An Efficient Method to Evaluate Intersections on Big Data Sets

YANGJUN CHEN and WEIXIN SHEN
The University of Winnipeg

Set intersections are important in computer science. Especially, intersection of inverted lists is a fundamental operation in information retrieval for text databases and Web search engines. In this paper, we discuss an efficient and effective way to implement this operation in the context of very big data sets. The main idea behind it is to do binary search over sorted interval sequences, each of which corresponds to an inverted list and is constructed by establishing a trie over the sequences of set identifiers as well as a kind of tree encoding, by which each node in the trie is assigned an interval. In many cases, an interval sequence is much shorter than its corresponding inverted list. In particular, the lowest common ancestors of intervals in a trie can be utilized to control a binary search to skip over useless interval containment checks, which enables us to reach an optimal off-line algorithm to do the task, and is theoretically better than any traditional on-line methods (at cost of more space). Experiments have been conducted, showing that the trade-off of space for time is worthwhile.

Categories and Subject Descriptors: F.2.2 [**Analysis of algorithms and Problem Complexity**]: Non-numerical Algorithms and Problems *Pattern matching*; *computation on discrete structures*

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Set intersection, inverted files, interval sequences, search engines.

1. INTRODUCTION

In mathematics, the *intersection* $A \cap B$ of two sets A and B is the set that contains all elements of A that also belong to B. In practice, however, the problem is typically related to a collection of sets $\boldsymbol{S} = \{S_1, S_2, \ldots, S_M\}$ and we are often asked to evaluate the intersection over a sub-collection of $\boldsymbol{S}$:

$$S_{i_1} \cap S_{i_2} \cap \ldots \cap S_{i_m}$$

for some $m \leq M$.

This is a key operation in information retrieval, especially for Web search engines and text databases, by which each S_i ($i \in \{1, \ldots, M\}$) is a subset of document identifiers containing a certain word, called an *inverted list*. Then, to find all the documents containing a set of words $w_1, \ldots, w_k$, a set intersection like the above over all the inverted lists associated with these words needs to be conducted.

This work is supported by NSERC, Canada, 239074-01 (242523). This is a modification and extension of two papers respectively published in *Int. Conf. on Advances in Big Data Analytics*, *IEEE*, July 21-24, 2014, USA [19]; and the *11th Int. Conf. on Foundations of Computer Science, IEEE,* July 26-30, 2015, USA [20].

Author's addresses: Dept. of Applied Computer Science, University of Winnipeg, 515 Portage Ave. Winnipeg, Manitoba, Canada R3B 2E9.

In the past several decades, there is a lot of research on this interesting topic, such as *adaptive* and *melding* algorithms [5, 6, 7, 8, 9, 23, 24], building additional data structures like *skipping lists* [43], *treaps* (a kind of balanced trees) [12], indexes [21], *hash tables* over sorted lists [3, 26], and so on. All of them can improve the time complexity at most by a constant factor, but none of them is able to bring it down by an order of magnitude.

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

- Put all the sets: S_1, S_2, ..., S_M in a sequence decreasingly sorted by their sizes. Then, represent each $e \in \bigcup_{i=1}^{M} S_i$ as a subsequence of set identifiers such that each set in this subsequence contains e, denoted as s_e.
- Construct a *trie* T over all s_e's.
- Replace each S_x ($x \in \{1, ..., M\}$) with an interval sequence X, where each interval in X is created by applying a kind of tree encoding over T.
- Associate each interval in X with a subset of S_x. In this way, we decompose S_x into a collection of disjoint subsets, and transform the comparison of elements (for doing set intersections) to the checking of interval containment. In many cases, an interval sequence is much shorter than the corresponding set. Therefore, the search of X can be faster than the search of the corresponding set S_x.
- For each interval sequence X constructed for a certain S_x, we will construct a second sequence, Γ_x, such that each element in it corresponds to the lowest common ancestor (*LCA* for short) of some nodes in the trie T, whose intervals make up a segment in X. Γ_x is used to control the binary search of X when a set intersection involving S_x is evaluated.

Let S_x, S_y be two sets with $|S_x| < |S_y|$. Up to now, the best on-line algorithm for intersecting S_x and S_y requires $O(|S_x|\cdot\log(|S_y|/|S_x|))$ time [5, 59]. In contrast, our off-line algorithm needs $O(|Y|\cdot\log\kappa)$ time by using indexes, where $\kappa < |\Gamma_x|/|Y|$, and Y is an interval sequence created for S_y. As can be seen later, we always have $|Y| \leq |X| \leq |S_x|$ and $|\Gamma_x| < |X|$. This time complexity is significantly better than the traditional on-line methods due to the following two key facts:

1. Each interval corresponds to a subset of some set. Therefore, in many cases, the length of an interval sequence created for a set can be much smaller than the corresponding set itself. Especially, the larger a set is, the smaller its corresponding interval sequence. Only for those very small sets, the sizes of their corresponding interval sequences may be near their sizes.
2. During the binary search of an interval sequence, the relationship between the intervals and their *LCA*s can be used to skip over a lot of useless interval containment checking while it is not possible by any comparison-based algorithm no matter how sets are stored using different data structures, such as sorted arrays, trees, skipping lists, hash tables, and so on.

The optimal running time of our method is at cost of the space for indexes. Since for each inverted list we will create an interval sequence which is not longer than the corresponding inverted list, the space requirement for all the interval sequences should be linear in the size of all inverted lists. However, since two integers are needed to represent an interval, the used space must be larger than the inverted lists. On the other hand, due to the property that the longer an inverted list is the shorter the corresponding interval sequence, the difference between interval sequences and inverted lists is not so big. For example, for the TREC GOV2 corpus, the size of the interval sequences for all those inverted lists of length smaller than 10k is 1.5 times the size of the corresponding inverted lists. But the size of all the interval sequences together is 1.105 times the size of all inverted lists.

The remainder of the paper is organized as follows. In Section 2, we review the related work. In Section 3, we present the new index structure in great detail. In Section 4, we discuss our algorithms to evaluate conjunctive queries based on this index structure, which are further improved by using *LCAs* in Section 5. In Section 6, we report the test results. Finally, a short conclusion is set forth in Section 7.

2. RELATED WORK

The evaluation of set intersections is very important in text databases and Web search engines [13, 40]. Much research on this topic has been done in the past several decades. Two methods have been widely advocated as efficient indexing schemes to handle large volume data. One is *inverted files* [56] and the other is *signature files* [27]. According to [57], for typical applications of full-text indexing, inverted files are superior to signature files in almost every respect, including speed, space, and functionality.

Inverted files

By the inverted file, each word will be associated with a sorted list of document identifiers containing the word, which was first reported as early as mid-1960s [34, 47]. The subsequent research on this index structure includes *integer coding* [3, 29], integer compression [1, 49, 55], bitmap compression [10, 41], caching [38, 45], parallelism [2, 48], and distributed computation [22]. Also, many methods have been proposed to speed up the intersection of inverted lists in different ways.

Adaptive and melding. An adaptive and melding method intersect all the lists in parallel so as to compute the intersection according to different measure of difficulty at each step [7, 23]. In [23], the galloping search is used to find a matching element in a sorted set while in [7] the interpolation is utilized. According to [8], for the intersection of k sets: $L_1 \cap \ldots \cap L_k$ ($k \geq 2$) the lower bound of the problem is $O(\delta\sum_{i=1}^{k}\log(|L_i|/\delta))$, where δ is the minimal number of intervals which cover a totally ordered space (i.e., all the docIds) such that for each interval I, if it covers only a singleton x, then $x \in \cap_i L_i$, otherwise, there exists at least a L_i ($i \in \{1, \ldots, k\}$) with $I \cap L_i = \Phi$. In the case of $k = 2$ with $|L_1| < |L_2|$, $\delta = O(|L_1|)$ and $O(\delta\sum_{i=1}^{k}\log(|L_i|/\delta)) = O(|L_1|\cdot\log(|L_2|/|L_1|))$. This is exactly the time complexity of Hwang and Lin's algorithm ([59], should be modified from doing the set union to the set intersection), and Baeza-Yates's [5], which is in fact a balanced version of the former, by which we choose the median element of L_1 at the first step and recursively divide L_1 and L_2 in the subsequent computation. For $k > 2$, the algorithm proposed by Barbay and Kenyon [8] reaches this time complexity.

Hierarchical representation. A set can be represented as a balanced binary search tree, such as treaps [12], skip-lists [43], or a compact two-level structure [44]. They also aim at reducing the number of comparisons. Especially, using treaps [12], the optimal time complexity $O(|L_1|\cdot\log(|L_2|/|L_1|))$ can be achieved. However, due to the tree searching process, this kind of algorithms not always outperforms the binary search over a sorted list. Their advantage mainly consists in the ease of maintenance of balanced data structures.

Hashing-based. There are various methods based on hash-functions to speed up the intersection, such as the algorithms discussed in [11, 26]. In [11], two sets L_1 and L_2 are mapped using a hash-function h to smaller representations $h(L_1)$ and $h(L_2)$, respectively. Then, the intersection is done on $h(L_1)$ and $h(L_2)$. Given k sets: $L_1, \ldots, L_k$ of total size M, their intersection can be computed in time $O((M\log^2 C)/C + k\cdot r)$ on average, where $r = |\bigcap_{i=1}^{k} L_i|$, and C is a constant, typically set to be the size of a memory unit. This running time is improved by Ding and König [26], who first divide a set into a collection of smaller subsets, and then map, using a hash-function, each subset into a bit string which can be packed in a single memory unit. In this way, the time complexity can be reduced to $O(M/\sqrt{C} + kr)$. However, we should notice that in both their time complexity analyses, the cost of hashing is not taken into account; but in practice each hash-function computation itself needs in fact a constant time.

Index-based. The method discussed in [21] is index-based, by which an unbalanced binary tree T is constructed with each node being used to store a matrix M over some sets from a database D. $M[i, j] = 1$ if lists i and j have a non-empty intersection. Otherwise, $M[i, j] = 0$. The size of M is bounded by $|D|$ (the number of sets in D.) Since the sets handled by two different nodes can be repeated, the space overhead of the whole index is bounded by $O(|T|\cdot|D|)$. The query time for evaluating $L_1 \cap L_2$ is bounded by $O(\sqrt{|D|\cdot r'} + r')$, where $r' = |L_1 \cap L_2|$. For a database D containing a very large amount of words (like TREC GOV2 corpus, which contains more than 38 million words), this time complexity is not much better than Baeza-Yates's [5].

SIMD-based. The SIMD (*single instruction/multiple data*) has also been used to reduce the running time, as reported in [33, 37, 46]. In [33], a simple algorithm is discussed, which extends the merge-based algorithm by reading multiple elements each time, instead of just one element, from each of two input arrays to use SIMD instructions. However, its theoretical time complexity is bounded by $O(b\cdot(|L_1| + |L_2|) - b^2)$, where b is the size of a block, i.e., the number of elements taken each time respectively from L_1 and L_2 for comparison. The methods discussed in [37, 46] have the same time complexities as [33]. In [46], the SIMD is used for the galloping search while in [37] the so-call *STNNI* instructions (STring and Text processing New Instruction) are used.

Inverted-list compression. Besides the above mentioned methods, there is quite different stream of research on the compression of inverted lists to reduce the space overhead, but not sacrificing too much query time, including document reordering, classification, and docId compression, such as *Rice coding* [62], *Simple16 coding* [62], *PForDelata coding* [32], and the modified *PForDelata coding* [61].

Signature files

By the signature file, a word is hashed to a bit string (called a *signature*) and all the words' signatures of a document are superimposed (bit-wise OR operation) into a document signature. When a query arrives, its signature will be created using the same hash-function and the document signatures are scanned and many nonqualifying documents are discarded. The rest are either checked (so that the 'false drops' are removed) or they are returned to the user as they are [26, 27]. The main disadvantage of this method is the false drop [31, 35], which needs extra time to check. Over the years, different ways to store signatures have been proposed, such as *bit-slice files* [35], *S-trees* [25, 51], and *signature trees* [15, 16]. By the bit-slice files, the signatures in the file are *vertically* stored in a set of files [31]. By the S-trees, a signature file is organized into a height balanced multiway tree. The signature tree works in a quite different way, which organizes a set of signatures into a binary tree structure and replaces a sequential search of signatures with a search of binary trees, improving performance by an order of magnitude or more.

