ISTANBUL UNIVERSITY ENGINEERING FACULTY
JOURNAL OF ELECTRICAL & ELECTRONICS

YEAR : 2002 (447-451)
VOLUME : 2
NUMBER : 1

ANALYSIS OF EFFICIENT FILE STRUCTURES FOR PARTIAL-MATCH RETRIEVAL

KISMÝ-EÞLEME ERÝÞÝMÝ ÝÇÝN UYGUN DOSYA YAPILARININ ANALÝZÝ

Oðuzhan ÖZTAÞ

Istanbul University, Engineering Faculty, Department of Computer Engineering
34850, Avcýlar, Ýstanbul, Turkey

e-mail: oguzhan@istanbul.edu.tr

ABSTRACT

In this study, the patial-match retrieval was examined. This access method is an access method to needed for different purposes. It is mostly used for office automation. There are different methods for these access type. We discussed here each method in general perspective and tried to obtain the best result of them by comparing these technics eachother on a simple data model.

Key Words: *Partial-Match Retrieval, Database, Index methods, Search*

ÖZET

Yapýlan çalýþmada, Kýsmi-Eºleme Eriºimi incelenmiºtir. Bu eriºim yöntemi çeºitli amaçlar için ihtiyaç duyulan bir eriþim yöntemidir. Özellikle ofis otomasyonunda çok kullanýlýr. Bu eriþim üzerine çeþitli yöntemler vardýr. Biz burada bu yapýlardan her birini genel hatlarýyla anlatýp, bu teknikleri genel ve basit bir yapý üzerinde birbiri ile mukayese ederek aralarýnda en iyi sonuç vereni saptamaya çalýþtýk.

Anahtar Kelimeler: *Kýsmi-Eºleme Eriºimi, Çokluortam, Veri Tabaný, Ýndeks yapýlarý, Arama*

1. INTRODUCTION

Partial-match retrieval is a problem involving searching on secondary keys. A number of approaches have been suggested. One way to tackle the problem is to search the entire file in response to each query, using a string or a regular expression based pattern matching algorithm. Although, regular expressions provide considerable flexibility in posing queries, particularly those concerned with text, the requirement of having to search through the entire file to answer each query limits this technique to files that are not too large, unless queries can be batched or special hardware is available.

A number of alternative schemes have been suggested to reduce the amount of information that needs to be searched to answer a query. One such class off schemes associates with each record a short representative code word. Searches can then be performed over the shorter code word file to determine which records are potential answers to a given query (or, equivalently, to exclude records that cannot possible be answers). Another comman approach is to construct an inverted index list for each field. A query can be answered by intersecting the index lists for each specified field.

Some of the methods for partial-match retrieval are shown in the following case:

1- Extendible Hashing
2- Indexed Descriptor Files
3- Signature Files

Review of Extendible Hashing and Its Application to Partial Match Retrieval

The file is contained in a number of pages (or buckets). There are two kinds of pages. Leaf pages contain the records themselves and directory pages contain the directory. The directory is organized as a linear array of $m=2^a$ entires where d is called the global depth of the directory. m is always a power of two and changes (doubled or halved) in response to the changes in the volume of the file. Each entry in the directory contains a pointer to a leaf page. Each leaf page has a header that contains its local depth d′. For a leaf page with local depth d′, there are $2^{(d-d')}$ directory entires pointing to it. Fig.1shows a file organization when d=3.

Associated with the file is a hashing function h:KEY→D where KEY is the domain of the primary key of the file, D={0, 1, 2, 3, …, 2^{dmax}-1}, and 2^{dmax} is the maximum allowable size of the directory. To locate or insert a record with primary key value V, we calculate the pseudokey V′ as the first d bits of h(V). V′ is then used as an index into the directory. The indexed directory entry contains a pointer to the leaf page where the record will be found or inserted.

When inserting a record into a full leaf page with local depth d′, d′< d, we split the bucket into two buckets, distribute the records between the two buckets according to their pseudokey values, and then change the pointers in the appropriate directory entries. However, when inserting a record into a full leaf page with local depth d′=d, we double the size of the directory; global depth d is then increased by one. Each directory entry becomes two complement directory entries with identical pointer values. Now the overflow leaf page can be split similarly as before.

