International Journal of Science and Research (IJSR)
ISSN: 2319-7064
ResearchGate Impact Factor (2018): 0.28 | SJIF (2018): 7.426

Analysis on Four Basic Types of Data Structure in Searching on External Storage

Nang Noon Kham

University of Computer Studies (Lashio), Burma

Abstract: ***The idea behind this article is to give an overview of a simple approach to organizing data in external storage rather than main memory and their search method with pseudo code. Searching is common fundamental operation and solve to searching problem in a different fields .This paper is presents the basic type of searching pseudo code like sequential ordering, b-tree, indexing , hashing for external storage and focus on how many disk access are necessary than on how many individual records they are.***

Keywords: Sequential Ordering, B-tree, Indexing, Hashing, Hash function

1. Introduction

In many situations the amount of data to be processed is too large to fit in main memory all at once. In this case a different kind of storage is necessary. Disk files generally have a much larger capacity then main memory. This is made possible by their lower cost per byte of storage. In a computer's main memory, any byte can be accessed in a fraction of a microsecond. Disk access times of around 10 millseconds are common. This is something like 10,000 times slower than main memory. This speed different means that different techniques must be used to handle it efficiently.The goal in external searching is to minimize the number of disk accesses, since each access takes so long compared to internal computation. Disk access is most efficient when data is read or write one block at a time. When the read- write head is correctly positioned and the reading (or writing) process begins , the drive can transfer a large amount of data to main memory fairly quickly. For this reason, and to simplify the drive control mechanism, data is stored on the disk in chunks called blocks, pages, allocation units or some other name, depending on the system. We'll call them block. Block size varies , depending on the operating system, the size of the disk drive, and other factors, but it is usually a power of 2.The selection of a power of 2 as a block size makes the translation of a logical address into a block number and block offset particularly easy.

2. External storage in Data Structure

Assume we have a database of 500000 records , each record is 512 bytes long , each block can store 16 records and a block size of 8192 bytes .The database will require 256000000 bytes divided by 8192 bytes per blocks. Which is 31250 blocks .And also assume that on the target machines this is too large to fit in main memory but small enough to fit on disk drive. So we can structure for a large amount of data on disk drive to provide the usual desirable characteristic, quick search by fours kind of external storage in data structure.

2.1 Sequential Ordering

The simple way to arrange the data in the disk file would be to order all the records according to some key, say alphabetically by last name.

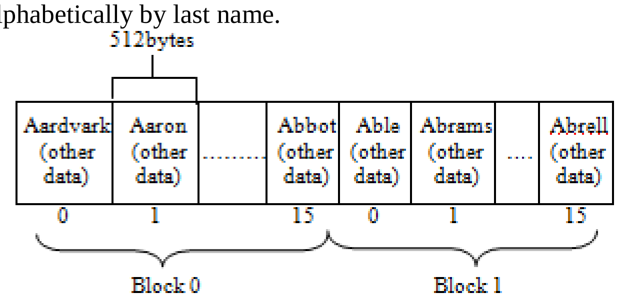

Figure 1: Sequential ordering

2.1.1 Operation on Sequential ordering Searching

To search a sequentially ordered file for a particular key, we could use a binary search. We would start by reading a block of records from the middle of the file. If the key of those records are equal to search key then return middle block. If the key is greater than those records then go to ¾ point in the file and read a block there. If the key is less than those records then go to ¼ point in the file. By continually dividing the range in half, we would eventually find the record you were looking for. If the search key isn't found and then return null.

Pseudo code:

BINARY_SEARCH (A,beg,end,mid,block,K)
While (beg <=end)
mid = (beg+end)/2
block=A.mid
DISK_READ (block)
i=0
While (i<= n[block] and K ≠ Key_i [block]

If (K == Key_i[Block]) then return block
Elseif (Key >Key_i[Block]) then beg= block+1
return BINARY_SEARCH (A, beg, end, mid,block, K)
Else end = block-1
return BINARY_SEARCH (A, begin, end, mid,block, K)
Return null

In this example there are 31250 blocks.Log2 of this number is about 15, so we'll need about 15 disk accesses to find the record we want. In practice this number is reduced because we read 16 records at once. In the beginning stages of a binary search, it doesn't help to have multiple records in memory because the next access will be in a distant part of the file. However, when we get close to the desired record, the next record we want may already be in memory because it's part of the same block of 16.This may reduce the number of comparisons by two or so. Thus, we'll need about 13 disk access (15-2),which at 10 milliseconds per access requires about 130 milliseconds, or 1/7 second.

2.2 B-tree

B-tree is a tree data structure that keeps data sorted and to provide fast searches, insertions and deletions times. B-tree are balanced search tree especially designed to be stored on disk based storage. It allows to keep both primary data records and search tree structure, out on disk. Only a few nodes from the tree and a single data record ever need be in primary memory. The order of a B-tree is the number of children each node can potentially have. B-tree is designed to store data in block fashion as it's efficient for operating systems to read and write data in blocks instead of writing individual bytes. B-tree nodes may have many children from a hundreds to thousands nodes. That is the "branching factor" of a B-tree. A large branching factor reduces the height of the tree and the number of disk accesses required to find any block and operations on B-tree are very fast.