Others

There are some other interesting methods to improve the efficiency of set intersections [48, 53, 58]. In [48, 53], the so-called *multi-core architecture* is utilized to speed up computation, which can also be used to parallelize our method. In [58], scoring functions are used to avoid computing full intersections. The main difficulty of this method is that a scoring function may not be easily established, and the machine-learning techniques have to be employed to solve the problem.

3. TRANSFORMATION OF SORTED LISTS TO SORTED INTERVAL SEQUENCES

Let $\aleph$ be a sequence of sets:

$S_1, S_2, \ldots, S_M$

such that for $1 \le i < j \le M$ $|S_i| \ge |S_j|$.

Denote $S = \bigcup_{i=1}^{M} S_i$. For simplicity, assume that each element in S is an integer > 0. Then, each element $i \in S$ corresponds to a subsequence of $\aleph$:

$S_{i_1}\ S_{i_2}\ \ldots\ S_{i_m}$

for some m such that for each $j \in \{1, \ldots, m\}$ $i \in S_{i_j}$. Clearly, for $k < l$, we have $|S_{i_k}| \ge |S_{i_l}|$. Assign each S_j an identifier w_j. Thus, we can represent i as a sequence $\aleph_i$:

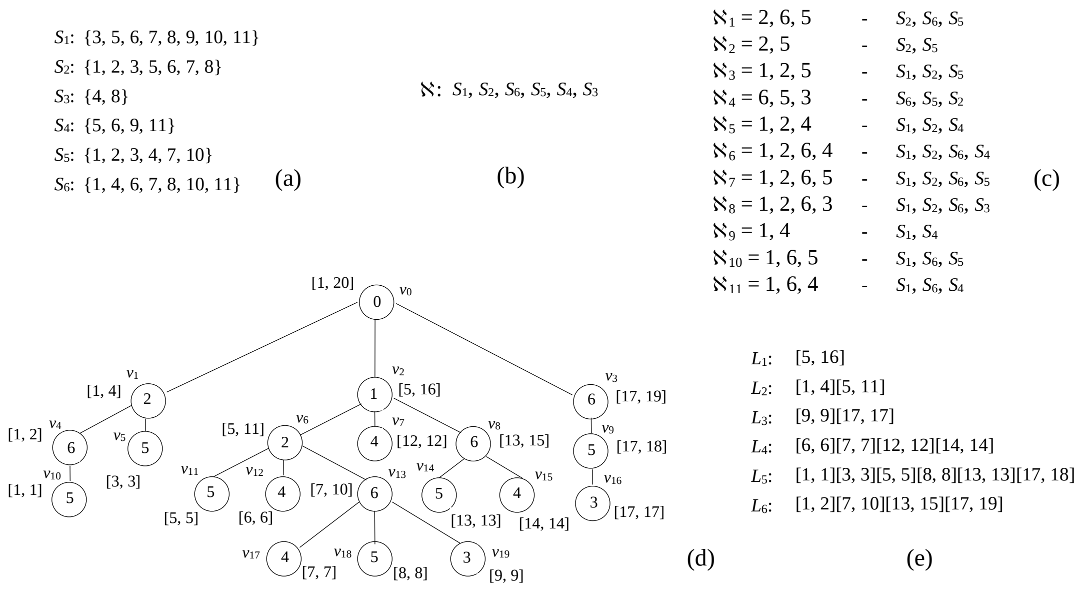
$w_{i_1}\ w_{i_2}\ \ldots\ w_{i_m}$.

Over all $\aleph_i$'s, a trie structure can be constructed as follows.

Let $D = \{\aleph_1, \aleph_2, \ldots, \aleph_N\}$ be all $\aleph_i$'s. Denote by *trie*(D) the trie constructed over D.

If $|D| = 0$, *trie*(D) is, of course, empty. For $|D| = 1$, *trie*(D) is a single node. If $|D| > 1$, D is split into M (possibly empty) subsets $D_1, D_2, \ldots, D_M$ so that a sequence $\aleph_i$ is in D_j if its first element is w_j ($1 \le j \le M$). The tries *trie*(D_1), *trie*(D_2), ..., *trie*(D_M) 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*(D).

See Fig. 1 for illustration.


S_1: {3, 5, 6, 7, 8, 9, 10, 11}
S_2: {1, 2, 3, 5, 6, 7, 8}
S_3: {4, 8}
S_4: {5, 6, 9, 11}
S_5: {1, 2, 3, 4, 7, 10}
S_6: {1, 4, 6, 7, 8, 10, 11}
(a)

$\aleph$: $S_1, S_2, S_6, S_5, S_4, S_3$
(b)

$\aleph_1$ = 2, 6, 5	-	S_2, S_6, S_5
$\aleph_2$ = 2, 5	-	S_2, S_5
$\aleph_3$ = 1, 2, 5	-	S_1, S_2, S_5
$\aleph_4$ = 6, 5, 3	-	S_6, S_5, S_2
$\aleph_5$ = 1, 2, 4	-	S_1, S_2, S_4
$\aleph_6$ = 1, 2, 6, 4	-	S_1, S_2, S_6, S_4
$\aleph_7$ = 1, 2, 6, 5	-	S_1, S_2, S_6, S_5
$\aleph_8$ = 1, 2, 6, 3	-	S_1, S_2, S_6, S_3
$\aleph_9$ = 1, 4	-	S_1, S_4
$\aleph_{10}$ = 1, 6, 5	-	S_1, S_6, S_5
$\aleph_{11}$ = 1, 6, 4	-	S_1, S_6, S_4

(c)

L_1:	[5, 16]
L_2:	[1, 4][5, 11]
L_3:	[9, 9][17, 17]
L_4:	[6, 6][7, 7][12, 12][14, 14]
L_5:	[1, 1][3, 3][5, 5][8, 8][13, 13][17, 18]
L_6:	[1, 2][7, 10][13, 15][17, 19]

(e)

Fig. 1: A collection of sets, a sequence $\aleph$ of sets, a set of subsequences of $\aleph$ and a trie.

In Fig. 1(a), we show a collection of six sets: $S_1, \ldots, S_6$ with each containing one or more positive integers. Fig. 1(b) shows a sequence $\aleph$ of all the sets sorted decreasingly by their sizes. Fig. 1(c) is a set of subsequences of $\aleph$, and each subsequence $\aleph_j$ represents a group of sets with each containing $j \in \bigcup_{i=1}^{6} S_i$. For example, $\aleph_7$ = 1, 2, 6, 5 represents four sets: S_1, S_2, S_6, S_5, each containing 7. In Fig. 1(d), we show a trie established over all $\aleph_i$'s.

In this *trie*, v_0 is a virtual root, labeled with 0 (representing an empty set) while any other node is labeled with a positive integer i, representing a set S_i. Hence, all the integers on a path from the root to a leaf make up a subsequencs of $\aleph$. For instance, the path from v_0 to v_{18} corresponds to a sequence $\aleph_7$ = 1, 2, 6, 5. Thus, to check whether two sets S_i and S_j contain a common element, we need only to check whether there exist two nodes v_1 and v_2 such that v_1 is labeled with i, v_2 with j, 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 it is asked whether a node v can reach another node u through a path (see [17, 18] for a detailed discussion.) If it is the case, it is denoted as $v \Rightarrow u$; otherwise, denoted as $v \not\Rightarrow u$.

It is well-known that the reachability checking can be done efficiently by using a kind of tree encoding [4, 17], which labels each node v in a tree with an interval $I_v = [\alpha_v, \beta_v]$, where β_v denotes the rank of v in a *post-order* traversal of the tree. 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(d), such a tree encoding is also exhibited, assuming that the children are ordered from left to right. It is easy to see that whether two nodes are on a same path can be checked by interval containment. For example, $v_2 \Rightarrow v_{19}$, since $I_{v_2} = [5, 16]$, $I_{v_{19}} = [9, 9]$, and $[9, 9] \subset [5, 16]$; but $v_1 \not\Rightarrow v_{16}$, since $I_{v_1} = [1, 4]$, $I_{v_{16}} = [17, 17]$, and $[17, 17] \not\subset [1, 4]$.

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

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

Proof. The lemma can be derived from the post-order traversal process of a tree or a forest. □

Since more than one node in a trie may be labeled with the same number, a number (representing a set) may be associated with more than one interval. Thus, to know whether two sets share common elements, multiple checks may be needed. For example, to check whether S_2 and S_3 contain common elements, we need to check v_1 and v_6 each against both v_{16} and v_{19}, by using the node's intervals.

For this reason, each number x in a trie will be associated with an interval sequence of the form: $X = I_1, I_2, \ldots, I_n$, where n is the number of all those nodes labeled with x and each $I_j = [I_j[1], I_j[2]]$ $(1 \le j \le n)$ is an interval associated with a certain node labeled with x. In addition, since any two of these intervals are not on a same path, we can sort X so that for $1 \le k < l \le n$ we have $I_k \prec I_l$ (then, $I_k[1] < I_k[2] < I_l[1]$), which will greatly reduce the time for checking reachability. We illustrate this in Fig. 1(e), in which each interval sequence corresponds to a set in Fig. 1(a).

Comparing Fig. 1(a) and Fig. 1(e), the following three properties can be easily observed

i) Any interval sequence cannot be larger than the corresponding set.

ii) An interval sequence can be much smaller than the corresponding set.

iii) The longer an inverted sequence is, the smaller the corresponding set.

For example, $S_1 = \{3, 5, 6, 7, 8, 9, 10, 11\}$ contains 8 elements while $L_1 = [5, 16]$ contains only one interval. In addition, we can associate each node v in a trie with a subset S' such that for each $i \in S'$, $\aleph_i$ has a prefix represented by the path from v_0 to v. For example, the sequence represented by a path: $v_0 \rightarrow v_2 \rightarrow v_6 \rightarrow v_{13}$ is 0, 1, 2, 6, a common prefix of $\aleph_6$, $\aleph_7$, $\aleph_8$. So the subset associated with v_{13} should be $\{6, 7, 8\}$. Procedurally, the subset associated with a node v can be constructed as below:

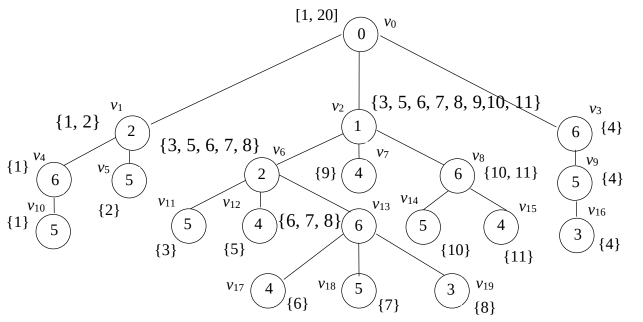
- If v is a leaf node, the subset assigned to v contains only one document identifier corresponding to the word sequence represented by the path from the root to v.

- If v is an internal node, the subset assigned to v is the disjoint union of all the subsets assigned to its children.

See Fig. 2(a) for illustration.

In this way, we also create a correspondence between intervals and subsets as illustrated in Fig. 2(b). Let I be an interval associated with a node v in T. We will denote by $\delta(I)$ (also, $\delta(v)$) the corresponding subset. Then, we get a different way to evaluate set intersections $S_x \cap S_y$ with S_y appearing before S_x in $\aleph$:

1. Let $X = I_1, I_2, \ldots, I_n$ and $Y = J_1, J_2, \ldots, J_m$ be the interval sequences for S_x and S_y, respectively.
2. Find $I_{n_1}, \ldots, I_{n_k}$ for some k such that for each $1 \leq l \leq k$ there exists some interval in Y, which covers I_{n_l}.
3. Return $\delta(I_{n_1}) \uplus \ldots \uplus \delta(I_{n_k})$ as the result of $S_x \cap S_y$, where $\uplus$ represents *disjoint union* over disjoint sets.



(a)

[1, 4] - {1, 2}	[7, 7] – {6}
[1, 2] – {1}	[8, 8] – {7}
[1, 1] - {1}	[9, 9] – {8}
[3, 3] – {2}	[12, 12] – {9}
[5, 16] – {3, 5, 6, 7, 8, 9, 10, 11}	[13, 13] – {10}
[5, 11] - {3, 5, 6, 7, 8}	[13, 15] – {10, 11}
[5, 5] – {3}	[14, 14] – {11}
[6, 6] – {5}	[17, 19] – {4}
[7, 10] – {6, 7, 8}	[17, 18] – {4}
	[17, 17] – {4}

(b)

Fig. 2: Association of intervals with subsets.

