A New Query Method of Regular Expression in Large Databases using MP-Tree

Xiaoyu ZHANG [1], Xiao LIN [2]

[1] *Department of Computer Science and Application*, Zhengzhou Institute of Aeronautic Industry Management, Zhengzhou, Henan 450015, China
[2] *School of Information Technology*, Luoyang Normal University, Luoyang, Henan 471934, China

Abstract — Pattern matching include regular expression matching with wide application and important research significance. This paper focuses on issue in this area of pattern matching, i.e. the query of regular expressions in large databases. We propose an effective index structure of MP-Trees, further develop a query processing algorithm based on this index structure, and analyze the complexity of this algorithm in detail. Finally, the efficiency and effectiveness of the method are proved by numerous experiments.

Keywords - Regular expression; Query processing; Large database; Indexing;MP-tree

I. INTRODUCTION

Pattern matching has been widely applied in computer science, such as text editing, symbol manipulation, data retrieval, network intrusion detection, etc.[1,2]A well-known science concept "regular expression" plays an important role in the pattern matching, which is often known as a pattern used for describing a series of character string conforming to a certain syntactic rule. In the past decades, the studies related to the regular expression has attracted many scholars' attention [3-5]. A classical method of the regular expression matching is to create a deterministic finite automaton (DFA), and then verify if the input information matches the target information through it [3]. Due to the possible exponential appearance of the number of states in DFA, the extendibility of this method is poor. Another method is Patricia tree-based approach [6] that overcomes the defects of the traditional method by establishing an effective index structure. This method is appropriate for the case that the whole index entries can be loaded into main memory, not suitable for processing the large data.

This paper mainly explained the regular expression queries in large databases. Specifically, instruct C to represent a character set, D to represent a database that stores n (a large positive integer) records. Assuming that each record has a field S with the attribute of string type; for the attribute field S of any record, it can assume that the upper limit of its string length is m and each character c of the character string is from the character set C, that is $c \in C$. Given a query string Q, in which all the characters in Q are from the set C, and $|Q|<m$, $|\cdot|$ represents the length of character string Q. The problem to be solved here is to find out all the records that the field S of the record matches the regular expression of query string Q, that is for the arbitrary record $r \in D$, if r. S matches the regular expression of string Q, then r will be returned as the result, in which r. S represents the value of field S in the record r. Considering a simple practical example, let's assume a customer information database, whose field includes customer ID, telephone, ID card number, license plate number, etc. If the user knows the customer's exact ID, then he/she can quickly find out other information of such customer. However, in some cases, if part of the query information packet is lost in the network transmission or for privacy protection [7], etc., only knowing part of the customer's ID information or partial information of other field, at this point, the regular expression query can find out the potential matching customer information.

The regular expression-related and large database-oriented subjects have received widespread attention at home and abroad. For example, the literature [8] discussed how to index the regular expression efficiently, the literature [9] discussed the parameterized pattern query, the literature [10] discussed the substring matching in the time series database, the literature [11] discussed the data flow-oriented regular expression query, etc.These work have similarities to the issues that this paper focuses on, however, the issues discussed above are semantically different from this paper. This paper proposes an effective index structure MP-tree, based on this index structure, an effective query processing algorithm is further developed, and the complexity of this algorithm is analyzed in detail, after that, the efficiency and effectiveness of the method proposed are proved by a lot of experiments.

II. PRELIMINARIES

MP-tree is the appropriate expansion conducted on the basis of the classical prefix tree [12] so as to adapt to the characteristics of the issues. This section firstly reviews the basic knowledge of the prefix tree, then officially defines the issue that this paper concerns.

A. Prefix Tree

In computer science, the prefix tree is a variation of Hash Tree. It is usually applied in the text word frequency statistics, statistics and sorting of character string, etc. in the search engine system. The prefix tree has the following main properties: (1) the root node doesn't contain character, every node other than the root node only contains one character. (2)The connected characters in the path from the root node to a certain node are the character string corresponding to such node. (3) All child nodes of each node contain different characters from each other.

Next, we will give a brief introduction to several operations commonly used in the prefix tree: insert, delete and search. Given a prefix tree and a new character string, inserting this character string into the given prefix tree includes the following steps:

Step 1: Set the current node as the root node, set the current character as the first character in the character string inserted.

