Frequency Based Indexing Technique for Pattern Matching

Manoj Kumar Gupta

Abstract*: Database management systems (DBMSs) play vital role in storing and managing the structured data in various application domains. Databases are queried by the users and the applications many times to access the stored data in different formats by using various search conditions. To improve the performance of the search operations, indexes are generally used by the DBMS. At present, almost all commercially available database management systems, performs full table scan (i.e. linear search) to answer the queries based on the LIKE '%...%' search even if the table is indexed based on the column being searched for. Although a number of indexing techniques have been proposed in the literature, but there is no index provided by the commercial RDBMS to efficiently answer the queries based on the LIKE '%...%' search. In order to improve the performance of the queries based on LIKE '%...%' operator (or pattern matching), a new indexing technique based on the frequency count of each character in the text is proposed in this paper. The proposed scheme is based on frequency count of each character in the string and the frequency is represented using B-Tree data structure. The proposed indexing technique is an attempt to answer the queries based on the LIKE '%...%' search without requiring full table scan which is shown through the empirical evaluation of the proposed scheme.

Index Terms*: Frequency Based Indexing Technique, Pattern Matching, LIKE Operator, B-Tree Index, Database Management System (DBMS)

I. INTRODUCTION

Database systems are widely used to store and manage the transactional data of the organizations. In some applications, large textual data is also stored in the database systems. Indexing techniques play a vital role in the database system to retrieve the requisite information from the databases [1, 2, 3]. In the absence of inappropriate indexing techniques, database search will be performed using full table scan. If there is an index available based on the searching key in the database then DBMS uses that index to answer the query using binary search else DBMS performs full table scan to answer the query [1, 2, 3, 4, 5, 6]. Index is a collection of data entries plus a way to quickly find entries with given search key values. Indexes should be used on the search keys in database systems where selection queries are frequent. After gone through the various indexes and index structures, it is found that [1, 7, 8, 9]:

- Clustered indexes are better if we require only one index for a table.
- Hash index provides best response time for the equality search.
- B-tree index works well in range and equality comparisons. It also provides the efficient sequential access.
- Bitmap index is best suited with low cardinality of key values. But it is not supported by the most commercial DBMSs.
- Reverse key index is better the monotonically increasing keys.
- Inverted index are suitable for full text searches and widely used by the search engines.
- R-tree index, grid file index and quad-tree indexes are suitable for multi-dimensional search.

II. RELATED WORK

A number of indexing techniques for database systems are presented in the literature for the numeric data and some of them are effectively used by many commercial RDBMS [7, 8, 9, 10]. The b-tree index [11, 12] is widely used in the DBMS and supported by the most commercial DBMSs as major queries are based on the combination of range and equality comparisons along with sequential search. Indexes based upon suffix trees [13, 14, 15, 16] and suffix arrays [16, 17] and related data structures are proficient when several searches are to be performed because the text need to be fully scanned only once when the indexes are created. In a huge variety of applications, the importance is given to suffix arrays and suffix trees besides string searching. Their range of applications is also growing in molecular biology, data compression, and text retrieval [17]. Suffix trees and suffix arrays based indexes occupy considerably more space than inverted lists which is a major criticism that limits the applicability of these indexes. Space requirement in these indexes are also especially crucial for large texts. For a text of n binary symbols, suffix arrays use n words of log n bits each i.e. O (n log n) bits, while suffix trees require between 4n and 5n words i.e. between O(4n log n) and O(5n log n) bits [18]. While, in the inverted lists, to create an index for a set of words consisting of a total of n bits, it require less than 0.l n / log n words (or 0.1 n bits) in many practical cases [19]. However, inverted files [27, 28] have less functionality than suffix arrays and suffix trees since only the words are indexed, whereas all substrings of the text are indexed in suffix arrays and suffix trees [17]. In order to reduce the space requirements in suffix arrays the concept of compressed suffix arrays (CSA) are introduced [17]. Similarly, compressed (or compact) suffix tree (CST) are introduced to reduce the space requirements.

Revised Manuscript Received on May 07 2019.
Dr. Manoj Kr. Gupta, Professor at Rukmini Devi Institute of Advanced Studies (Aff. to Guru Gobind Singh Indraprastha University), Delhi, India.

