**2021-2022** Assignment 2: Indexing/Hashing

This assignment is for understanding indexing and hashing.

Q1: An ordered index can be maintained using an indexed-sequential file, and an indexed-sequential file is to use a sequential file to maintain the sequential order of index entries, where an index entry is a pair of search key and the point pointing to the record in the data file. Let the index entry be denoted as *I* = (*S,P*), where *S* is a search key, and *P* is its pointer. Following the discussion on sequential files (Chapter 10), in the sequential file to be, for each index entry *I*, there is an additional pointer, denoted as *N*, pointing to the next index entry *I*′ following the sequential order. On the other hand, recall that a file is a sequence of blocks on disk. Therefore, all index entries in an ordered index are maintained in blocks on disk. The next pointer (*N*) is specified by a pair of block-id and slot-id, where the block-id indicates the block where the next index entry resides in the file, and the slot-id indicates the location where the next index entry is in that block (referring to the slotted page structure in Chapter 10). Answer the following regarding I/O cost.

1. Suppose that the indexed-sequential file is *B* blocks large in size without any overflow blocks, where a block can keep up *n* pairs of search key and pointer. And both the min and max search keys in the indexed-sequential file are known. Show a pseudo-code in brief to find whether an index entry is in the indexed-sequential file efficiently for a given search key. You have to make it clear for block access and index access in your pseudo-code. How many block accesses do you need to do so at most.

procedure search\_with\_key(key minKey, key maxKey, block B, search S)

if minKey is equal to S, then return the point P.

else

Set P= minKey's pointer.

Nnext=P.next

Nnext=(minKey', P')

Set minKey=minKey'.

do search\_with\_key(minKey, maxKey, B, S)

at most, need to access nB blocks.

1. Suppose that the indexed-sequential file is *B* blocks large in size with possible overflow blocks, where a block can keep up *n* pairs of search key and pointer. And both the min and max search keys in the indexed-sequential file are known. Can you use the similar pseudo-code in (1) to find it? Explain if you can or cannot in detail.

Show a pseudo-code in brief to deal with it when there are possible overflow blocks. You have to make it clear for block access (non-overflow and overflow blocks) and index access in your pseudo-code.

Note: an overflow block, *b*, may have another overflow block *b*′ if the overflow block *b* becomes full. How many block accesses do you need in the worst case, assuming each block can keep up to *n* index entries. Explain it in detail using some examples.

procedure search\_with\_key(key minKey, key maxKey, block B, search S, overblock b’)

if minKey is equal to S, then return the point P.

else

Set P= minKey's pointer.

Nnext=P.next

If Nnext is in overflow block b’, then access b‘

Nnext=(minKey', P')

Set minKey=minKey'.

do search\_with\_key(minKey, maxKey, B, S, overblock b’)

else

Nnext=(minKey', P')

Set minKey=minKey'.

do search\_with\_key(minKey, maxKey, B, S, overblock b’)

In the worst case, need to access nB+(n-1)b’ blocks.

1. Comparing (1) and (2), the efficiency are different. A question is how much update cost you want to pay when there are new index entries inserted, given the two options of (1) and (2). This question is also related how frequently the ordered index is used and how frequently the indexed-sequential file is updated. Do you want to keep the indexed-sequential file as one without any overflow blocks always, or allow some overflow blocks? Share you thought.

It will work well if there are relatively few records that need to be stored in the overflow block. However, the consistency between search key order and physical order may be broken, in which case sequential processing will be inefficient. At this point, the files should be reorganized to make the physical order of the files consistent, but this is expensive. There, I will allow some overflow blocks in order to avoiding the frequent file reorganizations.

Q2: Consider a relation *s* with one attribute *A*. Suppose you have to insert the following A = {100*,*400*,*600*,*500*,*550*,*520*,*522*,*521*,*101*,*510*,*300*,*112} into the relation *s*. Construct a B+-tree on the attribute *A* of *s*. Following the pseudo-code of B+-tree given in the textbook (or the slides in ch11.pptx), construct a B+-tree with *n* = 3. Assume that the B+-tree is initially empty.

1. Given A, insert the key values in such an order, and show the B+-tree after every insertion.
2. Given the B+-tree constructed in the step (1), show how the B+-tree is changed step-by-step following the pseudo-code when the key value 521 is deleted.
3. Given the B+-tree constructed in the step (2), show how the B+-tree is changed step-by-step following the ideas shown in pseudo-code when the key value 520 is deleted.

 **Question** 3: The B+-tree insertion/deletion pseudo-code is for a B+-tree where in the data file either the search keys are sorted (e.g., primary index) or every search key only appears once (e.g., search keys are unique). Now, consider the case where the data file is not sorted following the search keys, and the search keys are not unique. It is a typical case for secondary index. An example of secondary index is shown in the slide 11.11 in ch11.pptx. Consider the example when we have to use B+-tree. We can replace the indexed-sequential file on the left in the slide 11.11 with a B+-tree where the data file is on the right. The one in the middle is called an inverted file, which keeps inverted records. First, a pointer in a leaf-node in B+tree, that points to a data record, is now pointing to an inverted record in the inverted file. An inverted record instead points to the corresponding record in the data file. All inverted records for the same search keys are placed next to each other as a cluster. In other words, following the pointer in the leaf node for a search key, we can find all inverted records with the same search key, and we can access all data records in the data file following the inverted records.

