On the Intersection of Inverted Lists

Yangjun Chen[1] and Weixin Shen[2]
Dept. Applied Computer Science, University of Winnipeg, Canada
[1]y.chen@uwinnipeg.ca, [2]wxshen1985@gmail.com

Abstract— *In this paper, we discuss an efficient and effective index mechanism to support set intersections, which are important to evaluation of conjunctive queries by search engines. The main idea behind it is to decompose an inverted list associated with a word into a collection of disjoint sub-lists by arranging a set of word sequences into a trie structure. Then, by using a kind of tree encoding, we can replace each inverted list with a much shorter interval sequence. In this way, we can transform the comparison of document identifiers to the checking of interval containment by associating each interval with a sub-list. More importantly, for a sorted interval sequence the binary search can also be used. With the lowest common ancestors being utilized to control the search, a better theoretical time complexity than any traditional method can be achieved.*

Key words: *Search engine; inverted files; conjunctive queries; disjunctive queries.*

1. INTRODUCTION

Indexing the Web for fast keyword search is among the most challenging applications for scalable data management. In the past several decades, different indexing methods have been developed to speed up text search, such as *inverted files* [14, 15], *signature files* and *signature trees* for indexing texts [1, 5, 6, 11, 12]; and *suffix trees* and *tries* [13] for string matching. Especially, different variants of inverted files have been used by the Web search engines to find pages satisfying conjunctive queries of the form:

$$w_1 \wedge w_2 \wedge \ldots \wedge w_k.$$

A document D is an answer to such a query if it contains every w_i for $1 \leq i \leq k$. The algorithms developed to evaluate such a query typically use *inverted lists*, each of which comprises all those document identifiers containing a certain word. So, to find all the documents satisfying a query, set intersections have to be conducted.

There has been considerable study on this topic, such as *adaptive* algorithms [9], *melding* algorithms [2], building additional data structures like *skipping lists* [32], *treaps* (a kind of balanced trees) [4], *hash tables* over sorted lists [3, 10], and so on. All of them can improve the time complexity at most by a constant factor, but none of them is able to break through the linear time bottleneck.

In this work, we explore a different way to speed up the operation by constructing indexes, which are substantially different from any existing strategy. Concretely, our method works as follows.

- Represent each document as a word sequence, sorted decreasingly by the word appearance frequency (referred to as a *document word sequence*, or simply a *word sequence*), and then construct a trie structure over all such sequences.
- Associate each word with an interval sequence L, where each interval in L is created by applying a kind of tree encoding over the generated trie structure.
- Associate each interval, rather than a word, with a set of document identifiers. In this way, we decompose an inverted list associated with a word into a collection of disjoint sub-lists, and transform the comparison of document identifiers to the checking of interval containment.
- For each word w, instead of its interval sequence, we will construct a balanced binary tree over an even shorter interval sequence with each being an interval for a lowest common ancestor of some nodes labelled with w. The set intersection operation can then be done by searching a binary tree against a series of intervals.

Let δ_x and δ_y be two inverted lists associated with two words x and y, respectively. Without loss of generality, assume that $|\delta_x| < |\delta_y|$. Up to now, the best comparison-based algorithm for intersecting L_x and L_y requires $O(|\delta_x| \cdot \log \frac{|\delta_y|}{|\delta_x|})$ time. In contrast, our algorithm needs $O(|L_y| \cdot \log \frac{\lambda_x}{|L_y|})$ time, where L_x and L_y are the interval sequences created for L_x and L_y, respectively; and λ_x is the size of a subset of nodes with each being a lowest common ancestor of some nodes labeled with x in the trie. Generally, we have $|L_y| \leq |L_x| \leq |\delta_x|$ and $\lambda_x < |\delta_x|$. This time complexity is significantly better than the traditional methods due to the following two key facts:

1. Each interval corresponds to a sub-list of an inverted list. Therefore, in general, the length of an interval sequence associated with a word is much shorter than the inverted list for that word. Especially, the larger an inverted list is, the smaller its corresponding interval sequence. Only for those very short inverted lists (associated with low frequent words), the sizes of their corresponding interval sequences may be near their sizes.
2. During the search of a tree constructed over intervals, the relationship between a set of nodes and their lowest common ancestor can be used to skip over a lot of useless interval containment checkings while it is not possible by any tree built over an inverted list.

Moreover, our index structure can also be easily maintained.

2. NEW INDEX STRUCTURE

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 Interval sequences 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*. For each word $w \in \boldsymbol{W}$, we will associate it with an inverted list containing all the document identifiers with each containing w. Thus, to answer a conjunctive query, a set intersection over some inverted lists has to be conducted.

