On the Intersection of Inverted Lists

Yangjun Chen\textsuperscript{1} and Weixin Shen\textsuperscript{2}

Dept. Applied Computer Science, University of Winnipeg, Canada
\textsuperscript{1}y.chen@uwinnipeg.ca, \textsuperscript{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 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:

\[ w_1 \land w_2 \land \ldots \land w_k. \]

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.

- 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 \( \delta_x \) and \( \delta_y \) 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 \( O(|\delta_x| \log \frac{|\delta_y|}{|\delta_x|}) \) time. In contrast, our algorithm needs \( O(L_x \cdot \log \frac{x}{L_y}) \) time, where \( L_x \) and \( L_y \) are the interval sequences created for \( L_x \) and \( L_y \), respectively; and \( x \) 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 \( |L_x| \leq |L_y| \leq |\delta| \) and \( x \leq |\delta| \). This time complexity is significantly better than the traditional methods since an interval sequence can be much shorter than its corresponding inverted list. 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), may the sizes of their corresponding interval sequences be near their sizes. In fact, we only care about the cases of very long inverted lists. When all the inverted lists involved in an operation are short, all the methods work fast.

Finally, we indicate that 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 $D = \{D_1, ..., D_n\}$ be a set of documents. Let $W_i = \{w_{i1}, ..., w_{im}\}$ (i = 1, ..., n) be all of the words appearing in $D_i$ to be indexed. Denote $W = \bigcup_{i=1}^n W_i$, called the vocabulary. For each word $w \in W$, 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 $\mathcal{S} = w_1, w_2, ..., w_m$ ($m = |W|$) such that for any two words $w$ and $w'$ if the frequency of $w$ is higher than $w'$ then $w$ appears before $w'$ in $\mathcal{S}$, denoted as $w \prec w'$. Then, each document can be represented as a subsequence of $\mathcal{S}$, and over all these subsequences a trie structure can be established as illustrated in Fig. 1.

Documents and word sequences:

<table>
<thead>
<tr>
<th>Document</th>
<th>Words</th>
<th>Words</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a, f, d</td>
<td>d, f, a</td>
</tr>
<tr>
<td>2</td>
<td>a, d</td>
<td>a</td>
</tr>
<tr>
<td>3</td>
<td>a, e, d</td>
<td>e, a, d</td>
</tr>
<tr>
<td>4</td>
<td>f, b, a</td>
<td>f, a, b</td>
</tr>
<tr>
<td>5</td>
<td>c, d, e</td>
<td>e, d, e</td>
</tr>
<tr>
<td>6</td>
<td>d, f, e, c</td>
<td>e, d, f, c</td>
</tr>
<tr>
<td>7</td>
<td>f, d, e, a</td>
<td>d, f, a</td>
</tr>
<tr>
<td>8</td>
<td>f, d, e, b</td>
<td>d, f, b</td>
</tr>
<tr>
<td>9</td>
<td>c, c</td>
<td>e</td>
</tr>
<tr>
<td>10</td>
<td>e, f</td>
<td>e, f, a</td>
</tr>
<tr>
<td>11</td>
<td>f, e, c</td>
<td>e, f, c</td>
</tr>
</tbody>
</table>

Fig. 1(a): 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 $\varepsilon$ 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 $v_{11}$ 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 $I_v = [\alpha, \beta]$, where $\beta$ 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, $\alpha_v$ 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 $I_v \subseteq I_u$, 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, $v_3 \not\Rightarrow v_{19}$, since $v_3 = [8, 19]$, $I_{v_1} = [12, 12]$, and $[12, 12] \subseteq [8, 19]$; but $v_2 \not\Rightarrow v_{18}$, since $I_{v_2} = [5, 17]$, $I_{v_{19}} = [11, 11]$, and $[11, 11] \not\subseteq [5, 17]$.

Let $I = [\alpha, \beta]$ be an interval. We will refer to $\alpha$ and $\beta$ 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 $v_{12}$ and $v_{19}$ 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 $v_7$ each against both $v_{11}$ 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: $L_w = l_w^1, l_w^2, ..., l_w^k$, where $k$ is the number of all those nodes labeled with $w$ and each $l_w^i = [I_w^i[1], I_w^i[2]] \,(1 \leq i \leq k)$ 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 $1 \leq i < j \leq k$ we have $I_w^i[1] < I_w^j[1]$, 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.