The compressed suffix arrays are probably as good as inverted lists in terms of space usage, at least theoretically. No previous result supported this finding. In the worst case, asymptotically the space complexity of the both types of indexes i.e. compressed suffix arrays and inverted lists is same; however, compressed suffix arrays have more functionality because they support search for arbitrary substrings [17]. Compressed suffix trees can be implemented in O(n) bits by using compressed suffix arrays and the techniques for compact representation of Patricia tries presented in [20]. As a result, they occupy asymptotically the same space as that of the text string being indexed.

String B-trees are used as an external-memory indexing technique for pattern matching in contrast to the suffix array and suffix tree which are internal memory indexing techniques. It is implemented based on the concepts of B-trees and patricia tries . In worst case, it also gives best.

Almost all commercial relational database management system (RDBMS) available now days, performs full table scan to answer the queries based on the LIKE '%...%' search even if the table is indexed based on the column being searched for. Presently, there is a lack of index provided by the commercial RDBMS to efficiently answer the queries based on the LIKE '%...%' search [9, 29, 30, 31].

In order to avoid full table scan, a new indexing technique based on the frequency count of each character in the text is proposed for pattern matching in this paper. The detailed description of the proposed technique is presented in the subsequent sections. This technique is implemented in Visual C++ by using feature of basic in-built package B-Tree package of Visual C++. The empirical evaluation of the proposed technique shows the queries based on the LIKE '%...%' search can be answered without requiring full table scan.

III. TERMINOLOGIES USED

The terminologies used in this paper are: (i) the alphabet Σ is used to represent the finite ordered set of characters, (ii) the size of the set is represented with |Σ|, (iii) the string (or text) S of length n represented as an array of characters S[1..n] = S[1], S[2] . . . S[n], (iv) substring of S is represented as S[i..j] = S[i] . . . S[j] ($1 \le i \le j \le n$), (v) prefix of S is represented as S[i..n], (vi) suffix of S is represented as S[1..j], (vi) text T is a set of Strings S*, (vii) pattern string P is defined as P ε S (length |P|), (viii) problem size N is the total number of characters in the text, (ix) memory size M is the number of characters that fit into internal memory, (x) block size B is number of characters that fit into a disk block, (xi) K is the number of records in the table to be indexed, (xii) R is the size of the answer.

IV. FREQUENCY BASED INDEXING TECHNIQUE

The proposed indexing technique is based on the frequency count of each character in the string. Therefore, it is named as frequency index. This frequency based index is a kind of text index which is referred by the query processor for pattern matching. In this indexing technique, separate frequency index is created for each character of the set of characters.

A. Structure of the Index

As discussed earlier, N indexes are created where N is the number of characters in the set of characters. In other words, a separate index is created for each character supported by the system. In frequency index, each index comprises the physical address of the record and frequency of a particular character. For example, if we are considering only alphabets in the character set then 26 separate indexes will be created for each alphabet (a to z). The index for the alphabet 'a' consists of physical address of the record and the frequency (or occurrence) of 'a' in the indexed string for each record in the relation which is considered as key value of the index. Records in the index are arranged based on the frequency. The same process repeated for each character. During the search with pattern matching the frequency of each alphabet is also computed and the character with highest frequency is taken into consideration to select one the multiple indexes to search for the given pattern. From the selected index, the records that are having the frequency greater than or equals to the frequency of the selected character from the pattern are considered only for comparison. By using this method, the number of comparisons can be reduced and hence the full table scan is not required.

Example:

Consider the following table containing the string for which the frequency index is created (*Table 1*). The frequency of each of the 26 alphabets is counted and considered for creating the separate index for each alphabet (*Table 2*).

Separate indexes for each of 26 alphabets are to be created containing the physical address of the record and the frequency count of the respective alphabet in the string of the corresponding record. All these indexes are referred while answering the queries based on LIKE '%...%' search. Let's suppose we want to search for a pattern '%specialized database languages%'. Now, the frequency of each letter of the pattern is computed. The computed frequencies of the alphabets in the pattern are: a-6, b-1, c-1, d-2, e-4, g-2, i-2, l-2, n-1, p-1, s-3, t-1, u-1, z-1. After computing the frequency of the alphabet in the patter, it will select the highest frequency alphabet (say 'a' in the above example pattern) and then select the records, by referring the index of alphabet 'a', those having the frequency of 'a' greater than or equals to 6. The rows with rowIDs B081, B082, B083, B085, B086, B088, B090, B092, B093, B096, B097, B099, B100 are selected in which string contains the frequency of 'a' >= 6. At last, search the pattern within the selected rows only and will return the row with rowID B099 which contain the pattern being searched. Hence, the full table scan is not used.

