Full-Text Search on Data with Access Control using Generalized Suffix Tree

Ahmad Zaky
School of Electrical Engineering and Informatics
Institut Teknologi Bandung
Bandung, Indonesia
13512076@std.stei.itb.ac.id

Rinaldi Munir, S.T., M.T.
School of Electrical Engineering and Informatics
Institut Teknologi Bandung
Bandung, Indonesia
rinaldi.munir@itb.ac.id

Abstract—**Full-Text Search combined with access control have a wide range of applications, for example in a multi-users system that allows each user to create their own content (e.g. blog or social media). Unfortunately, there are only few (if not zero) researches that combine the two. It is also not supported in DBMS or modern search engines. The implication is that one should make their own implementation of full-text search with access control. While inverted index is already used widely for full-text searching, we try to use generalized suffix tree for its ability to search for any substring within a document, not only exact word occurrence. Theoretically, the time and memory needed to index a collection of documents is linear in the total size of the documents. However, our implementation requires memory more than 1200 times of the size of documents. A further analysis shows that at least 32 times is needed, but it will require longer indexing time. In conclusion, generalized suffix tree may not suitable for large amount of data. In the other hand, the search using generalized suffix tree is 3 times faster than inverted index. Suffix tree can be used only if substring search is mandatory (e.g. DNA processing) or where time is significantly more important than memory (e.g. search autocomplete system). The access control itself acts as filter after the documents yielded from searching through the index.**

Keywords—access control; full-text search; generalized suffix tree; indexing; inverted index

I. Introduction

Full-Text Search (FTS) is a pattern matching that allows one to search for multiple keywords at a time. It is important in information retrieval system as it increases the relevance of the search result. FTS has a wide range of use, especially in this era of computing. Pretty much every websites and web applications that support searching are using FTS for using traditional exact string matching yields less relevant results. One good example is Google, who is doing full-text indexing for almost all websites on the Internet.

On the other hand, combining FTS with access control will widen the application. Access control will limit the search result based on the user's access right. Each document has their own access control list (ACL), and the ACL will be compared to the access right to determine whether a certain user has the right to access certain document or not. Social media has applied FTS with access control to make sure that each content that is published by their users are not served to the unauthorized ones. Users has full control to set the ACL of each content they own. For example, users on Facebook can set who can see their posts and photos. Each post can be set independently.

Although there are a lot of researches on FTS already [1], [3], [4], [7], [8], there are so little (if not zero) researches about FTS when combined with access control. Various DBMS (e.g. MongoDB and PostgreSQL) and search engines (e.g. Lucene/Elasticsearch) already support FTS, but they do not support FTS with access control out of the box. It means that every system should implement it on their own. This is usually done in the application layer.

The idea is simple: to use the state-of-the-art FTS algorithms and modify them so they can support data with access control. As FTS is usually used for handling documents in large scale, and the users expect the result to appear instantly, then the evaluation is around the size of the index and the performance of search in terms of time.

This research paper will limit the scope to only the searching part. There are several steps that should be done in an information retrieval (IR) system. Some of them are (1) query refinement, (2) the document retrieval itself, and (3) ranking the results based on relevance of the documents and the query [5]. This research will focus on the second step, therefore transforming the whole IR problems into a string matching problem.

II. Full-Text Search with Access Control

FTS can be formulated as follows. Given a set of documents D, where each document $d \in D$ can be considered as a string, which represents the content of the document itself. Another representation of a document is considering them as a sequence of words. A query Q is defined as an array of strings, where each string represents a word in the query. A document d is said to

978-1-5090-5671-2/16/$31.00 ©2016 IEEE

match a query Q if and only if every word in the query appear in d as a substring or as a whole word when considering document as a sequence of words.

Formally, the abovementioned formulation can be written as follows. $Substring(a, b)$ is a function that checks whether a is a substring of b or not.

$$Search(Q, D) \rightarrow R \ni \forall d \in R, \forall q \in Q, Substring(q, d)$$

Before that part is done, it involves indexing, that is transforming the set of documents to some data structure to make the searching faster. So, often the search is performed on the *index* itself not the original set of documents. The indexing process can be viewed as a function $Index(D) \rightarrow S$ where S is the data structure, and the search is performed on S.