Step 2: Search the current character in the child node of the current node, if it exists, then the current node is set as the child node with the value of the current character; otherwise, create a new value as the child node of the current character, and then set the current node as the node newly created.

Step 3: set the current character as the next character in the string, if the current character is 0, then end it; otherwise, go to Step 2.

The search operation is similar to the insert operation. When the matched character can't be found, it should return a false value. If all the characters are matched, it should judge if the node that eventually stops on is the tree leaf. If so, then it should return a true value, otherwise, return a false value.

The delete operation is as follows: firstly, search this character string, in the search procedure, press the nodes in the path into a temporary stack one by one. If the search procedure fails, then it should return a false value; otherwise, it should judge if the stack top node is a tree leaf, if so, then delete this node, if not, it should return a true value.

B. *Definition of Symbols and Problems*

Instruct D represent a database storing n records. Assuming that there is a field S with an attribute of character string for any one record $r \in D$. for convenience, instruction r. S represents S value in the field S of record r. Instruct C represent a character set, and |C|is defined as the base of character set. Assuming that the upper limit of the length of character string of the attribute field is m for any one record $r \in D$, i.e.|r. S|<m, and assuming that each character c of the character string is generated from character set C, i.e. $\forall c \in$ r.S, $c \in C$. Instruct Q represent a query string, assuming that each character c of Q is generated from character set C,i.e. $\forall c \in Q, c \in C$; and assuming that the length of query string is less than m, i.e. |Q|<m, |.| represents the length of character string Q.

Definition 1(Regular expression query) With given database D and query string Q, regular expression query means to find out all records of the recorded field S matched with the regular expression of query string Q, i.e. for any record $r \in D$, ifr. S matches the regular expression of character string Q, r will be returned as a result, if not, ris not the query result.

For example, Table I provides a simple list of customer information list.Assuming that the query user has known part of the information of field ID, for example, '12', then the regular expression query will return to records 2 and 3. Similarly, assuming that the query user has known part of the information of field of the license plate, for example, 'Q3', then the regular expression query will return to record 3.

TABLE I. CUSTOMER INFORMATION LIST

ID	License Plate Number	Age	Gender
10834	ML4563	29	Male
10862	QY7834	38	Female
16542	QZ3965	45	Male
17634	ZL7983	37	Male
19405	HY3492	41	Female
20373	JF8943	28	Female
20673	JH7635	26	Female
……	……	……	……

III. PROPOSED METHOD

In order to effectively support for large database specified regular expression query, a method based on indexing is proposed. Its main idea is to effectively manage the record by constructing indexing. In order to adapt to the problem being solved, appropriate modification is made on the proposed indexed structure on the basis of traditional prefix tree (note: traditional prefix tree cannot be used for regular expression query, for the regular expression query is not a simple character string matching in the traditional sense). For the purpose of distinguishing, it is called Modified Prefix-tree (MP-tree). This section details this kind of index structure, and introduces the specific algorithm describing how to use the indexing to complete the query discussed above.

A. *MP-tree*

Similar to the prefix tree, the proposed index structure also has three main properties are described in Section II.A. Next, the focus is on the modified additional components in the indexing. For the purposes of discussion, a specific example is selected in this section to help us to explain the modified principle and details of the prefix tree.

TABLE II. DATABASE EXAMPLE

ID	S
1	wy
2	xz
3	wxy
4	xwy
5	ywz
6	zxy
7	zyw
8	wzxy

Considering that a database stores a series of records (see TableII), and each record consists of two fields: ID field and a character string S of the attribute field. Figure 1 shows the general picture of the modified indexed structure.