From this figure, we can see that for any two intervals $I$ and $I'$ in $L_w$ we must have $I \subset I'$, and $I' \subset I$ 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 \( L[j] \) to refer to the \( j \)th interval in \( L \), and \( L[i..j] \) to the segment from the \( i \)th to the \( j \)th 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: \( v_0 \rightarrow v_1 \rightarrow v_7 \rightarrow v_{14} \) in the trie shown in Fig. 1(c) are words: \( a, e, d, f \), respectively, 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 \( D_6, D_7, D_8 \) 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](image)

Let \( v \) be the ending node of a prefix \( P \), labeled with \( I \). We will use \( \delta_v \) interchangeably \( \delta \), to represent the set of document identifiers containing the words appearing on \( P \). Thus, we have, for example, \( \delta_v = \delta_{[10,13]} = \{6,7,8\} \). 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 \( W \). Then, we have \( \sum_{v \in T} \delta_v = \sum_{w \in W} \delta_w \).

**Proof.** Let \( v_1, \ldots, v_j \) be all the nodes labeled with a word \( w \) in \( T \). Then \( \delta_w = \sum_{i=1}^{j} \delta_{v_i} \). Since in \( T \) no node is labeled with more than one word, we have \( \sum_{w \in W} \delta_w = \sum_{i=1}^{j} \sum_{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 \( \delta_u \) and \( \delta_v \) are disjoint, i.e., \( \delta_u \cap \delta_v = \emptyset \).

**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, \( \delta_{v_1} \) the inverted list of \( w \) (i.e., the list of all the document identifiers containing \( w \)) is equal to \( \delta_{v_1} \cup \delta_{v_2} \cup \ldots \cup \delta_{v_j} \), where \( \cup \) represents disjoint union over disjoint sets that have no elements in common.

**Proof.** Obviously, \( \delta_{v_k} \) 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 \( \delta_{v_1} \cup \delta_{v_2} \cup \ldots \cup \delta_{v_j} \).

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 \( \delta_{v_1} \cup \delta_{v_7} = \{1, 2\} \cup \{3, 5, 6, 7, 8\} = \{1, 2, 3, 5, 6, 7, 8\} \).

### 3. Basic Query Evaluation

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

We first consider a query containing only two words \( w \land w' \) with \( w < w' \). 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 \( I \supseteq I' \). Therefore, such a query can be evaluated by running a process, denoted as \( \text{conj}(L_w, L_{w'}) \), 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 \( L_{w'} = I_{w'}^1, I_{w'}^2, \ldots, I_{w'}^\ell \). \( L \leftarrow \phi \).
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 \( I_{w}^p \supseteq I_{w'}^q \), append \( I_{w'}^q \) to the end of \( L \). Move to \( I_{w}^{p+1} \) if \( q < k' \) (then, in a next step, we will check \( I_{w}^p \) against \( I_{w'}^{q+1} \)). If \( q = k' \), stop.
   ii) If \( I_{w}^p [1] > I_{w'}^q [2] \), move to \( I_{w}^{p+1} \) if \( q < k' \). If \( q = k' \), stop.
   iii) If \( I_{w}^p [2] < I_{w'}^q [1] \), move to \( I_{w'}^{q+1} \) if \( p < k' \) (then, in a next step, we will check \( I_{w}^{p+1} \) against \( I_{w'}^{q+1} \)). If \( p = k' \), stop.

Assume that the result is \( L = I_1, I_2, \ldots, I_l \) (\( 0 \leq l \leq k' \)). Then, for each \( 1 \leq j \leq l \), there exists an interval \( I_j \in L_w \) such that \( I_j \subseteq I \), and we can return \( \delta_{I_1} \cup \ldots \cup \delta_{I_l} \) as the answer. In Fig. 4, we illustrate the working process on \( L_d \) and \( L_b \) shown in Fig. 2.
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 \( L_d' = [1, 4] \) against \( L_b = [5, 6] \). Since \( L_d' [2] = 4 < L_b [1] = 5 \), we will check \( L_d'' = [8, 14] \) against \( L_b = [5, 6] \) in a next step, and find \( L_b [2] = 5 < L_d'' [1] = 8 \). So we will have to do the third step, in which we will check \( L_d'' \) against \( L_b'' = [12, 12] \). Since \( L_d'' \supseteq L_b'' \), we get to know that \( d \) and \( b \) are in the same document.