As an example, consider L_5 = [1, 1][3, 3][5, 5][8, 8][13, 13][17, 18] (for S_5 in Fig. 1(a)) and L_2 = [1, 4][5, 11] (for S_2). To evaluate the intersection $S_5 \cap S_2$, we will find a subsequence in L_5: [1, 1][3, 3][5, 5][8, 8] such that [1, 1] $\subset$ [1, 4], and [3, 3] $\subset$ [1, 4], as well as [5, 5] $\subset$ [5, 11], and [8, 8] $\subset$ [5, 11]. Note that the δ([1, 1]) = {1}, δ([3, 3]) = {2}, δ([5, 5]) = {3}, and δ([8, 8]) = {7}. The answer is then their (disjoint) union: {1, 2, 3, 7}.

The correctness of the above algorithm is based on the following two lemmas.

Lemma 2 Let $\aleph = S_1, S_2, \ldots, S_M$ be a sorted sequence of sets. Let $\aleph_1, \aleph_2, \ldots, \aleph_N$ be all the subsequences of $\aleph$ created for all $j \in S = \bigcup_{i=1}^{M} S_i$ ($j = 1, \ldots, N$). Let $X = I_1, I_2, \ldots, I_n$ be the interval sequence for S_x ($x \in \{1, \ldots, M\}$). Then, $\delta(I_1) \cup \delta(I_2) \cup \ldots \cup \delta(I_n) = S_x$.

Proof. For each S_x, we can view x as its identifier. For each $y \in S_x$, we can view it as the identifier of another set Q_y which contains y. Let $v_1, \ldots, v_n$ be all the nodes labeled with x in T. Then, $\delta(v_1) \cup \delta(v_2) \cup \ldots \cup \delta(v_n)$ must be the identifiers of all those sets containing x, which are exactly equal to S_x. □

In a similar way, we can prove Lemma 3.

Lemma 3 Let u and v be two nodes in a trie T, labeled with two intervals I_u and I_v, respectively. If u and v are not on the same path in T, then $\delta(I_u)$ and $\delta(I_v)$ are disjoint, i.e., $\delta(I_u) \cap \delta(I_v) = \Phi$. □

Proposition 1 Let $X = I_1, I_2, \ldots, I_n$ be the interval sequence for S_x. Then, $\delta(I_1) \uplus \delta(I_2) \uplus \ldots \uplus \delta(I_n) = S_x$.

Proof. According to Lemma 2, we have $\delta(I_1) \cup \delta(I_2) \cup \ldots \cup \delta(I_n) = S_x$. According to Lemma 3, $\delta(I_1) \cup \delta(I_2) \cup \ldots \cup \delta(I_n)$ is equal to $\delta(I_1) \uplus \delta(I_2) \uplus \ldots \uplus \delta(I_n)$. □

As an example, consider the nodes v_1 and v_6 in Fig. 2(a). They are the only nodes labeled with 2. So S_2 is equal to $\delta(v_1) \uplus \delta(v_6) = \{1, 2\} \uplus \{3, 5, 6, 7, 8\} = \{1, 2, 3, 5, 6, 7, 8\}$.

4. EVALUATION OF SET INTERSECTIONS

In this section, we discuss how to efficiently evaluate set intersections by using interval sequences. For ease of explanation, we first discuss a method based on a linear search of interval sequences in 4.1. Then, in 4.2, we discuss a more interesting and efficient method based on a binary search of interval sequences, for which some more new concepts and techniques need to be introduced.

4.1 Evaluation based on linear search

As mentioned in Section 3, to evaluate the intersection of two sets S_x and S_y with S_y appearing before S_x in $\aleph$, we need to search both X and Y to find all those intervals in X such that each of them is covered by some interval in Y. Since both X and Y are sorted, a linear search process can be arranged to do the task as follows.

1. Let $X = I_1, I_2, \ldots, I_n$ and $Y = J_1, J_2, \ldots, J_m$. $L \leftarrow \phi$. (*L is used to store the result.*)
2. Step through X and Y from left to right. Let I_k and J_l be the intervals currently encountered. Compare I_k and J_l. We will have one of three possibilities:
   i) If $I_k \subset J_l$, append I_k to the end of L. Move to I_{k+1} if $k < n$ (then, in a next step, we will check I_{k+1} against J_l.) If $k = n$, stop.
   ii) If $I_k \prec J_l$, move to I_{k+1} if $k < n$. If $k = n$, stop.
   iii) If $J_l \prec I_k$, move to J_{l+1} if $l < m$ (then, in a next step, we will check I_k against J_{l+1}). If $l = m$, stop. □

Assume that the result is $L = I_1', I_2', \ldots, I_p'$ ($0 \le p \le n$). Then, for each $1 \le q \le p$, there exists an interval $J \in Y$ such that $I_q' \subset J$, and we can return $\delta(I_1') \uplus \delta(I_2') \uplus \ldots \uplus \delta(I_p')$ as the answer. In Fig. 3, we illustrate the working process on $X = L_6$ and $Y = L_2$ shown in Fig. 1(e).

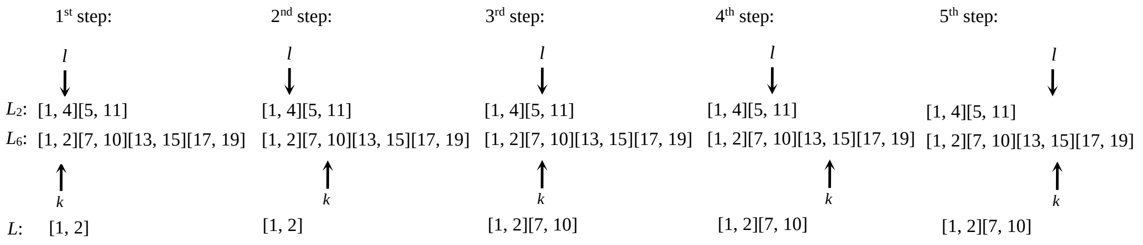


Fig. 3: Illustration for the linear search of interval sequences.

In Fig. 3, we first notice that $X = L_2 = [1, 4][5, 11]$ and $Y = L_6 = [1, 2][7, 10][13, 15][17, 19]$. In the 1st step, we will check $I_1 = [1, 4]$ against $J_1 = [1, 2]$. Since $[1, 2] \subset [1, 4]$, $J_1 = [1, 2]$ will be added to the result L. In the 2nd step, we will check $I_1 = [1, 4]$ against $J_2 = [7, 10]$. Since $[1, 4] \prec [7, 10]$, l will be increased by 1 and then we will check $I_1 = [5, 11]$ against $J_2 = [7, 10]$. Since $[7, 10] \subset [5, 11]$, $J_2 = [7, 10]$ will be added to L and k will be increased by 1. Comparing $I_2 = [5, 11]$ and $J_3 = [13, 15]$, we find $[5, 11] \prec [13, 15]$. Since now l is equal to $|X|$, the process stops. The result is $\delta([1, 2]) \uplus \delta([7, 10]) = \{1\} \uplus \{6, 7, 8\} = \{1, 6, 7, 8\}$.

Lemma 4 Let $L = I_1', \ldots, I_p'$ be the result of the above process when applied to X and Y respectively created for S_x and S_y with S_y appearing before S_x in $\aleph$. Then, for each I_q' ($1 \le q \le$

p), there must be an interval $J \in Y$ such that $I_q' \subset J$. For any interval $I \in X$ but $\notin L$, it is definitely not covered by any interval in Y.

Proof. The correctness of the lemma can be derived from the properties of X and Y, and the facts that both X and Y are sorted. □

4.2 Evaluation based on binary search

The set intersection can also be done by using binary search. However, due to the difference between the containment checking of intervals and the comparison of integers, the traditional binary searching cannot be simply utilized because the containment of an interval I (from Y) in another interval I' (from X) will not in general allow us to divide X into two parts. Therefore, a more sophisticated technique is needed to achieve an optimal running time. Specifically, *LCAs* (lowest common ancestors) have to be used to speed up the working process. In this subsection, we only give a basic algorithm for the binary search of interval sequences, while the discussion on how to employ *LCAs* will be shifted to the next section.

Before we present the algorithm, we first define two functions which will be used by it as two basic operations.

The first function is *search*(X, j, J) with three inputs: X – a sorted interval sequence, j – a position in X, and J – an interval from some other interval sequence Y, used to find a smallest $k \le j$ and a largest $l \ge j$ such that all the intervals in X between k and l (including k, l) can be contained in J. See Fig. 4 for illustration.

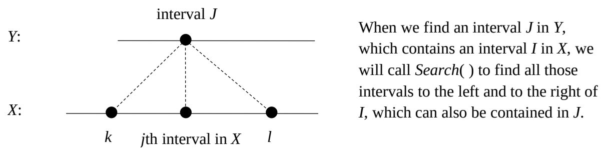

interval J
Y:
X:
k
jth interval in X
l
When we find an interval J in Y, which contains an interval I in X, we will call Search( ) to find all those intervals to the left and to the right of I, which can also be contained in J.

Fig. 4: Illustration for *Search*(X, j, J).

The second function is *binarySearch*(X, J, b), also with three inputs: X – a sorted interval sequence, J – an interval from some other interval sequence, and b – a Boolean value. If b = 0, it will try to find an interval in X by the binary search, which can be contained in J; otherwise (b = 1), to find an interval which contains J.

With the above two functions, our basic algorithm to do the binary set intersection can be described as follows, which is in essence a modification of the binary set union discussed in [59].

Let S_x and S_y be two sets. Let X and Y be their interval sequences containing distinct intervals of respective lengths n and m:

$I_1 \prec I_2 \prec \ldots \prec I_n$, and

$J_1 \prec J_2 \prec \ldots \prec J_m$.

Assume that S_y appears before S_x in $\aleph$. Then, we have $m \le n$.

The binary set intersection process can be mostly easily described recursively. When m = 0 (i.e., the shorter list is empty) there is no intersection to be done and the procedure terminates with *empty* as the result. Otherwise, we figure out J_m, the last interval in the shorter sequence Y, and attempt to find an interval in the longer sequence X, which can be contained in J_m. To do this, let $l = \left\lfloor \log \frac{n}{m} \right\rfloor$. Then, 2^l is the largest power of 2 not exceeding $\frac{n}{m}$.

Let $t = n - 2^l + 1$. Compare J_m and I_t. See Fig. 5 for illustration.

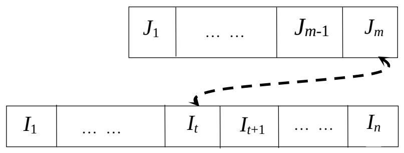


Fig. 5: First comparison during an interval intersection.

We first distinguish among three cases:

Case 1: $J_m \prec I_t$,

Case 2: $I_t \prec J_m$, and

Case 3: $J_m \supset I_t$.

If $J_m \prec I_t$ (case 1), then we will search the intervals to the left of I_t: I_1, …, I_t in Fig. 5. The problem immediately reduces to the situation illustrated in Fig. 6(a). We can finish the set intersection by recursively applying the binary set intersection to lists $X' = I_1, \ldots, I_t$, and $Y' = J_1, J_2, \ldots, J_m$.

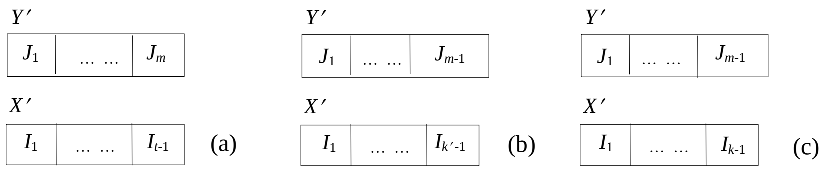


Fig. 6: Outcomes after first comparison.

If, on the other hand, $I_t \prec J_m$ (case 2), then we will search the intervals to the right of I_t: $L = I_{t+1}, \ldots, I_n$ in Fig. 5. By calling *binarySearch*(L, J_m, 0), we try to find, with exactly l more comparisons, an interval I_k either containable in J_m (case 2-1), or satisfying $I_{k-1} \prec J_m \prec I_k$ (case 2-2). In case 2-1, we will call *search*(X, k, J_m) to find k' and k'' such that all the intervals in X between k' and k'' (including k', k'') can be covered by J_m (then, all these intervals should be added to the result R); this information allows us to reduce the problem to the situation illustrated in Fig. 6(b). To complete the set intersection it is sufficient to perform the set intersection on the lists $Y' = J_1, J_2, \ldots, J_{m-1}$, and $X' = I_1, \ldots, I_{k'-1}$. In case 2-2, the problem is reduced to $Y' = J_1, J_2, \ldots, J_{m-1}$ and $L_x' = I_1, \ldots, I_{k-1}$. See Fig. 6(c). Note that X' may be longer than Y', so that in the recursive calls to the set intersection procedure the roles of X and Y may become reversed.

