A Space-Efficient Indexing Algorithm for Boolean Query Processing

Jianbin Qin[1], Chuan Xiao[2], Wei Wang[1], and Xuemin Lin[1]

[1] The University of New South Wales, Australia
[2] Nagoya University, Japan

Abstract. Inverted indexes are the fundamental index for information retrieval systems. Due to the correlation between terms, inverted lists in the index may have substantial overlap and hence redundancy. In this paper, we propose a new approach that *reduces* the size of inverted lists while retaining time-efficiency. Our solution is based on merging inverted lists that bear high overlap to each other and manage their content in the resulting *condensed index*. An efficient algorithm is designed to discover heavily-overlapped inverted lists and construct the condensed index for a given dataset. We demonstrate that our algorithm delivers considerable space saving while incurring little query performance overhead.

1 Introduction

Inverted index is a fundamental indexing data structure for information retrieval and has found its way into database systems. It associates tokens with their corresponding inverted lists; each list contains a sorted array of document identifiers in which the token appears. The primary advantage of the inverted index is that it supports *boolean queries* efficiently. For example, to retrieve documents containing both keywords x and y, we can intersect the inverted lists of x and y.

One issue with the traditional inverted index is its size. Currently, various compression techniques are used to reduce the size of each individual lists. However, little effort is paid to account for the redundancy among the inverted lists. Due to the existence of frequently co-occurring tokens (e.g., phrases), there will be high redundancy due to large overlaps.

In this paper, we propose a novel way to arrange the inverted index physically to achieve reducing the size of the inverted index by exploiting overlaps among inverted lists of groups of tokens. We name the resulting inverted index the *condensed inverted index*. The idea is to form groups of tokens and then explicitly represent the intersections of their corresponding inverted lists such that every document identifier only occurs at most once within the group. This not only reduces the overall size of the index but also accelerates certain queries. We present the query processing algorithm for boolean queries on the condensed index (Section 2).

One technical challenge is how to construct an optimal condensed index. We show that finding the minimum-sized condensed index is a very hard problem, and even a greedy algorithm is typically too expensive to be practical. We propose non-trivial optimizations to the greedy algorithm (Section 3).

We conducted experiments with several real-world datasets. It demonstrates the space and time trade-offs of the condensed index and the efficiency of the optimized index construction algorithm (Section 4).

Preliminaries. Let a record r be a *set* of tokens taken from a finite universe $\mathcal{U} = \{ w_1, w_2, \ldots, w_{|\mathcal{U}|} \}$, and R be a collection of records. A boolean query q is a sequence of tokens concatenated by boolean operators, AND, OR, and NOT. The task is to find all records r in R such that r satisfies the query q. The number of tokens in r is denoted as its *size*, or $|r|$.

An efficient way to answer boolean queries is to use *inverted indexes* [1]. An inverted list, l_w, is a data structure that maps the token w to a sorted list of record ids such that w is contained by the corresponding records. $l_w[i]$ denotes the i-th entry in the inverted list of token w.

After the inverted lists for all tokens in the record set are built, we can scan each token in the query q, probe the indexes using every token in x, and obtain a set of posting lists. Merging the posting lists using the boolean operators q will give us the final answer to the query.

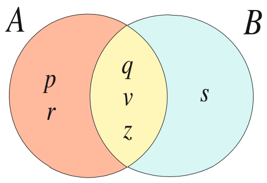
2 A New Index Structure for Boolean Queries

We design a new condensed index to exploit the correlation of multiple inverted lists.

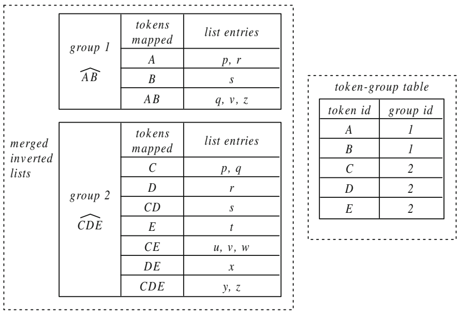
We illustrate the idea in Figure 1(b). Consider merging two lists l_A and l_B. It will produce a new list $l_{\widehat{AB}}$. The two tokens A and B will share the new list $l_{\widehat{AB}}$, and it will be traversed when either A or B appears in the query. This reduces the index size and the number of entries to be accessed when the query contains both A and B. However, it will probe more entries and introduce false positives when the query contains only A (or B). To address these issues, we divided the merged lists into *blocks*. Each block indexes a combination of the tokens in this list. For example, the first block maps to the records that contain only A, i.e., p and r. The second block maps to the records that contain only B, and the third block maps to the records that contain both A and B.