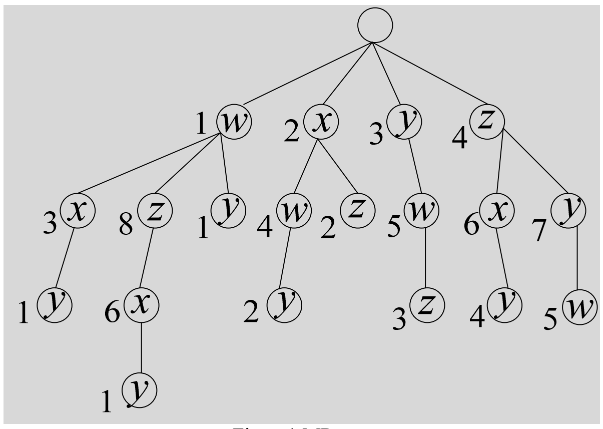

Figure1.MP-tree

Each path from root node to leaf node of the MP-tree corresponds to the value of the recorded field S of database (i.e. a character string). The layer of nodes corresponds to the depth of MP-tree, among which, the root node corresponds to the zero layer, and the layers of other nodes are the like. Give each node a place information which will help to access and store nodes. The place information consists of two elements: the layer of node and the order inserted in the layer by the node. For example, in the second layer, the order inserted by the node is: 'y' of the first record, 'z' of the second record, 'x' of the third record and so on. For the sake of brevity, instruct Node (x,y) represent the place information of nodes, x represents the layer number, and y represents the node position in this layer. For example, Node (3,6) represents that the node is located at the third layer, and the order is 6.

In order to quickly know the node itself and the corresponding records ID of the child node in assigned area of node, so create a list for each non-leaf node to save information of node itself and the child node ID, represented by I_{list}. For example, Node (1,1) will save 4 information I_{list} which respectively is null, 3, 8 and 1, among which, null means that the character string itself formed from the root node to the node does not exist in the database. If the regular expression has already satisfied matching in Node (1,1), it can simply return to records 3, 8 and 1, for the child nodes of Node (1,1) have a common prefix.

In addition to the above basic information, each node will be attached with a two-dimensional array. Before explainingthe specific composition of two-dimensional array, firstly, the following definitions are made.

Definition 2 (Key node) Given a node Nd and a character c, if the character c first appears in one subtree of given node Nd, and the node where the character c appears has no ancestor node (remove node Nd) also appears character c, and say the node where the character c appears is key node in subtree of character c on node Nd.

Notice: Given a node and a character, it may have multiple key nodes, for a given node may correspond to multiple subtrees. For example, the key node of characters 'x' to the root node are: Node(2,3), Node(3,6), Node(1,2), Node(2,6).

Definition 3 (First/Last key node) Given a node Nd and a character c, assuming that node n2, n3,… n_k are all the key nodes which appear in the layer l of tree of character c related to note n1, and the key node appears in the first time/last time is the first/last key node in the layer l of character c related to node n1. Particularly when the l layer has only one key node, this node is not only the first key node in the layer l of character c related to node n1, but also the last key node.

For example, the first key node of characters 'x' related to the root node in the second layer is Node(2,3), the last key node is Node(2,6). Note: The first/last key nodes of character 'x' related to the root node in the third layer are all Node(3,6), because there is only one key node for character 'x' related to the root node in this layer.

Now we will introduce the specific composition of two-dimensional array attached to each node. Clear the first key nodes in all the layers of subtree controlled by the node which are used by two-dimensional array to store each character. For example,TableIII and Table IVis the two-dimensional array attached by root node and Node(1,4) respectively. Take the first line of Table IIIas an example, it represents that the first key nodes of character 'w' related to the root node in the first, second, third and fourth layer are Node(1,1), Node(2,4), Node(3,5) and null respectively. (Note: It can be known that the first layer of Table IV can be omitted in fact according to the first key node, but keep the field of the first layer for the convenient subsequent discussion.)

For certain characters may have many key nodes in some layers, each node adds one key node pointer in order to access to all the key nodes easily. The pointer is used to point to the next key node in the same layer, using p_{next} to represent the pointer. For the last key nodes, make its p_{next} pointer point to null. Since then, it can traverse all the (key) nodes with the same symbol effectively based on two-dimensional array and the key node pointer p_{next} added by node. For example, if the initial position is in the root node, and want to visit all the nodes of the characters'y' in the third layer, can know the first key node of the characters'y' in the third layer is Node(3,1) through first key node array of the root node.Then know its next key node is Node(3,2) by pointer p_{next} of Node(3,1).Similarly, can know next key node is Node(3,4) by pointer p_{next} of Node(3,2). In the end, the traversal ends at this step for the pointer p_{next} of Node(3,4) is null. So far, the primary member of MP-tree has been discussed. In the next section, we will introduce how to use MP-tree to support regular expression query. It's worth noticing that MP-tree needs to be updated accordingly when add or delete database records, its update operation is similar to those discussed in Section I.A, and needs extra overhead to update the newly added components of MP-tree.