If $J_m \supset I_t$ (case 3), as case 1 above, we will call *search*(X, t, J_m) to find all the intervals in X which can be covered by J_m and insert them into the result R. The set intersection can be completed by a recursive call on $Y' = J_1, J_2, \ldots, J_{m-1}$ and $X' = I_1, \ldots, I_{k'-1}$, where k' is the left-most position in X which can be covered by J_m, just as case 2-1.

Finally, due to the possible interchange of rolls played by X' and Y', we need yet to consider a fourth case: $J_m \subset I_t$. In this case, we simply insert J_m into R and the problem is reduced to the set intersection over $Y' = J_1, J_2, \ldots, J_{m-1}$ and $X' = I_1, \ldots, I_t$, also as case 2-1.

In terms of the above analysis, we give the following algorithm, in which besides *search*( ) and *binarySearch*( ), another two simple subfunctions B_0( ) and B_1( ) are also used, respectively for $b = 0$ and $b = 1$, to handle the result and to determine the interval subsequences for a next recursive call.

The algorithm takes two interval sequences X and Y with $|X| \geq |Y|$, and a Boolean variable

b as the inputs.

```
ALGORITHM setIntersect(X, Y, b) (*Initially, b = 0.*)
begin
1.  Let $X = I_1, I_2, \dots, I_n$ and $L_2 = J_1, J_2, \dots, J_m$;
2.  if $m = 0$ then return;
3.  $l \leftarrow \left\lfloor \lg \frac{n}{m} \right\rfloor$; $t \leftarrow n - 2^l + 1$; $I \leftarrow J_m$;
4.  if $I \prec I_t$ then {$X' \leftarrow X[1 .. t - 1]$; $Y' \leftarrow Y$;}
5.  if $I_t \prec I$
6.  then $z \leftarrow binarySearch(X[t + 1 .. n], I, b)$;
7.       if $z = 0$ then { $X' \leftarrow X$; $Y' \leftarrow Y[1 .. m-1]$;}
8.       else if $b = 1$ then $<X', Y'> \leftarrow B_1(X, Y, t + z, I, R)$;
9.            else $<X', Y'> \leftarrow B_0(X, Y, t + z, I, R)$;
10. if $I \supset I_t$ then $<X', Y'> \leftarrow B_0(X, Y, t, I, R)$;
11. if $I \subset I_t$ then $<X', Y'> \leftarrow B_1(X, Y, t, I, R)$;
12. if $|Y'| \leq |X'|$ then setIntersect(X', Y', b)
13. else setIntersect($Y', X', \bar{b}$);
end
```

```
FUNCTION B_0(X, Y, t, I, R)
Begin
1. $<j', j''> \leftarrow Search(X, t, I)$;
2. $R \leftarrow R \cup \{X[j' .. j'']\}$;
3. $X' \leftarrow X[1 .. j' - 1]$; $Y' \leftarrow Y[1 .. |Y| - 1]$;
4. return $<X', Y'>$;
end
```

```
FUNCTION B_1(X, Y, t, I, R)
begin
1. $R \leftarrow R \cup \{I\}$;
2. $X' \leftarrow X[1 .. t]$; $Y' \leftarrow Y[1 .. |Y| - 1]$;
3. return $<X', Y'>$;
end
```

The algorithm *setIntersect*(X, Y, b) can be viewed as composed of six parts.

Part 1 (lines 1 – 3). In this part, we do the initialization work.

Part 2 (line 4). In this part, we handle the case $J_m \prec I_t$ by determining two interval sequences $X' = X[1 .. t - 1]$ and $Y' = Y$, to which the next recursive call will be applied.

Part 3 (lines 5 – 9). In this part, the most complicated case $I_t \prec J_m$ is handled. First, a traditional binary search over $X[t + 1 .. n]$ will be carried out to find an interval, which contains J_m (if $b = 1$), or is containable in J_m (if $b = 0$) (see line 6.) If the corresponding interval cannot be found, we will determine two interval sequences $X' = X$ and $Y' = Y[1 .. m - 1]$ for the next recursive call (see line 7.) Otherwise, depending on whether $b = 1$ or $b = 0$, we will respectively call $B_1(X, Y, t, I, R)$, or $B_0(X, Y, t, I, R)$, where $I = J_m$ (see lines 8 and 9.) In $B_1(X, Y, t, I, R)$, we will first add I to R and then determine two interval sequences $X' = X[1 .. t]$ and $Y' = Y[1 .. |Y| - 1]$ for the next recursive call. In $B_0(X, Y, t, I, R)$, we will first call *Search*(X, t, I) to find a pair $<j', j''>$ such that all the intervals between j', j'' (including j', j'') in X are containable in I; and then add all these intervals to R. Afterwards, two interval sequences $X' = X[1 .. j' - 1]$ and $Y' = Y[1 .. |Y| - 1]$ will be figured out for the next recursive call.

Part 4 (line 10). In this part, we call $B_1(X, Y, t, I_t, R)$ to handle the case $J_m \supset I_t$. (This case occurs only when $b = 0$.)

Part 5 (line 11). In this part, we call $B_0(X, Y, t, I, R)$ to handle the case $J_m \subset I_t$. (This case occurs only when $b = 1$.)

Part 6 (line 12 – 13). In this part, we will apply a recursive call to the interval sequences figured out in the previous computation (i.e, in a part i for $i \in \{2, ..., 5\}$). In terms of whether $|Y'| \leq |X'|$ or $|Y'| > |X'|$, the new value of b for the recursive call is set to be the same as the old value of b (see line 12) or changed to its negation (see line 13).

Example 1 Consider L_2 = [1, 4][5, 11] and L_5 = [1, 1][3, 3][5, 5][8, 8][13, 13][17, 18]. By calling *setIntersect*(L_5, L_2, 0), the following operations will be conducted:

Step 1: check L_2[2] = [5, 11] against L_5. $l = \left\lfloor \log\frac{6}{2} \right\rfloor = 1$, $t = n - 2^l + 1 = 6 - 2 + 1 = 5$, L_5[5] = [13, 13]. Since [8, 11] $\prec$ [13, 13], we will make a recursive call *setIntersect*(L_5[1 .. 4], L_2, 0).

Step 2: In the execution of *setIntersect*(L_5[1 .. 4], L_2, 0), check L_2[2] = [5, 11] against L_5[1 .. 4]. $l = \left\lfloor \log\frac{4}{2} \right\rfloor = 1$, $t = n - 2^l + 1 = 4 - 2 + 1 = 3$, L_5[3] = [5, 5]. Since [5, 5] $\subset$ [5, 11], we will call *Search*(L_5[1 .. 4], 3, [5, 11]), which returns $\langle j', j'' \rangle = \langle 3, 4 \rangle$. Then, we will recursively call *setIntersect*(L_5[1 .. 2], L_2[1 .. 1], 0) in the next step.

Step 3: check L_2[1] = [1, 4] against L_5[1 .. 2]. $l = \left\lfloor \lg\frac{2}{1} \right\rfloor = 1$, $t = 3 - 2^1 + 1 = 2$, L_5[2] = [3, 3]. Since [3, 3] $\subset$ [1, 4], we will call *Search*(L_5[1 .. 2], 2, [1, 4])]), which returns $\langle j', j'' \rangle = \langle 1, 2 \rangle$. Then, we will recursively call *setIntersect*(L_5[1 .. 0], L_2[1 .. 0], 0) in the next step.

Step 4: $|L_2[1 .. 0]| = 0$. Stop.

The result R = [1, 1][3, 3][5, 5][8, 8]. □

5. IMPROVEMENTS

In this section, we discuss an improvement of the algorithm discussed in the previous section. First, we show how *LCAs* (least common ancestors) can be used to speed up the binary search of a sorted interval sequence in 5.1. Then, in 5.2, we discuss how *LCAs* can be efficiently figured out.

5.1 Integrating *LCAs* into binary search

First of all, we notice that the time complexity of *setIntersect*(X, Y, b) is bounded by $O(|Y| \cdot \log|X|)$ with $|X| \geq |Y|$ since in the worst case *Search*(X, t, I) requires $O(\log|X|)$ time, where $1 \leq t \leq |X|$ and I is an interval in Y. However, by using *LCAs*, both *Search*(X, t, I) and *binarySearch*(L, I, b), can be non-trivially improved.

- *LCA sequences*

Denote by V_x all the nodes labeled with x in T. All the *LCAs* of the nodes in V_x, denoted as V_x', can be efficiently recognized using a way to be discussed in 5.2. For example, for the set of nodes labeled with number 5: $V_5 = \{v_{10}, v_5, v_{11}, v_{18}, v_{14}, v_9\}$, we can find another set of nodes: $V_5' = \{v_1, v_6, v_2, v_0\}$ with v_1 being the *LCA* of $\{v_{10}, v_5\}$, v_6 the *LCA* of $\{v_{11}, v_{18}\}$, v_2 the *LCA* of $\{v_{11}, v_{18}, v_{14}\}$, and v_0 the *LCA* of $\{v_{10}, v_5, v_{11}, v_{18}, v_{14}, v_9\}$. Now we construct a tree structure, called an *LCA-tree* and denoted as T_x, which contains all the nodes in $V_x \cup V_x'$. In T_x, 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 $V_x \cup V_x'$. In Fig. 7(a), we show T_5 for illustration.

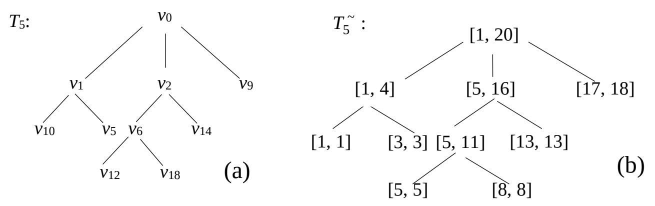


Fig. 7: Illustration for T_x and $T_x^{\sim}$.

Replacing each node in T_x with the corresponding interval, we get another tree, denoted as $T_x^{\sim}$, 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 $T_x^{\sim}$ [v] (the subtree rooted at v in $T_x^{\sim}$). See $T_5^{\sim}$ 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, 11] is the smallest interval covering [5, 5] and [8, 8]; and [5, 16] is the smallest interval covering [5, 5], [8, 8] and [13, 13]. Finally, [1, 20] is the smallest interval covering all the intervals in L_5: [1, 1], [3, 3], [5, 5], [8, 8], [13, 13], [17, 18]. Here, our intention is to associate each interval I_j in X with a second interval, which is the parent of I_j in $T_x^{\sim}$, denoted as $c(j)$. For this purpose, we will keep a sequence Γ_x containing all the LCA-intervals in the post-order of $T_x^{\sim}$. (This can be obtained by traversing $T_x^{\sim}$ in post-order, but with all the leaf nodes removed). For example, $\Gamma_5 = \gamma_1\gamma_2\gamma_3\gamma_4$ = [1, 4][8, 11][5, 16][1, 20]. Each γ in Γ_x will be associated with two links, denoted as $l(\gamma)$ and $r(\gamma)$, pointing to two intervals in L_x, which are respectively the left-most and right-most leaf nodes in $T_x^{\sim}$ [γ] (the subtree rooted at γ in $T_x^{\sim}$). Fig. 8 helps for illustration.

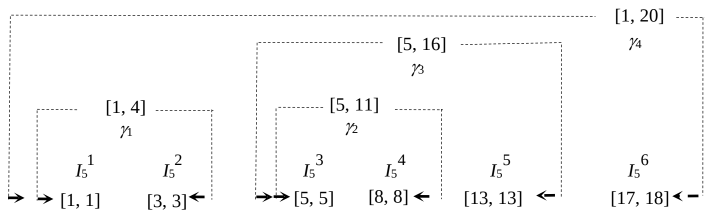


Fig. 8: Illustration for links associated with intervals in $T_x^{\sim}$.

In Fig. 8, I_5^6 = [17, 18] is associated with an LCA-interval $c(6) = \gamma_4$ = [1, 20], which is the parent of I_5^6 in the corresponding $T_x^{\sim}$ shown in Fig. 7(b). In addition, $l(\gamma_4)$ is a link pointing to I_5^1 and $r(\gamma_4)$ is a link pointing to I_5^6 . They are respectively the left-most and the right-most interval in L_5 covered by γ_4. In the same way, we can check all the other intervals and links shown in Fig. 8.

- *Improving search( ) by using LCAs*

By using LCA intervals and the corresponding links, *search*(X, t, I) should be changed to *search*(X, Γ_x, t, I), which can be done more efficiently as follows.