Lloyd has Ramamohanarao extended this single-key file organization to handle partial-match retrievals in the following way. Let n be the number of fields in the record structure of the file. With each field of the record structure, associate a hashing function $h_i:F_i \rightarrow B_i$, i=1,2, …,n where F_i is the key space of the ith field and Bi is the pseudokey space of a certain length m_i. The index of a record $(r_1, r_2, \dots, r_n)$ into the directory is assembled using the pseudokeys $h_1(r_1), h_2(r_2), \dots, h_n(r_n)$ and the choice vector (…, i_3, i_2, i_1). If $i_j=p$ in the choice vector, then the jth (rightmost) bit in the index should come from the pth pseudokey $h_p(r_p)$.

The choice vector has important consequences on the performance of the file. It dynamically adapts to the current state of the system according to the distribution of occurrences of each field in the record structure in the file operations performed thus far. This is done by deferring the calculation of ij until the directory size grows to 2^j. The procedure, called Minimal Marginal Increase (MMI), used to calculate the choice vector, and the theory behined it are discussed in [7].

The operations of insertion, searching, bucket splitting, directory doubling, etc., are then similar to the single-key case.

Example 1: Let the global depth=5, record R=(v_1, v_2, v_3, v_4) and the choice vector=(…, 4, 2, 2, 3, 2). Let $h(v_1)=a_m\dots a_1a_0$, $h(v_2)=b_m\dots b_1b_0$, $h(v_3)=c_m\dots c_1c_0$, and $h(v_4)=d_m\dots d_1d_0$ where $a_m\dots a_1a_0$, $b_m\dots b_1b_0$, $c_m\dots c_1c_0$, and $d_m\dots d_1d_0$ are binary numbers with a sufficiently large number m of digits. Then the address of record $R=d_0b_2b_1c_0b_0$.

Indexed Descriptor Files

Pfaltz et al. applied the technique of disjoint coding to a file structure called the indexed descriptor file. The idea behind the technique is to speed up retrieval of records by encoding information about the records in an efficient manner. Basically, the information in a single

record is represented by a descriptor word and the descriptor words of all records stored in one bucket (on disk) are bitwise OR'd together to form a bucket descriptor D_B. The directory of an indexed descriptor file is just the collection of all bucket descriptors.

A descriptor D_r of a record $r=(r_1, r_2, \dots, r_k)$ where $r_i \in D_i$ for $1 \le i \le k$ is a bit string of w (for width) bits. Each descriptor is divided into k disjoint fields. Each field F_i consists of w_i bits; therefor,

$$\sum_{i=1}^{k} w_i = w \qquad (1)$$

There are k functions $H_i: D_i \rightarrow \{1, 2, \dots, w_i\}$ for $1 \le i \le k$. In a descriptor D_r of a record $r=\{r1, r2, \dots, rk\}$, the $H_i(r_i)$th bit in F_i is set to 1 (the remaining w_i-1 bits are set to 0). There are exactly k bits set to 1 in a record descriptor.

A descriptor D_q for a query $q=(A_1=a_1, A_2=a_2, \dots, A_k=a_k)$ is a bit string of w bits with the $H_i(a_i)$th bit in F_i set to 1 if a_i=* for $1 \le i \le k$ and the rest of the bits are 0. Note that the number of bits set to 1 in a query descriptor is equal to the number of uniquely specified attributes in the query.

The search algorithm for a given query in an indexed descriptor file is simply to compare D_q to the descriptor D_B of each bucket B and access all buckets B whose descriptors have 1's in all the same positions as D_q; the values contained in the other bit positions in D_B do not need to be considered. Note for a bucket B whose bucket descriptor satisfies the above criterion, this method does not guarantee that B contains at least one record in R(q) (a "false hit").

Signature Files

A content-addressable scheme for a message file system has been proposed here. This scheme uses an idea similar to that of the indexed descriptor files described Fig.2. In this scheme, a descriptor file called a signature file is created for each message in the file system. The descriptor file F' for a message F is created by concatenating the descriptors of all uncommon words in message F. Whereas Pfaltz and Cagley set exactly one bit to 1 in each field descriptor, a word descriptor in the message-file scheme is a string of bits which represents (not uniquely) the word.

There are many algorithms for transforming a word into its descriptor. One method is to use a hash function to hash each word into a string of bits. Another method is to divide each word into overlapping triplets of letters and then to hash each triplet into into a fixed number of bits. The main idea behind both methods is that a descriptor requires much less storage space than an actual word. This reduces the amount of data requiring access when comparison are made.