TABLE III. THE TWO-DIMENSIONAL ARRAY ATTACHED TO THE NODES(THE FIRST KEY NODE OF NODE(1,4))

Character	The first layer	The second layer	The third layer	The fourth layer
w	1	4	5	Φ
x	2	3	6	φ
y	3	1	1	1
Z	4	2	3	φ

TABLE IV. THE TWO-DIMENSIONAL ARRAY ATTACHED TO THE NODES(THE FIRST KEY NODE OF ROOT NODE)

Character	The first layer	The second layer	The third layer	The fourth layer
w	φ	φ	5	φ
x	φ	6	φ	φ
y	φ	7	4	φ
Z	φ	φ	φ	φ

B. *Regular Expression Query Processing*

Instruct record List to represent the linked list which is applied to store result set. ch represents the variable applied to store characters, Q[i] represents the i(th) character in query string and root represents the root node of MP-tree. Algorithm 1 describes the pseudo code of regular expression query processing. The general steps are to determine whether the query string is null or whether the first character of query string exists from the i(th) layer to the (m-i+1)(th) layer in the first key node array of root node. If the query string is null or the first character of query string does not exist from the i(th) layer to the (m-i+1)(th) layer in the first key node array of root node, the direct withdrawal result will be Φ (refer to line 1-4, note: m means the maximal length of record field S in database, see Section I.B). Otherwise, find key nodes of the first character from the i(th) layer to the (m-i+1)(th) layer related to root node according to the first key node array of root node and p_{next} pointer of node (refer to line 6-8). As for each above-mentioned key node that has been found, call the sub-function FindRecordInSubtree( ), then combine all the results found by sub-function one by one. And finally, return the result set record List (refer to line 9-11).

Algorithm 1*RegExpQuery*(string *Q*, Node *root*)

```
1. Set i←0, recordList←Φ, sameLevelList←Φ
2.ch ← Q [i],   i←i+1
3. ifch=null || does not exist the first key node of character ch from the i(th) layer to the (m-i+1)(th) layer in the first key node array of root node, then
4.   returnrecordList
5. else
6.   rootFirstKeyNode ←all first key nodes from the i(th) layer to the (m-i+1)(th) layer in the first key node array of root node,related to character ch
7.    foreachnodeNd ∈ rootFirstKeyNodedo
8. rootKeyNode←all other key nodes of this layer related to root node, which are found by according to $p_{next}$ pointer of node Nd, set sameLevelList← Nd ⋃ rootKeyNode
9.    for each node Nd ∈ sameLevelListdo
10.   tempRecords←FindRecordInSubtree(Q,Nd,i);
          setrecordList←recordList ⋃ tempRecords
11.   returnrecordList
```

The sub-function FindRecordInSubtree() in algorithm 1 belongs to recursive function, which will further find detailed records that can match in the subtree under the control of current node. Algorithm 2 describes the pseudo code of this function. The general steps are to estimate whether they are all matched up to the current node; if it is, return the ID of all records in this node I_{list} (refer to line 1-4); it should be noted that I_{list} is a linked list to store the ID of the node itself and child node (refer to Section III.A). If it fails to determine whether it is matched, continue to find the first key node array of this node, and if the first key node of corresponding characters cannot be found in this array, it means that it is unnecessary to keep finding those following this node but directly return Φ instead (refer to line 5-6). Otherwise, find all key nodes of corresponding characters related to the current node according to the first key node array of current node and p_{next} pointer of node (refer to line 8-10). Then focusing on the above-mentioned key nodes that have been found, continue to call this recursive function and return after combining the results (refer to line 11-13).

Algorithm 2*FindRecordInSubtree*(string *Q*, Node *Nd*, int*i*)

```
1. set ch ←Q [i], tempRecords←Φ, i←i+1
2. ifch=null   then
3. empRecords←allrecords IDs in node Nd
4.   returntempRecords
5.else if the first key node of character ch does not exist
          from the i(th) layer to the (m-i+1)(th) layer in
          the first key node array of node Nd, then
6.   returntempRecords
7. else
8.NdFirstKeyNode←all key nodes from thei(th) layer to the
        (m-i+1)(th) layer related to character ch in the first
        key node array of node Nd
9.   foreach node Nd ∈ NdFirstKeyNodedo
10. NdKeyNode← all other key nodes of this layer related
          to character ch, which are found by according to
          $p_{next}$ pointer of node Nd; set
          NdSameLevelList← NdNdKeyNode
11.   for each node Nd ∈ NdSameLevelListdo
12.tempRecords←tempRecords ⋃ FindRecordInSubtree
```

```
        (Q,Nd,i)
13.**return**tempRecords
```

