A Space-Efficient Indexing Algorithm for Boolean Query Processing

Jianbin Qin[1] Chuan Xiao[2] Wei Wang[1]

Xuemin Lin[1]

[1] University of New South Wales, Australia
{jqin,weiw,lxue}@cse.unsw.edu.au
[2] Nagoya University, Japan
chuanx@db.itc.nagoya-u.ac.jp

Technical Report
UNSW-CSE-TR-201225
November 2012

Abstract

Boolean search is the most common information retrieval technique that requires users to input the exact words they are looking for into a text field. Existing algorithms use inverted list indexing schemes to answer boolean search queries. These approaches occupy large index size as the inverted lists are highly overlapping and redundant. In this paper, we propose a novel 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. A new algorithm is designed to discover overlapping inverted lists and construct condensed index for a given dataset. We conduct extensive experiments on several publicly available datasets. The proposed algorithm delivers considerable space usage improvement while exhibits tolerable time performance penalties.

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 lists contains an 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 keyword 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 named 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 3).

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 4).

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 5).

1.1 Related Work

There has been a body of work on keyword and boolean search in IR [1, 2, 3] and Web search [4]. Major RDBMSs have also integrated full-text search capabilities that incorporate IR relevance ranking strategies. Existing systems such as DBXplorer [5] and DISCOVER [6] developed efficient algorithms for answering keyword queries but limited to AND semantics. [7] presented a system for efficient keyword search over relational databases that can handle queries with both AND and OR semantics.

The inverted index data structure is a central component of a typical search engine indexing algorithm [8]. Many inverted index compression techniques have been studied, see [9, 10] for a survey and [11, 12, 13, 14, 15] for recent work. Most techniques first subtract each document id by the preceding document id, and obtain a number called d-gap. The document ids are replaced by d-gaps, and then encoded with some integer compression algorithm. A higher compression ratio can be achieved due to the smaller average value of d-gaps.

2 Preliminaries

Let a record r be a *set* of tokens taken from a finite universe $\mathcal{U} = \{ w_1, w_2, \dots, w_{|\mathcal{U}|} \}$, 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.

Consider the example of search engine. Each web page on the internet is parsed, cleaned, and then transformed into a set of tokens. We use the term *token* to refer to the basic element in a (string) record. A token can be a stemmed word or a q-gram A user may ask: Give me all the documents that contain the word "`christmas`" or the word "`tree`". The user will frame his query in a syntax supported by the system: "`christmas` OR `tree`".

We denote the size of a record x as $|x|$, which is the number of tokens in x. Given a document "`the lord of the rings`", we can tokenize the document with white spaces into the following record $x = \{ A, B, C, D \}$, where A stands for "`the`", B for "`lord`", and so on. Note that we remove duplicate tokens, and hence the second "`the`" has been removed from the tokenized set.

An efficient way to answer boolean queries is to use *inverted indexes* [8]. An inverted list, l_w, is a data structure that maps the token w to a sorted list of record ids such that w in contained by the corresponding records. Supposing the inverted list is sorted according to the order that the record ids are inserted, $l_w[i]$ indicates 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.

Example 1 *Suppose we have three documents D_x = "`the lerd of two kings`", D_y = "`the lord of the rings`", and D_z = "`the rings of two kings`", and a query document q = "`lord` AND `rings`". We tokenize them with white spaces. They are transformed and canonicalized according to decreasing idf order into: $x = [A, C, D, F, G]$, $y = [B, E, F, G]$, and $z = [C, D, E, F, G]$, with the following word-token mapping table*

Word	*lerd*	*lord*	*two*	*kings*	*rings*	*the*	*of*
Token	A	B	C	D	E	F	G

We regard "`the`" and "`of`" as stop words, and do not build inverted index for them. Then the inverted indexes are built

$$A = [x], \quad B = [y], \quad C = [x, z], \quad D = [x, z], \quad E = [y, z].$$

The query q contains two tokens B and E. Probing the inverted indexes yields two posting lists $[y]$ and $[y, z]$. We merge the two lists using the operator AND and obtain the intersection $[y]$. Therefore D_y = "`the lord of the rings`" is confirmed as the result.