For the purpose of the new index structure, we will put all the words in a sorted sequence $\vartheta = w_1, w_2, \ldots, w_m$ $(m = |\boldsymbol{W}|)$ such that for any two words w and w' if the frequency of w is higher than w' then w appears before w' in ϑ, denoted as $w \prec w'$. Then, each document can be represented as a subsequence of ϑ; and over all these subsequences a trie structure can be established as illustrated in Fig. 1.

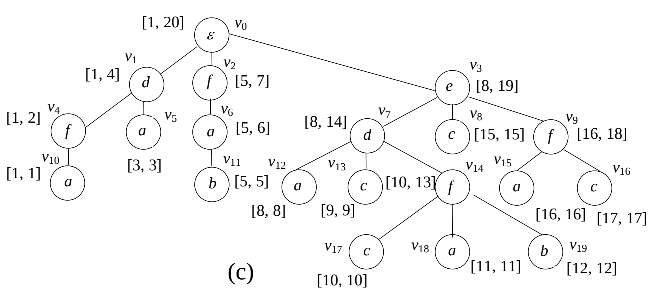
Documents and word sequences:

DocId	words	words
1	*a, f, d*	*d, f, a*
2	*a, d*	*d, a*
3	*a, e, d*	*e, d, a*
4	*f, b, a*	*f, a, b*
5	*c, d, e*	*e, d, c*
6	*d, f, e, c*	*e, d, f, c*
7	*f, d, e, a*	*e, d, f, a*
8	*f, d, e, b*	*e, d, f, b*
9	*e, c*	*e, c*
10	*a, e, f*	*e, f, a*
11	*f, e, c*	*e, f, c*

(a)

e: {4, 5, 6, 7, 8, 9, 10, 11}
d: {1, 2, 4, 5, 6, 7, 8}
f: {1, 3, 5, 6, 7, 9, 10}
a: {1, 2, 3, 4, 7, 10}
c: {5, 6, 9, 11}
b: {3, 8}

(b)



(c)

Fig. 1: A trie and a set of sorted interval sequences

In Fig. 1(a), we show a document database containing 11 documents, their words, and their sorted sequences by the word frequency, where we use a character to represent a word for simplicity. In Fig. 1(b), we show the inverted lists for all the words in the database. The trie over all the sorted sequences is shown in Fig. 1(c).

In this 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*, *p*, *m*. 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 [7][8], 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 visits a node after all of its children have been accessed. In Fig. 1(c), 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, $v_3 \Rightarrow v_{19}$, since $I_{v_3} = [8, 19]$, $I_{v_{19}} = [12, 12]$, and $[12, 12] \subset [8, 19]$; but $v_2 \not\Rightarrow v_{18}$, since $I_{v_2} = [5, 7]$, $I_{v_{18}} = [11, 11]$, and $[11, 11] \not\subset [5, 7]$.

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 $I'[2] < I[1]$.

Proof. It is easy to prove. □

However, more than one node may be labeled with the same word, such as nodes v_1, and v_6 in Fig. 1(c). Both are labeled with word *d*. 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 *d* and *b* are in the same document, we need to check v_1 and v_6 each against both v_{16} and v_{19}, 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 Fig. 1(a) is associated with an interval sequence.

e: [8,19]
d: [1, 4][8, 14]
f: [1, 2][5, 7][10, 13][16, 18]
a: [1, 1][3, 3][5, 6][8, 8][11, 11][16, 16]
c: [9, 9][10, 10][15, 15][17, 17]
b: [5, 5][12, 12]

Fig. 2: a set of interval sequences

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.

In addition, for any interval sequence L, we will use $L[i]$ to refer to the ith interval in L, and $L[i .. j]$ to the segment from the ith to the jth interval in L.

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: $v_0 \to v_3 \to v_7 \to v_{14}$ in the trie shown in Fig. 1(c) are words: ε, *e*, *d*, and *f*, represented by the interval [10, 13] associated with v_{14}. So, the document identifiers assigned to [10, 13] should be {6, 7, 8}, indicating that documents D_6, D_7 and D_8 all contain those three words. See the trie shown in Fig. 3 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.

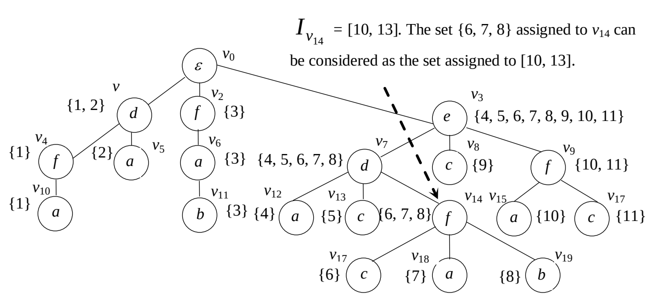


Fig. 3: Illustration for assignment of document IDs

Let v be the ending node of a prefix P, labeled with I. We will use δ_I, interchangeably δ_v, to represent the set of document identifiers containing the words appearing on P. Thus, we have, for example, $\delta_{v_{14}} = \delta_{[10,\ 13]} = \{6, 7, 8\}$. Concerning the decomposition of inverted lists, the following two lemmas can be easily proved.

