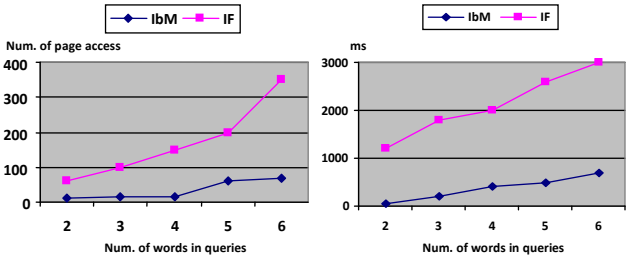

Figure 10. Test of disjunctive queries with page size 12KB

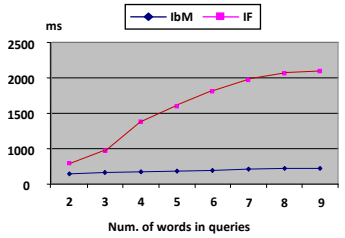

Figure 11. Test of disjunctive queries with whole index in main memory

5 Conclusion

In this paper, a new method is discussed to evaluate conjunctive queries. The main idea is to transform an evaluation of queries to a series of reachability checkings, which improves the traditional method by an order of magnitude or more.

6. References

[1] Anh, V.N. and A. Moffat, A, 2005. Inverted index compression using word-alinged binary codes, *Kluwer Int. Journal of Information Retrieval 8, 1*, pp. 151-166.

[2] Chen, Y. and Chen, Y.B. 2006. On the Signature Tree Construction and Analysis, *IEEE TKDE*, Vol.18, No. 9, pp 1207 – 1224.

[3] Y. Chen and Y.B. Chen. An Efficient Algorithm for Answering Graph Reachability Queries, in *Proc. 24th Int. Conf. on Data Engineering (ICDE 2008)*, IEEE, April 2008, pp. 892-901.

[4] B. Ding, A.C. König, Fast set intersection in memory, *Proc. of the VLDB Endowment, v.4 n.4*, p.255-266, January 2011.

[5] Faloutsos, C. 1985. Access Methods for Text, *ACM Computing Surveys*, vol. 17, no. 1, pp. 49-74.

[6] Faloutsos, C. and Chan, R. 1988. Fast Text Access Methods for Optical and Large Magnetic Disks: Designs and Performance Comparison, *Proc. 14th Int'l Conf. Very Large Data Bases*, pp. 280-293.

[7] D.E. Knuth, *The Art of Computer Programming, Vol. 3*, Massachusetts, Addison-Wesley Publish Com., 1975.

[8] R. Lempel and S. Moran, Predictive caching and prefetching of query results in search engines, in *Proc. the World Wide Web Conf.*, Budapest, Hungary, ACM, 19-28, 2003.