3 A New Index Structure for Boolean Queries

In this section, we first give an example to motivate the need to looking for condensed inverted index, and then introduce a new index structure that exploits the correlation among inverted lists.

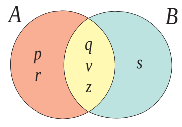


Figure 3.1: Combine Two Inverted Lists

Example 2 *Consider the a tokenized query $q = A \cap B$, and the following two inverted lists for token A and B*

$$l_A = [\,p, q, r, v, z\,], \qquad l_B = [\,q, s, v, z\,].$$

The size of the index is 5+4=9. We sequentially scan the inverted lists of A and B, and then find the six results $\{p, q, r, s, v, z\}$. The number of entries accessed in the inverted lists is 9.

However, we can combine the two inverted lists and mark the common entries of them underlined.

$$l_{\widehat{AB}} = [\,p, r, \underline{q, v, z}, s\,].$$

The size of the index is now reduced to 6, and the number of entries accessed is reduced to 3 if we probe only the common part of the two lists.

Seeing the correlation among inverted lists, it is desirable to design a new index structure in order to both reduce index size and improve query performance. We illustrate the idea in Figure 3.1. 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 3.2 shows the structure of the merged inverted lists. We call the lists that 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.

Example 3 *Consider a tokenized query $q = B \cup (C \cap E)$ and the new inverted index in Figure 3.2.*

We first look up the token-group table. B maps to group 1, and both C and E map to group 2. For group 1, we check the blocks that contain B's inverted list,

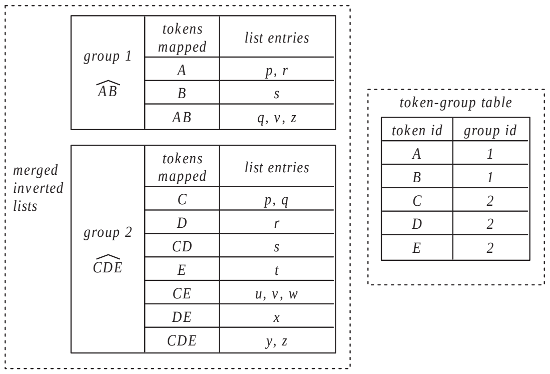


Figure 3.2: Condensed Index Structure

i.e., the blocks marked "B" and "AB". Four records $\{s, q, v, z\}$ *are obtained. For group 2, we check the blocks that contain both C and E's inverted list, i.e., the blocks marked "CE" and "CDE". We obtain five records* $\{u, v, w, y, z\}$*. Hence the final results are* $\{q, s, u, v, w, y, z\}$*.*

Algorithm 1: AnswerBooleanQueriesWithNewIndex

```
1 $G \leftarrow \emptyset$;
2 for each $w \in q$ do
3     $u \leftarrow w$'s group id, $G \leftarrow G \cup \{l_u\}$;
4 for each $l_i \in G$ do
5     if the operator is "AND" then
6         probe the blocks that index all requested tokens;
7     if the operator is "OR" then
8         probe the blocks that index any requested token;
```

We state the algorithm to answering boolean queries using the new index structure in Algorithm 1. Compared with the basic algorithm using traditional index structure, this algorithm makes three major modifications: (1) A new condensed inverted index is used to merge the inverted lists that bear strong correlation on each other. The index size is significantly reduced. This also improves the running time performance since the intersection and union operators within a group can be handled by accessing the entries in the blocks only without processing the expensive set operations. (2) A token-group table is used to determine the groups of inverted lists that we are going to probe. (3) For every probed group, we need to determine the blocks to be probed according to the operators that act on the tokens in this group.