For example,assuming that the query string is 'zx'.Firstly, find the first key node from the first layer to the fourth layer related to character 'z' according to the first key node array of root node, where Node(1,4), Node(2,2) and Node(3,3) are the first key nodes of the first, second and third layer respectively. Secondly, find all key nodes related to character 'z' in this layer according to the pointer p_{next} of node, where p_{next} of Node(1,4) is null, so there is only one key node related to character 'z' in the first layer.Also there is only one key node related to character 'z' in the third layer.But the pointer p_{next} of Node(2,2) is Node(2,8) and that p_{next} of Node(2,8) is null, so there are two key nodes in the second layer. Next, focusing on each key node related to character 'z', find the first key node related to character 'x' in the second and third layer of the key node array of this node (notes: character 'x' is the second character of query string). Then find all key nodes related to character 'x' according to the p_{next} pointer of the first key node related to character 'x'. Where, the first key node of Node(1,4) 1related to character 'x' is Node(2,6) and the p_{next} of Node(2,6) is null, so there is only one key node of Node(1,4) related to character 'x', i.e. Node(2,6). Similarly, the key nodes of Node(2,2), Node(2,8) and Node(3,3) related to character 'x' can also be found, where the key node of Node(2,2) related to character 'x' is null, that of Node(2,8) is Node(3,6) and that of Node(3,3) is null. Therefore, so far, only the path where the two key nodes of Node(2,6) and Node(3,6) are located has been left as the alternative results. At this time, the two characters of query string 'zx' have been checked, so directly return the tuple ID in I_{list} of the two nodes with the final return result of {6,8}.

C. Complexity Analysis

In Section II.B, n has been defined as the total number of records, m as the upper limited length of S field in each record,|C|as the base of character set and |Q| as the length of query string. The complexity of algorithm will be measured by virtue of these parameters next.

It is easy to know that n record(s) may have m*n characters at the worst. However, different records have a common prefix in MP-tree, so the total number of node tends to be O(n) when all the combinations of |C| character are included in the tree. Besides, as for every node, a two-dimensional array is applied to store the first key node. Where, the line of two-dimensional array represents the characters and column represents the number of layers. Number of columns is |C| and number of layers is m+1 at the worst. So the space limit of each node applied to store two-dimensional array is |C|*(m+1) at maximum. Besides, each node will occupy an additional space of O(1) to store the key node pointer. Therefore, the overall space usage of each node is O(|C|*(m+1)+1). So the overall space complexity is O(n*(|C|*(m+1)+1)), namely, O(n*m*|C|), where m(or |C|)<< n. (It should be noted that much space responding to the first key node in two-dimensional array is empty, which indicates that the space used for actual needs can be decreased further in theory. It will be considered that how to further optimize space requirement in the future work.)

Time complexity of algorithm will be analyzed next. As previously mentioned, the total number of node tends to be O (n) when all the combinations of |C| character are included in the tree. At the worst, it is necessary to check |Q| layer(s) downwards each node to determine whether its path can meet query requirements, so its time complexity is O(|Q|*n), where |Q|<<n. But in practice, a majority of paths can be pruned (namely, unnecessary to be checked). For example, if query string is comparatively long, only the node of upper layer needs to be checked, or in the situation that the amount of number of characters in parts that are not checked in query string and number of layers that the current node is located is greater than the total number of layer, m+1, etc. Therefore, the work efficiency of algorithm is ideal in fact comparing with that in theory.

IV. EXPERIMENTAL RESULTS

A comparison is made between the proposed method and the method in reference [4] to verify the effectiveness and efficiency of the proposed one. It should be noted that the algorithm in reference [4] should be modified simply to adapt to the issues under discussion. In particular, the query string should be converted to a regular expression in the form of '.*' firstly, then verify the matching degree of each record by means of regular expression matching algorithm, and finally, return the ID information of all matching records. For the sake of brevity, this method is called as CM (Compared Method). The method proposed in this paper is called as MPM (MP-tree based Method).