Figure 1 shows the structure of the merged inverted lists. We call the lists formed by merging *groups*, and assign a group id to each of them. We keep the *token-group table* that maps token id to group id, so as to locate the group that stores the token's inverted list. At the stage of index probing, the tokens in the query are first collected according to their groups. For each of these groups, we probe the blocks that contain the (combination of) tokens. To handle the boolean operators within a group, we need to probe the following blocks: (1) AND The blocks that index all the tokens (of this group) in the query. (2) OR The blocks the index any of the tokens (of this group) in the query. Then we take the union of the results. Note that we do not need to remove duplicates when processing union within a group since the blocks are disjoint in indexed entries.

In the interest of space, we refer readers to [2] for the detailed query processing algorithm.



(a) Combinding Two Inverted Lists


merged inverted lists

group 1 $\widehat{AB}$	tokens mapped	list entries
	A	p, r
	B	s
	AB	q, v, z

group 2 $\widehat{CDE}$	tokens mapped	list entries
	C	p, q
	D	r
	CD	s
	E	t
	CE	u, v, w
	DE	x
	CDE	y, z

token-group table

token id	group id
A	1
B	1
C	2
D	2
E	2

(b) Data Structure of Condensed Indexes

Fig. 1. Condensed Indexes

3 Choosing Inverted Lists to Merge

The condensed index structure can be implemented using a small amount of application-level code. Both space and time efficiency of the index structure depends on which of lists are chosen to be merged, yet this is not an easy task. In this section, we provide an efficient greedy algorithm that chooses lists to merge considering both space and time factors.

3.1 Greedy Algorithm

We start with a basic greedy algorithm that repeatedly merges the two lists that yield the most space saving.

Algorithm 1 describes the algorithm. Suppose the input lists L have been sorted by increasing token id. We initialize the groups by treating each inverted list as a single group (Line 1), and assign a group id according to their token id. Then we search for the pair of lists with the largest overlap in each iteration (Line 2 and 6), merge them into one group (Line 4 and 5), and assign a new group id, which is required to be greater than all of the current ones. To strike a balance between space saving and time efficiency, we use a parameter M to limit the maximum size of a group. The resulting group serves as a new inverted list to replace the two merged ones. The algorithm repeats until no pair of lists can be found to improve the overall space saving.

Algorithm 2 captures the process searching for the pair of lists with the largest overlap. We scan every inverted list, denoted l_i, searching for the list that has the most overlap with l_i (Line 2). We call this list the *partner* of l_i if we can safely merge the two lists without exceeding the size limit M.[1] We compute the overlap between each l_i and its partner, and arrange them in a max-heap E. The pair of lists at the top of E is the pair that yields the largest overlap.

In order to find the partner of each inverted list l_i, we use an array of counters to calculate the overlap between l_i and the other lists in L. The records indexed

[1] Note that the definition of partner is not symmetric.

Algorithm 1: MergeLists (R, L)

```
$E \leftarrow \emptyset$; $g_i \leftarrow 1 (1 \leq i \leq |L|)$ ;          /* $E$ is a max-heap */
$(l_x, l_y, score) \leftarrow$ SearchListPair $(R, L, E)$;
while $l_x \neq \emptyset$ do
    $g_{new} \leftarrow g_x + g_y$ ;                                     /* increase group size */
    $l_{new} \leftarrow l_x \cup l_y$, $L \leftarrow L \setminus \{l_x, l_y\} \cup \{l_{new}\}$;
    $(l_x, l_y, score) \leftarrow$ SearchListPair $(R, L, E)$;
return $L$
```

Algorithm 2: SearchListPair (R, L, E)

```
for $i = 1$ to $|L|$ do
    $(l_i, l_j, score) \leftarrow$ SearchPartner $(l_i)$;
    $E.push(l_i, l_j, score)$;
$(l_x, l_y, score) \leftarrow E.pop()$;
return $(l_x, l_y, score)$
```

by l_i is sequentially scanned. For each token w in each record, we increase the counter corresponding to l_w by one. The inverted list with the greatest value among the counters is reported as l_i's partner. The pseudo-code is given in Algorithm 3.