4 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.

4.1 Greedy Algorithm

Seeing the difficulty in finding optimal solution for the above problem, we design a greedy method for choosing lists to merge.

The algorithm takes input as a set of inverted lists built on the corpus R. We greedily choose and merge the two lists that yield the most space saving. Note that the overlap between the two lists is exactly the increment to the overall space saving, [1] and therefore our task is to choose the pair of lists with the largest overlap.

Algorithm 2 describes the greedy 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 2), 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 3 and 7), merge them into one group (Line 5 and 6), and assign a new group id, which is required to be greater than all of the current ones. The resulting group either (1) serves as a new inverted list to replace the two merged ones if its size is smaller than M; or else (2) is ignored from further consideration. The algorithm repeats until no pair of lists can be found to improve the overall space saving. Algorithm 3 captures the process searching

Algorithm 2: MergeLists (R, L)

```
Input   : $R$ is a collection of token arrays sorted by the increasing order of
          sizes. $L$ is the inverted lists built on $R$.
1 $E \leftarrow \emptyset$ ;                                   /* $E$ is a max-heap */
2 $g_i \leftarrow 1 (1 \leq i \leq |L|)$;
3 $(l_x, l_y, score) \leftarrow$ SearchListPair $(R, L, E)$;
4 while $l_x \neq \emptyset$ do
5     $g_{new} \leftarrow g_x + g_y$ ;                        /* increase group size */
6     $l_{new} \leftarrow l_x \cup l_y$, $L \leftarrow L \backslash \{l_x, l_y\} \cup \{l_{new}\}$;
7     $(l_x, l_y, score) \leftarrow$ SearchListPair $(R, L, E)$;
8 return $L$
```

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. [2] We compute the overlap between each l_i and its partner, and use a max-heap E to arrange the overlaps. The pair of lists at the top of E is the pair that yields the largest overlap.

After the merging of the pair, we need to update some inverted lists for their partners as the partners might have been merged with other lists during the previous iteration. We recompute the partners for these affected lists, and insert the result into E. 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 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 4.

[1] This also holds for the case when more than three lists are merged into one group.

[2] The definition of partner is not symmetric, i.e., x is not necessarily y's partner given that y is x's partner.

Algorithm 3: SearchListPair (R, L, E)

```
1  for $i = 1$ to $|L|$ do
2      $(l_i, l_j, score) \leftarrow$ SearchPartner $(l_i)$;
3      $E.push(l_i, l_j, score)$;
4  $(l_x, l_y, score) \leftarrow E.pop()$;
5  while either $l_x$ or $l_y$ has been merged do
6      $(l_x, l_y, score) \leftarrow E.pop()$;
7  return $(l_x, l_y, score)$
```