1. Let $X = I_1, I_2, \ldots, I_n$. Let Γ_x be the corresponding LCA interval sequence. Assume that $1 \leq t \leq n$ such that $I_t \subset I$ in Y.
2. Compare I and $c(t) = \gamma_i$ for some i, where $c(t)$ represents the LCA-interval of I_t.
   i) If $\gamma_i \supset I$, return $<t, t>$.
   ii) If $\gamma_i = I$, return $<l(\gamma_i), r(\gamma_i)>$.

iii) If $\gamma_i \subset I$, search the intervals to the right of γ_i in Γ_x to find a largest $f \geq i$ such that γ_f is covered by I. Return $<l(\gamma_f),\ r(\gamma_f)>$.

3. In the first two cases of (2) (i.e., in 2-(i) and 2-(ii)), Γ_x will be changed to $\Gamma_x[1\ ..\ k]$, which will be used for the next recursive execution of *setIntersect*( ), k is the position just prior to γ_i in Γ_x. In the third case of (2), Γ_x will be changed to $\Gamma_x[1\ ..\ g]$, where g is the position just prior to γ_f in Γ_x.

Special attention should be paid to (2).

In the case of $\gamma_i \supset I$, I_t must be the unique interval, which can be covered by I. Therefore, $<t, t>$ is simply returned.

In the case of $\gamma_i = I$, I can only cover all those intervals between $l(\gamma_i)$ and $r(\gamma_i)$ (including $l(\gamma_i)$, $r(\gamma_i)$) in X. So $<l(\gamma_i), r(\gamma_i)>$ will be returned.

In the case of $\gamma_i \subset I$, we will try to find a highest interval (in $T_x^{\sim}$) which can be covered in I. This can be found only among those intervals to the right of γ_i (if any) since Γ_x is an interval sequence in the post-order of $T_x^{\sim}$. (Recall that by using the original *search*(X, t, I) we may need to search the whole X.)

In this way, the time complexity of *Search*(X, Γ_x, t, I) is dramatically decreased. First, in the case of $c(t) = \gamma_i \supset I$ or $\gamma_i = I$, no search of X is performed at all. Secondly, in the case of $\gamma_t \subset I$, only part of Γ_x (to the right of γ_i) is explored. Using the traditional binary search, the time for this task must be bounded by $O(\log|X|/|Y|)$ since we always have $|\Gamma_x| \leq |X|$.

Example 2 To see how the *LCA*s can be used to skip over useless checks, we check several single intervals against L_5 in Fig. 8 to demonstrate the working process.

- Assume that $I = [13, 15]$ is compared with $I_5^5 = [13, 13]$ in L_5. We have $[13, 13] \subset [13, 15]$. Since $c(5) = \gamma_3 = [5, 16] \supset I = [13, 15]$, we immediately know that $I_5^5 = [13, 13]$ is the only intervals which can be covered by $I = [13, 15]$ and simply return $<5, 5>$.
- Assume that $I = [5, 11]$ is compared with $I_5^4 = [8, 8]$ in L_5. We have $[8, 8] \subset [5, 11]$. However, $I = c(4) = \gamma_2 = [5, 11]$. We will return $<l(\gamma_2), r(\gamma_2)> = <3, 4>$. No further search is necessary.
- Assume that $I = [5, 16]$ is compared with $I_5^4 = [8, 8]$ in L_5. We have $[8, 8] \subset [5, 16]$. But we also have $c(4) = \gamma_2 = [5, 11] \subset [5, 16]$. So we will search part of Γ_5 to the right of $[5, 11]$ (it is a subsequence: $[5, 16][1, 20]$) to find a largest interval which can covered by $I = [5, 16]$. It is $\gamma_3 = [5, 16]$. So we return $<l(\gamma_3), r(\gamma_3)> = <3, 5>$. □

- *Improving binarySearch( ) by using LCAs*

We need further change *binarySearch*(X, I, b) to *binarySearch*(X, Γ_x, I, b), by which the search can also be done more efficiently by using *LCA*-intervals.

1. Let $X = I_1, I_2, \ldots, I_n$. Let $\Gamma_x = \gamma_1, \ldots, \gamma_j$ be the corresponding *LCA* interval sequence. Let $t = \lfloor n/2 \rfloor$. Compare I and I_t.
2. If $b = 0$ and $I \supset I_t$, return t.
3. If $b = 1$ and $I \subset I_t$, return t.
4. If $I \prec I_t$, compare I and $c(t) = \gamma_i$ for some $1 \leq i \leq |\Gamma_x|$, where $c(t)$ represents the *LCA*-interval of I_t. If $I \not\subset \gamma_i$, we will explore $X[1\ ..\ l(\gamma_i) - 1]$ in a next step; otherwise, $X[l(\gamma_i)\ ..\ t - 1]$.
5. If $I_t \prec I$, also compare I and $c(t) = \gamma_i$ for some $1 \leq i \leq |\Gamma_x|$. If $I \not\subset \gamma_i$, we will explore $X[r(\gamma_i) + 1\ ..\ n]$ in a next step; otherwise, $X[t + 1\ ..\ r(\gamma_i)]$.

In the above process, if $b = 0$ we will check whether $I \supset I_t$ (line 2) while if $b = 1$ we will check whether $I \subset I_t$ (line 3). However, for the case $I \prec I_t$ (line 4) or $I_t \prec I$ (line 5), the *LCA*-intervals can be used to control the binary search in the same manner no matter whether $b = 0$ or $b = 1$, due to the following two lemmas.

Lemma 5 Let I be an interval in an interval sequence L and γ its *LCA*-interval. Let I' be another interval such that $I' \prec I$ and $I' \not\subset \gamma$. Then, for any interval I'' between $l(\gamma)$ and $r(\gamma)$ (including $l(\gamma)$ and $r(\gamma)$), we have neither $I'' \supset I'$, nor $I'' \subset I'$.

Proof. If $I'' \supset I'$, we would have $I' \subset \gamma$ since $I'' \subset \gamma$. This contradicts the fact that $I' \not\subset \gamma$. If $I'' \subset I'$, we would have $I' \subset \gamma$ or $\gamma \subset I'$. Since $I' \not\subset \gamma$, we would have $\gamma \subset I'$, which contradicts the fact that $I' \prec I$. □

Lemma 6 Let I be an interval in an interval sequence L and γ its *LCA*-interval. Let I' be another interval such that $I \prec I'$ and $I' \not\subset \gamma$. Then, for any interval I'' between $l(\gamma)$ and $r(\gamma)$ (including $l(\gamma)$ and $r(\gamma)$), we have neither $I'' \supset I'$, nor $I'' \subset I'$.

Proof. The lemma can be proved in a way similar to Lemma 5. □

In this way, the time for the binary search of an interval sequence X can be reduced to $O(\log|\Gamma_x| + \log\lambda)$, where λ is the largest number of intervals in X, which share the same *LCA*. We always have $|\Gamma_x| \leq |X|$ and $\lambda \leq |X|$. Especially, if $|\Gamma_x| \geq 2$, $|\Gamma_x| < |X|$ and $\lambda < |X|$.

Finally, part 1 (line 4) in *setIntersect*( ) should be accordingly slightly changed to use *LCA*-intervals to speed up the process as described below.

```
if $I \prec I_t$ then { if $I \not\subset c(t)$ then $X' \leftarrow X[1 .. l(c(t)) - 1]$ else $X' \leftarrow X[1 .. t - 1]$; $Y' \leftarrow Y$; }
```

From the above discussion, we can clearly see that *LCAs* are quite useful for speeding up the operation. However, all of them have to be first efficiently recognized. In the next Subsection, we address this issue in great detail.

5.2 Construction of *LCA*-trees

For constructing T_x, the *LCAs* for all the nodes labeled with x (in T) have to be recognized. A simple approach is to search T for each x, which obviously needs $O(|S|\cdot|T|)$ time, where $S = \bigcup_{i=1}^{M} S_i$. But we will describe an algorithm, whose time complexity is bounded by $O(|T|)$.

For this purpose, we will search T bottom-up, and the nodes labeled with different x's and the corresponding *LCAs* will be inserted into different T_x's. We will attach each node v with two links when inserted into a T_x, denoted as *parent*(v) and *left-sibling*(v), respectively. *parent*(v) is used to point to the parent of v in T_x while *left-sibling*(v) points to a node in T_x inserted just before v, which is not a descendant of v in T (of course, not in T_x, either). Concretely, the following operations will be conducted.

(i) Let v be the node currently inserted into a T_x.

(ii) If v is not the first node inserted into T_x, we do the following:

Let v' be the node inserted into T_x just before v. If v' is not a child (descendant) of v, create a link from v to v', denoted as *left-sibling*(v) = v'. If v' is a child (descendant) of v, we will first create a link from v' to v, denoted as *parent*(v') = v. Then, v must be an *LAC* of some nodes labeled w. We will go along the left-sibling chain starting from v' until we meet a node v'' which is not a child (descendant) of v in T_w. For each encountered node u except v'', set *parent*(u) $\leftarrow v$. Finally, set *left-sibling*(v) $\leftarrow v'$.

Fig. 9 is a pictorial illustration of this process.

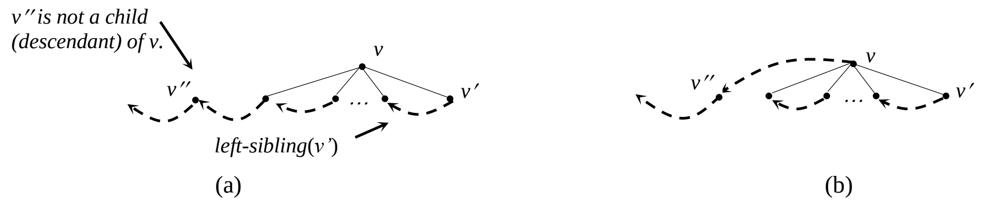


Fig. 9. Illustration for the construction of a T_x.

In Fig. 9(a), we show the navigation along a left-sibling chain starting from v' when we find that v' is a child (descendant) of v. This process stops whenever we meet v'', a node that is not a child (descendant) of v. Fig. 9(b) shows that the left-sibling link of v is set to point to v'', which is previously pointed to by the left-sibling link of v's left-most child.

Combining the above process with a bottom-up search of T, we get an efficient algorithm *find-LCA*(T) (see below) for finding all the *LCAs*.

```
ALGORITHM find-LCA(T)
begin
1. For each $w \in S$, $T_w \leftarrow \emptyset$.
2. Let $u$ (labeled with an integer $x$) be the first node encountered during the bottom-up
   searching of $T$. Insert $u$ in $T_x$.
3. Let $v$ be the currently encountered node in $T$ and labeled with an integer $y$. Let $v'$ be the
   node visited just before $v$. Do (4) or (5), depending on whether $v$ is the parent of $v'$ or not.
4. If $v$ is not the parent of $v'$, then insert $v$ into $T_y$.
5. If $v$ is a parent of $v'$, then for each child $z$ of $v$ in $T$, let $K(z)$ be the set of LCA-trees, into
   which $z$ is inserted. Then, for each $z$ and for each $T' \in K(z)$, we will go along a left-sibling
   chain starting from $z$ in $T'$ until we meet a node $v''$ which is not a child (descendant) of $v$
   in $T'$. If the number of the nodes encountered on this navigation along the left-sibling
   chain is larger than 1, insert $v$ into $T'$, set parent($u$) $\leftarrow v$ for each $u$ on the chain, and set
   left-sibling($v$) $\leftarrow v''$; otherwise, $v$ will not be inserted into $T'$.
end
```

In the algorithm, special attention should be paid to (5), by which the parent/child relationship of the nodes in each *LCA*-tree T' is established. Denote by $d_{T'}(v)$ the outdegree of a node v in T'. Since for each node v added to T' as a parent of some other nodes a left-sibling chain will be navigated, $O(d_{T'}(v))$ time is required for this task. Therefore, the time used for establishing a single T' is bounded by

$$\sum_{v \in T'} d_{T'}(v) = O(|T'|).$$

On the other hand, since for each internal node v in T' we have $d_{T'}(v) \geq 2$, $|T'|$ must be $\leq 2|leaves(T')|$, where $leaves(T')$ represents all the leaf nodes of T' and must be exactly all those nodes labeled with a same integer in T. Thus, the time complexity for constructing all the *LCA*-trees must be bounded by

$$\sum_{all\, T'} O(|T'|) \leq \sum_{all\, T'} O(2\,|leaves(T')|) = O(2\,|T|) = O(|T|).$$

Example 3 Consider the *trie* T shown in Fig. 1(d). Applying the above algorithm to T, we will generate a series of subtrees as illustrated in Fig. 10. Due to space limitation, only first 7 steps are traced.

