A Data Indexing Technique to Improve the Search Latency of AND Queries for Large Scale Textual Documents

Abdulla Kalandar Mohideen*, Shikharesh Majumdar*, Marc St-Hilaire* and Ali El-Haraki†
*Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada
†Telus, Ottawa, Canada
Email: {abdullakalandarmohid@cmail., majumdar@sce., marc_st_hilaire@}carleton.ca, ali.el-haraki@telus.com

***Abstract*—Boolean AND queries (BAQ) are one of the most important types of queries used in text searching. In this paper, a graph-based indexing technique is proposed to improve the search latency of BAQ. It shows how a graph structure represented using a hash table can reduce the number of intersections needed for the execution of BAQ. The performance of the proposed technique is compared with one of the most widely used index structures for textual documents called Inverted Index. A detailed performance analysis is performed through prototyping and measurement on a system subjected to a synthetic workload. To get further performance insights, the proposed graph-based indexing technique is also compared with an enterprise-level search engine called Elasticsearch which uses Inverted Index at its core. The analysis shows that the graph-based indexing technique can reduce the latency for executing BAQ significantly in comparison to the other techniques.**

***Index Terms*—text indexing, keyword search, Boolean AND queries, Graph-Based Index**

I. Introduction

The advancement of Internet search engine capability to provide more relevant and useful results to users has led to an increased number of search operations on the Internet. Indexing is one of the common techniques adopted to perform search operations faster involving a large set of documents. Examples of such documents include tweets and web pages. Search engines like Google[1] and Elasticsearch[2] use Inverted Index as index structure for full-text searching. Inverted Index also finds its application in the field of micro-blogs [1]. A micro-blog is like a normal blog except that the content is restricted to a smaller size.

Inverted Index is an index structure that exhibits an association between different keywords extracted from a document and their corresponding postings list. A document is any textual data source from which distinctive keywords can be extracted. Each document is associated with an identifier so that it can be represented individually. A postings list is defined as a group of document ids. A postings list for a keyword indicates that the keyword appears in all the documents represented by its ids present in the postings list [2].

The direct association between a keyword and its postings list in Inverted Index is highly helpful in executing normal queries. Such a normal query contains only a single search keyword and results in a set of documents containing the search keyword. A simple lookup for the keyword in Inverted Index will get all the ids of the documents containing the search keyword. These queries are not very popular as they typically return large sets of uninteresting documents which are difficult to narrow down as per user need. In the case of Boolean queries, two or more search keywords postings lists have to be accessed. Then, depending on the query that is being executed (AND, OR, or NOT), different operations such as intersection, union, or difference have to be carried out. This paper focuses on the efficient execution of Boolean AND queries (BAQ) by reducing the number of intersection operations needed in comparison to Inverted Index. BAQ are commonly used to narrow down the search results to relevant links in web search engines like Google[3].

The authors of [3] describe general concepts of graph theory and outline the various representations of graph data structures and their application in social network analysis in websites like Facebook and Twitter. A graph data structure can be used in situations where two or more different artifacts that have common attributes have to be related. In the case of textual data indexing, different keywords extracted from a single document can be considered as artifacts. The id of the document from which these keywords are extracted is the common attribute connecting these keywords. Thus, the relationship between different keywords that appears in a document can be modelled using a graph structure. The result of a BAQ is the set of document ids in which all the search keywords appear together. Since the graph structure stores the relationships between keywords, the proposed technique uses this relationship information to reduce the number of intersections needed to compute the result.

This research focuses on indexing and executing BAQ in a large set of textual documents. Preliminary research focusing on the basic concepts underlying the technique was published in a 4-page poster paper [4]. This full paper describes the results of our complete research and presents new materials: figures, description of algorithms, techniques, and experimental results. The main contributions of this paper include the

[1]https://gofishdigital.com/googles-indexer-caffeine/
[2]https://www.elastic.co/blog/found-elasticsearch-from-the-bottom-up
[3]https://library.uaf.edu/ls101-boolean

following:

1) A detailed description of the Graph-Based Index (GBI) structure and the proposed indexing algorithm.
2) A new algorithm for Boolean AND search using GBI.
3) A proof of concept prototype that implements the algorithms described earlier.
4) A detailed set of insights into system behaviour and performance resulting from extensive measurements made on the prototype subjected to a synthetic workload.
5) A performance comparison between GBI and Elasticsearch, a popular enterprise-level search engine that uses Inverted Index.

The rest of this paper is organized as follows. Section II describes a representative set of related works. Section III introduces the GBI structure and discusses the indexing algorithm used in GBI. Section IV discusses the BAQ execution using GBI. Section V presents a performance evaluation of GBI and compares its performance to the Inverted Index prototype and Elasticsearch. Finally, Section VI provides conclusions and directions for future work.

II. Related Works

When many textual documents are indexed, the execution of BAQ using Inverted Index involves huge intersection computations. The authors of [5] explained the concept of using a skip list to speed up the postings list traversal for intersection computation. In the skip list method, skip pointers are placed on certain document ids in all the postings lists. This method helps avoid iterating over all the document ids in the postings list by bypassing the skip pointers that are unessential for intersection computation.

Another method to reduce the Boolean queries processing time is to adopt the concept of scoring or ranking for the documents before indexing. For example, in applications like Twitter, when the user searches using keywords, not all tweets (micro-blogs) containing the keywords are listed. Each tweet is ranked before indexing based on measures such as the popularity of the author, the number of followers, or the popularity of the topic. Then, during the search, only the top K results are retrieved where K represents the number of resultant documents matching the search keywords [6] [7]. The ranking concept has been used in web page indexing as well [8].

Some of the research highlighting the scoring concept are discussed next. The authors of [9] proposed an optimized scoring mechanism for extended Boolean retrieval to get more relevant documents. In an extended Boolean retrieval query, in addition to having normal Boolean operators (AND, OR, NOT), it also has a proximity operator. A proximity operator tells how close two keywords should appear together in the document to be eligible to be part of the resultant set. The closeness is defined in terms of the number of words, sentences, or paragraphs that should appear between two keywords. The authors of [10] pruned the search keywords in a query instead of search results to lower the query execution time. The existence of common keywords in the queries has the least impact on the result. Thus, by pruning those search keywords before the execution of the ranked queries, the authors claim that they were able to attain reasonable precision in the results. The authors of [11] proposed a controlled ranking and pruning strategy where the quality of the result of the queries depends on the time and the system resource that has been allocated to the search queries. The authors used early search termination heuristics to get results with a reasonable quality during heavy system load and strategies for getting more refined results under less system load.