Algorithm 4: SearchPartner (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  for each $r \in l_x$ do
5      for each $w \in r$ do
6          $y \leftarrow w$'s group id;
7          if $g_x + g_y \le M$ and $A[y] \ne r$ then
8              $O[y] \leftarrow O[y] + 1$, $A[y] = r$;
9              if $O[y] > O_{\max}$ then
10                 $O_{\max} \leftarrow O[y]$, $l_{\max} \leftarrow l_y$;
11 return $(l_x, l_{\max}, O_{\max})$
```

Example 4 *Consider the following five inverted lists and a group size limit of* $M = 3$.

$$A = [a, b, c], \quad B = [a, c, d], \quad C = [a, e, f, g, h], \quad D = [a, f, g, h, i], \quad E = [b, e, f, j, k]$$

At first, we scan each list searching for their partners. The lists, , partners (underlined) and their overlaps are $(A, \underline{B}, 2)$, $(B, \underline{A}, 2)$, $(C, \underline{D}, 4)$, $(D, \underline{C}, 4)$ *and* $(E, \underline{C}, 2)$. *The largest among them is* $(C, \underline{D}, 4)$, *so we merge the two lists* C *and* D *into a new list* $\widehat{CD}$. *Now we have four lists* $A = [a, b, c]$, $B = [a, c, d]$, $E = [b, e, f, j, k]$, $\widehat{CD} = [a, e, f, g, h, i]$. *The partners are* $(A, \underline{B}, 2)$, $(B, \underline{A}, 2)$, $(E, \underline{A}, 1)$ *and* $(\widehat{CD}, \underline{E}, 2)$. *We choose to merge* A *and* B, *and the resulting lists are* $E = [b, e, f, j, k]$, $\widehat{CD} = [a, e, f, g, h, i]$, $\widehat{AB} = [a, b, c, d]$. *Similarly, we choose to merge* E *and* $\widehat{CD}$, *and then* $\widehat{AB} = [a, b, c, d]$, $\widehat{CDE} = [a, b, e, f, g, h, i, j, k]$. *The group size of* $\widehat{CDE}$ *has reached the limit* M. *Since there is no pair of lists that can improve the space saving, we finish the merging and return the two lists* $\widehat{AB}$ *and* $\widehat{CDE}$ *as the final inverted index. The index size before merging is* $3 + 3 + 5 + 5 + 5 = 21$. *After merging, the index size is* $4 + 9 = 13$, *and thus a space of 8 entries is saved.*

4.2 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. This repeated computations incur significant overhead, and render the algorithm unable to output results for large-scale datasets in reasonable time. We use an example to illustrate the the difficulty in identifying the lists

Algorithm 5: 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  $(l_x, l_y, score) \leftarrow E.pop()$;
6  while either $l_x$ or $l_y$ has been merged do
7      if $l_x$ has not been merged then
8          $(l_x, l_z, score) \leftarrow$ SearchPartner $(l_x)$ ;  /* search $x$'s new partner  */
9          $E.push(l_x, l_z, score)$;
10     $(l_x, l_y, score) \leftarrow E.pop()$;
11 return $(l_x, l_y, score)$
```

that change their partners due to merging.

Example 5 *Consider the following four inverted lists.*

$$A = [\,i, j, k, l, m\,],\ \ B = [\,a, b, c, d, e, f\,],\ \ C = [\,c, d, e, f, g, h\,],\ \ D = [\,a, b, g, h, i, j, k\,]$$

We merge B and C that have the largest overlap among all the pairs, and obtain $\widehat{BC} = [\,a, b, c, d, e, f, h, h\,]$. Before the merging, the partner of D is A, with an overlap of 3, but changes to $\widehat{BC}$, with an overlap of 4 after the merging. We have seen that this issue cannot be addressed by simply memorizing each list's partner, since D is neither the partner of B nor C before the merging. On the other hand, declining to update D's partner may lead to inexact answer and therefore compromise the overall space saving.

Nevertheless, we can avoid this dilemma 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.

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.*

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 5.

Another important optimization is to speed up the count algorithm we use in Algorithm 4. 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*

Algorithm 6: OptimizedSearchPartner

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

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 6, and is used to replace the original partner searching algorithm provided in Algorithm 4. In addition, we can infer an initial lower bound of $O_{\max}$ before performing partner search, given that l_x is the list formed by merging two lists during previous iteration. Suppose the two lists are l_u and l_v, and without loss of generality, $u < v$. Since we have obtained the overlap between l_u and its partner l_i, it is guaranteed that the overlap between l_x and l_i is no less than this value. This is because

Proposition 1 $|l_i \cap l_x| = |l_i \cap (l_u \cup l_v)| \geq |l_i \cap l_u|$.

We provide the pseudo-code in Algorithm 7, and invoke it in Line 5 of Algorithm 6.

Algorithm 7: GetO$_{\max}$ForNewList (l_x)

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

5 Experiments

We present our experimental results and analyses in this section.

5.1 Experiment Setup

We used the following publicly available real datasets. They covered a wide range of data distributions and application domains.