Algorithm 3. SearchPartner (l_x)

```
$O_{\max} \leftarrow 0$, $l_{\max} \leftarrow \emptyset$;
$O \leftarrow$ empty map from group id to int;
$A \leftarrow$ empty map from group id to record id;
for each $r \in l_x$ do
    for each $w \in r$ do
        $y \leftarrow w$'s group id;
        if $g_x + g_y \leq M$ and $A[y] \neq r$ then
            $O[y] \leftarrow O[y] + 1$, $A[y] = r$;
            if $O[y] > O_{\max}$ then
                $O_{\max} \leftarrow O[y]$, $l_{\max} \leftarrow l_y$;
return $(l_x, l_{\max}, O_{\max})$
```

3.2 Further Optimizations

The above greedy algorithm returns the condensed inverted lists. An important issue is that it has to recompute the partner of each list once two lists l_x and l_y are merged. These repeated computations incur significant overhead, and render the algorithm unable to output results for large-scale datasets in reasonable time. Nevertheless, we can avoid such computation by enforcing a constraint that a list's group id should always be greater than its partner's group id. We formally state the principle in the lemma below.

Algorithm 4. OptimizedSearchListPair (R, L, E)

```
1  if this function is called for the first time then
2      for $i = 1$ to $|L|$ do
3          $(l_i, l_j, score) \leftarrow$ SearchPartner $(l_i)$;
4          $E.push(l_i, l_j, score)$;
5  else
6      $(l_{new}, l_j, score) \leftarrow$ SearchPartner $(l_{new})$;
7      $E.push(l_{new}, l_j, score)$;
8  $(l_x, l_y, score) \leftarrow E.pop()$;
9  while either $l_x$ or $l_y$ has been merged do
10     if $l_x$ has not been merged then
11         $(l_x, l_z, score) \leftarrow$ SearchPartner $(l_x)$ ;   /* search $x$'s new partner */
12         $E.push(l_x, l_z, score)$;
13     $(l_x, l_y, score) \leftarrow E.pop()$;
14 return $(l_x, l_y, score)$
```

Lemma 1. *Let the partner of a list l_i be the list whose (1) group id is smaller than the group id of l_i; (2) group size will not exceed M if it is merged with l_i; (3) overlap with l_i is the largest among all the lists that satisfy the first two conditions. If a list changes its partner after merging l_x and l_y, then the partner of this list must be either l_x or l_y before the merging.*

Algorithm 5. OptimizedSearchPartner (l_x)

```
1  $O_{\max} \leftarrow 0$; $l_{\max} \leftarrow \emptyset$;
2  $O \leftarrow$ empty map from group id to int;
3  $A \leftarrow$ empty map from group id to record id;
4  if $l_x$ is the new list formed in the previous iteration then
5      $(O_{\max}, l_{\max}) \leftarrow$ GetO$_{\max}$ForNewList $(l_x)$;
6  for $i = 1$ to $|l_x| - O_{\max}$ do
7      $r \leftarrow l_x[i]$;
8      for each $w \in r$ do
9          $y \leftarrow w$' group id;
10         if $y < x$ and $g_x + g_y \leq M$ and $A[y] \neq r$ then
11             $O[y] \leftarrow O[y] + 1$, $A[y] = r$;
12             if $O[y] > O_{\max}$ then
13                 $O_{\max} \leftarrow O[y]$, $l_{\max} \leftarrow l_y$;
14 for each $y$ such that $O[y] > 0$ do
15     $O[y] \leftarrow |l_x \cap l_y|$ ;   /* evaluate exact overlap */
16     if $O[y] > O_{\max}$ then
17         $O_{\max} \leftarrow O[y]$, $l_{\max} \leftarrow l_y$;
18 return $(l_x, l_{\max}, O_{\max})$
```

This principle enables us to avoid committing the costly scanning over the set of lists L. Instead, only the lists whose partners are l_x or l_y need to be assigned with new partners. Additionally, we perform a *lazy* update to postpone the searching for such lists' partners. Only if these lists are popped from the max-heap E, we search for new partners for them. The merging algorithm will benefit since these lists may have been merged and discarded from further consideration before we are forced to seek new partners. We give the pseudo-code for the above method in Algorithm 4, and replace Algorithm 2 with it.