A series of character strings produced at random are used as testing data set. Each letter of character string is selected from the character set C. Assuming C={a,b,c,…,y,z}, and the length of each character string produced is 10. In order to test the effect of data set size, several groups of set with different data set sizes are produced, and the number of their character strings is: 1e+3, 1e+4, 1e+5, 1e+6, 1e+7, 5e+7 and 1e+8 respectively (where 1e+7 belongs to default setting). In addition, in order to test the effect of query string length, character strings with length of 3, 4, 5, 6, 7, 8 and 9 respectively are used as query string (where 5 belongs to default setting). These strings are produced at random with each group producing 10 strings. Each test should be conducted for 10 times, and then calculate the average I/O time and average query time (sum of CUP time and I/O time).

The operating environment of experiments is as follows: 2.6GHz CPU, 2G internal storage, 4K page size and Windows 7 operating system. All the codes should apply the C++ programming language with the development tool of Microsoft Visual Studio 2010. Main experimental results will be reported next.

Figure2 shows the average query time of increasing the data set size from 1e+3 to 1e+8. It can be concluded from the figure that the query time of CM can increase sharply when the data set is expanded, while the query time of MPM increases relatively slowly, which shows a better extendibility. The query time of MPM is several order of magnitudes less than that of baseline method especially when the data set size reaches a certain scale, which indicates the effectiveness of proposed method.

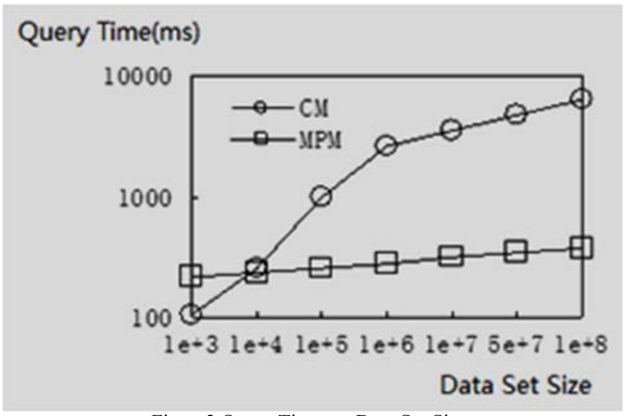

Figure2.Query Time vs. Data Set Size.

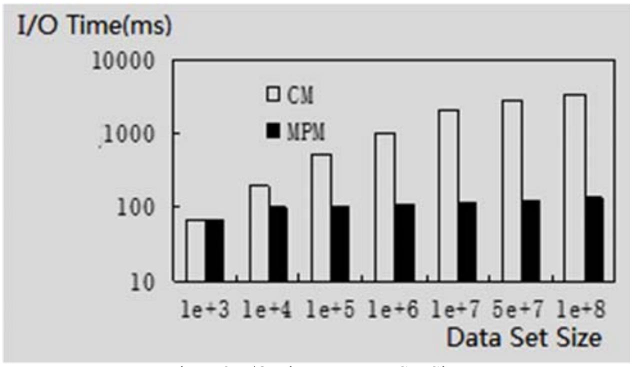

Figure3. I/O Time vs. Data Set Size

Figure3 shows the effect of data set size on I/O time. As might be expected, the I/O time of CM can increase rapidly with the increase of data set size, while the I/O time of MPM has an insignificant effect (merely increasing slightly). The result of this group shows that comparing with CP, MPM is more suitable for the query of regular expression in large databases.

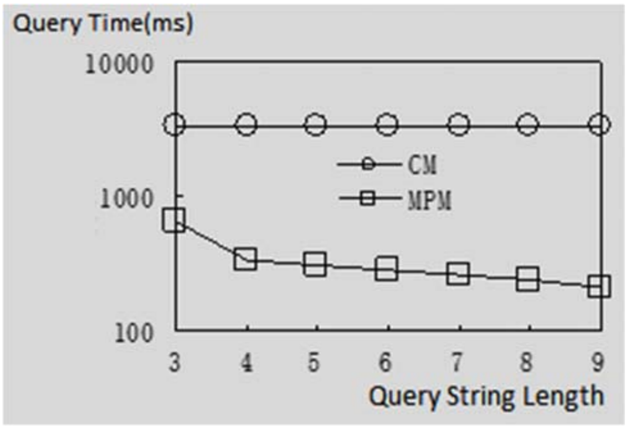

Figure4. Query Time vs. Query String Length

Figure4 shows the query time of query string length increasing from 3 to 8. It can be easily concluded from the figure that the length of query string can hardly affect the query time of CM, but when the query string increases, the query time of proposed method will witness a decline trend. The main reason of which is that query scope will be limited more seriously with the increasing length of query string, so that most nodes can be pruned in advance. At the same time, the figure shows that the query time of MPM is several order of magnitudes less than that of CM, which further indicates the effectiveness of proposed method.