In step 1, node v_{10} labeled 5 is met and inserted into T_5. In step 2, v_4 labeled 6 is encountered and inserted into T_6. But it is not inserted into T_5 since v_{10} is its unique child. In step 3, we meet v_3 labeled 5. Since it is not an ancestor of v_{10}, a link *left-sibling*(v_3) = v_{10} is created. In step 4, v_1 labeled 2 is met and inserted into T_2. At the same time, it is also inserted into T_5 since v_3 is a child of v_1 and when we navigate along the *left-sibling* chain starting from v_3, we will meet v_{10} which is a descendant of v_1. In step 5, we meet v_{11} labeled with 5. Since it does not have any child, it will be inserted into T_5, and a link *left-sibling*$(v_{11}) = v_1$ will also be created. In step 6, we will meet v_{12} labeled 4 and insert it into T_4. In step 7, v_{17} labeled 4 is encountered and inserted into T_4 with a link *left-sibling*$(v_{17}) = v_{11}$ being created. □

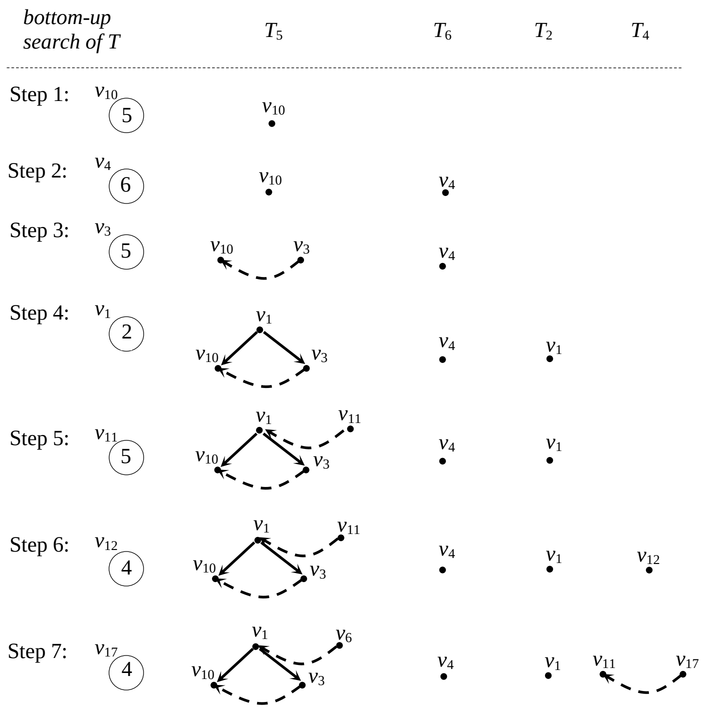


Fig. 10. Sample trace.

6. CORRECTNESS AND TIME COMPLEXITY

In this section, we prove the correctness of *setIntersect*(L_x, L_y, b) ($|L_x| \geq |L_y|$) with *LCAs*, and analyze its time complexity .

Lemma 7 Let u and v be two nodes in T such that $I_u \prec I_v$. Let u' and v' be their respective *LCAs*. Then, u' and v' must be in one of three relationships: u' and v' are identical, $I_{u'} \prec I_{v'}$, or u' is an ancestor of v'.

Proof. Since $I_u \prec I_v$, it is possible that u' and v' are identical, or $I_{u'} \prec I_{v'}$. If u' and v' are not identical, nor $I_{u'} \prec I_{v'}$ holds, u' must be an ancestor of v'. Otherwise, $I_{v'} \prec I_{u'}$ (i.e., $I_{v'}[2] < I_{u'}[1]$). This contradicts the fact that $I_{u'} \supset I_u$, which implies that $I_{u'}[1] < I_u[1] \leq I_u[2] < I_{v'}[2]$. □

Corollary 1 Let u and v be two nodes labeled with the same integer x in T such that $I_u \prec I_v$. Then, if in Γ_x the *LCA* u' of u appears after the *LCA* v' of v, u' must be an ancestor of v'. □

Based on the above Corollary, we can immediately get the correctness of the modified *Search*(X, t, I). We recall that by the original *Search*(X, t, I), for $I \supset I_t$, we will try to find a smallest $j' \leq t$ and a largest $j'' \geq t$ such that all the intervals in X between j' and j'' (including j', j'') can be contained in I. By the modified *Search*(X, t, I), we use the *LCA*-intervals in Γ_x and distinguish among three cases: $c(t) \supset I$, $c(t) = I$, and $c(t) \subset I$, where $c(t)$ represents the *LCA* of I_t in Γ_x. In the first case, we simply return $<t, t>$ since I_t is the only interval in X which can be contained in I. In the second case, we will return $<l(c(t)), r(c(t))>$. This is obviously correct. In the third case, we will search part of Γ_x to the right of $c(t)$ to find the last interval γ which is still containable in I and return $<l(\gamma), r(\gamma)>$. The correctness of this process is guaranteed by the Corollary. Finally, for each case, Γ_x is shortened for efficiency. For the first two cases, the subsequence starting from $c(t)$) to the end of Γ_x is cut off. For the third case, we take away all the intervals starting γ to the end from Γ_x. This will definitely not impact the correctness since we proceed for X and Γ_x from right to left.

Lemma 8 The modified *Search*(X, Γ_x, t, I) (with $I \supset I_t$) will find a smallest $j' \leq t$ and a largest $j'' \geq t$ such that all the intervals in X between j' and j'' (including j', j'') can be contained in I.

Proof. See the above analysis. □

Proposition 2 Let X and Y be two interval sequences with $|X| \geq |Y|$. *setIntersect*(X, Y, b) with *LCAs* will return a correct answer.

Proof. To prove the correctness of *setIntersect*(X, Y, b) with *LCAs* , where $|X| \geq |Y|$, we simply check all the following four cases one by one.

Case 1: $J_m \prec I_t$ (line 4),

Case 2: $I_t \prec J_m$ (lines 5 – 9),

Case 3: $J_m \supset I_t$ (line 10), and

Case 4: $J_m \subset I_t$ (line 11).

In Case 1, the problem is reduced. Depending on whether $J_m \subset c(t)$, it is reduced to $X' = X[1 .. t - 1]$ and $Y' = Y$; or $X' = X[1 .. l(c(t)) - 1]$ and $Y' = Y$. This is obviously correct.

In Case 2, we first try to find an interval in the part to the right of I_t in X, which can be contained in J_m or contain J_m (depending on whether $b = 1$ or $b = 0$), by using the modified *binarySearch*(L, Γ, J_m, b). According to Lemma 4 and 5, it is correct. If such an interval can be found, the problem is reduced to $X' = X$ and $Y' = Y[1 .. m\text{-}1]$. Otherwise, depending on the value of b, we call $B_1(\,)$ or $B_0(\,)$. $B_1(\,)$ is simply correct. $B_0(\,)$ is correct in terms of Lemma 7.

In Case 3, we call $B_0(\,)$. In Case 4, we call $B_1(\,)$. According to the analysis of Case 2, we can see that these two cases are also correctly handled. This completes the proof of the proposition. □

Lemma 9 Let $m = |Y|$ and $n = |X|$. Assume that $m \leq n$. The time complexity of *setIntersect*(L_x, L_y, b) with *LCAs* is bounded by $O((1 + 2l)m)$, where $l = \left\lfloor \log \frac{n}{m} \right\rfloor$.

Proof. We prove the proposition by induction on $h = m + n$.

Basis. When $h = 1, 2$, the proposition trivially holds.

Hypothesis. Assume that the proposition holds when $h \leq k$. We will prove that when $h = k + 1$, the proposition also holds.

Denote by $\alpha(m, n)$ the number of comparisons made by the algorithm. Then, According to the four cases checked in the algorithm, we have

1. $J_m \prec I_t$: $\alpha(m, n) = 1 + \alpha(m, n - 2^l - 1)$;
2. $I_t \prec J_m$: $\alpha(m, n) = 1 + 2l + \alpha(m - 1, n)$, where $2l$ is due to the two searches: one in X and one in Γ_x;
3. $J_m \supset I_t$: $\alpha(m, n) = 1 + l + \alpha(m - 1, n - 2^l - 1)$, where l is due to the search in Γ_x; and
4. $J_m \subset I_t$: $\alpha(m, n) = 1 + \alpha(m - 1, n)$,

where $t = n - 2^l - 1$.

(3) and (4) are obviously smaller than (1) or (2). So we need only to solve recursions (1) and (2). To solve the recursion (1), we represent n as $2^l m + \lambda$ with $0 \leq \lambda < m$. Thus, we have

$$n - 2^l - 1 = 2^l m + \lambda - 2^l - 1.$$

If $\lambda - 2^l - 1 < 0$, $n - 2^l - 1 = 2^{(l-1)}m + \lambda'$, where $\lambda' = m + \lambda - 2^l - 1 < m$. By induction, $\alpha(m, n) = 1 + \alpha(m, n - 2^l - 1) = (1 + 2(l - 1))m = (-1 + 2l)m \leq (1 + 2l)m$. In the case of $\lambda - 2^l - 1 \geq 0$, we must have $\lambda - 2^l - 1 < \lambda < m$. By induction, the proposition still holds.

For recursion (2), we have

$$1 + 2l + \alpha(m - 1, n) = 1 + 2l + (1 + 2l)(m - 1) = (1 + 2l)m,$$

which shows that the proposition also holds in this case. □

Finally, we notice that by using the modified *binarySearch*( ) (see 4.1), the running time to search $X[t + 1 .. |X|]$ is actually bounded by $O(\log|\Gamma_x[c(t) .. |\Gamma_x|]| + \log\lambda)$ (instead of $O(\lfloor \log \frac{n}{m} \rfloor)$), where we use $c(t)$ to represent the position of I_t's *LCA*-interval in Γ_x, and λ is the largest number of intervals in $X[t + 1 .. |L_x|]$, which share the same *LCA*; and in general we have $|\Gamma_x[c(t) .. |\Gamma_x|]|$ and λ both smaller than $|X[t + 1 .. |X|]| = \frac{n}{m}$. So we have the following proposition on the time complexity of our algorithm.

Proposition 3 Let X and Y be two interval sequences with $|Y| = m \leq |X| = n$. The time complexity of the modified *setIntersect*(X, Y, b) with *LCAs* is bounded by $O(m \cdot \log\kappa)$, where $\kappa < \frac{n}{m}$.

Proof. See Lemma 9 and the above analysis. □

Finally, we consider a combined sequence L formed by inserting all the intervals of Y into X in such a way that for any two intervals I and I' in L if I is to the left of I' or $I \subseteq I'$, I appears before I' in L. Then, there are $\binom{m+n}{n}$ possible placements of the intervals of Y in the combined sequence; it follows that $\left\lceil \lg\binom{m+n}{n} \right\rceil$ comparisons are necessary to distinguish these possible orderings. Since each of such combined sequences corresponds to an intersection of Y and X, we can take $\left\lceil \lg\binom{m+n}{n} \right\rceil = \Omega\left(m \cdot \lg\frac{n}{m}\right)$ as a lower bound of the problem. In this sense, our algorithm reaches the optimality.

7. EXPERIMENTS

In order to show that our method has not only the best theoretical time complexity, but also works quite well in practice, we have made a bunch of tests.

In the experiments, we have tested seven methods:

- *Hwang-Lin's* [57] (*HL* for short),
- *Baeza-Yates's* [5] (*BY* for short),
- *Barbay*-Ortiz-Lu's [7] (*BOL* for short),
- *Hashing-based* (*RanGroupScan* in [26]; *Hb* for short),
- *Skip-list-based* [43] (*SkipL* for short),
- *setIntersect* (discussed in the paper; *sI* for short),
- *setIntersect with LCAs* (discussed in the paper; *sIL* for short).

All our experiments are performed on a 64-bit Windows operating system. The processor is Intel Core(TM) i5-3210M CPU @ 2.50GHZ with 8GB 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. For all the tests the indexes are put entirely in memory.

- *Data Sets*

To evaluate the algorithms, we use both synthetic and real data.

For the experiments with synthetic data, we first create a vocabulary containing 3 million words (short strings each containing less than 20 characters). Then, we randomly choose words to form a document. The length of each document is between 500 and 1000 words.

For the experiments with real data, we use the TREC GOV2 corpus. The characteristics of this collection are shown in Table 2.