Table 1: Sample Table

RowID	Physical Address	String
B075	005094B0	Reduced updating errors and increased consistency
B076	0050FA10	Improved data security
B077	0050EC40	Reduced data entry, storage, and retrieval costs
B078	0050F590	Facilitated development of new applications program
B079	0050F7F0	Reduced data entry, storage, and retrieval costs
B080	0050F330	Database systems are complex, difficult, and time-consuming to design
B081	005041A8	Substantial hardware and software start-up costs
B082	005083A0	Damage to database affects virtually all applications programs
B083	00508600	Extensive conversion costs in moving form a file-based system to a database system
B084	00508060	Initial training required for all programmers and users
B085	00508820	Handheld computers do away with the tedium of digitizing data from paper check sheets.
B086	0050C530	The term database system implies that the data is managed to some level
B087	00508030	A general-purpose DBMS is typically a complex software system
B088	0050BEE0	The utilization of databases is now spread to such a wide degree
B089	0050F090	heavily depend on databases for their operations
B090	0050D870	The design, construction, and maintenance of a complex database
B091	0050DA90	Their tasks are supported by tools provided either as part of the DBMS
B092	0050D610	These can be seen as special-purpose programming languages
B093	0050BC80	Database languages are generally specific to one data model
B094	0050D030	A way to classify databases involves the type of their contents
B095	0050D3A0	Another way is by their application areas
B096	0050CD60	The term database may be narrowed to specify particular aspects
B097	0050D3A4	the databases that it maintains are often large
B098	0051F460	This definition is very general, and is independent of the technology used.
B099	0051F464	These tools include specialized database languages including data definition languages
B100	0051F468	The utilization of databases is now spread to such a wide degree

Table 2: Frequency Count of Each Alphabet

Row ID	Physical Address	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B075	005094B0	3	0	4	5	6	0	1	0	3	0	0	0	0	5	2	1	0	4	4	2	2	0	0	0	1	0
B076	0050FA10	2	0	1	2	2	0	0	0	1	0	0	0	1	0	1	1	0	2	1	2	1	1	0	0	1	0
B077	0050EC40	5	0	2	4	6	0	1	0	1	0	0	1	0	2	2	0	0	4	3	5	1	1	0	0	1	0
B078	0050F590	5	0	2	2	5	1	1	0	4	0	0	3	2	3	4	4	0	2	1	4	0	1	1	0	0	0
B079	0050F7F0	5	0	2	4	6	0	1	0	1	0	0	1	0	2	2	0	0	4	3	5	1	1	0	0	1	0
B080	0050F330	5	1	3	3	6	2	2	0	5	0	0	2	4	4	3	1	0	1	6	5	2	0	0	1	1	0
B081	005041A8	7	1	1	2	2	1	0	1	1	0	0	1	0	2	2	1	0	4	5	6	2	0	2	0	0	0
B082	005083A0	11	1	2	1	3	2	2	0	3	0	0	5	2	1	3	3	0	3	4	5	1	1	0	0	1	0
B083	00508600	6	2	2	2	8	2	1	0	5	0	0	1	4	5	6	0	0	2	10	6	0	3	0	1	2	0
B084	00508060	5	0	0	2	4	1	2	0	5	0	0	3	2	4	2	1	1	8	3	2	2	0	0	0	0	0
B085	00508820	6	0	3	6	8	2	2	5	6	0	1	1	3	2	4	3	0	3	3	7	2	0	2	0	1	1
B086	0050C530	8	1	0	3	10	0	1	3	3	0	0	3	5	1	2	1	0	1	6	8	0	1	0	0	1	0
B087	00508030	4	0	2	0	6	1	1	0	2	0	0	4	2	1	3	4	0	3	5	3	1	0	1	1	3	0
B088	0050BEE0	6	1	1	4	7	1	1	2	5	0	0	1	0	2	4	1	0	2	5	4	2	0	2	0	0	1
B089	0050F090	5	1	0	3	6	1	0	2	3	0	0	1	0	3	4	2	0	3	3	3	0	1	0	0	1	0
B090	0050D870	7	1	4	3	6	1	1	1	3	0	0	1	2	7	4	1	0	1	3	4	1	0	0	1	0	0
B091	0050DA90	4	1	0	3	7	1	0	3	3	0	1	1	0	0	5	4	0	6	5	6	1	1	0	0	1	0
B092	0050D610	6	1	2	0	8	0	4	1	2	0	0	2	2	4	2	4	0	3	6	0	2	0	0	0	0	0
B093	0050BC80	9	1	2	2	8	1	3	0	2	0	0	4	1	3	3	1	0	2	3	3	1	0	0	0	1	0
B094	0050D030	5	1	2	1	6	2	0	2	3	0	0	2	0	3	4	1	0	1	6	7	0	2	1	0	3	0
B095	0050D3A0	5	1	1	0	3	0	0	2	4	0	0	1	0	2	2	2	0	3	2	3	0	0	1	0	2	0
B096	0050CD60	8	2	3	2	7	1	0	1	2	0	0	1	2	1	2	3	0	5	4	5	1	0	1	0	2	0
B097	0050D3A4	8	1	0	1	5	1	1	2	3	0	0	1	1	3	1	0	0	2	3	7	0	0	0	0	0	0
B098	0051F460	2	0	1	5	10	2	2	3	7	0	0	2	0	8	4	1	0	2	4	4	1	1	0	0	2	0
B099	0051F464	10	1	3	6	9	1	5	1	8	0	0	6	0	7	3	1	0	0	6	4	4	0	0	0	0	1
B100	0051F468	6	1	1	4	7	1	1	2	5	0	0	1	0	2	4	1	0	2	5	4	2	0	2	1	0	1