Appending access control to FTS increases the complexity of the problem. Each document has their own access control, and the search function includes an additional parameter, the user's access rights. The final result should conform with the access right itself. The formulation is as follows, where $ACL(d)$ returns the document's ACL, and $Authorized(ar, acl)$ determines whether a user with access right ar is allowed to access resources with ACL acl.

$$Search(Q, ar, D) \rightarrow R \ni \forall d \in R, \forall q \in Q, Substring(q, d) \land Authorized(ar, ACL(d))$$

III. Related Works

As mentioned in the introduction, there are so little researches about the subject. However, there are many researches about FTS itself. There are two major algorithms that are used for FTS: inverted index and suffix tree.

A. Inverted Index

Inverted index is the most commonly used method for FTS. It is, for example, used by the number one open-source search engine, Elasticsearch [2]. The idea of inverted index is to create a map from the word to the occurrence in the documents. Table 1 shows an example of inverted index. When searching "QUICK BROWN FOX" to the index, the result is $\{4,8\}$, the intersection between the set of documents that is containing "QUICK", "BROWN", and "FOX".

Table 1. Inverted index example

Term	Document IDs
QUICK	1, 2, 4, 8
THE	1, 2, 3, 4, 5, 6, 7, 8, 9, 10
BROWN	2, 3, 4, 6, 8
DOG	3, 5, 7
FOX	3, 4, 5, 7, 8, 10
IN	1, 2, 4, 5, 6, 8, 9
JUMP	3, 10
LAZY	2, 4, 5, 6, 8, 10
LEAP	3, 5, 7, 9, 10
OVER	1, 3, 4, 5, 6, 9
SUMMER	5, 6

Main advantage of inverted index is the size of the index itself. On most cases, the size of the index is much smaller than the document's size. Elasticsearch reports that the size of index is only 50%-130% of the whole document's size [6]. But in terms of complexity, it is linear with respect to the total length of the documents.

The complexity time of the search is proportional to the time of retrieving the document IDs and calculating the intersection of those sets of IDs. It depends on the data structure which implements the map. Hash map will achieve constant time of retrieval, but will perform poorly on larger dataset. B-Tree is preferred, with complexity logarithmic to the number of terms. The intersection itself can be calculated linearly with respect to the number of results.

B. Suffix Tree

Suffix tree is a prefix tree which contains all suffixes of some string. Storing them as is will require unnecessary huge amount of memory, since the number of nodes in the resulting tree is quadratic of the length of string. One way to optimize it is to squeeze a sequence of nodes which has only one child into one, and represent the edge as a pair of index which refers to a substring of the original string. Figure 1 shows an example for this. The construction can be performed in time and memory linear in the length of the string. The algorithm is explained in [1].

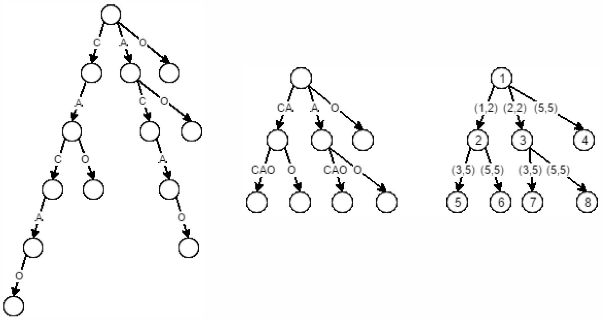

Figure 1. Suffix tree of string CACAO. The left one is the raw prefix tree of the suffixes, the middle one is the tree after compressing the nodes, and the right one is the final representation of the tree using reference index to the original string

Determining whether a string P is a substring of a string T is easy when we already have suffix tree for T: just traverse the tree from the root following the characters in P. The substring exists if and only if there is some node (explicitly or implicitly) in the tree that represents P. The time complexity is linear in the length of P and the memory is constant.

The plain suffix tree cannot handle FTS right away, for it can only hold one single string. There is a workaround for this: to append all documents into one and create a suffix tree for the concatenated string. Each leaf should contain the position of the suffix, and the ID of the document can be derived from the position.