**Lemma 3** Let \( L = L_1, \ldots, L_k \) be the result of \( \text{conj}(L_w, L_w') \). Then, for each \( I_j (1 \leq j \leq k) \), there must be an interval \( I \subseteq L_w \) such that \( I \supseteq I_j \). For any interval \( I' \in L_w' \) but \( I \), 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 \( Q = w_1 \land w_2 \land \ldots \land w_l \) with \( w_1 \land w_2 \ldots \land w_l \) with \( w_1 \land w_2 \ldots \land w_l \), and \( I \supseteq I \) or \( I \supseteq I \\

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) of short 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 \( o < w \). Then, \( m = |L_o| \leq n = |L_w| \).

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 \( \text{conj}(L_o, L_w) \), called \( \text{conjB}( ) \), which can be mostly easily described recursively. When \( m = 0 \), there is no conjunction to be done and the result is \( \emptyset \). Otherwise, we will first check \( I_o ^m \) against \( L_w \). As with \( [46] \), let \( l = \lfloor \log m \rfloor \). 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 \( I \) more comparisons. Around \( I \), we will continually (by a simple linear search) find the leftmost interval \( x \) in \( L_w \), which can be covered by \( I_o ^m \); and then with \( I \) more comparisons, we will find the rightmost 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 \( L_o ^t \) and \( L_w ^t \), where \( L_o ^t = I_o ^1, \ldots, I_o ^{m-1} \) and \( L_w ^t = I_w ^1, \ldots, I_w ^{t-1} \).

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 \( L_o ^t = L_o \) against \( L_w ^t = L_w [1 \ldots t - 1] \). We can complete the operation by applying the above process to \( L_o ^t \) and \( L_w ^t \).
However, 

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

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

**Algorithm conjB(L_o, L_n, b)**

begin
1. \( m \leftarrow |L_o|; n \leftarrow |L_n|; \)
2. if \( m = 0 \) then return;
3. \( I \leftarrow [\frac{n}{m}] \); \( t \leftarrow n - 2I + 1 ; I \leftarrow I_w^m ; \)
4. if \( I[2] < I_o^1 [1] \) then \( \{ L_o' \leftarrow L_o; L_n' \leftarrow L_n[1 .. t - 1]\}; \)
5. if \( I[1] > I_w^t [2] \)
6. then if \( b = 1 \) then \( z \leftarrow \text{binaryS-1}(I, L_o[t + 1 .. n]) \)
7. if \( z = 0 \) then \( \{ L_o' \leftarrow L_o[1 .. m - 1]; L_n' \leftarrow L_n; \}
8. else \( R := R \cup \{I\}; \)
9. \( L_o' \leftarrow L_o[1 .. m - 1]; \)
10. \( L_n' \leftarrow L_n[t + 1 .. m - 1]; \)
11. else \( <x, y> \leftarrow \text{binaryS-2}(I, L_o, L_n') \)
12. if \( x = 0 \) then \( \{ L_o' \leftarrow L_o[1 .. m - 1]; L_n' \leftarrow L_n \} \)
13. else \( R \leftarrow R \cup \{all interval between x and y, including x and y\}; \)
14. \( L_o' \leftarrow L_o[1 .. m - 1]; L_n' \leftarrow L_n[1 .. x - 1]; \)
15. if \( I \supset I_w^t \) then \( <x, y> \leftarrow \text{linearSearch}(I, L_o, I_w^t) \)
16. \( R \leftarrow R \cup \{all interval between x and y, including x and y\}; \)
17. \( L_o' \leftarrow L_o[1 .. m - 1]; L_n' \leftarrow L_n[1 .. x - 1]; \)
18. if \( I \subset I_w^t \) then \( R := R \cup \{I\}; \)
19. \( L_o' \leftarrow L_o[1 .. m - 1]; L_n' \leftarrow L_n[1 .. t]; \)
20. if \( |L_o'| \leq |L_n'| \) then \( \text{conjB}(L_o', L_n', b) \)
21. else \( \text{conjB}(L_o', L_n', \bar{b}) \);
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 last interval \( I_w^m \) in \( L_o \) against \( L_n' \). According to the above discussion, four cases are distinguished: \( I_w^m [2] < I_o^1 [1] \) (line 4), \( I_w^m [1] > I_o^1 [2] \) (lines 4 – 14), \( I_o^m [1] \supset I_o^1 [1] \) (lines 15 – 17), and \( I_o^m [1] \subset I_o^1 [18 – 19] \). Special attention should be paid to the use of \( b \), which indicates whether we check \( I_w^m \) to find a covering interval in \( L_n' \) (by calling \( \text{binaryS-I}(\cdot) \)) or to find all those intervals that can be covered by \( I_w^m \) (by calling \( \text{binaryS-2}(\cdot) \)).