All of the existing methods discussed above try to adopt a type of ranking mechanism to avoid returning all the matching documents at once. However, for applications requiring all the results of BAQ at once, scoring becomes irrelevant. All the documents are considered equally important. Thus, returning only a certain number of top results alone will not justify the BAQ execution.

A graph data structure called Directed Acyclic Word Graph (DAWG) has found its application in the field of text processing where closely related strings are indexed to perform string pattern matching and prefix search efficiently [12]. In this graph structure, the unique character in a set of related strings forms a node of DAWG. This inspired the research discussed in this paper to use unique keywords extracted from the document as nodes in the proposed GBI structure. The authors of [13] proposed a graph-based indexing scheme for Arabic documents. In this work, the entire document is modeled as a graph and a term weighting feature is applied to find the relevancy of a term to the document. The research work also claims that graph-based indexing is better for contextual indexing and outperforms statistics-based methods like Term Frequency - Inverse Document Frequency (TF-IDF).

A representative set of related works are presented above. The proposed graph-based indexing technique focuses on obtaining all matching document ids at once without compromising BAQ execution time. This forms an important motivating factor for undertaking this research. To address this issue, the GBI structure aims to reduce the number of intersection operations needed by cutting down the number of postings lists needed for BAQ execution. Since the search time enhancement is done by only reducing the number of postings list needed, the type of intersection algorithm used is irrelevant, the intersection algorithm that can be applied to Inverted Index can also be applied in systems that use GBI as well as long as the format of the postings list is the same in both the cases.

III. Graph-Based Index

In general, not all the words found in a document are indexed. The document to be indexed undergoes various keyword filtering and extraction processes to get the potential keywords for indexing. One such extraction process involves removing common words that frequently occur in almost any textual data like articles and prepositions to name a few [14]. In this research paper, we focus only on the strategies used to index and search the extracted keywords using a Graph-Based Index (GBI).

GBI consists of a directed graph where each unique keyword extracted from the document to be indexed forms a node. An edge is added between two keyword nodes if the two keywords exist in the same document. The direction of the edge is always pointed from a lower alphabetical order keyword to a higher alphabetical order keyword. The ids of the documents that contain both keywords are used as the labels of the edge. A slightly modified version of the well-known adjacency matrix representation [3] is used to represent GBI. In the proposed method, the adjacency matrix stores only the keywords (nodes) that have occurred together in a document. This allows GBI to consume less memory. This is different from the usual adjacency matrix representation of a graph where the nodes which do not have an edge between them are also stored. The authors of [15] gained a superior search performance by using a hash table to build Inverted Index. This inspired the adoption of a two-dimensional hash table to store this modified version of the adjacency matrix. In a two-dimensional hash table, two keys are used to uniquely identify a set of values. The keywords that occur together in a document are grouped into a pair of two keywords and are used as the key for the hash table entry containing the ids of the documents as value.

Consider, for example, the following keywords being extracted from two documents after the keyword extraction procedure.

- Document 1: *Justin*, *Trump* and *Clinton*. Let the id of document 1 be 1.
- Document 2: *Clinton* and *Trump*. Let the id of document 2 be 2.

Initially, let us consider the extracted keywords from document 1. To index these extracted keywords, the keywords are sorted in lexicographical order. Thus, the sorted keywords set becomes {*Clinton*, *Justin*, *Trump*}. Three nodes, each one representing one of the extracted keywords, are created. Then, an edge is placed from each of the lower alphabetical order keyword nodes to the higher alphabetical order keyword nodes in the sorted set. The document id 1 is added as the label for all the edges created. The same procedure is repeated for document 2 as well. Since the nodes *Clinton* and *Trump* already exist, the document id 2 is directly added to the list of labels in the edge between *Clinton* and *Trump*. The resulting graph structure of GBI after indexing the keywords extracted from these two documents is presented in Figure 1. As can be seen, the graph represents relationships between keywords. For example, the keywords *Clinton* and *Trump* are related by the documents 1 & 2. Thus, a Boolean AND query "*Clinton* AND *Trump*" will directly return the document ids 1 & 2 without performing any intersection operation reducing the computation time.

Algorithm 1 exhibits the construction of GBI for BAQ execution. At a high level, Algorithm 1 gets extracted keywords from the document and establishes a relationship (connection) between them using the id of the document. The inputs are a set of keywords (S) and the identifier (Id) for the document. The output is the Graph-Based Index structure in the form of a hash table (G). A new GBI structure is created when the first document is indexed and gets updated when more documents are indexed. The condition is that at least two keywords should represent a document. The documents which result in only one keyword from the keyword extraction process can be ignored from indexing. Since BAQ needs at least two keywords to execute, these documents will not be part of any results. By ignoring the single keyword documents from indexing, the memory consumption of GBI is also reduced. Initially, the keywords in S are sorted in ascending order [Line 1]. If the algorithm is processing the last keyword in the set S, then the indexing process is complete. This is because all the necessary information about the last keyword can be found in preceding keywords hash table entries so there is no need of creating a separate entry for the last keyword in the set. This further helps the saving of memory space for GBI [Line 3-5]. If the current keyword under processing is not the last in S, then for each keyword in S, a series of entries are created in G. Each entry contains the current keyword as $key1$ and keyword occurring next to the current keyword in S as $key2$ with Id as value [Line 6-9]. This procedure is repeated for all the keywords in S. This results in creating edges between the keyword nodes with document ids as labels.

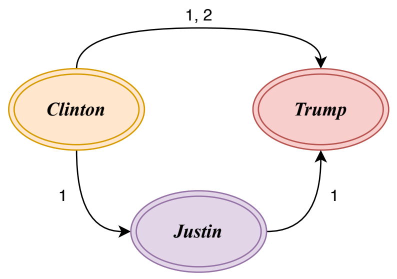

Fig. 1. The graph structure obtained after indexing documents 1 and 2.