B. Data Structures Used

B-Tree is used as the data structure for this proposed index. The frequency of each character is categorized into 16 categories i.e. 0 to 14 and more than 14. Hence, two level using 4-ways b-tree is used in the proposed technique. The structure of the index is represented in the figure 1 for the sample data given table 1 and 2 for alphabet 'a'.

B-Tree is used in the proposed indexing technique because it is widely used by most of the existing techniques and it is height balanced tree. The number of the levels can be reduced by m-way B-Tree. The leaf nodes of the B-Tree points to the data pages which contains the physical address of the records belongs to the particular value of the search key. These data pages are also points to each other which also provide the sequential access of the records based on the search key.

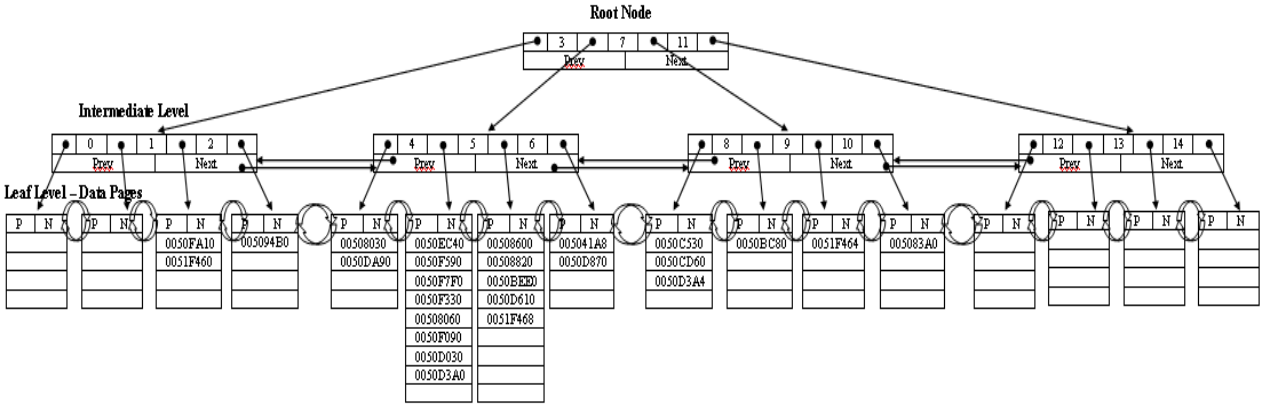

Figure 1: Index Structure for the Proposed Indexing Technique

C. Algorithms

This section presents the algorithms for various operations of the proposed indexing technique e.g. creation of index (CreateIndex), updation of the index UpdateIndex), and searching of the pattern (Search).