In the second part (lines 20 – 21), we make a recursive call to check \( L_o' \) and \( L_n' \), which are determined respectively from \( L_o \) and \( L_n' \) during the execution of the first part. If \( |L_o'| \leq |L_n'| \), we simply call \( \text{conjB}(L_o', L_n', b) \) (see line 14.) Otherwise, the roles of \( L_o \) and \( L_n \) should be interchanged and we will call \( \text{conjB}(L_w', L_o', \bar{b}) \), where \( \bar{b} \) represents the negation of \( b \) (see line 21.)

It \( \text{binaryS-I}(I, L) \), we will find, by the binary search, an interval \( I_\ell \) in \( L \) which covers \( I \). If \( z = 0 \), it shows that such an interval does not exist.

**Function binaryS-I(I, L)**

begin
1. \( z \leftarrow 0; \)
2. binary search of \( L \) to find an interval \( z \), which covers \( I \);
3. return \( z; \)
end

In \( \text{binaryS-2}(I, L) \), we will first find a pair \( <x, y> \) such that \( I_\ell \) is the left-most interval in \( L \), which can be covered by \( I \); and \( I_r \) 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);
end

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

FUNCTION linearSearch(I, L, I)
begin
1. Let \( I' \) be \( I \);
2. Search \( I_{t-1}, I_{t-2}, \ldots \) until \( I_t \) such that \( I_t \) is covered by \( I \) but \( I_{t+1} \) not;
3. Search \( I_{t+1}, I_{t+2}, \ldots \) until \( I_t \) such that \( I_t \) is covered by \( I \), but \( I_{t+1} \) 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 \( \text{conjB}(L_f, L_w, 0) \), the following operations will be conducted:

Step 1: checking \( L_d[2] = [8, 14] \) against \( L_a \), \( l = \left\lceil \log \frac{2}{1} \right\rceil = 1, t \)
\[
= n - 2^t + 1 = 6 - 2 + 1 = 5, \quad L_a[5] = [11, 11].
\]
Since \([11, 11] \subset [8, 14], \) we will call \( \text{linearSearch}(\cdot) \) to find \( x = 4 \) and \( y = 5 \).

Step 2: checking \( L_a[1] = [1, 4] \) against \( L_a[1 \ldots 3] \), \( l = \left\lceil \log \frac{3}{1} \right\rceil = 1, t = 3 - 2^1 + 1 = 2, \quad L_a[2] = [3, 3]. \)
Since \([3, 3] \subset [1, 4], \) we will call \( \text{linearSearch}(\cdot) \) to find \( x = 1 \) and \( y = 2 \).

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 \( \{v_{10}, v_2\}, \) \( v_7 \) being LCA of \( \{v_{12}, v_8\}, \) \( v_2 \) being LCA of \( \{v_6, v_{12}, v_{18}, v_{15}\}, \) and \( v_0 \) being LCA of \( \{v_{10}, v_5, v_{12}, v_{16}, v_{15}\} \).

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 an 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 \( T_w \) for illustration.

Replacing each node in \( T_w \) with the corresponding interval, we get another tree, denoted as \( T_w^- \), 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 \( T_w^- [v] \) (the subtree rooted at \( v \) in \( T_w^- \)). See \( T_w^- \) shown in Fig. 7(b) for illustration. In \( T_w^- \), 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 \( L_w : [1, 1], [3, 3], [5, 6], [8, 8], [11, 11], [16, 16]. \)

Here, our intention is to associate each interval \( I_o^\prime \) in \( L_w \) with a second interval \( y_J \), which is the parent of \( I_o^\prime \) in \( T_w^- \), and two links, denoted as \( I_l \) and \( r_J \), respectively pointing to two intervals in \( L_w \), which are respectively the left-most and right-most leaf nodes in \( T_w^- [y_J] \). Fig. 8 helps for illustration.