It is important to note that an inverted file is a file that keeps inverted records in blocks. First, write pseudo-code to maintain inverted records in an inverted file for insertion case only. Hint: you do not need to keep all inverted records sorted in a sequential order, you only need to keep the inverted records that have the same search key as a cluster together. Here, we assume that all inverted records for the same search key can be possibly kept in a block, and you have to do so to reduce the I/O cost. Second, show where you have to modify the pseudo-code for the B+-tree insertion when you have to support B+-tree which is for secondary index.

I have to modify the pseudo-code for the B+-tree insertion of inserting a child, here I need to support more than one search key with the same value. So modify “if(K<=L.K1) then insert P,K into L just before L.P1” to support B+-tree for secondary index.

procedure insert\_reverted(value K, pointer P)

if (tree is empty) create an empty leaf node L, which is the root

else Find the lead node L that should contain key value K

if (L had less than n-1 key values)

then insert\_in\_leaf(L, K, P)

else begin

Create node L'

Copy L.P1...L.Kn-1 to block of memory T that can

holed n(pointer, key-value) pairs

insert\_in\_leaf(T, K, P)

Set L'.Pn= L.Pn; Set L.Pn=L'

Erase L.P1 throught L.Kn-1 from L

Copy T.P1 through T.Kn/2 from T into L starting at L.P1

Copy T.Pn/2+1 through T.Kn from T into L' starting at L'.P1

Let K' be the smallest key-value in L'

insert\_in\_parent(L,K', L')

end

procedure insert\_in\_leaf(value K, pointer P)

if(K<=L.K1)

then insert P,K into L just before L.P1

else begin

Let Ki be the highest value in L that is less than K.

Insert P,K into L just after T.Ki

end

proceduer insert\_in\_parent(node N, value K', node N')

if (N is the root of the tree)

then begin

Create a new node R containing N, K', N'

Make R the root of tree

return

end

Let P=parent(N)

if P has less than n pointers

then insert(K',N') in P just after N

else begin

Copy P to a block of memory T that can hold P and (K',N')

Insert (K',N') into T just after N

Erase all entries from P; Create node P'

Copy T.P1...T.Pn/2 into P1

Let K''=T.Kn/2

Copy T.Pn/2+1 ...T.Pn\_1 into P'

insert\_in\_parent(P,K'',P')

end

Q4: Consider the same relation *s* with one attribute *A*. Suppose you have to insert the following A = {100*,*400*,*600*,*500*,*550*,*520*,*522*,*521*,*101*,*510*,* 300*,* 104} into the relation *s*. Construct a dynamic hash index on the attribute *A* of *s* based on extendable hashing. Assume the hash function used is *h*(*k*) = *k*%16, where “%” is the modular operation and the hash function returns the remainder of *k* divided by 16. In other words, in this question, the hash function *h*(·) generates a *b*-bit binary integer, for *b* = 4, from 0000 to 1111. For simplicity, suppose that, in the hash index, a bucket keeps 2 entries, where an entry is a pair of search-key and pointer to the record in *s* that has the key. Assume that the dynamic hash index is with one bucket address table and one empty bucket initially.

1. Given A, insert the key values in such an order, and show the dynamic hash index (the bucket address table and the buckets) after every insertion.
2. In the textbook, there are no algorithms given on how to handle deletion by the extendable hashing. Following the ideas given for extendable hashing to deal with deletion (the slide 11.23 in ch11-hash.pptx), give the steps to handle extendable hashing for deletion in detail.
3. Show how your algorithm works for deletion using examples when it ends up reducing the number of buckets and shrinking the bucket address table. Hint: you do not necessarily use the same example used in this question.

(1)

ℎ(𝑥)=𝑝𝑟𝑒𝑓𝑖𝑥(𝑏, 𝒉(𝑥))

h(k)=k%16

h(k)=prefix(b, k%16))

|  |  |  |
| --- | --- | --- |
| x | x%16 | h(x) |
| 100 | 100%16=4 | 0100 |
| 400 | 400%16=0 | 0000 |
| 600 | 600%16=8 | 1000 |
| 500 | 500%16=4 | 0100 |
| 550 | 550%16=6 | 0110 |
| 520 | 520%16=8 | 1000 |
| 522 | 522%16=10 | 1010 |
| 521 | 521%16=9 | 1001 |
| 101 | 101%16=5 | 0101 |
| 510 | 510%16=14 | 1110 |
| 300 | 300%16=12 | 1100 |
| 114 | 114%16=2 | 0010 |

(2)

Procedure delete\_key(value V, prefix I, buckets B)

Find bucket B of the V,

If Find, then remove it;

If the V is empty, then coalescing buckets. The bucket can be removed, and coalescing buckets with same prefix I.

If need decrease buckets, then decreasing bucket address size.