Algorithm 6. GetO$_{\max}$ForNewList (l_x)

```
1 $(l_u, l_v) \leftarrow$ the two lists that were merged to form $l_x$;
2 **if** $u < v$ **then** $w \leftarrow u$; **else** $w \leftarrow v$;
3 $z \leftarrow w$'s partner;
4 **if** $z$ has not been merged **and** $g_z + g_x \leq M$ **then**
5     $O_{\max} \leftarrow |l_w \cap l_z|$, $l_{\max} \leftarrow z$;
6 **else**
7     $O_{\max} \leftarrow 0$, $l_{\max} \leftarrow \emptyset$;
8 **return** $(O_{\max}, l_{\max})$
```

Another important optimization is to speed up the count algorithm we use in Algorithm 3. Since we are looking for the partner that has most overlap with l_x, a filtering condition can be developed using the current maximum overlap $O_{\max}$. Considering the following prefix filtering principle.

Lemma 2 (Prefix Filtering Principle). *Consider an ordering $\mathcal{O}$ of the token universe $\mathcal{U}$ and a set of records, each sorted by $\mathcal{O}$. Let the p-prefix of a record x be the first p tokens of x. If $|x \cap y| \geq \alpha$, then the $(|x| - \alpha + 1)$-prefix of x and the $(|y| - \alpha + 1)$-prefix of y must share at least one token.*

If there exist l_y such that $|l_x \cap l_y| > O_{\max}$, then l_y must share at least one token with the $(|l_x| - O_{\max})$-prefix of l_x. Therefore, only the first $(|l_x| - O_{\max})$ entries in l_x need to be probed in order to generate the candidate lists that have potential to become l_x's partner. This filtering condition is tightened as $O_{\max}$ increases. Finally, the candidate lists are verified for the exact overlap. The improved algorithm is captured in Algorithm 5, and is used to replace the original partner searching algorithm in Algorithm 3. In addition, we can infer an initial lower bound of $O_{\max}$ before partner search, given that l_x is the list formed by merging two lists during previous iteration, supposing they are l_u and l_v, and $u < v$. Since we have obtained the overlap between l_u and its partner l_i, it is guaranteed the overlap between l_x and l_i is no less than this value. This is because $|l_i \cap l_x| = |l_i \cap (l_u \cup l_v)| \geq |l_i \cap l_u|$. The pseudo-code is given in Algorithm 6, and invoked in Line 5 of Algorithm 5.

4 Experiments

In the interest of space, we briefly present our experimental results in this section. Please refer to [2] for the full version of experimental evaluation.

Three publicly available datasets were used in our experiments: DBLP bibliography records, TREC-9 Filtering Track Collections, and Enron email collection. We generated queries for each dataset by sampling records from the dataset and randomly selecting a number of consecutive tokens containing no stop words.

We evaluated the optimization methods proposed in Section 3. On DBLP, the lazy update technique exhibits a speed-up up to 9.3x over the basic greedy list merging algorithm with the constraint exploiting Lemma 1. Further applying prefix filtering principle achieves an additional speed-up of 2.6x, and runs in 10 to 20 seconds with varying maximum group size.

We compared the condensed index sizes with the original index sizes. The total index sizes decrease as the maximum group size M grows, bottoms at 6 or 7, and then rebounds. The overall space savings against the original inverted index are 16.4% on DBLP, 26.8% on TREC, and 39.2% on ENRON.

We evaluated the query processing time with varying numbers of tokens in a query. The best choice of M increases when more tokens are introduced. $M = 2$ yields the best runtime performance when a query contains two or three tokens.

5 Conclusion

We propose a novel inverted index structure to support boolean queries efficiently. By exploiting the overlaps among inverted lists of groups of tokens, the condensed structure is able to represent the intersections of their corresponding inverted lists, so that the redundancy among the inverted lists of frequently co-occurring tokens can be avoided. We design an efficient greedy algorithm to find a good condensed index. Experimental results show that our condensed index structure occupies less space yet achieves accepatable runtime performance.

References

1. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval, 1st edn. Addison-Wesley (May 1999)
2. Qin, J., Xiao, C., Wang, W., Lin, X.: Condensed inverted index: A space-efficient index for boolean queries. Technical report, University of New South Wales (2012)