Algorithm 1: CreateIndex

```
Input: Name of the Table T, Name of the Column C --
for which index to be created.
Output: Index
1.  Declare an array f(1..K, 1..|Σ|) and initialize
    with 0.
2.  For each Record r in the Table t
3.  loop
4.     For each character c in Σ
5.     loop
6.       Increment f(r, c)
7.     end loop
8.  end loop
9.  For each character c in Σ
10. loop
11.     Create B-Tree based on the frequency f(1..K,
        c) as described in section 5.1.2.
12. end loop
```

Algorithm 2: UpdateIndex

```
Input: Name of the Table T, Name of the Column C --
for which index to be updated.
Output: Updated Index
1.  Declare an array f(1..|Σ|) and initialize with 0.
2.  For each Updated Record r in the Table t
3.  loop
4.     For each character c in Σ
5.     loop
6.       Increment f(c)
7.     end loop
       Update the B-Tree based on the frequency f(c)
       for the record r
8.  end loop
```

Algorithm 3: Search

```
Input: Name of the Table T, Name of the Column C,
Pattern P -- for which index to be updated.
Output: Set of Records containing P
1.  Declare an array f(1..|Σ|) and initialize with 0.
2.  Initialize the output list with null.
3.  For each character c in Σ in P
4.  loop
5.     Increment f(c)
6.  end loop
7.  Select the character c with Maximum Frequency mf
8.  Select the B-Tree for character c from the Index
9.  Select the set of records sr with frequency >= mf
10. For each r in sr
11. loop
12.    Search the pattern P in r
13.    if found then include in output list else ignore
       the record
14. end loop
15. return the output list
16. end loop
```

D. Complexity of the Algorithms

This section presents the time and space complexity of the above mentioned three algorithms viz. CreateIndex, UpdateIndex and Search.

Complexity of the Algorithm 1: CreateIndex

The complexity of the algorithm CreateIndex is computed as under:

	Time Complexity	Space Complexity
Declare an array f(1..K, 1..|Σ|) and initialize with 0.	O(K)	O(K.|Σ|)
For each Record r in the Table t loop	O(|Σ|)..O(K) times	
For each character c in Σ loop Increment f(r, c) end loop	O(K.|Σ|)	
end loop	O(|Σ|)	
For each character c in Σ loop Create B-Tree based on the frequency f(1..K, c) as described in section 5.1.2. end loop	O($\log_4$16.K) i.e. O(K) ---------- O(K.|Σ|..2K)	O((N/B). $\log_{(M/B)}$(N/B)) ----------

The statement at step 6, which calculates the frequency of each character for all records, will execute K.|Σ| times i.e. the time complexity is O(K.|Σ|). The statement at step 11, which creates the index based on the frequencies calculated by creating one B-Tree for each character, will execute $\log_4$16.K i.e. 2K times i.e. the time complexity is O(K). Therefore, the total time complexity of this algorithm is O(K.|Σ|) + O(K) = O(K.|Σ|). The space complexity for internal memory is O(K.|Σ|) and for external memory (i.e. disk) is O((N/B). $\log_{(M/B)}$(N/B)).

Complexity of Algorithm 2: UpdateIndex

The complexity of the algorithm UpdateIndex is computed as under:

	Time Complexity	Space Complexity
Declare an array f(1..|Σ|) and initialize with 0.		O(Σ|)
For each Updated Record r in the Table t loop	O(K/2) or O(K)	
For each character c in Σ loop Increment f(c) end loop	O(|Σ|), O(K) times O(K.|Σ|)	
Update the B-Tree based on the frequency f(c) for the record r end loop	O($\log_4$16.K) i.e. O(K) ---------- O(K.|Σ| + 2K)	O((N/B). $\log_{(M/B)}$(N/B)) ----------

The statement at step 6, which calculates the frequency of each character for all updated records whose search key is updated, will execute from |Σ| to K.|Σ| times that is on average K/2.|Σ| i.e. the time complexity is O(K/2.|Σ|) and 2 is a constant than it can be O(K.|Σ|). The statement at step 8, which creates the index based on the frequencies calculated by creating one B-Tree for each character, will execute $\log_4$16.K i.e. 2K times i.e. the time complexity is O(K). Therefore, the total time complexity of this algorithm is O(K.|Σ|) + O(K) = O(K.|Σ|). The space complexity for internal memory is O(K.|Σ|) and for external memory (i.e. disk) is O((N/B). $\log_{(M/B)}$(N/B)).