Lemma 1 Let T be a trie constructed over a set of word sequences (sorted by the appearance frequency) over $\boldsymbol{W}$. Then, we have $\sum_{v \in T} \delta_v = \sum_{w \in W} \delta_w$.

Proof. Let $v_1, \ldots, v_{l_w}$ be all the nodes labeled with a word w in T. Then $\delta_w = \sum_{i=1}^{l_w} \delta_{v_i}$. Since in T no node is labeled with more than one word, we have $\sum_{w \in W} \delta_w = \sum_{w \in W} \sum_{i=1}^{l_w} \delta_{v_i} = \sum_{v \in T} \delta_v$. □

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 δ_u and δ_v are disjoint, i.e., $\delta_u \cap \delta_v = \Phi$.

Proof. It is easy to prove. □

Proposition 1 Assume that $v_1, v_2, \ldots, v_j$ be all the nodes labeled with the same word w in T. Then, δ_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, δ_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 1, $\delta_{v_1} \cup \delta_{v_2} \cup \ldots \cup \delta_{v_j}$ is equal to $\delta_{v_1} \uplus \delta_{v_2} \uplus \ldots \uplus \delta_{v_j}$. □

As an example, see the nodes v_1 and v_7 in Fig. 2. Both are labeled with word *d*. So the inverted list of *d* is $\delta_{v_1} \uplus \delta_{v_7}$ = {1, 2} ⊎ {4, 5, 6, 7, 8} = {1, 2, 4, 5, 6, 7, 8}.

3. BASIC UERY EVALUATION

Based on the new index structure, we design our basic algorithms.

We first consider a query containing only two words $w \wedge w'$ 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 $I \supset I'$. Therefore, such a query can be evaluated by running a process, denoted as *conj*(L_w, $L_{w'}$), to find all those intervals in $L_{w'}$ with each being contained in some interval in L_w, stored in a new sequence L.

1. Let $L_w = I_w^1, I_w^2, \ldots, I_w^k$. Let $L_{w'} = I_{w'}^1, I_{w'}^2, \ldots, I_{w'}^{k'}$. $L \leftarrow \phi$.
2. 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 checkings:
   i) If $I_w^p \supset I_{w'}^q$, append $I_{w'}^q$ to the end of L. Move to $I_{w'}^{q+1}$ if $q < k'$ (then, in a next step, we will check I_w^p against $I_{w'}^{q+1}$).
   ii) If $I_w^p[1] > I_{w'}^q[2]$, move to $I_{w'}^{q+1}$ if $q < k'$. If $q = k'$, stop.
   iii) 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 $I_{w'}^q$). If $p = k$, stop. □

Assume that the result is $L = I_1, I_2, \ldots, I_l$ $(0 \le l \le k')$. Then, for each $1 \le j \le l$, there exists an interval $I \in L_w$ such that $I_j \subset I$, and we can return $\delta_{I_1} \uplus \ldots \uplus \delta_{I_k}$ as the answer. In Fig. 4, we illustrate the working process on L_p and L_b shown in Fig. 1(b).

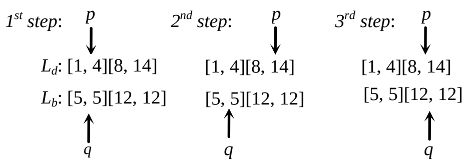


Fig. 4: Illustration for $conj(L_w, L_{w'})$

In Fig. 4, we first notice that L_d = [1, 4][8, 14] and L_b = [5, 5][12, 12]. In the 1st step, we will check L_d^1 = [1, 4] against L_b^1 = [5, 6]. Since $L_d^1[2] = 4 < L_b^1[1] = 5$, we will check L_d^2 = [8, 14] against L_b^1 in a next step, and find $L_b^1[2] = 5 < L_d^2[1] = 8$. So we will have to do the third step, in which we will check L_d^2 against L_b^2 = [12, 12]. Since $L_d^2 \supset L_b^2$, we get to know that *d* and *b* are in the same document.

Lemma 3 Let $L = I_1, \ldots, I_k$ be the result of $conj(L_w, L_{w'})$. Then, for each I_j ($1 \le j \le k$), there must be an interval $I \in L_w$ such that $I \supset I_j$. For any interval $I' \in L_{w'}$ but $\notin L$, it definitely does not belong to any interval in L_w.