The use of the signature file is illustrated in the following: a user generates a query that specifies a variable number of words and a pattern among those words. The pattern is a boolean expression involving combinations of words. For example, a user can specify five words where the pattern is the conjunction of those five words, i.e., all messages containing those five words should be retrieved. A sequential search is performed on the descriptor file to determine if the five words are contained in any entry in the file; if the search is successful, the corresponding document is retrieved. Note, however, that although the descriptor file contains the five words, it is possible that the actual corresponding message need not contain those five words. This is because there is no one-to-one correspondence between a word and its descriptor.

Partial-Match Retrieval Using Hashing and Descriptors

A partial-match retrieval scheme based solely on descriptors seems to have some disadvantages, especiallyfor dynamic files. However, a small, simplified descriptor file, built on top of a hashing scheme, is a practical and effective solution to the problem of large key spaces.

Let us now be more precise about descriptor files. A descriptor is a fixed-length bit string consisting of w bits. Typically, w would be 100 to 500. Each page in the main file has a descriptor associated with it. The descriptors are numbered in the same way as the pages of the main file, and the collection of all descriptors is called the descriptor file. Each field f, has associated with it a transformation T_i, which maps from the key space of f_i to the subset of bit strings of length w_i, where each bit string has exactly one bit set to 1, with the remainder 0. Furthermore, $w_1 + \dots + w_k = w$. Now the descriptor associated with a page is obtained by the (bitwise) ORing of the record descriptors for each record in the page and any associated

overflow pages. A record descriptor is obtained by applying the transformation T_i to each field f_i (i=1, 2, … , k) and forming the bit string of length w, which is the concatenation of each of the resulting strings. Only page descriptors are actually stored. When a record is added to or deleted from a page, the descriptor must be updated. A major difference between the descriptor file here and that in indexed descriptor file is that we have only one level of descriptor file. Furthermore, the descriptors in indexed descriptor file are larger than ours because the scheme in indexed descriptor file relies solely on the descriptors for retrieval. Thus our descriptor file is smaller. The size of the descriptor file is $w2^a$ bits.

2. ANALYSIS

The index methods are compared for add and query records. The same record structure are used to compare methods. The record pattern as shown below :

$$R_k = \begin{bmatrix} v_1 & (isim) & = & ALÝ\_KAYA \\ v_2 & (Dogum\_tarihi) & = & 12/12/1966 \\ v_3 & (Telefon\_numarasý) & = & 3324671 \\ v_4 & (Meslegi) & = & 4 \end{bmatrix}$$

Average data access time and average data insertion time graphics are shown as below :

3. CONCLUSIONS

In this study, we eximined the index technics and compared them with average data access time and average data insertion time. As a result of this process we obtained that the indexed descriptor file is the best partial-match retrieval method.

4. REFERENCES

1. **C. Faloutsos**, March 1985, “Access methods for text”, ACM Comput. Surveys, Vol. 17, No. 1, pp. 49-74
2. **C.J.Date**, 1990, “Database Systems”, Addison-Wesley Pub.Comp., Vol. 1
3. **J.Llyod and K.Ramamohanarao**, 1982, “Partial Match Retriavel For Dynamic Files”, BIT, Vol. 22, pp. 150-168.
4. **J.Pfaltz, W.Berman and E.Cagley,** September 1980, “Partial-Match Retrieval Using Indexed Descriptor Files”, Commun. ACM, Vol. 23, No. 9, pp. 522-528.
5. **K.Ramamohanarao, John W.Lloyd and James A. Thom,** December 1983, “Partial-Match Retrieval Using Hashing and Descriptors”, ACM Transactions on Database Systems, Vol. 8, No. 4, pp. 552-576.
6. **Philip J. Pratt, Joseph J. Adamski,** 1987, “Database Systems Management and Design”, Boyd & Fraser Publishing Com.
7. **R.Fagin, J.Nievergelt, N.Pippenger and R.Strong,** September 1979, “Extendible Hashing – A Fast Access Method for Dynamic Files”, ACM Transactions on Database Systems, Vol. 4, No. 3, pp. 315-344.

Oðuzhan Öztaþ was born in Izmir, Turkey, 1966. He received B. Sc. Degree in Mathematical Engineering from the Faculty of Science and Letters, Ýstanbul Technical University, Turkey in 1989. He received M.Sc. Degree in System Analysis, from the Institute of Science and Technology of the same University in 1992. He received Ph. D. Degree in Computer Engineering, from the Institute of Science and Technology, Ýstanbul University in 2001. He has been teaching computer science courses in the Computer Engineering Department of Istanbul University. His research interests include computer graphics, fuzzy systems, multimedia and database systems. He is married and has one son.