IV. BAQ EXECUTION USING GBI

A Boolean AND search query finds a set of documents in which all the search keywords specified in the query appear together [14]. The position of the search keywords within a document is irrelevant. If any of the search keywords is not present in a document, then the document is discarded.

Consider an example where the keyword *Clinton* appears in documents *1, 2* and *5* and the keyword *Justin* appears in documents *1, 2, 3* and *4*. Then, a Boolean AND query "*Clinton* AND *Justin*" gives the result *1* and *2*. This is because *1* and *2* are the ids of the documents in which both *Clinton* and *Justin* appears together.

BAQ execution using GBI is achieved by executing Algorithm 2. Algorithm 2 takes a set of search keywords and finds whether the search keywords are connected in the GBI. If the keywords are connected, then an intersection between the keyword pair's postings lists is performed and the output is returned. If none of the search keywords are connected, then NULL is returned without performing any intersection operation. The inputs are set S containing the search keywords

Algorithm 1: Construction of GBI

```
Input: Set ($S$) of keywords and identifier ($Id$) of
        document
Output: New or updated hash table representing the
         Graph-Based Index ($G$)
Condition: At least two keywords exists in $S$
1  Sort set $S$ in ascending order
2  foreach keyword $key1$ in list $S$ do
3      if $key1$ is last entry in list $S$ then
           /* Indexing process is complete */
4          break
5      end
       /* When a document has more than one keyword
          to index */
6      foreach keyword $key2$ occurring after $key1$ do
7          $Key \leftarrow$ hash table key [$key1$][$key2$]
8          If an entry identified by $Key$ exists in $G$,
            append $Id$ to the list of values, otherwise
            create an entry using $Key$ with $Id$ as value.
9      end
10 end
```

and the GBI hash table G. The output is either an array containing the matching document ids or NULL. The condition for running the Boolean AND search algorithm is that at least two search keywords should be provided. Initially, S is sorted in lexicographical order and the size of S is determined [Line 1-3]. Then, the existence of an edge between all the search keywords is checked. The postings list is collected only if keyword pairs are non-overlapping or it is the last keyword pair in list S [Line 5-11]. This is to reduce the total number of postings lists needed. To illustrate this, consider four keywords *A, B, C* and *D*. Since the keywords are indexed in pairs (i.e. *AB, AC, AD, BC, BD*, and *CD*) only non-overlapping keyword pairs (in this case AB and CD) are needed to compute the intersection. Therefore, only the document ids of these two pairs are collected. Now, let's consider five search keywords say *A, B, C, D* and *E*. The postings lists will be collected for *AB, CD* and also the last pair *DE* as one more pair is needed to include keyword *E*. Thus, the document ids are collected from the last pair which is *DE*. If any of the search keywords are not connected to the rest of the search keywords or the keyword does not exist in GBI, then the search procedure stops as there is no possibility of a common document, and NULL is returned [Line 12-20]. If all search keywords are connected in the underlying graph structure of GBI and if there are only two search keywords, then the result is directly obtained as they are already indexed in pairs [Line 22-24]. If there are more than two search keywords, then the intersection operation is performed between the collected postings lists [Line 25-28]. In this paper, the traditional Small-Vs-Small (SvS) intersection algorithm is used for demonstration purposes. The SvS algorithm is chosen as it performs better when the data is stored as it is without any compression [16].

Algorithm 2: Boolean AND Search using GBI

```
Input: Set of search keywords ($S$), hash table
        representing the Graph-Based Index ($G$)
Output: NULL or the matching document ids.
Condition: At least two unique keywords exists in $S$
1  Sort $S$ in ascending order
2  $postings\_list\_collection \leftarrow$ an empty hash table
3  $search\_count \leftarrow$ size of list $S$
4  foreach keyword $key1$ in $S$ do
5      if $key1$ is not the last entry in $S$ then
           /* Check if all the search keywords are
              connected to each other in GBI */
6          foreach keyword $key2$ occurring next to $key1$
            in $S$ do
7              $Key \leftarrow$ hash table key [$key1$][$key2$]
8              if $Key$ exists in hash table $G$ then
9                  if $Key$ is non-overlapping or last
                    keyword pair in $S$ then
                       /* Collect selected postings
                          list */
10                     create entry in
                        $postings\_list\_collection$ with
                        $key1$ as key and postings list at
                        $G[Key]$ as value
11                 end
12             else
13                 $postings\_list\_collection \leftarrow$ NULL
                   /* If any of the search keywords
                      are not connected, then
                      terminate keyword relationship
                      checking. No possibility of
                      common documents */
14                 break
15             end
16         end
17         if $postings\_list\_collection$ is NULL then
               /* Terminate search process */
18             return NULL
19         end
20     end
21 end
22 if $postings\_list\_collection$ is not NULL then
       /* If all the search keywords are connected
          to each other then common documents are
          possible */
23     if $search\_count$ equals 2 then
           /* If only two search keywords then
              return result directly */
24         return $postings\_list\_collection$
25     else
26         Perform Small-Vs-Small (SvS) intersection
            algorithm on $postings\_list\_collection$ as
            described in Algorithm 3
27     end
28 end
```

Algorithm 3: Small-Vs-Small Intersection Algorithm [16]

```
Input: A hash table containing a set of postings lists ($P$)
Output: NULL or an array containing common document ids among the postings lists
Condition: At least two postings lists should be provided for performing intersection
1  Sort $P$ by the size of postings lists in ascending order
2  $output \leftarrow$ first postings list in $P$
3  remove first postings list from $P$
4  foreach remaining postings list $PL$ in $P$ do
5      foreach $document\_id$ in $output$ do
           /* remove document ids from the smallest postings list if it does not exists in other postings lists */
6          if $document\_id$ does not exist in $PL$ then
7              remove the $document\_id$ from $output$ array
8          end
9      end
10     if $output$ is empty then
11         $output \leftarrow$ NULL
12         break
13     end
14 end
15 return $output$
```

Algorithm 3 shows the execution of the SvS algorithm. Initially, the postings lists are sorted, and the smallest postings list is assumed as the candidate solution [Line 1-3]. Then, the SvS algorithm works by comparing the document ids in the smallest postings list with the rest of the collected lists. A document id is removed from the smallest postings list if it does not appear in other lists [Line 4-14]. Finally, the resultant document ids are returned [Line 15].