Proof. It is easy to prove. □

Since in this process, each interval in both L_w and $L_{w'}$ is accessed only once, the time complexity of this process is bounded by $O(|L_w| + |L_{w'}|)$. In addition, the above approach can be easily extended to evaluate general queries of the form $Q = w_1 \wedge w_2 \wedge \ldots \wedge w_l$ with $w_1 \prec w_2 \prec \ldots \prec w_l$ and $l \ge 1$ based on the transitivity of intervals: $I \supseteq I' \supseteq I'' \rightarrow I \supseteq I''$.

What we need to do is to repeatedly apply *conj*( ) to the corresponding interval sequences associated with the query words one by one. The following is a formal description of the process.

ALGORITHM $conEvaluation(Q)$

begin

1. let $|Q| = l$; assume that $Q[1] \prec Q[2] \prec \ldots \prec Q[l]$;
2. $L := Q[1]$;
3. **for** (j = 2 to l) **do**
4. { $L \leftarrow conj(L, L_{Q[j]})$; }
5. let $L = I_1, \ldots, I_k$;
6. return $\delta_{I_1} \uplus \ldots \uplus \delta_{I_k}$.

end

It is easy to see that the time complexity of the algorithm is bounded by $O(\sum_{w \in Q} |L_w|)$.

Proposition 2 Let $Q = w_1 \wedge w_2 \wedge \ldots \wedge w_l$ with $w_1 \prec w_2 \prec \ldots \prec w_l$ and $l \ge 1$. The answer produced by algorithm *conEvaluation*(*Q*) is correct.

Proof. Let $L = I_1, \ldots, I_k$ be the interval sequence produced by the main **for**-loop (line 3 – 4). Then, according to Lemma 3, for each I_j ($1 \le j \le k$), there must exist an interval sequence $\iota_1, \iota_2, \ldots, \iota_{l-1}$ such that $\iota_i \in L_{w_i}$ ($1 \le i \le l - 1$) and $\iota_1 \supset \iota_2 \supset \ldots \supset \iota_{l-1} \supset I_i$. Next, according to Proposition 1, we know that $\delta_{I_1} \uplus \ldots \uplus \delta_{I_k}$ must be the correct answer. □

Example 1 Consider Fig. 2 and 3. Let $Q = d \wedge f \wedge a$. Then, in the first iteration, we will get $L = conj(L_d, L_f)$ = [1, 2][10, 13]. In the second iteration, we will get $L = conj(L, L_p)$ = [1, 1][11, 11]. The results is then $R = \delta_{[1,1]} \uplus \delta_{[11,11]} = \{1\} \uplus \{7\} = \{1, 7\}$. □

4. Improvements

In this section, we discuss a new algorithm to improve the naïve method shown in the previous subsection. The main idea is to use lowest common ancestors (*LCAs* for short) of nodes (in *T*) to control a binary search process. First, in 4.1, we discuss the binary search of an L_w. Then, we show how to use *LCAs* to speed up such a search in 4.2.

4.1 Set intersection based on binary search

Each interval sequence is sorted. So we can do the conjunction of interval sequences based on binary search.

Let $L_o = I_o^1, I_o^2, \ldots, I_o^m$ and $L_w = I_w^1, I_w^2, \ldots, I_w^n$ be two interval sequences with $w \prec o$. Then, $m = |L_o| \le n = |L_w|$.

By using the binary search technique, we need to work from the end to the start of L_w to incorporate the *LCAs* into the process. To this end, we design an algorithm different from $conj(L_o, L_w)$, called *conjB*( ), which can be mostly easily described recursively. When $m = 0$, there is no conjunction to be done and the result is ϕ. Otherwise, we will first check I_o^m against L_w. As with [46], let $l = \left\lfloor \lg \frac{n}{m} \right\rfloor$. Then, 2^l is the largest power of two not exceeding $\frac{n}{m}$. Let $t = n - 2^l + 1$. Compare I_o^m and I_w^t.

1. If $I_o^m[1] > I_w^t[2]$, we should look for the intervals (in L_w) covered by I_o^m somewhere to the right of I_w^t. By using the traditional binary search, we try to find an interval *I* covered by I_o^m with *l* more comparisons. Around *I*, we will continually (by a simple linear search) find the left-most interval *x* in L_w, which can be covered by I_o^m; and then with *l* more comparisons, we will find the right-most interval *y* covered by I_o^m, in a similar way. Obviously, all the intervals between *x* and *y*, including *x* and *y*, can be covered by I_o^m. (See Fig. 5(a).) This information allows us to reduce the problem to the situation illustrated in Fig. 5(b). To complete the whole operation, it is sufficient to apply the above process to L_o' and L_w', where $L_o' = I_o^1, \ldots, I_o^{m-1}$ and $L_w' = I_w^1, \ldots, I_w^{x-1}$.
2. If, on the other hand, $I_o^m[2] < I_w^t[1]$, we should check the intervals to the left of I_w^t, and the problem immediately reduces to the checking of $L_o' = L_o$ against $L_w' = L_w[1 .. t -$

1]. We can complete the operation by applying the above process to L_o' and L_w'.