Complexity of Algorithm 3: Search

The complexity of the algorithm UpdateIndex is computed as under:

	Time Complexity	Space Complexity
Declare an array f(1..|Σ|) and initialize with 0. Initialize the output list with null.		O(Σ|)
For each character c in Σ in P loop		O(K)
Increment f(c) end loop	O(|P|)	
Select the character c with Maximum Frequency mf Select the B-Tree for character c from the Index Select the set of records sr with frequency >= mf	O(|P|) O($\log_B$N) O(K)	
For each r in sr loop Search the pattern P in r if found then include in output list else ignore the record end loop		O(K)
return the output list end loop	O(K) ---------- O(|P|)+K)	--------- O(K)

The statement at step 5, which calculates the frequency of each character of the pattern to be seached for, will execute |P| times i.e. the time complexity is O(|P|). The statement at step 12, which searches the pattern in the selected records of the step 9, will execute less than or equals to K times i.e. the time complexity is O(K). Therefore, the total time complexity of this algorithm is O(|P|) + O(K) = O(K.|P|). *Asymptotically the time complexity is O(K) but actually the time complexity is less than O(K)*. The space complexity is *O(K).*

V. CONCLUSION AND FUTURE WORK

A number of indexing techniques viz. bitmap index [25] suffix array, suffix tree [24], patricia tree [20] , pat tree [21, 22], suffix binary search tree [23] and string b-tree based indexes, etc. are presented in the literature for pattern matching. However, a few indexing techniques are suitable for the database queries based on pattern matching. As a result, in order to answer the pattern matching based queries such as queries based on the LIKE '%...%', DBMSs perform full table scan (i.e. linear search) [26].

In order to avoid the full table scan for the pattern matching based queries, a frequency based pattern matching indexing technique is proposed in this paper. The proposed algorithm is implemented in Visual ++ using in-built B-Tree Package. The algorithm was executed on the set of 26 hypothetic strings mentioned in *Table 1*. The empirical results and the B-Tree represented in Figure 1 shows that the proposed indexing technique is better for the queries based on the comparison based on LIKE '%...%' operator because of not using the full table scan to answer the queries based on pattern matching. Although the asymptotically, the complexity of the algorithm signifies the full table scan but practical it reduces the number of records comparisons. On the negative side, the cost of updation of this index is high therefore, it is most suitable for the text mining and other related applications where the updatation of the search key is minimum.

Further, the proposed technique can be improved to reduce the space requirement and the cost of updation by optimizing the methods.