Table 2: Characteristics of Wikipedia Data

	TREC GOV2
Documents (in HTML or PDF)	25,197,000
Size (gigabytes)	360
Word occurrences (without markup)	38,515,000

The corpus is associated with a query log containing 100,000 queries. For this test, we randomly choose 6000 queries which contain more than one key word. Table 3 shows the distribution of the numbers of key words in queries.

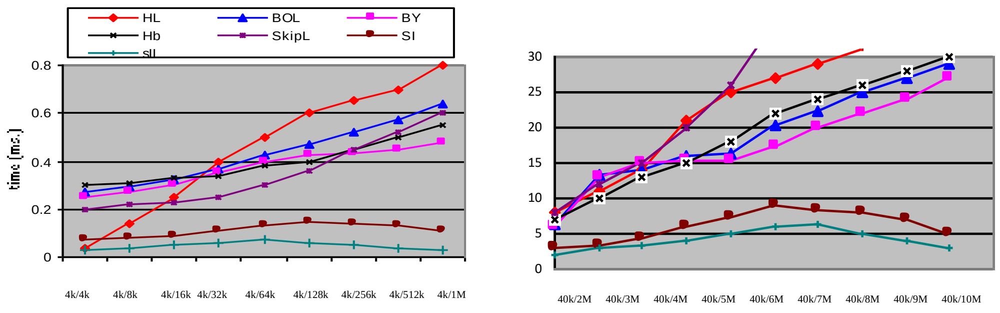
Table 3: Distribution of key word numbers in queries

#words	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
#queries	1549	1362	1053	873	546	365	89	63	51	24	5	11	2	4	1	1	1

7.1 Test on synthetic data sets

- *Two word queries with varying list sizes*

First, we test two-word queries with varying list sizes. We use the synthetic data. The results are shown in Fig. 11, where the time is measured in milliseconds. In Fig. 11(a), we demonstrate the results over short lists ranging from 4k/4k (both lists taking parting in the operation are of length 4k integers) to 4k/1M (one of the lists contains 4k integers while the other 1 million integers). In Fig. 11(b), we show the results over long lists ranging from 40k/2M to 40k/10M.



(a) (b)

Fig. 11: Two word queries over synthetic data.

From these two figures, we can see that our methods uniformly outperform the other strategies. Even the binary search of intervals without *LCA*s works better than the others. Especially, as the length of inverted lists increases, the running time of our methods decreases. For short inverted lists (Fig. 11(a)), no significant difference among *hash-based*, *SkipList*, and all the adaptive methods can be observed. However, for large inverted lists (Fig. 11(b)), we can clearly see that *SkipList* has the worst performance. We can also see that Hwang and Lin's is slightly worse than Baeza-Yates's, Barbay-Ortiz-Lu's, and *hash-based* methods.

In Fig. 12, we show the ratios of the interval sequence sizes over their corresponding inverted lists, which is obtained by dividing all the inverted lists into ten groups according to theirs sizes and calculating the average size of each group and the average size of the corresponding interval sequences.

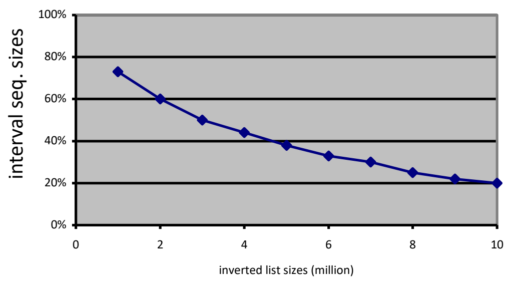


Fig. 12: Ratio of intervals sequences over inverted lists.

This figure demonstrates that as the size of inverted lists increases the size of the corresponding interval sequences become shorter. Together with the binary search and the use of *LACs*, this property makes our methods superior.

- *Tests on queries with varying number of key words*

In this experiment, we vary the number k of words in a query with k = 2, 3, …, 7. For each query, the words are chosen randomly, but with a control being imposed so that the size of the inverted lists associated with each word is larger than 100,000 since for short lists all the algorithms work fast (as shown in Fig. 11(a), they all run within one millisecond) and no significant difference can be observed.

In Fig. 13, we report the test results on the queries with 2, 3, and 4 key words while in Fig. 14 on the queries with 5, 6, and 7 key words.

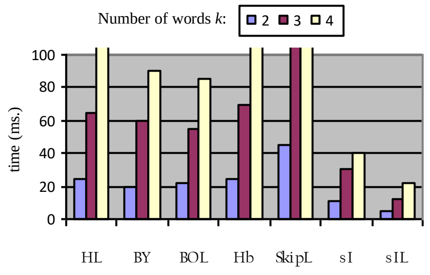


Fig. 13: Test results on varying number of words in a query.

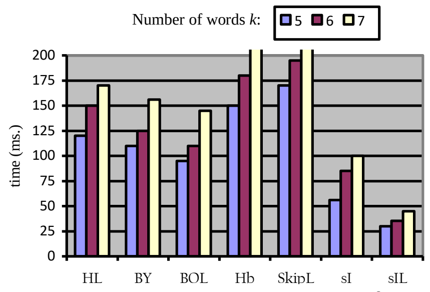


Fig. 14: Test results on varying number of words in a query.

From these two figures, we can see that the *SkipList*, and the *Hash-based* both perform poorly. The reason for this it is due to the auxiliary data (such as new hash values) or new data structures (such as new skip lists) that have to be established for the intermediate results. In the opposite, for all the remaining algorithms, no such a task is required. Even though for our algorithm with *LCAs* extra data structures are used, they are automatically changed for the intermediate results and no extra effort is required to reproduce them.

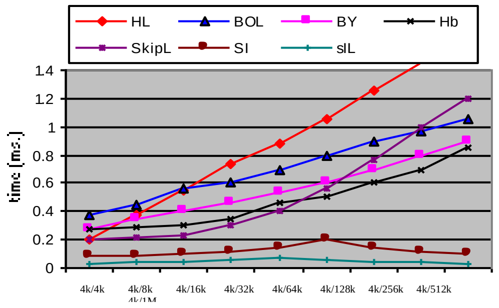
So both our algorithms work much better than the *SkipList* and the *Hash-based* when more words are involved in a query. They are also better than all the other three methods since much shorter interval sequences are checked.

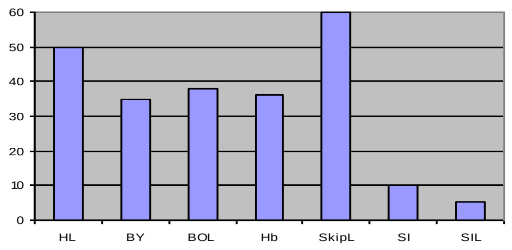
7.2 Tests on real data sets

In this experiment, we use TREC GOV2 corpus. The most important characteristics of this text corpus are given in Table 2. Over this database, the same tests are performed as over the synthetic database. However, for the tests on the two-word queries, we have evaluated queries from the query log, categorized into four groups with each containing 25 queries. In the first three groups, both the words are with high appearance frequency, middle appearance frequency, and low appearance frequency, respectively; and the ratio $|\mathfrak{I}_w|/|\mathfrak{I}_{w'}|$ for all the queries is set to be between 1.0 and 1.22, where $\mathfrak{I}_w$ represents the inverted list associated with w. In the fourth group, the appearance frequencies of the two words in a query is greatly different with the ratio $|\mathfrak{I}_w|/|\mathfrak{I}_{w'}| \geq 8$.

For the queries with varying number of words, denoted by w_i the ith word in a query and $\mathfrak{I}_i$ the inverted list associated with w_i. In all the queries, the words are ordered such that $|\mathfrak{I}_i| \leq |\mathfrak{I}_{i+1}|$. For all the queries tested, the ratio $|\mathfrak{I}_{i+1}|/|\mathfrak{I}_i|$ is between 1.0 and 2.0. But for our methods, the words are reordered according to the length of interval sequences.

In Fig. 15, we show the average time of each strategy on two-word queries. In Fig. 15(a) we show the running time of queries over short lists while in Fig. 15(b) we show the average time of all the tested queries.





(a) (b)

Fig. 15: Two word queries over real data.

In Fig. 16 and 17, we show the test results on the queries with 2, 3, 4 words, and the queries with 5, 6, 7 words, respectively.

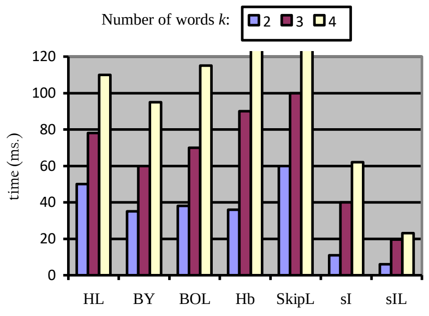


Fig. 16: Test results on queries with 2, 3, 4 words.

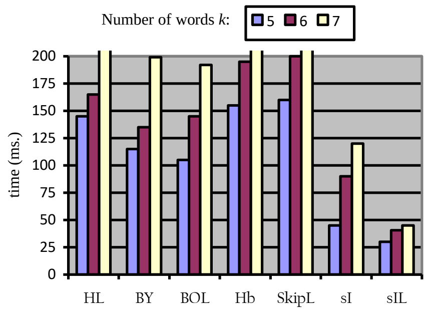


Fig. 17: Test results on queries with 5, 6, 7 words.

From these figures, we can see that for the two-word queries Hwang and Lin's is slightly worse than Baeza-Yates's, but Barbay-Ortiz-Lu's has a comparable performance. They all are much better than the *Hash-based* and the *SkipLst*. Again, for the queries with more than two words, the performance of the *Hash-based* degrades due to the recalculation of hash values for the intermediate results. For a similar reason, the *SkipList* also works poorly.

Our algorithms still work best, consistent with our theoretical time analysis.

7.3 Space requirements

The high performance of our methods is at cost of extra space for storing interval sequence. However, in comparison with some other methods with auxiliary data structure, like the *SkipList*, and the *Hash-based*, our method does not impose higher space requirements than them. Especially, by using different integer encoding schemes, such as δ-coding, γ-coding (p. 116 in [60]), or *Golomb*-coding [29] for integers (gaps between consecutive document IDs) in inverted lists and interval sequences, the problem can be further mitigated to some extent.

On the synthetic data sets, we have tested several algorithms with the *Golomb*-coding being used. The results are shown in Fig. 18 and 19. In Fig. 18, we show the space requirements. From this, we can see that our method with no *LCAs* uses a little bit less space than the *SkipList*, more space than the *Hash-based* and *Baeza-Yates's*. Particularly, *Baeza-Yates's* uses only simple (compressed) inverted list, and therefore has the lowest space requirement. Our method with *LCAs* requires most space. Its storage requirement is between 1.1 and 1.5 times of the size of the compressed inverted lists and between 1.05 and 1.21 times of the size of the *Hash-based* algorithm.

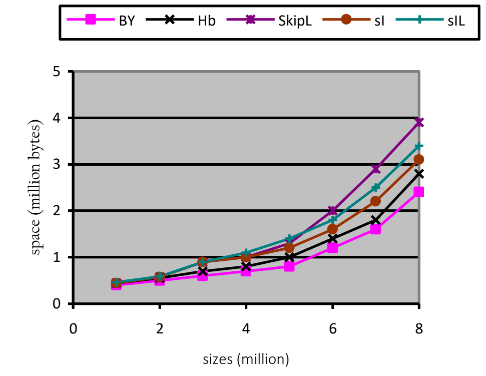


Fig. 18: Space requirements.

In Fig. 19, we show the running time. Comparing this figure and Fig. 13, we can clearly see that more time is required for every algorithm to evaluate the same kind of queries.