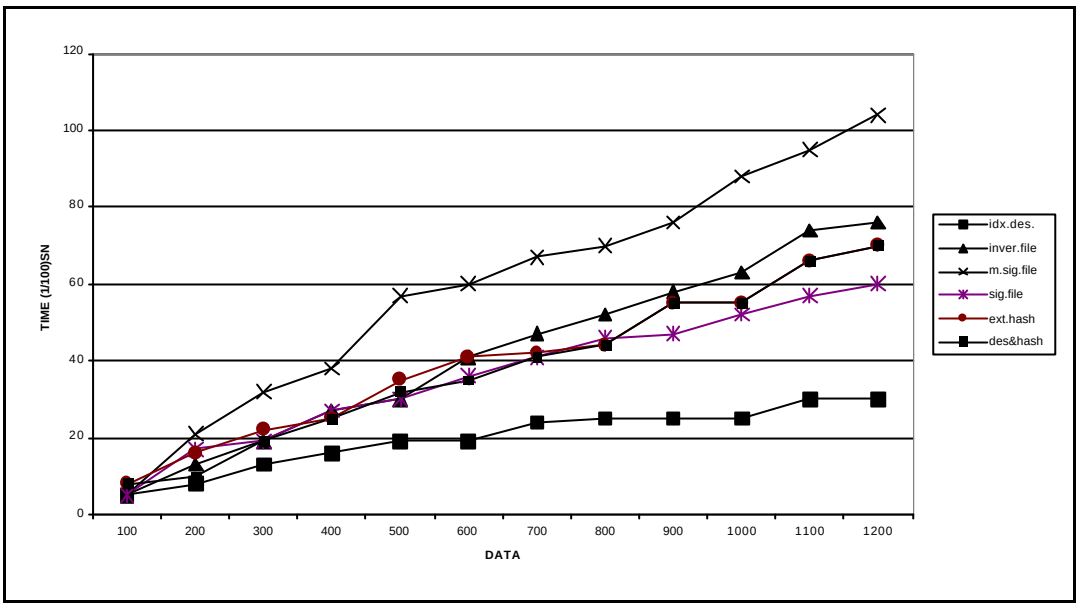

Fig. 1.Average access time.

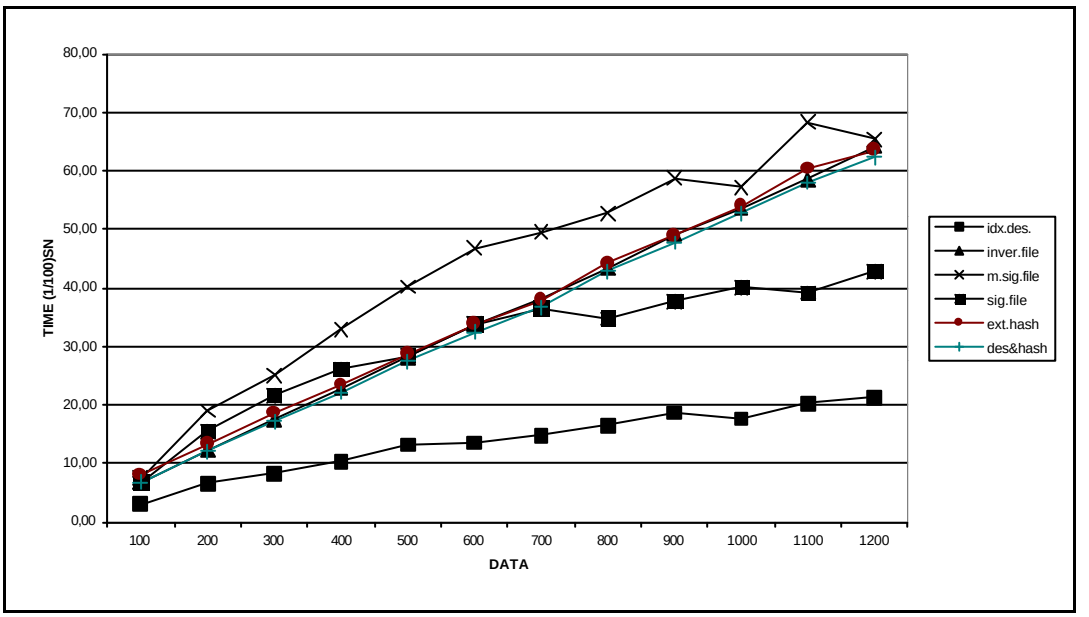

Fig. 2. Average data insertion time.