- **DBLP** is a snapshot of the bibliography records from the DBLP Web site.Each record is the title of a publication.
- **TREC** is from the TREC-9 Filtering Track Collections. Each record is a concatenation of the author, title, and abstract fields.
- **ENRON** This dataset is from the Enron email collection. Each record consists of the email title and body.

The datasets are tokenized by following methods: (1) For DBLP, TREC, and ENRON, we tokenize the datasets with white spaces and punctuations. (2) For DBLP, TREC, we extract q-grams from the records. White spaces and punctuations are converted into underscores, and letters into their lowercases for DBLP and TREC. We set the q-gram length for DBLP and TREC as 5. The tokenized records are then sorted in ascending order of length. Some important statistics about these datasets after the cleaning are listed in Table 5.1. We generate

Table 5.1: Statistics of Cleaned Datasets

Dataset	*# of records*	*avg. length*	$\|U\|$
DBLP	851,570	9.2	153,716
DBLP 5-GRAM	851,570	66.3	482,815
TREC	45,244	168.9	275,629
TREC 5-GRAM	45,244	768.4	564,545
ENRON	12,977	2022.3	1,169,334

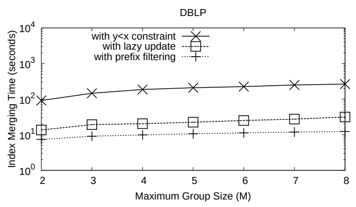
10,000 queries for each dataset by sampling 10,000 records from the dataset and randomly selecting a number of consecutive tokens that do not contain stop words.

All algorithms are implemented as disk-resident algorithms in C++. We store inverted lists on disks and load token-group table into memory. The experiments are carried out on a PC with Xeon X3220@2.40GHz CPU and 4GB RAM.

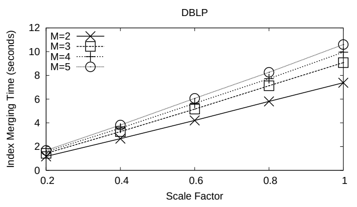
5.2 Effect of Optimization on Index Merging Algorithm

We evaluate the optimization methods over the index merging algorithm proposed in Section 4 by comparing three algorithms: (1) The basic greedy list merging algorithm with the constraint on the partner of an inverted list. We require the partner's group id to be smaller than the group id of the list (denoted "with $y < x$ constraint"). (2) The above algorithm equipped with the lazy update technique. (denoted "with lazy update"). (3) The above algorithm equipped with the optimization using prefix filtering principle. (denoted "with prefix filtering").

We plot the running time of constructing condensed index structure from the original inverted lists and show the results on DBLP in Figure 5.1(a). The

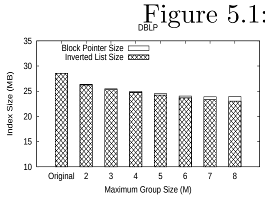


(a) Optimization on Index Merging Algorithm

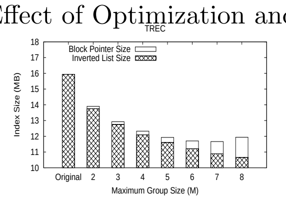


(b) Scalability of Index Merging Algorithm

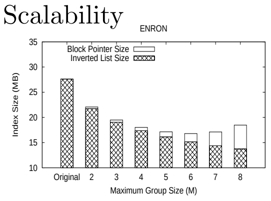
Figure 5.1: Effect of Optimization and Scalability