B-tree of Order m has the following properties:

- All the leaf nodes must be at same level.
- All nodes except root must have at least [m/2]-1 keys and maximum of m-1 keys.
- All non leaf nodes except root (i.e. all internal nodes) must have at least m/2 children.

We assume, each node as one block and one block contains 16 records (1 record = 512 bytes) and in 16 records we reduce the number of records to 15 to make room for the links(which means links to other blocks).For more efficient to have an even number of records per node we reduce the record size to 507 bytes. There will be 17 child links so the links will require 68 bytes(17*4).This leaves room for 16-507 byte record with 12 bytes left over (507*16+68=8180).

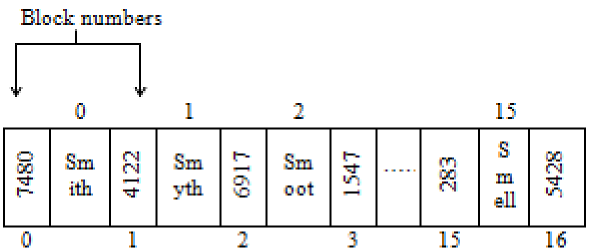

Figure 2: A node in a B-tree of order 17

2.2.1 Operation on B-tree

Searching

Within each node the data is ordered sequentially by key, a search for a record with a specified key is carried out .First, the block containing the root is read into memory. The search algorithm then starts examining each of the 15 record starting at 0.

When search key k is in the B-tree,B-TREE_SEARCH returns the ordered pair(x, i) consisting of a node root x and an index i such that $key_i[x]$= k. If x is a leaf return the null values or recur to search the appropriate subtree of x (if it find a record with a greater key, it knows to go to the child whose link lies between this record and the next one and if it find a record with a less key, it knows to go to the child whose link lies between this record and the previous one) after performing the necessary DISK-READ on that child. This process continues until the correct node is found.

Pseudo code:

B-TREE_SEARCH (x , k)
i=0
While i< n[x] and k >= key $_i$[x]
Do i=i+1
If i<= n[x] and k = key $_i$[x] then return (x , i)
If leaf [x] then return null
Else DISK-READ ($child_i$[x])
return B-TREE_SEARCH(child $_i$[x] , k)

Operations on B-tree are very fast, because there are so many records per node and so many nodes per lever. All the nodes in the B-tree are at least half full, so they contain at least 8 records and 9 links to children. The height of the tree is less than log9N, where N is 500000.This is 5.972, so there will be about 6 levels in the tree. Thus, using a B-tree, only six disk accesses are necessary to find any record in a file of 500000 records. At 10 milliseconds per access, this takes about 60 milliseconds, or 6/100 of a second .

2.3 Indexing

A different approach to speeding up file access is to store records in sequential order but use a file index along with the data itself. A file index is a list of key/block pairs, arranged with the keys in order. Assuming our search keys is last name, every entry in the index contains two items: The key, and the number of the block where the last name record is located within the file. These numbers run from 0 to 31249.Let's say we use a string 28 bytes long for the key (bit enough for most last names) and 4 bytes for the block number. Each entry in our index requires 32 bytes. This is only 1/16 the amount necessary for each record. The entry in the index are arranged sequentially by last name. The original records on the disk can be arranged in any convenient order. This means that new records are simply appended to the end of the file, so the records are ordered by time of insertion.

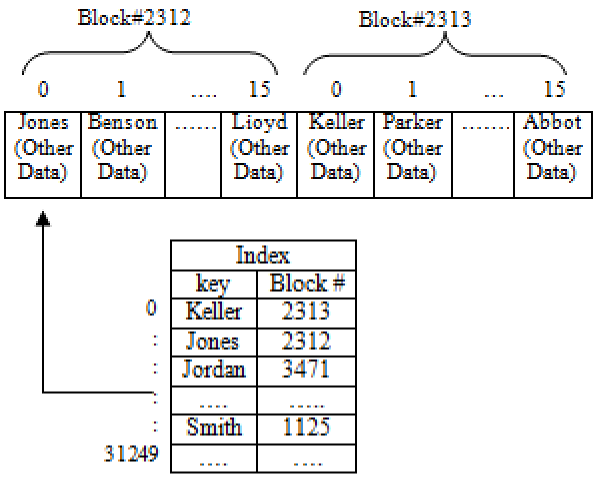

Figure 3: A file index

2.3.1 Operation on Indexed file

Searching

Index file is a list of key/block# pairs, arranged with the keys in order. We first compare to search with key with the mid key of the Indexfile. If search key is greater than to mid key then go to lower part of Index file. If search key is less than to mid key then go to upper part of Index file .Otherwise we get block# from Indexfile and read that block using Linear Search. If search key is match then return that block. If do not match return null.