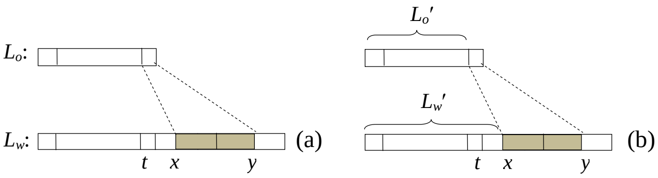


Fig. 5: First comparison during an interval intersection

However, L_o' may become larger than L_w'. So in the recursive call to *conjB*( ), the roles of L_o' and L_w' may be reversed, by which we will check each interval I in L_w' against L_o' to find an interval I' in L_w' such that $I' \supset$ the last interval in L_o'. See Fig. 6 for illustration. Assume that that the last interval I_w^{x-1} in L_w' is covered by an interval I_o^j $(1 \leq j \leq m - 2)$ in L_o'. Then, by the next recursive call, we will check $L_w'' = I_w^1, \ldots, I_w^{x-2}$ and $L_u'' = I_o^1, \ldots, I_o^{j-2}$.

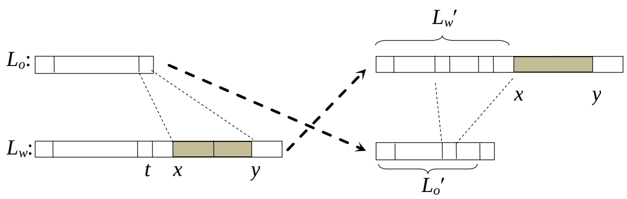


Fig. 6: Illustration for interchanging rolls of L_w' and L_u'

3. If $I_o^m \supset I_w^t$, we will check linearly I_w^{t-1}, I_w^{t-2}, … until we meet a first interval x' which is to the left of I_w^t and not covered by I_o^m. Then, check I_w^{t+1}, I_w^{t+2}, … until a first interval y' which is to the right of I_w^t and not covered by I_o^m. All the encountered nodes, except x' and y', must be covered by I_o^m. This reduces the problem to a checking of $L_o' = L_o[1 .. m - 1]$ against $L_w' = L_w[1 .. x']$.
4. If $I_o^m \subset I_w^t$ (we may have this case due to the roll interchange), we add I_o^m to the result and the problem reduces to a checking $L_o' = L_o[1 .. m - 1]$ against $L_w' = L_w[1 .. t]$.

According to the above discussion, we give the following recursive algorithm, which takes three inputs: L_o, L_w, b with $|L_o| \leq |L_w|$, where b is a Boolean value used to indicate how I_o^m is checked against L_w. If $o \prec w$, $b = 0$. Otherwise ($w \prec o$), $b = 1$. In addition, in the Algorithm a global variable R is used to store the result.

ALGORITHM *conjB*(L_o, L_w, b)

begin

1. $m \leftarrow |L_o|$; $n \leftarrow |L_w|$;
2. **if** $m = 0$ **then** return;
3. $l \leftarrow \left\lfloor \lg \frac{n}{m} \right\rfloor$; $t \leftarrow n - 2^l + 1$; $I \leftarrow I_o^m$;
4. **if** $I[2] < I_w^t[1]$ **then** $\{L_o' \leftarrow L_o; L_w' \leftarrow L_w[1 .. t - 1];\}$
5. **if** $I[1] > I_w^t[2]$
6. **then if** $b = 1$ **then** $z \leftarrow$ *binaryS-1*(I, $L_w[t + 1 .. n]$)
7. **if** $z = 0$ **then** $\{L_o' \leftarrow L_o[1..m-1]; L_w' \leftarrow L_w;\}$
8. **else** $R := R \cup \{I\}$;
9. $L_o' \leftarrow L_o[1 .. m - 1]$;
10. $L_w' \leftarrow L_w[1 .. t + z - 1]$;
11. **else** $<x, y> \leftarrow$ *binaryS-2*(I, $L_w[t + 1 .. n]$)
12. **if** $x = 0$ **then** $\{L_o' \leftarrow L_o[1 .. m - 1]; L_w' \leftarrow L_w;\}$
13. **else** $R \leftarrow R \cup$ {all interval between x and y, including x and y};
14. $L_o' \leftarrow L_o[1 .. m - 1]$; $L_w' \leftarrow L_w[1 .. x - 1]$;
15. **if** $I \supset I_w^t$ **then** $<x, y> \leftarrow$ *linearSearch*(I, L_w, I_w^t)
16. $L_o' \leftarrow L_o[1 .. m - 1]$; $L_w' \leftarrow L_w[1 .. x - 1]$;
17. $R \leftarrow R \cup$ {all interval between x and y, including x and y};
18. **if** $I \subset I_w^t$ **then** $R := R \cup \{I\}$;
19. $L_o' \leftarrow L_o[1 .. m - 1]$; $L_w' \leftarrow L_w[1 .. t]$;
20. **if** $|L_o'| \leq |L_w'|$ **then** *conjB*(L_o', L_w', b)
21. **else** *conjB*(L_w', L_o', $\bar{b}$);