REFERENCES

1. Desai, B. C. (2000) "An Introduction to Database Systems," Galgotia, 2000.
2. Elmasri, E. and Navathe, S.B (2008) "Fundamentals of Database Systems," Pearson Education, 2008.
3. Ramakrishnan, R. and Gehrke, J. (2003) "Database Management Systems," McGraw-Hill, 2003
4. Pravin Chandra, Anurag Jain, and Manoj Kr. Gupta, Query Optimization in Oracle, Voyager-The Journal of Computer Science and Information Technology, ISSN 0973- 4872, Vol. 6, No.1, p.p. 18-22, July-Dec. 2007
5. Manoj Gupta & Pravin Chandra, "An Empirical Evaluation of LIKE Operator in Oracle", International Journal of Information Technology, 2011. Vol. 3 No. 2 PP 23-29.
6. Manoj Gupta, Pravin Chandra & Dharmendra Badal, "An Empirical Evaluation of Join Operation in Different Versions of ORACLE", National Conference on Business Intelligence and Data Warehousing organized by USMS, GGSIPU jointly with IETE, CSI and IEEE, 2012. pp 167-176
7. Manoj Gupta & Dharmendra Badal, "Comparative Study of Indexing Techniques in DBMS" National Conference on Business Intelligence and Data Warehousing organized by USMS, GGSIPU jointly with IETE, CSI and IEEE, 2012. pp 177-185
8. Manoj Gupta & Dharmendra Badal, "A Study on Indexes and Index Structures", INTENSITY:IJASSR, 2013, Vol. 2 No. 1, pp 212-222.
9. Manoj Gupta & Dharmendra Badal, "A Study of Indexing Techniques for Text Indexing and Pattern Matching", IFRSA's International journal of Computing, Vol. 3 No. 3 pp. 53-57
10. Krugel, Johannes. Approximate Pattern Matching with Index Structures. Diss. Technische Universität München, 2016.
11. P. Ferragina and R. Grossi. The String B-tree: a new data structure for string search in external memory and its applications. Journal of the ACM, 46(2):236-280, Mar. 1999.
12. P. Ferragina and R. Grossi. The String B-tree: A New Data Structure for String Searching in External Memory and Its Application. Journal of the ACM, 46(2):236-280, 1999.
13. A. Andersson and S. Nilsson. Efficient implementation of suffix trees. Software Practice and Experience, 25(2):129-141, Feb. 1995.
14. A. Andersson, N. J. Larsson, and K. Swanson. Suffix trees on words. Algorithmica, 23(3):246-260, 1999.
15. J. I. Munro, V. Raman, and S. Srinivasa Rao. Space efficient suffix trees. In Proceedings of Foundations of Software Technology and Theoretical Computer Science, volume 1530 of Lecture Notes in Computer Science, pages 186-195, Berlin, Germany, 1998, Springer-Verlag.
16. J. Käirkkäinen. Suffix cactus: A cross between suffix tree and suffix array. In Combinatorial Pattern Matching, olume 937 of Lecture Notes in Computer Science, pages 191-204. Springer, 1995.
17. Roberto Grossi and Jeffrey Scott Vitter. Compressed Suffix Arrays and Suffix Trees Applications to Text Indexing and String Matching. ACM STOC 2000 Portland Oregon USA: 397-406, 2000
18. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. SIAM Journal on Computing, 22(5):935-948, 1993.
19. A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. A CM Transactions on Information Systems, 14(4):349-379, Oct. 1996.
20. J. I. Munro, V. Raman, and S. Srinivasa Rao. Space efficient suffix trees. In Proceedings of Foundations of Software Technology and Theoretical Computer Science, volume 1530 of Lecture Notes in Computer Science, pages 186-195, Berlin, Germany, 1998, Springer-Verlag.
21. D. Clark. Compact Pat trees. PhD Thesis, Department of Computer Science, University of Waterloo, 1996.
22. G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. In Information Retrieval: Data Structures And Algorithms, chapter 5, pages 66-82. Prentice-Hall, 1992.
23. R. W. Irving. Suffix binary search trees. Technical Report TR-1995-7, Computing Science Department, University of Glasgow, 1995.
24. J. Käirkkäinen and E. Ukkonen. Sparse suffix trees. Lecture Notes in Computer Science, 1090:219-230, 1996.
25. Elizabeth O'Neil and Patrick O'Neil, Kesheng Wu. Bitmap Index Design Choices and Their Performance Implications, LBNL-62756.
26. Zobel, J., Moffat, A., and Sacks-Davis, R. (1992) "An Efficient Indexing Technique for Full-Text Database Systems," VLDB 1992.
27. Zobel, Justin; Moffat, Alistair; Ramamohanarao, Kotagiri (December 1998). "Inverted files versus signature files for text indexing". ACM Transactions on Database Systems (New York: Association for Computing Machinery) 23 (4): pp. 453–490. doi:10.1145/296854.277632.
28. Zobel, Justin RMIT University, Australia; Moffat, Alistair The University of Melbourne, Australia (July 2006). "Inverted Files for Text Search Engines". ACM Computing Surveys (New York: Association for Computing Machinery) 38 (2): 6. doi:10.1145/1132956.1132959.
29. Lieponienė, J. (2018). A study of relational and document databases queries performance. Informatika, 12-22.
30. Lewenstein, Moshe. "Orthogonal range searching for text indexing." Space-Efficient Data Structures, Streams, and Algorithms. Springer, Berlin, Heidelberg, 2013. 267-302.
31. Bille, Philip, et al. "String indexing for patterns with wildcards." Theory of Computing Systems 55.1 (2014): 41-60.

AUTHORS PROFILE

Dr. Manoj Kr. Gupta is presently working as Professor at Rukmini Devi Institute of Advanced Studies (Aff. to Guru Gobind Singh Indraprastha University), Delhi, India. He is also Dean Examination, Admission and Administration in the Institute. He has more than 20 years of experience in teaching and administration. His interest areas are Database Systems, Data Warehousing and Data Mining. He has 4 books and 20+ international / national research papers to his credit.