IV. Proposed Method

While inverted index is the most commonly used method for FTS, it has one little limitation: it can only search for exact word matches. This can be handled by normalization before indexing the whole documents (*achieve*, *achieved*, *achieving* will become *achiev*). But, the proposed method is to use suffix tree for its ability to find any substring. In the other hand, searching in suffix tree is theoretically faster than inverted index, because searching in inverted index requires to search in map which takes logarithmic time.

But in order for suffix tree to work with multiple documents, we will not use concatenation, but use a more appropriate method named *generalized suffix tree* (GST). GST is a suffix tree that can store multiple strings. The construction algorithm is very similar to the ordinary suffix tree. The differences are:

1. Edges are represented by three integers, x, k, p which represents substring $D[x][k..p]$ of the x-th document.
2. Each leaf has special attribute, which indicates the ID of the document it represents.

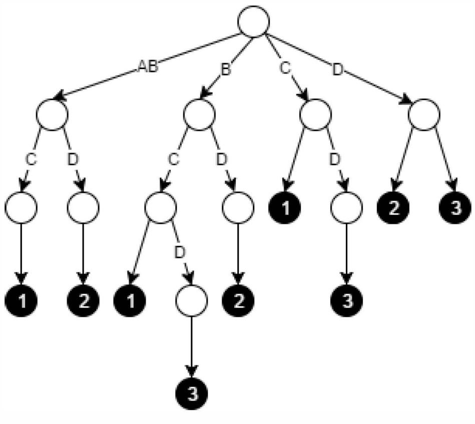

Figure 2. GST for set of strings $\{ABC, ABD, BCD\}$. The black nodes represent the ID of document of each suffix. A suffix may have more than one black nodes, which means that there are multiple documents that contain it.

Figure 2 shows an example of GST containing the strings "ABC", "ABD", and "BCD". Each suffix has a special child-node (shown as black leaf) which indicates the ID of the document the suffix belongs to. Searching is as simple as traversing the tree from root according to the query. When searching for string "BC", for example, we traverse from the root to node "BC", and we find that all IDs of documents shown in the subtree of the node are $\{1,3\}$. This means that documents 1 and 3 contain "BC" as their common substring.

Some modifications of construction algorithm in [1] are necessary. Four algorithms below show the modified algorithms of the four procedures given in [1]. Other modification that is made is that the auxiliary state is omitted in this version. The complexity of time and memory is still linear in the total length of the documents.

The algorithm can be explained as follows. Construction process is performed one character at a time for each document, as shown in Figure 3. The algorithm calls GST-Update for each character, which is shown in Figure 4. GST-Update tries to insert one character to the tree, making new branches as necessary. Function GST-Test-And-Split given in Figure 5 tests whether current state needs a new branch or now. GST-Update does this repeatedly, from currently active point, following the suffix link (a link that connects each node to other node which represent the suffix one character less than it) until it reaches root or no more split is necessary. Lastly, GST-Canonize as shown in Figure 6 is a helper function to canonize a state in the tree. It is called each time after splitting is done, and after GST-Update is done.

```
function GST-Construct (D)

 { Input: D is the set of documents for which the GST
 will be built.
    Output: GST of D }

    root = new Node
    S = new Set(Node)
    S.insert(root)
    g = new Map((Node, (integer, integer, integer)) ->
    Node)
    { f denotes the suffix links. f[r] = r' means that
    r' there is some character x such that r = xr' }
    f = new Map(Node -> Node)
    for i = 1 to length(D) do
        { (x, s, k) is the active state. }
        (x, s, k) = (i, root, 1)
        for j = 1 to length(D[i]) do
            (x, s, k) = GST-Update(s, (x, k, j))
            (x, s, k) = GST-Canonize(s, (x, k, j))
    return (root, S, g, D)
```

Figure 3. GST-Construct algorithm

```
function GST-Update (s, (x, k, i))

{ Input: (s, (x, k, i - 1)) is the canonical reference
of the current active point.
   Output: the new active point after i-th character
   is inserted }

oldr = root
(isEndpoint, r) = GST-Test-And-Split(s, (x, k, I -
1), D[x][i])
while (!isEndpoint) do
   insert new transition g(r, (x, i, ∞)) = r'
   if (oldr != root) then
      f[oldr] = r
   oldr = r
   if (r = root) then
      k = k + 1
   else
      s = f[s]
   (x, s, k) = GST-Canonize(s, (x, k, i - 1))
if (oldr != root) then
   f[oldr] = s
return (x, s, k)
```