Pseudocode:

IndexBinarySearch(index,beg,end,mid,sk,sr)
While(beg <= end)
Mid = (beg+end)/2
If (SK >index.mid.key) then beg = mid+1
Else if (SK <index.mid.key) then end = mid-1
Else block = index.mid.block#
DISK_READ(block)
i = 0
For(i = 0,i<n[block],i++)
If (SR == r_i[block]) then return block
If (i == n[block] then return null

The index is much smaller than the file containing actual records. It may even be small enough to fit entirely in main memory. Operations on the index can take place in memory and faster operations on the file . There is the time to read the actual record from the file, once its block number has been found in the index. This is only one disk access using indexing.

2.4 Hashing

Hashing for disk files is called external hashing. One of the main goals in External hashing is to reduce the number of accesses(probes) to secondary storage. External hashing is a hash table containing block number, which refer to blocks in external storage. Each block may hold multiple records of sequential order. All records with keys that hash to the same value are located in the same block. The hash table can be stored in main memory or, if it is too large, stored externally on disk, with only part of it being read into main memory at a time. The index (hash table) in main memory holds pointers to the file blocks.

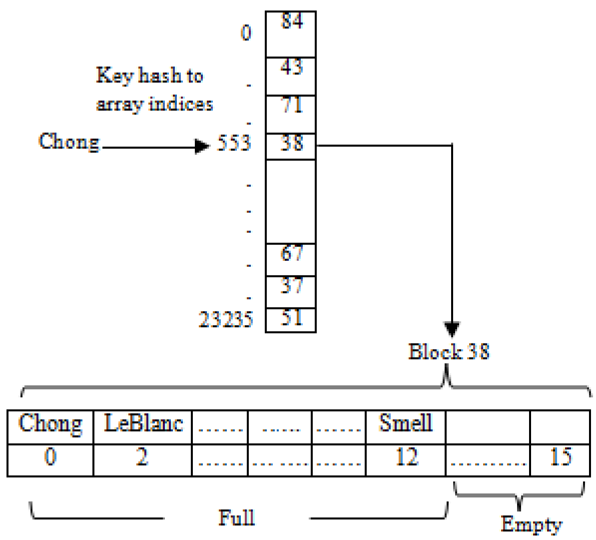

Figure 4: External Hashing

2.4.1 Operation on Hashing

Search

To find a record with a particular key, the search algorithm hashes the key, uses the hash value as an index to the hash table, gets the block number at that index and reads the block. If item is not in that block , this situation can be handled using linear probing method .In linear probing method the step size is always 1 so if x is the array index calculated by the hash function , the probe goes to x,x+1,x+2,x+3 and so on. In external hashing full block are undesirable because an additional disk access is necessary for the second block: this doubles the access time. Only ¾ percent of block are full and 12 records are hold in one block. Which is 23436 block for 500000 records and the hash function compute a value in the range 0 to 23235.

A hash function

A hash function H(k)transforms a key into an a address / hash index .The contents of index position is block number.If keys are strings, get integers by converting the letters to their numerical equivalents, multiply them by appropriate powers of 27(because there are 27 possible characters, including the blank, e.g. “a” = 1, “b” =2, “c” = 3, “d” = 4 etc.)

Pseudocode:

HASH_FUNCTION(char key)
HashIndex(key) = StringInt (key) mod ArraySize
HashBlock = HashArray[HashIndex]
Return HASHBLOCK_SEARCH(HashBlock,SKey)

HASHBLOCK_SEARCH(HashBlock,key)
{ i=0
if(i<n[HashBlock] and HashBlocki.key<= key)
{if(HashBlocki.key = = key) then return HashBlocki
Else ++ i
}
Else if (i>= n) then HashBlock = ++ HashBlock
Disk_read(HashBlock)
Return
HASHBLOCK_SEARCH(HashBlock,key)
Else return null
If ArrayIndex[HashBlock] = = ArraySize then ArrayIndex% = ArraySize
}

This process is efficient because only one disk access is necessary to locate a given item.

3. Conclusion

This paper discusses how to organize data in external storage using four basis types of data structure and how many disk accesses are used by each searching method .Sequential storage might be satisfactory for a small amount of records, not for access time .B-tree can work on very large files.The larger the branching factor the smaller the height of the tree and disk access is required to find any block and operations. As a result, B-tree is very fast.External hashing has the same access time as Index files, but can handle larger file and might be a good choice.

References

[1] ABRAHAM SILBERSCHATZ,PETER BAER GALVIN,GREG GAGNE:OPERATING SYSTEM PRINCIPLES,Seven Edition
[2] Robert Lafore:Data Structure and Algorithm in JAVA™,Second Edition
[3] SEYMOUR LIPSCHUTZ:THEORY AND PROBLEMS OF DATA STRUCTURES
[4] Cormen T.H., Leiserson C.E., Rivest R.L.: Introduction to algorithms, McGraw-Hill Book Company, New York, USA, 2000
[5] KamleshkumarPamdey,NarendraPradhan:"A comparison and selection on Basic Type of Searching Algorithm in Data Structure",International Journal of Computer Science and Mobile Computing,Vol.3 Issue 7,July-2014
[6] Petra Koruga, Miroslav Baca:"Analysis of B-tree data structure and its usage in computer forensics",The 21st Central European Conference on Information and Intelligent Systems,September 22-24 , 2010