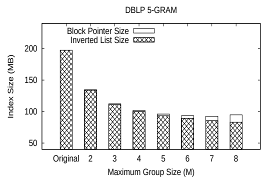
(a) DBLP, Index Size



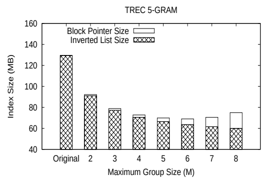
(b) TREC, Index Size



(c) ENRON, Index Size



(d) DBLP 5-GRAM, Index Size



(e) TREC 5-GRAM, Index Size

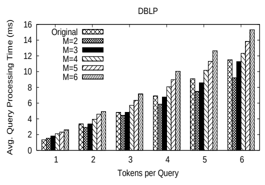
Figure 5.2: Evaluation of Index Size

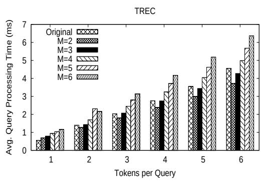
running time is drastically reduced after applying lazy update technique, and the speed-up can be up to 9.3x. The result reveals that a majority of the lists have been merged and discarded from further consideration before we are forced to seek new partners for them. We postpone the process of searching partners for them and thus to avoid redundant computations. The performance is further improved after applying prefix filtering conditions, with an additional speed-up of 2.6x. Another trend is that the running time increases steadily when we move the maximum group size M towards larger values, because we are able to merge more lists into one group.

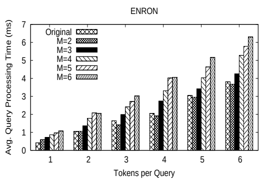
5.3 Scalability against Data Sizes

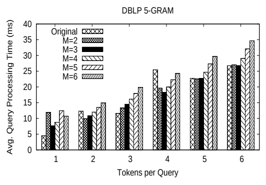
We study the scalability of our index merging algorithm with varying data sizes and measure the index construction time on DBLP dataset. We randomly take a sample from 20% to 100% of the records. We choose maximum group size M from 2 to 5 and show the running times in Figure 5.1(b).

It is clear that the running time of the index merging algorithm grows linearly with the expansion of data size. This is due to the number of iterations to choose lists to merge, which grows linearly with the increase of data size. E.g., there are 14,924 iterations on 20% of data, and 58,731 iterations on the whole corpus.


(a) DBLP, Query Time


(b) TREC, Query Time


(c) ENRON, Query Time


(d) DBLP 5-GRAM, Query Time

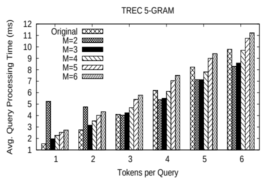

(e) TREC 5-GRAM, Query Time

Figure 5.3: Evaluation of Query Processing Time

5.4 Evaluation of Index Size

On Datasets Tokenized by Words. We compare the condensed index sizes with the original index sizes. Figures 5.2(a) – 5.2(c) show the results on DBLP, TREC, and ENRON. It is the original inverted index when $M = 1$, so we mark it as "Original" in the figures. Since the groups in the condensed inverted index are divided into blocks, we need a pointer mapping a block to the inverted lists that stored on disks. Therefore, we decompose the condensed index size into *inverted list size* and *block pointer size*, as shown in the figures. The general trend is that the total index sizes decrease rapidly as the maximum group size M grows, bottoms at $M = 6$ or 7, and then rebounds when M continues to grow. There are two main factors: (1) the inverted list size is always reduced when we allow to merge more lists in a group. (2) the block pointer size grows exponentially with M since a group is divided into $2^M - 1$ blocks for its M tokens. Both factors affect the total condensed index size, and the latter is more significant for large M values. With respect to the overall space saving against the original inverted index, the space consumption can be reduced by up to 16.4% on DBLP, 26.8% on TREC, and 39.2% on ENRON. Another interesting finding is that the space consumption reduces significantly when we move M from 1 to 2, which showcases the overlap among the original inverted lists.

On Datasets Tokenized by q-grams. We plot the index size on DBLP 5-Gram, TREC 5-Gram in Figures 5.2(d) – 5.2(e).The result displays similar trend as does on the datasets tokenized by words. A major difference is that overall space saving is more remarkable on q-gram datasets. For example, the space consumption can be reduced by up to 53.1% on DBLP 5-Gram and 46.7% on TREC 5-Gram. This is mainly because q-grams are more correlated if they are extracted from a word or a phrase, and therefore have more chance to overlap in the inverted lists than do the words tokenized by white spaces.