In GBI, the number of intersection operation performed depends on the number of connected search keywords (*n*). If $n > 2$ and n is even, then only *n/2 - 1* intersections are performed. If $n > 2$ and n is odd, then only $\lfloor n/2 \rfloor$ intersections are performed. If $n = 2$, then no intersection operation is performed.

In Inverted Index, an intersection operation is performed between individual keyword postings lists as opposed to keyword pairs postings lists as in the case of GBI. Also, in Inverted Index, at least one intersection operation is needed to validate that all the search keywords have a common document. For Inverted Index, the number of intersections performed is *n-1* for n keywords where $n > 1$. Thus, irrespective of the intersection algorithm that is used, GBI will always perform better than Inverted Index as GBI always uses a smaller number of postings lists and tries to avoid intersection.

V. Experiments and Performance analysis

This section focuses on experiments conducted to evaluate GBI in terms of indexing and BAQ search times and compare its performance with that of Inverted Index prototype and Elasticsearch. A proof of concept prototype is built for both GBI and Inverted Index using the PHP scripting language version 7.3. Elasticsearch is a popular search engine built using the Apache Lucene[4]. It provides full-text searching capabilities and uses Inverted Index as its index structure. Apache Lucene uses a skip list for intersection operations [9]. Therefore, Elasticsearch uses a skip list for performing intersection operations. Though the prototypes built by us can support the skip list, for simplicity, the skip list is not currently used in the prototypes. Incorporating the skip list in our prototypes forms an important direction for future research. The experiments are performed on the Carleton University Research Computing and Development Cloud (RCDC) system[5] with the following resource specification: IBM POWER8E processor with 64 GB memory running Ubuntu as the operating system. GBI and Inverted Index prototypes are implemented using a hash table. The SvS algorithm with hash table lookup as a searching methodology is used with the two indexing prototypes for performing the intersection operations. Each experiment was repeated 10 times and the resulting values were averaged. A standard deviation of less than 5% is observed for the multiple runs of each experiment presented in this paper. Two different categories of experiments are performed for evaluating the performance of GBI, Inverted Index, and Elasticsearch.

Experiment Category 1 - In this category, the documents are created by randomly selecting the keywords from a fixed keyword pool. The goal of this experiment category is to analyze the performance of GBI, Inverted Index, and Elasticsearch when the number of documents that contain the search keywords is not known *apriori*.

Experiment Category 2 - In this category, the set of keywords that a document can contain is selected beforehand from a given pool of keywords to control the way the multiple keywords can co-occur within a document. The goal is to study the impact of the relationship between the search keywords.

A. Elasticsearch Setup

One instance of Elasticsearch version 7.6 is used for experimentation. The heap memory required for Elasticsearch instance to run is set to 30 Gigabytes as recommended[6]. The Elasticsearch-PHP client API is used to send requests and receive responses from Elasticsearch. OpenJDK version 11 is used in the system and G1 garbage collection is enabled for Elasticsearch. Heap memory is allocated in such a way to avoid garbage collection during experimentation. Elasticsearch is configured to use only one shard and no replicas and to store only document ids in its Inverted Index. Elasticsearch, by default, enables scoring of the results and returns only the

[4]https://www.elastic.co/what-is/elasticsearch
[5]https://carleton.ca/rcs/research-computing-and-development-cloud-rcdc/
[6]https: //www.elastic.co/guide/en/elasticsearch/reference/current/heap-size.html

top ten results. The result also contains source data from which the keywords are extracted. For the experimentation purpose, the scoring of the document is disabled and Elasticsearch is configured to return all the matching document ids at once ignoring the source data. Caching of the search results is also disabled in Elasticsearch to get a consistent result while running an experiment multiple times. Indexing is done in bulk with refresh interval disabled for better indexing performance[7].

B. Workload Generation

In this section, the workload generations for the experiments conducted are discussed. Table I presents the list of workload parameters and their values. A factor at a time experimentation is performed: one parameter is varied while the other parameters are held at their default values (presented in boldface in Table I).

TABLE I
WORKLOAD PARAMETERS

Parameter	Values	Description
D_{count}	2M, 4M, **6M**, 8M, 10M	Number of documents
SK_{count}	2, **4**, 6, 8, 10	Number of search keywords
K_{pool}	An array of unique names	Keyword pool
$K_{poolsize}$	4, **10**	Size of keyword pool

1) Workload generation for Experiment Category 1: To generate the documents needed for this experiment category, a fixed keyword pool, (K_{pool}) is defined with unique names. The size of the keyword pool ($K_{poolsize}$) is 10 for this experiment category. A document is generated randomly by selecting the keywords from the keyword pool. There are two integer random numbers (X_{random} and Y_{random}) involved in the process of creating a set of keywords for a document to be indexed. The two random numbers follow a uniform distribution U(1, 10). X_{random} specifies the number of keywords that the document will contain. Y_{random} determines which keyword from K_{pool} goes into that document. After generating X_{random}, a loop is ran for X_{random} number of times. In each iteration, a random number Y_{random}, which specifies the index of the keyword in the array K_{pool}, is generated. Y_{random} is generated repetitively until a unique keyword is drawn from K_{pool} which is not in the set of already selected keywords. Finally, a set of unique keywords for that document is generated after the termination of the loop. The indexing algorithm described in Algorithm 1 is used to construct GBI and the same set of keywords is also used to construct Inverted Index. This procedure is repeated for generating D_{count} documents.

2) Workload generation for Experiment Category 2: In this category, we define three different type of relationship between the search keywords: *No*, *Partial* and *Full* relationship. These relationships are discussed next.

No relationship: A *No* relationship between keywords refers to the situation in which none of the search keywords appear together in any of the documents indexed.

Partial relationship: A *Partial* relationship between keywords refers to the situation in which some keywords among the entire search keyword set appear together in one or more documents.

Full relationship: A *Full* relationship between keywords refers to the situation in which all the keywords in the search keyword set appear together in one or more documents.

To perform experiments for these three relationships, SK_{count} and $K_{poolsize}$ are fixed to 4. This is because most Google search queries have less than 4 keywords in length[8]. In the three relationship experiments, all the 4 keywords in K_{pool} are utilized. The ways the 4 keywords are utilized to create documents are described next.