end

The above algorithm can be divided into two parts. The first part consists of lines 1 – 10; and the second part lines 20 – 21. In the first part, we will check the first interval I_o^m in L_o against L_w. According to the above discussion, four cases are distinguished: $I_o^m[2] < I_w^t[1]$ (line 4), $I_o^m[1] > I_w^t[2]$ (lines 4 – 14), $I_o^m[1] \supset I_w^t$ (lines 15 – 17), and $I_o^m[1] \subset I_w^t$ (18 – 19). Special attention should be paid to the use of b, which indicates whether we check I_o^m to find a covering interval in L_w (by calling *binaryS-1*( )) or to find all those intervals that can be covered by I_o^m (by calling *binaryS-2*( ))). In the second part (lines 20 – 21), we make a recursive call to check L_o' and L_w', which are determined respectively from L_o and L_w during the execution of the first part. If $|L_o'| \leq |L_w'|$, we simply call *conjB*(L_o', L_w', b) (see line 14.) Otherwise, the rolls of L_o and L_w should be interchanged and we will call *conjB*(L_w', L_o', $\bar{b}$), where $\bar{b}$ represents the negation of b (see line 21.)

It *binaryS-1*(I, L), we will find, by the binary search, an interval I_z in L which covers I. If $z = 0$, it shows that such an interval does not exist.

FUNCTION *binaryS-1*(I, L)

begin

- $z \leftarrow 0$;
- binary search of L to find an interval z, which covers I;
- return z;

end

In *binaryS-2*(I, L), we will first find a pair $<x, y>$ such that I_x is the left-most interval in L, which can be covered by I;

and I_y the right-most interval covered by I. Then, x = 0 indicates that no interval in L is covered by I.

FUNCTION *binaryS-2*(I, L)

---

begin
1. $x \leftarrow 0$; $y \leftarrow 0$;
2. binary search of L to find an interval I_z which is covered by I;
3. return *linearSearch*(I, L, I_z);

end

In *linearSearch*(I, L, I'), we will find a pair <x, y> such that I_x, I_{x+1}, …, I', …, I_{y-1}, I_y are all the intervals that can be covered by I.

FUNCTION *linearSearch*(I, L, I')

---

begin
1. Let I' be I_z;
2. Search I_{z-1}, I_{z-2}, … until I_x such that I_x is covered by I, but I_{x-1} not;
3. Search I_{z+1}, I_{z+2}, … until I_y such that I_y is covered by I, but I_{y+1} not;
2. return <x, y>;

end

Example 2 Consider L_d = [1, 4][8, 14] and L_a = [1, 1][3, 3][5, 6][8, 8][11, 11][16, 16]. By calling *conjB*(L_f, L_a, *false*), the following operations will be conducted:

Step 1: checking L_d[1] = [1, 4] against L_a. $l = \left\lfloor \lg\frac{6}{2} \right\rfloor = 1$, $t = 2^l = 2$, L_a[2] = [3, 3]. Since [1, 4] ⊃ [3, 3], we will call *linearSearch*( ) to find x = 1 and y = 2.

Step 2: checking L_d[2] = [8, 14] against L_a[3 .. 6]. $l = \left\lfloor \lg\frac{4}{1} \right\rfloor = 2$, $t = 2^l = 4$, L_a[4] = [16, 16]. Since [8, 14] is to the left of [16, 16], we will make a binary search of L_a[3 .. 5], by which we will find x = 4 and y = 5. □

4.2 Search control by using *LCAs*

The method discussed in 4.1 can be significantly improved by using *LCAs*. Given a word w, denote by V_w all the nodes labeled with w. All the *LCAs* of the nodes in V_w (in T), denoted as V_w', can be efficiently recognized using a way to be discussed in Section 6. For example, for the set of nodes labeled with word a: $V_a = \{v_{10}, v_5, v_6, v_{12}, v_{18}, v_{15}\}$, we can find another set of nodes: $V_a' = \{v_1, v_7, v_2, v_0\}$ with v_1 being *LCA* of $\{v_{10}, v_5\}$, v_7 being *LCA* of $\{v_{12}, v_{18}\}$, v_2 being *LCA* of $\{v_6, v_{12}, v_{18}, v_{15}\}$, and v_0 being *LCA* of $\{v_{10}, v_5, v_6, v_{12}, v_{18}, v_{15}\}$. Now we construct a tree structure, called an *LCA-tree* and denoted as T_w, which contains all the nodes in $V_w \cup V_w'$. In T_w, there is arc from v_1 to v_2 iff there exists a path P from v_1 to v_2 in T and P does not pass any other node in $V_w \cup V_w'$. In Fig. 7(a), we show T_a for illustration.