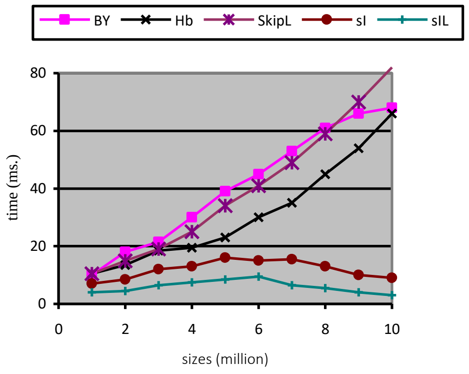
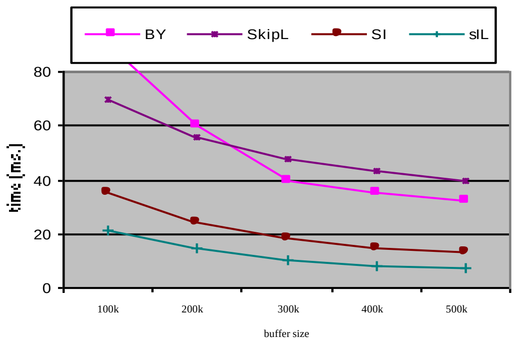


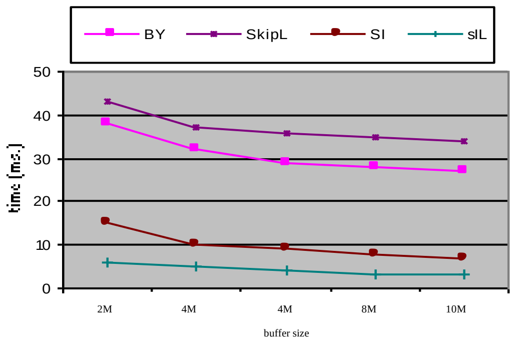
Fig. 19: Running time.

7.4 Indexes on disk

In all the above tests, all the indexes are maintained in main memory. However, indexes are normally disk-resident. When a query arrives, the corresponding inverted lists or the corresponding interval sequences will be taken from hard disk into main memory. This can be done quickly by using a hash table. In order to check the impact of space usage on query time, we have done two more experiments. One is with small buffer sizes, and the other is with large ones. Altogether four methods are tested in such an environment: *SkipList*, *Baeza-Yates's*, *interval sequence with* and *without LCAs*. In the tests, we have run 500 two-word queries (from the query log) with each word associated with an inverted list of length larger than 100,000. The average running times are shown in Fig. 20(a) and (b).



(a)



(b)

Fig. 20: Two word queries over disk-residential indexes.

From Fig. 20(a), we can see that the small buffer sizes do not impact much our methods. It is because for a long inverted list the corresponding interval sequence tends short. In the opposite, the impact of small buffer sizes for *Baeza-Yates's* is relatively big due to the way it works. By this method, at each step a median element of the shorter list will be figured out to do a binary search in the longer list. In a small buffer, however, only part of the lists is stored and each time only a locally central element (i.e., the median of the part of the shorter list, which is currently stored in the buffer) can be used, which will substantially degrade the efficiency of this method. Comparing Fig. 20(a) and Fig. 11(b), we can see that, with large buffers, no significant difference between in-memory and on-disk for all the four strategies can be observed.

8. CONCLUSION

In this paper, a new off-line algorithm for doing set intersections is discussed. Based on the transformation of inverted lists to interval sequences by establishing a trie T over the sequences of set identifiers, a binary search over interval sequences is designed with their least common ancestors being used to skip over useless interval containment checking. In this way, an optimal time complexity is achieved. Let X and Y be two interval sequences corresponding to two inverted lists created for two words x and y, respectively. Our algorithm needs only $O(|Y|\cdot\log(|\Gamma_x|/|Y|))$ time, where Γ_x is a sequence of nodes with each being the least common ancestors of some nodes in T, whose intervals make up a segment in X. Since in many cases an interval sequence is much shorter than the corresponding inverted list, the time complexity is theoretically better than any existing on-line method. Extensive Experiments have been conducted, which shows that our method also has optimal performance in practice.

REFERENCES

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

[2] N. Ao, F. Zhang, D. Stones, et al.: Efficient Parallel Lists Intersection and Index Compression Algorithms using Graphics Processing Units, *PVLDB* 2011, Seattle, USA.

[3] T. Apaydin, G. Canahuate, H. Ferhatosmanoglu, A. Saman Tosun: Approximate Encoding for Direct Access and Query Processing over Compressed Bitmaps. *VLDB* 2006: 846-857.

[4] D. Arroyuelo, S. González, M. Oyarzún and V. Sepulveda: Document identifier reassignment and run-length-compressed inverted indexes for improved search performance. SIGIR 2013: 173-182.

[5] R. A. Baeza-Yates: A Fast Set Intersection Algorithm for Sorted Sequences. CPM 2004: 400-408.

[6] R.A. Baeza-Yates, and A. Salinger: Experimental analysis of a fast intersection algorithm for sorted sequences, in *Proc. 12th Intl. Conf. on String Processing and Information*, Springer, Berlin, 13-24, 2005

[7] J. Barbay, A. López-Ortiz, and T. Lu: Faster adaptive set intersections for text searching. In *Proc. of the 5th International Workshop on Experimental Algorithms (WEA)*, volume 4007 of Lecture Notes in Computer Science (LNCS), pages 146–157. Springer Berlin / Heidelberg, 2006. 7.

[8] J. Barbay, C. Kenyon: Alternation and redundancy analysis of the intersection problem. *ACM Transactions on Algorithms* 4(1) (2008).

[9] 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).

[10] T.A. Bjørklund, N. Grimsmo, J. Gehrke, Ø. Torbjørnsen: Inverted indexes vs. bitmap indexes in decision support systems. *CIKM* 2009: 1509-1512.

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

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

[13] B. Croft, D. Metzler, T. Strohman: Search Engines: Information Retrieval in Practice – Feb 6 2009, Amazon.com.

[14] D. Carmel, D. Cohen, R. Fagin, E. Farchi, M. Hercovici, Y.S. Maarek, and A. Soffer: Static index pruning for information retrieval systems, in *Proc. 24th Annual Intl. Conf. on Research and Development in formation Retrieval*, New Orlean, LA, 43-50, 2001.

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

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

[17] 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.

[18] 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.

[19] Y. Chen and W. Shen, Decomposition of Inverted Lists and Word Labeling: A New Index Structure for Text Search, in *Proc. 2014 Int. Conf. on Advances in Big Data Analytics*, *IEEE*, July 21-24, 2014, Las Vegas, Nevada, USA.

[20] Y. Chen and W. Shen, On the Intersection of Inverted Lists, in *Proc. the 11th Int. Conf. on Foundations of Computer Science, IEEE,* July 26-30, 2015, Las Vegas, Nevada, USA.

[21] H. Cohen and E. Porat: Fast set intersection and two-patterns matching. Theoretical Computer Science 411(40-42): 3795-3800 (2010).

[22] C.L.A. Clarke and G.V. Cormack: Dynamic inverted indexs for a distributed full-text retrieval systems, Tech. rep. MT-95-01, Dept. Computer Science, University of Waterloo, Waterloo, Canada, 1995.

[23] 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.

[24] K.D. Demaine, T.R. Jones, and M. Patrascu: Interpolation search for non-independent data, in *Proc. 15th , ACM-SIAM Symposium on Discrete Algorithms,* Philadelphia, 529-530, 2004.

[25] U. Deppisch: S-Tree: A Dynamic Balanced Signature Index for Office Retrieval, *Proc. ACM SIGIR conf*, Sep 1986, pp 77 – 87.

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

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

[28] 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.

[29] S.W. Golomb: Run-length encodings, *IEEE Trans. Inform. Theory, IT-12,* 3 (July), 399-401, 1966.
[30] D. Harman, W. McCoy, R. Toense, and G. Candela: Prototyping a distributed information retrieval system using statistical ranking, *Inform. Proc. Manag. 27,* 5, 449-460, 1991.
[31] Y. Ishikawa, H. Kitagawa and N. Ohbo: Evaluation of signature files as set access facilities in OODBs, in *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, Washington D.C., May 1993, pp. 247-256.
[32] S. Heman. Super-scalar database compression between RAM and CPU-cache. MS Thesis, Centrum voor Wiskunde en Informatica, Amsterdam, Netherlands, July 2005.
[33] H. Inoue, M. Ohara, and K. Taura, Faster Set Intersection with SIMD instructions by Reducing Branch Mispredictions, in *Proc. VLDB*, pp. 293 – 304, 2015.
[34] E.L. Ivie: *Search procedure based on measures of relatedness between documents, Ph.D. thesis*, MIT, Cambridge, MA, 1966.
[35] H. Kitagawa and Y. Ishikawa: False Drop Analysis of Set Retrieval with Signature Files, *IEICE TRANS. INF. & SYST*, VOL. E80-D, NO. 6 JUNE 1997.
[36] D.E. Knuth, *The Art of Computer Programming*, *Vol. 3*, Massachusetts, Addison-Wesley Publish Com., 1975.
[37] D. Lemire, L. Boytsov and N. Kurz. SIMD Compression and the Intersection of Sorted Integers. arXiv:1401.6399, 2014.
[38] 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.
[39] J. Lovins: Development of a Stemming Algorithm, *Mechanical Translation and Computational Linguistics*, 11, 22—31, 1968.
[40] *Lucene in Action, Second Edition*: Covers Apache Lucene 3.0 by Michael McCandless, E. Hatcher and O. Gospodnetic (July 28, 2010).
[41] A. Moffat and J. Zobel: Parameterized compression for sparse bitmaps, in *Proc. 5th Annual Intl. ACM SIGIR Conf. on Research and Development in Information Retrieval*, Copenhagen, Denmark, ACM, 274-285, 1992.
[42] A. Moffat and J. Zobel: What does it means to "measures performance"? in *Proc. 5th Intl. Conf. on Web Information Systems*, Brisbane, Australia, *Lecture Notes in Computer Science*, vol. 3306, Springer, 1-12, 2004.
[43] W. Pugh. A skip list cookbook, technical report, UMIACS-TR-89-72, University of Maryland, 1990.
[44] P. Sanders and F. Transier. Intersection in Integer Inverted Indices. In ALENEX, pp. 71-83, 2007
[45] P.C. Saraiva et al.: Rank-preserving two-level caching for scalable search engines, in *Proc. 24th Annual Intl. Conf. on Research and Development in Information Retrieval*, New Orlean, LA, 51-58, 2001.
[46] B. Schlegel, T. Willhalm, and W. Lehner: Fast Sorted-Set Intersection using SIMD Instructions. In *Proc. of the Second International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, 2011.
[47] S.S. Skiena: *The Algorithm Design manual*, TELOS, State University of New York, Stony Brook, NY, 1997.
[48] S. Tatikonda, F. Junqueira, B.B. Cambazoglu, V. Plachouras: On efficient posting list intersection with multicore processors, *SIGIR 2009*: 738-739.
[49] Transier and P. Sanders: Compressed inverted indexes for in-memory search engines, in *ALENEX*, pp. 3-12, 2008.
[50] M. Terrovitis, S. Passas, P. Vassiliadis, and T.K. Sellis: A combination of trie-trees and inverted files for the indexing of set-valued attributes, in *Proc. CIKM*, 2006, pp.728-737.
[51] E. Tousidou, A. Nanopoulos, and Y. Manolopoulos: Improved Methods for Signature-Tree Construction, *Computer J.*, vol. 43, no. 4, pp. 301-314, 2000.
[52] E. Tousidou, P. Bozanis, Y. Manolopoulos: Signature-based structures for objects with set-values attributes, *Infromation Systems*, 27(2):93-121, 2002.
[53] D. Tsirogiannis, S. Guha, N. Koudas: Improving the Performance of List Intersection, *PVLDB* 2009.
[54] P. Willett: The Porter stemming algorithm: then and now. *Program: electronic library and information systems*. 40(3). pp. 219 – 233, 2006.
[55] H.E. Williams, J. Yiannis, and J. Zobel: Compression of inverted indexes for fast query evaluation, in *Proc. 25th Conf. on Research and development in information retrieval*, 2002, 222-229.
[56] J. Zobel and A. Moffat: Inverted Files for Text Search Engines, *ACM Computing Surveys*, 38(2):1-56, July 2006.

[57] J. Zobel, A. Moffat, and K. Ramamohanarao: Inverted Files Versus Signature Files for Text Indexing, in *ACM Trans. Database Syst.*, 1998, pp.453-490.
[58] A.Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien: Efficient query evaluation using a two-level retrieval process, in *Proc. CIKM*, pp. 426-434, 2003.
[59] F.K. Hwang and S. Lin, A Simple Algorithm for Merging Two Distinct Linear Ordered Sets, *SIAM J. Comput.*, Vol. 1. No. 1, March 1972.
[60] I. H. Witten, A. Moffat, and T. C. Bell, Managing Gigabytes – Compressing Indexing Document and Images, Morgan Kaufman Publisher, 1999.
[61] H. Yan, S. Ding and T. Suel: Inverted index compression and query processing with optimized document ordering. *WWW* 2009: 401-410.
[62] J. Zhang, X. Long, and T. Suel: Performance of compressed inverted list caching in search engines. In *Proc. of the 17th Int. World Wide Web Conf.*, April 2008.