5.5 Evaluation of Boolean Query Processing Time

On Datasets Tokenized by Words. We evaluate the query processing performance with varying numbers of tokens in a query, and report the running time on DBLP, TREC, and ENRON in Figures 5.3(a) – 5.3(c). Several observations can be made: (1) For all the M settings, the running time grows linearly with the number of tokens in a query. (2) The best choice of M increases when we introduce more tokens to a query. (3) Large M values give the worse performance. The original index structure is the best choice when a query contains only one token. When we allow more tokens in the query, the condensed index exhibits better runtime performance since both intersection and union operations can benefit from the index structure. In addition, $M = 2$ yields the best runtime performance on the three datasets when we have two or more tokens in a query.

On Datasets Tokenized by q-grams. We plot the query processing time on DBLP 5-Gram, TREC 5-Gram in Figures 5.3(d) – 5.3(e). Similar trend can be observed, while the major difference is the best M choice. For DBLP 5-Gram, the best M is 2 when we have 2 or 3 tokens in a query, and 3 when the number of tokens in a query passes 4. For TREC 5-Gram, the best M is 2 when the number of tokens per query is 2, 3 for the range of [3, 5], and 4 when the number of tokens reaches 6. This can be explained as q-grams are more correlated and have more overlap in their corresponding inverted lists. Therefore merging more tokens into one group will improve the runtime performance, especially when the number of tokens in a query is large.

6 Conclusion

In this paper, 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 high redundancy among the inverted lists of frequently co-occurring tokens can be avoided. We design an efficient greedy algorithm to tackle the problem of finding a good condensed index. Experimental results show that our proposed condensed index structure occupies less space yet achieves accepatable runtime performance.

Bibliography

[1] Singhal, A., Buckley, C., Mitra, M.: Pivoted document length normalization. In: SIGIR. (1996) 21–29

[2] Singhal, A.: Modern information retrieval: A brief overview. IEEE Data Eng. Bull. **24**(4) (2001) 35–43

[3] Grossman, D.A., Frieder, O.: Information Retrieval: Algorithms and Heuristics. Springer (2004)

[4] Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. Computer Networks **30**(1-7) (1998) 107–117

[5] Agrawal, S., Chaudhuri, S., Das, G.: Dbxplorer: A system for keyword-based search over relational databases. In: ICDE. (2002) 5–16

[6] Hristidis, V., Papakonstantinou, Y.: Discover: Keyword search in relational databases. In: VLDB. (2002) 670–681

[7] Hristidis, V., Gravano, L., Papakonstantinou, Y.: Efficient ir-style keyword search over relational databases. In: VLDB. (2003) 850–861

[8] Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. 1st edition edn. Addison Wesley (May 1999)

[9] Witten, I.H., Moffat, A., Bell, T.C.: Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition. Morgan Kaufmann (1999)

[10] Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Comput. Surv. **38**(2) (2006)

[11] Anh, V.N., Moffat, A.: Inverted index compression using word-aligned binary codes. Inf. Retr. **8**(1) (2005) 151–166

[12] Anh, V.N., Moffat, A.: Improved word-aligned binary compression for text indexing. IEEE Trans. Knowl. Data Eng. **18**(6) (2006) 857–861

[13] Zukowski, M., Héman, S., Nes, N., Boncz, P.A.: Super-scalar ram-cpu cache compression. In: ICDE. (2006) 59

[14] Zhang, J., Long, X., Suel, T.: Performance of compressed inverted list caching in search engines. In: WWW. (2008) 387–396

[15] Yan, H., Ding, S., Suel, T.: Inverted index compression and query processing with optimized document ordering. In: WWW. (2009) 401–410