In the experiment for *No* relationship, the documents are created in such a way that no search keywords among the 4 keywords appear together in any documents. To achieve this, each of the D_{count} document indexed contains only 1 of the 4 search keywords.

In the experiment for *Partial* relationship, the documents are created in such a way that any 2 or 3 keywords out of 4 search keywords appear together in one or more documents such that all the keywords do not appear together in any documents out of the D_{count} documents indexed.

In the experiment for *Full* relationship, the documents are created in such a way that all the 4 search keywords appear together in one or more documents. To achieve this, the total number of documents D_{count} to index is divided into 2 parts. One half of the documents contains all the search keywords together and the other half of the documents contains only 2 keywords together (out of 4). This is done to avoid all the documents indexed containing all the search keywords together. All the documents indexed containing all the 4 search keywords is a special case of the experiments concerning the *Full* relationship. The motivation for doing this special experiment is to analyze how much better GBI performs in comparison to Inverted Index in the worst-case scenario of all the documents containing all the search keywords. This is the worst case because the size of the postings list of all the search keywords (in Inverted Index) or keyword pairs (in GBI) is equal to the total number of documents D_{count}. Thus, it leads to the case where the smallest postings list will contain the ids of all the documents indexed, with each id being compared with the rest of all the search keyword's postings list leading to the worst-case search time possible.

C. Performance Metrics

Table II describes the performance metrics used to evaluate GBI, Inverted Index, and Elasticsearch. Three performance metrics are collected: T_s, T_i and M_i. A definition of each metric is provided next.

- T_s: The search latency associated with BAQ execution. It is the difference between the end and start times of the Boolean AND search algorithm in GBI and Inverted Index. To avoid any additional overhead, the search and

[7]https://www.elastic.co/guide/en/elasticsearch/reference/current/tune-for-indexing-speed.html

[8]https://www.wordstream.com/blog/ws/2019/02/07/google-search-statistics

TABLE II
PERFORMANCE METRICS

Parameter	Description	Unit
T_s	Boolean AND query search execution time	milliseconds (ms)
T_i	Indexing time for a set of documents	seconds (s)
M_i	Index memory consumption	Megabytes (MB)

indexing times are measured by a set of info APIs provided by Elasticsearch. The info APIs provide a detailed report on the time and memory consumed when handling a search or indexing request. For measuring a query execution time, Elasticsearch has provided Profile API for obtaining time consumed by each component of the search operation[9]. Hence, for Elasticsearch, T_s is the time consumed for a Boolean query execution reported in the "BooleanQuery" field of the Elasticsearch Profile API response.

- T_i: The latency associated with indexing a set of documents. To calculate T_i for GBI and Inverted Index, the difference between the end and the start times for indexing each document is calculated and all the resulting latencies are added to get the final value. For measuring indexing time in Elasticsearch, Index stats API provided by Elasticsearch is utilized[10]. Hence, for Elasticsearch, T_i is the time value reported in "index.time_in_millis" field of the Index stats API response.
- M_i: Amount of memory consumed for indexing a given set of documents. For GBI and Inverted Index, M_i is the amount of main memory consumed by the GBI and Inverted Index hash table. M_i for Elasticsearch is the memory value reported in the "store.size_in_bytes" field of the Index stats API response.

The results of the experiments are presented in the following sections. Note that a log scale has been used for the Y-axis in figures 3, 4, 8, and 9.

D. Analysis of the Search Latency in Experiment Category 1

Two experiments (Experiment A and B) of this category have been conducted based on varying the two workload parameters - D_{count} and SK_{count}, one at a time.

In Experiment A, the number of documents indexed D_{count} has been increased from 2M to 10M in steps of 2M while SK_{count} is held at 4. The 4 keywords to search in the index are also chosen from K_{pool} such that all the search keywords occur together in one or more documents. Figure 2 shows that as D_{count} increases, the search time (T_s) for Inverted Index is increasing at a much faster rate compared to GBI. This is because as D_{count} increases, there is a sharp growth in the size of the smallest postings list among all the search keywords postings lists in Inverted Index. This leads to the comparison

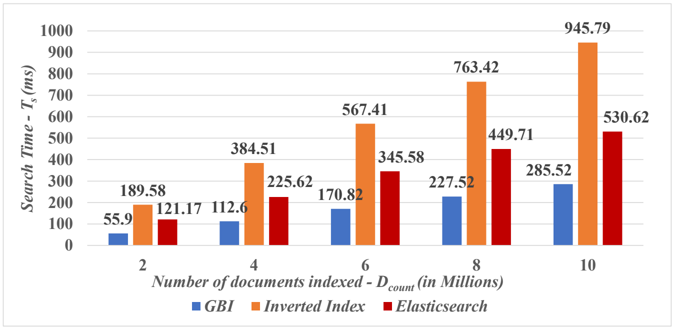

Fig. 2. Effect of number of documents indexed (D_{count}) on the search latency (T_s) when the number of search keyword (SK_{count}) is kept at the constant value of 4.

of more document ids. Even if the postings list for each search keyword pairs also grows with D_{count} in GBI, only a single intersection operation is needed in this case as discussed in Algorithm 2. The presence of a skip list helped Elasticsearch to have a lower search latency than the Inverted Index prototype. In GBI, though the postings list is traversed normally, the reduction in the number of postings list and document ids to be compared for intersection operation lead to a smaller GBI search time. Thus, GBI shows an average improvement of 70% and 50% over Inverted Index and Elasticsearch respectively for this experiment.

In Experiment B, D_{count} has been fixed at 6M and SK_{count} is increased from 2 to 10 in steps of 2. Figure 3 shows that for any given SK_{count}, GBI is performing better than Inverted Index. The search time (T_s) is almost negligible (around 0.03 ms) for GBI with a search latency improvement of 99.9% when there are only two keywords involved. This is because the ids of the documents are stored in keyword pairs so single direct access (hash table lookup) gives the result in GBI. For $SK_{count} > 2$, as the number of search keyword increases, the number of intersection operations increases in Inverted Index resulting in higher search time latency. Although the number of intersection operations increases in GBI as well, it will always be lower than Inverted Index as discussed in Section IV. This results in GBI performing well even when the number of search keywords increases. Similar to Experiment A, the skip list helped Elasticsearch to avoid a large increase in search time. For $SK_{count} > 2$, the average overall improvement in GBI is 62% and 24% in comparison to Inverted Index and Elasticsearch respectively.