Replacing each node in T_w with the corresponding interval, we get another tree, denoted as $T_w^{\sim}$, in which each internal node v must be an interval that is the smallest interval covering all the intervals represented by the leaf nodes in $T_w^{\sim}[v]$ (the subtree rooted at v in $T_w^{\sim}$). See $T_a^{\sim}$ shown in Fig. 7(b) for illustration. From this, we can see that [1, 4] is the smallest interval covering [1, 1] and [3, 3]; [8, 14] is the smallest interval covering [8, 8] and [11, 11]; and [8, 19] is the smallest interval covering [8, 8], [11, 11] and [16, 16]. Finally, [1, 20] is the smallest interval covering all the intervals in L_a: [1, 1], [3, 3], [5, 6], [8, 8], [11, 11], [16, 16].

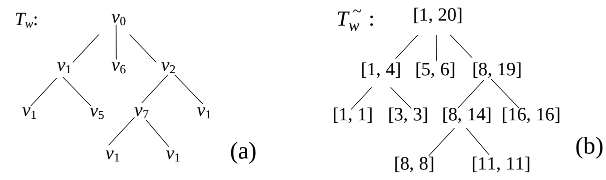


Fig. 7: Illustration for T_w and $T_w^{\sim}$

Here, our intention is to associate interval I_w^j in L_w with a second interval γ_j, which is the parent of I_w^j in $T_w^{\sim}$, and two links, denoted as l_j and r_j, respectively pointing to two intervals in L_w, which are respectively the left-most and right-most leaf nodes in $T_w^{\sim}[\gamma_j]$. Fig. 8 helps for illustration.

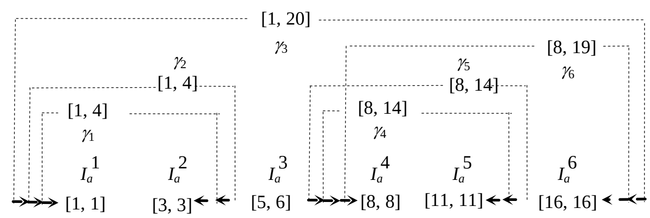


Fig. 8: Illustration for links associated with intervals in L_w

In Fig. 8, I_a^3 = [5, 6] is associated with an *LCA* interval γ_3 = [8, 14], which is the parent of I_a^3 in the corresponding $T_a^{\sim}$ shown in Fig. 7(b). In addition, l_3 is a link pointing to I_a^1 and r_3 is a link pointing to I_a^6. They are respectively the laft-most interval and the right-most interval covered by γ_3. In the same way, we can check all the other intervals and links shown in Fig. 8.

In addition, we will keep a sequence Γ_w containing all the *LCA* intervals in the post-order of $T_w^{\sim}$. For example, $\Gamma_a = \gamma_1\gamma_4\gamma_6\gamma_3$ = [1, 4][8, 14][8, 19][1, 20]. With such intervals and links, the binary search of L_w against a certain interval (in L_o) can be done much more efficiently by skipping over useless checkings. Concretely, the checking of I_o^m against L_w will be done as follows.

1. If $I_o^m[1] > I_w^t[2]$, compare I_o^m and γ_t. If $I_o^m \not\subset \gamma_t$, explore $L_w[r_t + 1 .. n]$ by the binary search. Otherwise, explore $L_w[t + 1 .. r_t]$.
2. If $I_o^m[2] < I_w^t[1]$, compare I_o^m and γ_t. If $I_o^m \not\subset \gamma_t$, explore $L_w[1 .. l_t - 1]$. Otherwise ($I_o^m \subset \gamma_t$), explore $L_w[l_t .. t - 1]$.
3. If $I_o^m \supset I_w^t$, compare I_o^m and γ_t. If $\gamma_t \supset I_o^m$, I_w^t must be the

unique interval which can be covered by I_o^m . Therefore, I_w^t is the result and the search stops. The problem reduces to a checking of $L_o[1 .. m - 1]$ against $L_w[1 .. t - 1]$ with $\Gamma_w[1 .. k]$ to be used for control, where k is the position prior to γ_t in Γ_u. If $\gamma_t = I_o^m$, we will return all those intervals between l_t and r_t, including both l_t and r_t. The search also stops and the problem reduces to a checking of $L_o[1 .. m - 1]$ against $L_w[1 .. l_t - 1]$ with $\Gamma_w[1 .. k]$. If $\gamma_t \subset I_o^m$, we will search part of Γ_w to the right of γ_t to find the right-most interval γ_f covered by I_o^m . Then, return all the intervals between l_f and r_f, including l_f and r_f, which allows us to reduce the problem to check $L_o[1 .. m - 1]$ against $L_w[1 .. l_f - 1]$ with $\Gamma_w[1 .. g]$, where g is the position prior to γ_f in Γ_w.