In Fig. 8, \( I_o^\prime = [5, 6] \) is associated with an LCA interval \( y_3 = [8, 14], \) which is the parent of \( I_o^\prime \) in the corresponding \( T_w^- \) shown in Fig. 7(b). In addition, \( I_l \) is a link pointing to \( I_o^3 \) and \( r_J \) is a link pointing to \( I_o^6 \). They are respectively the left-most interval and the right-most interval covered by \( y_J \). In the same way, we can check all the other intervals and links shown in Fig. 8.

In addition, we will keep a sequence \( \Gamma \) containing all the LCA intervals in the post-order of \( T_w^- \). For example, \( \Gamma = [1, 4][8, 14][8, 19][1, 20] \). 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 \( I_o^m [1] > I_o^w [2] \), compare \( I_o^m \) and \( y_J \). If \( I_o^m \not\subset y_J \), explore \( L_w[r_J + 1 \ldots n] \) by the binary search. Otherwise, explore \( L_w[t +1 \ldots r_J] \).
2. If \( I_o^m [2] < I_o^w [1] \), compare \( I_o^m \) and \( y_J \). If \( I_o^m \not\subset y_J \), explore \( L_w[1 \ldots t - 1] \). Otherwise (\( I_o^m \subset y_J \)), explore \( L_w[l \ldots t - 1] \).
3. If \( I_o^m \supset I_o^w \), compare \( I_o^m \) and \( y_J \). If \( y_J \supset I_o^m \), \( I_o^w \) must be the unique interval which can be covered by \( I_o^m \). Therefore, \( I_o^w \) is the result and the search stops. The problem reduces to a
4. Assume that $O(\log k)$ to be used for control, where $k$ is the position prior to $\gamma_t$ in $\Gamma_u$. If $\gamma_t = I_{m}^u$, we will return all those intervals between $l_i$ and $r_i$, including both $l_i$ and $r_i$. The search also stops and the problem reduces to a checking of $L_a[1 .. m - 1] = I_{m}^u$. Then, return all the intervals between $l_i$ and $r_i$, including $l_i$ and $r_i$, which allows us to reduce the problem to check $L_a[1 .. m - 1] = I_{m}^u[1 .. g]$, where $g$ is the position prior to $\gamma_t$ in $\Gamma_u$.

4. If $I_{m}^u \subset L_a$, 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 $\text{conj}(\cdot)$.

Example 3 To see how the LCAs 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 $L_a = [11, 11]$ in $\Gamma_u$. Since $[5, 7]$ is to the left of $[11, 11]$, we will compare $[5, 7]$ with $\gamma_t = [8, 14]$ and $[5, 7] \subset [8, 14]$. So we will check $[5, 7]$ against $L_a[1 .. I_s - 1] = L_a[1 .. 3]$ in a next step, instead of the sequence containing all the intervals to the left of $I_s$.

2. Assume that $I = [10, 13]$ is compared with $L_a = [8, 8]$ in $\Gamma_u$. Since $[10, 13]$ is to the right of $[8, 8]$, $[10, 13]$ and $\gamma_t = [8, 14]$ will be compared and $[10, 13] \subset [8, 14]$. So, in the next step, we will check $[10, 13]$ against $L_a[4 + 1 .. r_s] = L_a[5 .. 5]$, not the sequence containing all the intervals to the right of $I_s$.

3. Assume that $I = [10, 13]$ is compared with $L_a = [11, 11]$ in $\Gamma_u$. We have $[10, 13] \supset [11, 11]$. However, $[10, 13] \subset \gamma_t = [8, 14]$. It shows that $[11, 11]$ is the only interval in $L_a$, which can be covered by $[10, 13]$. No further search is necessary.

4. Assume that $I = [8, 14]$ is compared with $L_a = [8, 8]$ in $\Gamma_u$. We have $[8, 14] \supset [8, 8]$. But we also have $[8, 14] = \gamma_t$. Then, we know immediately that only the intervals in $L_a[4 .. r_s] = L_a[4 .. 5]$ 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 $A(w)$ 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 outperforms the inverted file and the signature tree by an order of magnitude or more.

REFERENCES

[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.