E. Analysis of the Search Latency in Experiment Category 2

Figure 4 shows the performance of GBI, Inverted Index, and Elasticsearch when the search keywords do not appear together in any document (i.e. *No* relationship). For a given D_{count}, GBI shows a significant performance improvement of 99.9% over the two others with a negligible search time (T_s) of around 0.02 ms. This high performance is because GBI checks whether all the search keywords occur together in any of the documents before performing an intersection. Checking for the simultaneous occurrence of two keywords

[9]https://www.elastic.co/guide/en/elasticsearch/reference/current/search-profile.html
[10]https://www.elastic.co/guide/en/elasticsearch/reference/current/indices-stats.html

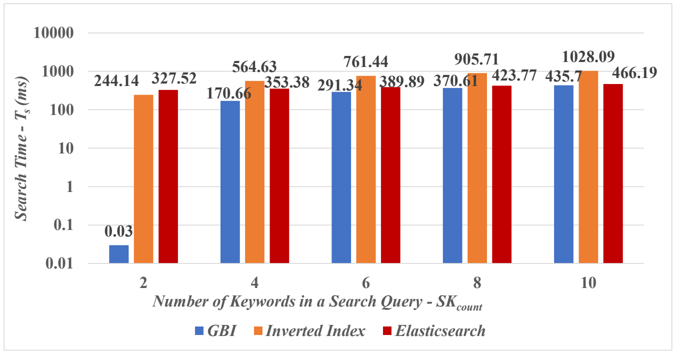

Fig. 3. Effect of number of search keyword (SK_{count}) on search latency (T_s) when number of documents indexed (D_{count}) is kept at the constant value of 6M.

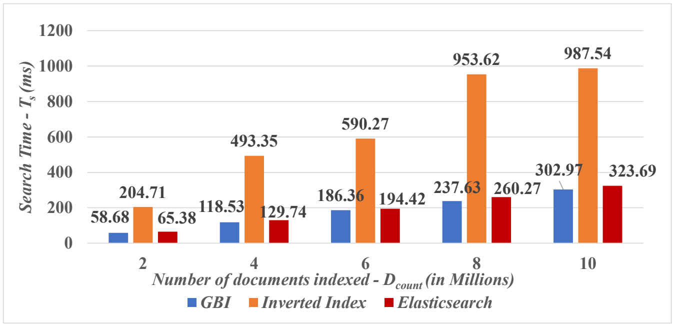

Fig. 5. Effect of number of documents indexed (D_{count}) on search latency (T_s) for a workload with *Partial* relationship among search keywords.

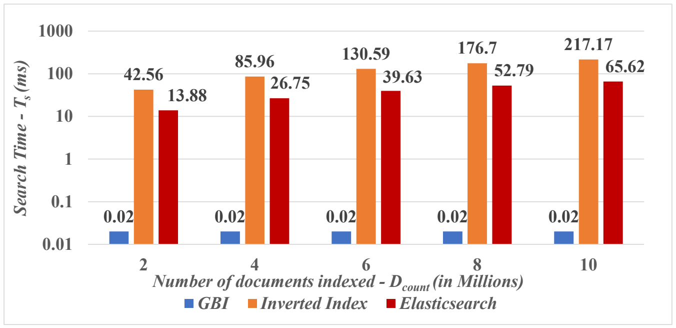

Fig. 4. Effect of number of documents indexed (D_{count}) on search latency (T_s) for a workload with *No* relationship among search keywords.

in a document is a hash table lookup in GBI. Since there is no simultaneous occurrence of all the search keywords, GBI stops further action and does not perform any intersection operation. If there are more connections between keywords to check, then the execution time for GBI increases. Since the hash table lookup takes a negligibly small amount of time, the search time for checking more connections does not increase significantly. As the number of search keywords is kept at a constant value of 4, the number of connections to check remains the same. Thus, a similar time is observed for GBI for different values of D_{count}. With Inverted Index, the execution time for BAQ depends on the size of the smallest postings list among the search keywords. As the total number of documents indexed (D_{count}) increases, the size of the smallest postings list also increases leading to an increase in the search time for Inverted Index. With Elasticsearch, the presence of skip list had reduced the search time but the search time increases as D_{count} increases, unlike GBI for which it stays the same irrespective of D_{count}.

Figure 5 shows the performance of GBI, Inverted Index, and Elasticsearch when some of the search keywords appear together in one or more documents (i.e. *Partial* relationship). For a given D_{count}, once again, GBI shows a performance improvement of around 72% and 8% over Inverted Index and Elasticsearch respectively. As per the partial relationship workload, any 2 or 3 keywords out of 4 keywords can appear together in a document to be indexed. It is observed from the generated workload that though there is no document with all the keywords appearing together, each of the keywords appears together with other keywords in some documents. This means that there is a connection between all of the keyword nodes in the GBI structure. Thus, it passes the initial connection verification. Since there are only 4 keywords, a single intersection operation is enough to obtain the result. With Inverted Index, the execution time depends on how many keywords among the search keywords set appear together in a document. If more keywords are appearing together in many documents, then Inverted Index takes more time to perform the intersection. In this case, 3 intersections are needed to obtain the result. With Elasticsearch, the presence of the skip list helped bring down the search time close to GBI but still, GBI performs better. GBI traverses the postings list normally meaning GBI visits all the document ids in the postings list, unlike Elasticsearch which uses the skip list to avoid visiting certain document ids. Without the presence of a skip list, GBI can perform better in this case. Thus, it is expected that the usage of skip list in GBI can further lower the search time leading to an even higher performance improvement in comparison to Elasticsearch.