4. If $I_o^m \subset I_w^t$, the above data structure cannot be utilized to speed up the search. Thus, this case will be handled in the same way as described for *conjB*( ).

Example 3 To see how the *LCA*s can be used to skip over useless checkings, we check several single intervals against L_a in Fig. 8 to show the working process.

1. Assume that $I = [5, 7]$ is compared with $I_5 = [11, 11]$ in L_a. Since [5, 7] is to the left of [11, 11], we will compare [5, 7] with $\gamma_5 = [8, 14]$ and $[5, 7] \not\subset [8, 14]$. So we will check [5, 7] against $L_a[1 .. l_5 - 1] = L_a[1 .. 3]$ in a next step, instead of the sequence containing all the intervals to the left of I_5.
2. Assume that $I = [10, 13]$ is compared with $I_4 = [8, 8]$ in L_a. Since [10, 13] is to the right of [8, 8], [10, 13] and $\gamma_4 = [8, 14]$ will be compared and $[10, 13] \subset [8, 14]$. So, in the next step, we will check [10, 13] against $L_a[4 + 1 .. r_5] = L_a[5 .. 5]$, not the sequence containing all the intervals to the right of I_4.
3. Assume that $I = [10, 13]$ is compared with $I_5 = [11, 11]$ in L_a. We have $[10, 13] \supset [11, 11]$. However, $[10, 13] \subset \gamma_5 = [8, 14]$. It shows that [11, 11] is the only interval in *La*, which can be covered by [10, 13]. No further search is necessary.
4. Assume that $I = [8, 14]$ is compared with $I_4 = [8, 8]$ in L_a. We have $[8, 14] \supset [8, 8]$. But we also have $[8, 14] = \gamma_4$. Then, we know immediately that only the intervals in $L_a[l_4 .. r_4] = L_a[4 .. 5]$ can be covered by [8, 14]. □

By Example 3, we can clearly see that *LCAs* are quite useful to speed up the operation. However, all of them should be efficiently recognized. We will discuss this in the next Section.

5. Conclusion

In this paper, a new index structure is discussed. It associates each word w with a sequence of intervals, which partition the inverted list $\delta(w)$ into a set of disjoint subsets, and transform the evaluation of conjunctive queries to a series of checkings of interval containment. Especially, the intervals can be organized into a compact interval graph, which enables us to skip over any useless checking of interval containment. On average, to evaluate a two-word query, only O(log$n$) time is required, where n is the number of documents. This is much more efficient than any existing method for set intersection. Also, how to maintain such an index is described in great detail. Although the index is of a more complicated structure, the cost of maintaining it in the cases of addition and deletion of documents is (theoretically) comparable to the inverted file. Extensive experiments have been conducted, which show that our method outperformances the inverted file and the signature tree by an order of magnitude or more.

REFERENCES

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

[2] J. Barbay, A. López-Ortiz, T. Lu, A. Salinger: An experimental investigation of set intersection algorithms for text searching, *ACM Journal of Experimental Algorithmics 14*: (2009).

[3] P. Bille, A. Pagh, and R. Pagh. Fast-Evaluation of Union-Intersection Expression. In ISAAC, pp. 739-750, 2007.

[4] G.E. Blelloch and M. Reid-Miller. Fast Set Operations using Treaps. In ACM SPAA, pp. 16-26, 1998.

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

[6] Y. Chen: Building Signature Trees into OODBs, *Journal of Information Science and Engineering*, 20, 275-304 (2004).

[7] 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.

[8] Y. Chen and Y.B. Chen: Decomposing DAGs into spanning trees: A new way to compress transitive closures, in *Proc. 27th Int. Conf. on Data Engineering (ICDE 2011)*, IEEE, April 2011, pp. 1007-1018.

[9] K.D. Demaine, A. López-Ortiz, and J.I. Munro: Adaptive set intersections, unions, and differences, in *Proc. 11th ACM-SIAM Symposium on Discrete Algorithms*, Philadelphia, 743-752, 2000.

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

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

[12] C. Faloutsos and R. Chan: 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, Aug. 1988.

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

[14] J. Zobel and A. Moffat: Inverted Files for Text Search Engines, *ACM Computing Surveys*, 38(2):1-56, July 2006.

[15] J. Zobel, A. Moffat, and K. Ramamohanarao: Inverted Files Versus Signature Files for Text Indexing, in *ACM Trans. Database Syst.*, 1998, pp.453-490.