Figure 4. GST-Update algorithm

```
function GST-Test-And-Split (s, (x, k, p), t)

{ Input: (s, (x, k, p)) is the canonical reference of
a state that is needed to be tested to be endpoint or
not in the process of inserting t.
   Output: (isEndpoint, r) where r is the new state
   upon the splitting of the input state }

if (k <= p) then
   (s', (x', k', p')) = D[x][k] transition from s
   if t = D[x'][k' + p - k + 1] then
      return (true, s)
   else
      replace D[x][k] transition by g(s, (x', k',
      k' + p - k)) = r and g(r, (x', k' + p - k +
      1, p') = s'
      return (false, r)
else if no t-transition from s then
   return (false, s)
else
   return (true, s)
```

Figure 5. GST-Test-And-Split algorithm

```
function GST-Canonize (s, (x, k, p))

{ Input: (s, (x, k, p)) is some state that is not need
to be canonical.
   Output: Canonical reference of the same state. The
   p value is omitted as it stays the same. }

if (p < k) then
   return (x, s, k)
else
   find D[x][k]-transition g(s, (x', k', p')) = s'
   while (p' - k' <= p - k) do
      k = k + p' - k' + 1
      s = s'
      if k <= p then
         find D[x][k] transition g(s, (x', k', p')
         = s'
   return (x, s, k)
```

Figure 6. GST-Canonize algorithm

What is left is to handle access control in the search process. The result of the search is filtered after we find all the documents matching documents by their content only. There are many advantages of handling access control this way:

1. Flexibility: the access control can be implemented in any way using any available methods.

2. Simplicity: the original string-matching algorithm itself is not changed.

3. Efficiency: the amount of memory used to index the documents is linear with respect to the *number* of documents, not the total length of documents.

Therefore, the index consists of two parts: the GST and the mapping between documents (or their IDs) to the corresponding ACL. The whole indexing process is straightforward; therefore, the full algorithm will not be included in this paper.

V. Implementation and Experiments

The proposal of using GST is tested against inverted index. The metrics that is took into account in the experiment is the performance of both of them in terms of time and memory of indexing and searching. There are some assumptions before the implementation takes place:

1. All processes will be done in the main memory without involving secondary memory e.g. disk (except for when the documents are loaded for the first time). This significantly simplifies the implementation.

2. The access control will be implemented using access control matrix. There will be randomly generated ACLs and access rights and the ACLs will be randomly assigned to the documents.

The specification of the testing environment is shown in Table 2. The algorithms will be implemented using Java. The dataset used in the experiment is OHSUMED test collection, which is the abstracts from 270 medical journals over a five-year period (1987-1991). It contains 56984 documents with the total size of around 60 MBs.

Table 2. Testing environment specification

Aspect	Value
Processor	Intel ® Core™ i7-6700HQ CPU @ 2.60 GHz
RAM	16 GB
Operating System	Windows 10 64 bit
Language	Java 1.8.092

A. *Indexing Performance*

Inverted index performs significantly better than GST in the indexing process. GST is 50 times slower than inverted index, and it requires memory more than 1200 times of the size of the original documents. Inverted index, in comparison, only requires less than half size of the original document. The full result can be seen in Table 3.

Table 3. Time and memory of indexing

Metrics	Inverted Index	GST
Document Count	3 000	
Document Size (KB)	3 595	
Time (ms/document)	0.023	1 155
Memory (index size / document size)	0.420	1 224

B. *Searching Performance*

We generate a total of 500 random queries of length 1 to 10 words. The access controls are randomly generated. The time shown is the average and percentiles of search times, where each time of all queries is the median of the same search performed 25 times. It turns out that GST performs about 3 times faster than inverted index on average.