Figure 6 shows the performance of the GBI, Inverted Index, and Elasticsearch when all the search keywords occur together in some of the documents indexed (i.e. *Full* relationship). Figure 7 corresponds to the special case of the *Full* relationship experiment where all the search keywords appear in all the documents indexed. In both figures, for any given value of D_{count}, GBI shows better performance over Inverted Index and Elasticsearch. This is because intersections in GBI are done in pairs and so the number of intersections performed is only one as opposed to three intersections performed with Inverted Index as discussed in Algorithm 2. Elasticsearch shows a higher search time than GBI. This is because all the search keywords appear in all the documents indexed. Hence, more number of document id comparison has to be performed. The presence of the skip list in Elasticsearch does not seem to offer much help for these workloads as none of the skip pointers in the skip list can be avoided. In both cases, the performance improvement achieved by GBI is becoming much

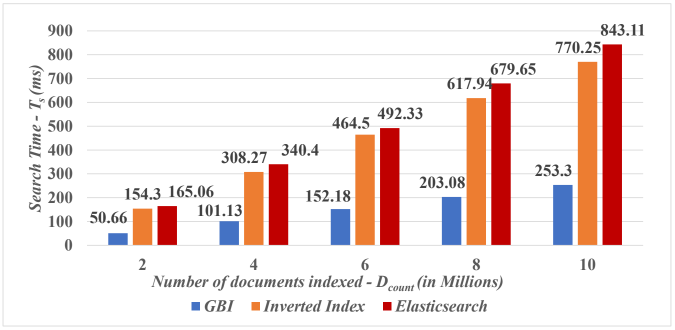

Fig. 6. Effect of number of documents indexed (D_{count}) on search latency (T_s) for a workload with *Full* relationship among search keywords.

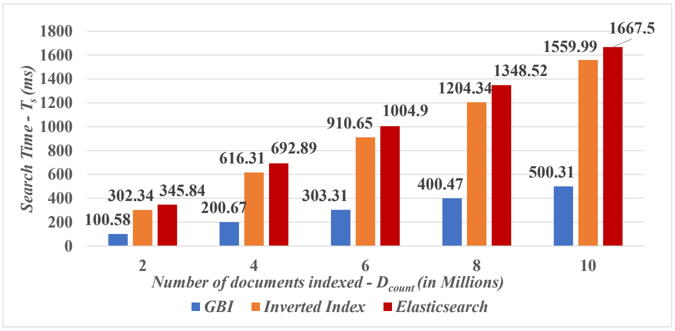

Fig. 7. Effect of number of documents indexed (D_{count}) on search latency (T_s) for a workload with *Full* relationship among search keywords when the search keywords appear in all the documents indexed.

more evident for larger values of D_{count} with an overall average improvement of approximately 67% and 70% over Inverted Index and Elasticsearch respectively.

F. Analysis of the Indexing Performance

In this subsection, GBI is compared with Inverted Index, and Elasticsearch on the time and memory required to index D_{count} set of documents. The random workload used for Experiment Category 1 is used for analyzing the indexing performance. In the first experiment, the number of documents D_{count} has been increased from 2M to 10M in steps of 2M and the time for indexing these documents in GBI, Inverted Index, and Elasticsearch is calculated. In the second experiment, the amount of memory used while performing indexing is determined. As described in Section V-B1, the number of keywords in a given document is generated using a uniform distribution. The results of the two experiments can be seen in Figure 8 and 9 respectively.

In Figure 8, it is observed that Inverted Index is performing better than GBI. This is evident as in Inverted Index, each keyword has its postings list. But in the case of GBI, relationships are established in pairs of two keywords. Thus, there will be a postings list for every such keyword pairs. In GBI, for n keywords (nodes), the number of postings lists needed are nC_2 (n choose 2). Whereas in Inverted Index, for n keywords, n postings lists are created. GBI takes time to create and form connections between these keywords and hence the increase

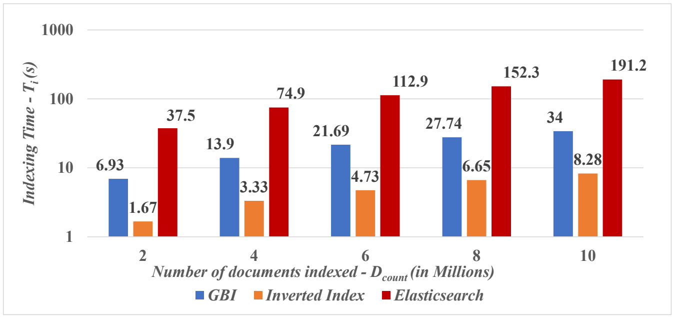

Fig. 8. Effect of number of documents indexed (D_{count}) on indexing latency (T_i).

in indexing time. For any given D_{count}, the indexing time with GBI is higher than what is required with Inverted Index. The indexing time for Elasticsearch is higher than GBI. In the proposed experimental prototypes, the time taken for indexing the keywords alone is considered. In the case of Elasticsearch, Index stats API gives the complete indexing time for the total number of documents indexed. Therefore, the time reported by Elasticsearch may include the time for processing the request, analyzing the keywords, and then indexing in its index. Thus, T_i for Elasticsearch is higher than GBI. Further investigation into analyzing the individual component of indexing time in Elasticsearch is an important component of future research.

Figure 9 infers that Inverted Index is performing better than GBI. For any given D_{count}, M_i is higher in GBI than Inverted Index. It can also be noted that the memory usage is the same for GBI when D_{count} ranges from 6M to 10M. This can be explained in terms of hash table resizing. To accommodate more data, the hash table uses strategies like increasing the size in the power of two to avoid resizing often[11]. This large increase in size gives the capacity for more data. It is because of the same reason that there is a large difference in memory consumption between GBI and Inverted Index at D_{count} = 6M when GBI expands its hash table capacity before Inverted Index does. To conclude, GBI consumes more memory than Inverted Index. M_i for Elasticsearch is much lower in comparison to the two other prototypes. This is because Elasticsearch compresses and stores its index on a disk. Elasticsearch uses a lossless compression algorithm called LZ4 to compress its data[12]. Thus, the significant reduction in search time of GBI comes at the cost of spending more time in performing the indexing operations. GBI is nevertheless preferable to Inverted Index in the common use case where the indexing operations are less frequent than the search operations.