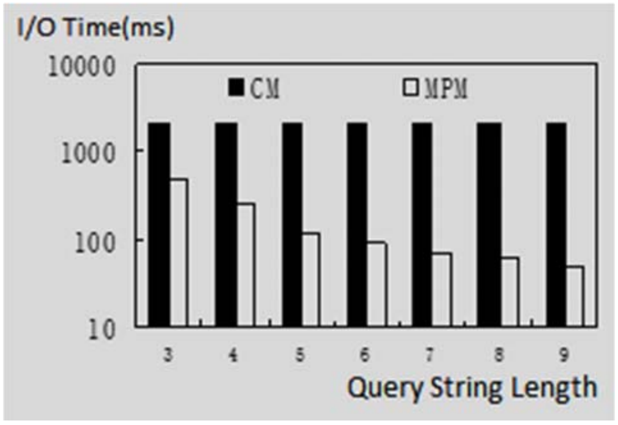

Figure5. I/O Time vs. Query String Length

Figure5 shows the effect of query string length on I/O. It can be easily seen that the I/O time of CM nearly has nothing to do with query string length, but the I/O time of MPM is inversely proportional to query string length. That it to say, I/O time will decrease with the increase of query string length, and the main reason is as previously stated that longer query string can help to define (or decrease) the hunting zone of MP-tree.

V. CONCLUSION

This paper has proposed a method based on MP-tree to solve the problem of regular expression query in large databases, analyzed and discussed the algorithm in detail and verified its efficiency and effectiveness in combination of a large number of experiments. We will try to explore other more efficient solutions in the future work.

REFERENCES

[1] FAN Aijing, YANG Zhaofeng. New method of pattern-matching for network intrusion detection[J]. Journal of Computer Applications, 2011,31(11): 2961-2964.

[2] SUN Qingdong, HUANG Xinbo, WANG Qian. Multiple pattern matching on Chinese/English mixed texts[J]. Journal of Software, 2008,19(3): 674-686.

[3] HUANG Kun, ZHANG Dafang, XIE Gaogang, JIN Junhang. Acompact regular expression matching algorithm for deep packet inspection[J]. Science China Information Sciences, 2010, 40(2): 356-370.

[4] ZHANG Shuzhuang, LUO Hao, FANG Binxing, YUN Xiachun. An efficient regular expression matching algorithm for network security inspection[J]. Chinese Journal of Computers, 2010, 33(10): 1976-1986.

[5] WANG Peifeng, LI Li. Application of an improved multi-pattern matching algorithm in snort[J]. Computer Science, 2012, 39(02): 72-75.

[6] Ricardo A. Baeza-Yates, Gaston H. Gonnet. Fast Text Searching for Regular Expressions or Automaton Searching on Tries[J]. Journal of ACM, 1996,43(6):915-936.

[7] ZHU Qing, ZHAO Tong, WANG San. Privacy preservation algorithm for service-oriented information search[J]. Chinese Journal of Computers, 2010, 33(8): 1315-1323.

[8] Junghoo Cho, Sridhar Rajagopalan. A Fast Regular Expression Indexing Engine[C].ICDE, 2002, pp:419-430.

[9] Cédric du Mouza, Philippe Rigaux, Michel Scholl. Efficient evaluation of parameterized pattern queries[C]. CIKM, 2005, pp: 728-735.

[10] Wook-Shin Han, Jinsoo Lee, Yang-Sae Moon, Seung-won Hwang, Hwanjo Yu. A new approach for processing ranked subsequence matching based on ranked union [C]. SIGMOD, 2011, pp:457-468.

[11] Anirban Majumder, Rajeev Rastogi, Sriram Vanama: Scalable regular expression matching on data streams[C]. SIGMOD, 2008, pp: 161-172

[12] CHENG Jiang, YI Yunfei, LIN jianhui, YU Qigang. Mining algorithm for fuzzy association rules based on prefix tree[J]. Computer Engineering, 2009, 35(7): 68-70