Table 4. Search performance

Index type	Search time (µs)				
	Average	10%	20%	80%	90%
Inverted Index	14.25	2.37	3.16	17.46	27.26
GST	5.04	3.16	3.56	6.32	7.11

C. *Analysis*

The current implementation of GST, which requires an incredible amount of memory, can be explained as follows. The most memory-consuming part of GST is the mapping $g[s,(x,k,p)]$. Such mapping requires $4 \times \#nodes \times \#child$, where $\#nodes$ is the maximum number of nodes, which is $2N$, and $\#child$ is the maximum number of child of each node, which is 36 (10 + 26 alphanumeric characters). From this mapping alone, it takes us $4 \times 2N \times 36 = 288N$ 32-bit integers. It is 1152 times larger than the original document size, assumed that each letter in the document takes 8-bit memory.

This actually can be improved. The factor 36 is not necessary, for the size of mapping g is equal to the number of edges of the tree, which is not more than $2N$. Therefore, theoretically it only takes us $4 \times 2N = 8N$ 32-bit integers, or 32 times the size of the original document. But, in order to achieve this, it will take longer indexing time. Right now, the indexing time of GST is already 50 times slower than inverted index. Hence, such a thing is not recommended.

However, suffix tree is better at searching once the indexing process is done. On average, it is 3 times faster than inverted index. This is the fundamental space-time tradeoff in computer science. Not only space-time tradeoff, but suffix tree allows us to search for any substring occurrence within the document. It can be concluded that we need to use suffix tree only if:

1. Substring search is required, e.g. DNA processing where every substring of the DNA string is important [7], or
2. The system does not need to index huge number of documents, but critically need faster amount of time, then suffix tree is preferred. A good example for this is search autocomplete. The size of documents not large, but it requires very fast response time as the result is needed for every characters typed.

VI. Conclusion

While perform faster than inverted index (roughly 3 times for most of the queries), GST cannot be applied to huge number of documents, as it requires about 1200 times of the original document size. It, however, can be applied to some systems that requires very fast response time and do not have large amount of data, e.g. autocomplete in search boxes, or where substring search is needed, e.g. DNA processing.

Access control can be applied to any search algorithms easily using the proposed method, as it only acts as a filter after the search is being done. The complexity is not hardly affected; it is still linear in the amount of result when the search is not involving access control.

VII. Future Work

The most straightforward work to do is to improve the GST again to use as least memory as possible. An example is to use compression method, as used by the succinct data structure which allows suffix array to its theoretical lower bound [4]. Tree structure or how information are stored among the nodes and edges may differ, hence further research should be conducted.

One other limitation is that all of the process should fit into the main memory. In order to work with larger scale of data, the secondary memory is required. However, the proposed method cannot be used right away, as we should design how the data is persisted in the disk to reduce the amount of I/O.

B-Tree is a good example of efficient use of indexing using secondary memory. Inverted index can be implemented using B-Tree right away. A related work about this is a data structure called String B-Tree, which can utilize B-Tree to store a PATRICIA tree [3]. Further research may combine the work to improve the GST to work with secondary memory.

References

[1] Ukkonen, E, “On-line Construction of Suffix Trees” in Algorithmica, 1995, pp. 249-260.

[2] C. Gomley, Z. Tong, “Elasticsearch: The Definitive Guide”, O’Reilly Media, Inc., 2015.

[3] P. Ferragina, R. Grossi, “The String B-Tree: A New Data Structure for String Search in External Memory and its Applications”, ACM Symposium of Theory of Computing, 1995.

[4] N. Tanida, M. Inaba, K. Hiraki, T. Yoshino, “Hardware Accelerator for Full-Text Search (HAFTS) with Succinct Data Structure”, International Conference on Reconfigurable Computing and FPGAs, 2009, pp. 155-160.

[5] I.H. Witten, A. Moffat, T. C. Bell, “Managing Gigabytes: Compressing and Indexing Documents and Images”, Academic Press, 1999.

[6] P. Kim, “The true story behind Elasticsearch storage requirements” (https://www.elastic.co/blog/elasticsearch-storage-the-true-story), April 2015 (Accessed on September 3, 2016).

[7] P. Bieganski, J. Riedl, J. V. Carlis, “Generalized Suffix Trees for Biological Sequence Data: Applications and Implementations”, Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences, 1994, pp. 35-44.

[8] E. Atlam, E. Ghada, M. Fuketa, K. Morita, J. Aoe, “A Compact Memory Space of Dynamic Full-Text Search using Bi-Gram Index”, The 9th International Symposium of Computers and Communications, pp. 104-109.