VI. Conclusions

Searching for relevant content through a large volume of textual data is a difficult problem. In this paper, a graph-based indexing technique is proposed to effectively evaluate Boolean

[11]https: //nikic.github.io/2014/12/22/PHPs-new-hashtable-implementation.html

[12]https: //www.elastic.co/blog/store-compression-in-lucene-and-elasticsearch

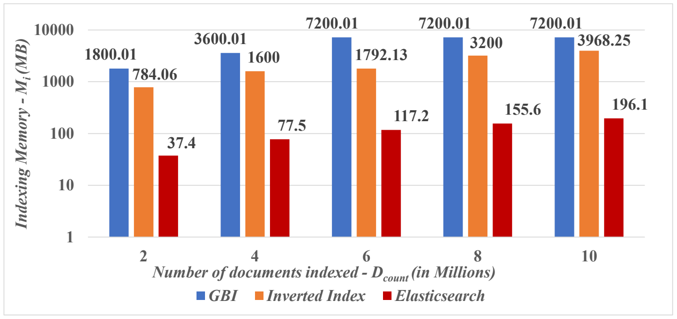

Fig. 9. Effect of number of documents indexed (D_{count}) on memory consumed (M_i).

AND queries. Key insights resulting from the performance analysis of GBI, Inverted Index, and Elasticsearch for the experiments reported in this paper are presented next.

1) GBI shows a significant performance improvement over Inverted Index and Elasticsearch when executing BAQ in randomly created documents for different range of values of D_{count} and SK_{count} (see Section V-D). This is because GBI always performs intersection in pairs and the number of postings lists needed in GBI is always less than that for Inverted Index or Elasticsearch. (see Section IV).
2) GBI shows an outstanding performance improvement over Inverted Index and Elasticsearch when none of the search keywords appear together in any of the documents indexed (i.e. *No* relationship) (see Section V-E). This is because GBI takes advantage of checking whether the search keywords are connected before performing any intersection operations. If such a connection does not exist, then the expensive intersection operations are avoided.
3) GBI shows a notable performance improvement over Inverted Index and Elasticsearch when executing BAQ containing some of the search keywords appear together in a set of documents indexed (i.e. *Partial* relationship) (see Section V-E). It is expected that the usage of a skip list in GBI can lower the search time further.
4) As reported in Section V-E, GBI shows a remarkable performance improvement over Inverted Index and Elasticsearch when all the search keywords appear together in one or all the documents indexed (i.e. *Full* relationship). Thus, GBI shows superior performance even for the case where all the documents indexed need to be returned as a result of a Boolean AND query.

Directions for further research include:

1) The memory and indexing time for GBI is higher than Inverted Index (see Section V-F). This is because the same document ids for the different keyword pairs are stored multiple times. Further Investigations to reduce memory consumption and indexing time for GBI are warranted.
2) Research is ongoing to expand GBI capability to address other queries such as single keyword queries, Boolean NOT, and OR queries. Studying the effect of change in keyword pool size on indexing performance also forms an important direction for future research.

Acknowledgment

We are thankful for the computing resource support provided by RCDC of Carleton University. We are also grateful to TELUS and the Natural Sciences and Engineering Research Council of Canada (NSERC) for providing financial support for this research.

References

[1] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin, "Early-bird: Real-Time Search at Twitter," in *2012 IEEE 28th International Conference on Data Engineering*, 2012, pp. 1360–1369.
[2] C. D. Manning, H. Schütze, and P. Raghavan, *Introduction to Information Retrieval*. Cambridge university press, 2008.
[3] A. Chakraborty, T. Dutta, S. Mondal, and A. Nath, "Application of Graph Theory in Social Media," *International Journal of Computer Sciences and Engineering*, vol. 6, no. 10, pp. 722–729, 2018.
[4] A. K. Mohideen, S. Majumdar, M. St-Hilaire, and A. El-Haraki, "A Graph-Based Indexing Technique to Enhance the Performance of Boolean AND Queries in Big Data Systems," in *20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020, pp. 677–680.
[5] A. Moffat and J. Zobel, "Self-Indexing Inverted Files for Fast Text Retrieval," *ACM Transactions on Information Systems (TOIS)*, vol. 14, no. 4, pp. 349–379, 1996.
[6] R. Nagmoti, A. Teredesai, and M. De Cock, "Ranking Approaches for Microblog Search," in *2010 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, 2010, pp. 153–157.
[7] C. Chen, F. Li, B. C. Ooi, and S. Wu, "TI: An Efficient Indexing Mechanism for Real-Time Search on Tweets," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, 2011, pp. 649–660.
[8] R. Sharma, A. Kandpal, P. Bhakuni, R. Chauhan, R. H. Goudar, and A. Tyagi, "Web Page Indexing through Page Ranking for Effective Semantic Search," in *2013 7th International Conference on Intelligent Systems and Control (ISCO)*, 2013, pp. 389–392.
[9] S. Pohl, A. Moffat, and J. Zobel, "Efficient Extended Boolean Retrieval," *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, no. 6, pp. 1014–1024, 2012.
[10] V. N. Anh and A. Moffat, "Pruned Query Evaluation Using Pre-Computed Impacts," in *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, 2006, pp. 372–379.
[11] V. N. Anh, O. de Kretser, and A. Moffat, "Vector-Space Ranking with Effective Early Termination," in *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, 2001, pp. 35–42.
[12] S. Yata, K. Morita, M. Fuketa, and J. Aoe, "Fast String Matching with Space-efficient Word Graphs," in *2008 International Conference on Innovations in Information Technology*, 2008, pp. 79–83.
[13] M. S. El Bazzi, D. Mammass, T. Zaki, and A. Ennaji, "A graph based method for Arabic document indexing," in *2016 7th International Conference on Sciences of Electronics, Technologies of Information and Telecommunications (SETIT)*, 2016, pp. 308–312.
[14] A. K. Mahapatra and S. Biswa, "Inverted indexes: Types and techniques," *International Journal of Computer Science Issues (IJCSI)*, vol. 8, no. 4, pp. 384–392, 2011.
[15] S. Shah and A. Shaikh, "Hash Based Optimization for Faster Access to Inverted Index," in *2016 International Conference on Inventive Computation Technologies (ICICT)*, 2016, pp. 1–5.
[16] J. S. Culpepper and A. Moffat, "Efficient Set Intersection for Inverted Indexing," *ACM Transactions on Information Systems*, vol. 29, no. 1, pp. 1:1–1:25, 2010.