CS 455: Principles of Database Systems

Review Guide 4: Hashing and Indexing (Selected Soln)

Your Name:

1. **[Disk Scheduling]** None of the disk-scheduling policies, except FCFS, is truly fair (starvation may occur). Explain why this assertion is true.
Solution: C-LOOK, LOOK, SSTF all prioritize head locality (and direction in the case of C-LOOK and LOOK). It is possible that the query processor requests blocks near or at the head's current track at a rate that is faster than the service rate.

2. **[Hashing]** Consider a hash file that using the function $h(k) = k \ mod \ n$, where n is an arbitrary positive constant and k is the search key for a tuple. Answer the following questions:

   (a) With regards to $h(k)$, discuss the pros and cons of a small n (as it approaches 1) versus a large n (as it approaches the total number of blocks taken up by this file).
   Solution: Small n – It's good use of space when the table is small. When the table is large, you'd need to chain blocks together in a small number of buckets to hold all the tuples, so the equality-search performance approximates that of heap files. Large n – may be a bad use of space (i.e., lots of blocks allocated for empty buckets) when the data set is small. Fast $O(1)$ operation to find the search key's bucket the block within (fewer chances for overflow chains).

   (b) A follow up to the previous question, the maximum length of bucket overflow is often an indication of the "goodness" of a hash function. Explain why.
   Solution: If the max length of the bucket-overflow list is small, that means the hash function that was chosen is probably distributing tuples in a uniform fashion, keeping disk blocks as full as possible. A bad hash function maps a disproportionate amount of keys to only a few buckets, causing a drastic increase in the length of those overflow lists.

3. **[Disk]** Hard disks store more sectors on the outer tracks than the inner tracks. Since the time required for a full revolution is constant, the sequential data transfer rate is also higher on the outer tracks. Seek time and rotational latency are unchanged. Considering this information, explain good strategies for placing files on disk with the following kinds of access patterns: (9pts)

   (a) Frequent, random accesses to a small file.
   Solution: Due to its random access pattern (jumping around in this file), placing it on the outer tracks would not benefit from the faster rotational speeds observed there. Furthermore, placing the file on the outer tracks would take valuable blocks away from larger files that could actually benefit from being stored there. I would place small files requiring random access toward the middle-to-inner tracks.

   (b) Sequential scans of a large file.
   Solution: Place file on the outer tracks. Outer tracks occupy more surface area, and stores more data. This maximizes the chances that most, if not all, blocks pertaining to the file are stored contiguously on the same or adjacent tracks, maximizing spatial locality. Second, given a constant unit of time, laws of physics tell us that outer tracks must move *faster* than the inner tracks, allowing more data to be read over the same period of time.

   (c) Sequential scans of a small file.
   Solution: Here's a good candidate for middle-track placement. Sequential scans benefit from

the faster speeds observed towards the outer tracks, but you still don't want to take those outer blocks away from the large files. Placing them towards too far inside doesn't make sense either, as you'll require entire revolution(s) to read only a small amount of data. The DAT will be dominated by the initial head seek, and placing it in the middle exploits common disk-scheduling policies that prioritize head location.

4. **[Joins]** Assume that you want to join two relations $R(A, B)$ and $S(B, C)$. The two relations are stored as simple (unsorted) heap files. When would you prefer a hash join to a sort-merge join, and when would you prefer a sort-merge join (SMJ) to a hash-join (HJ)?

   You would prefer a HJ normally, since both files must be sorted to perform SMJ. However, HJ can only be used if B contains unique values for you to build the hashmap. If B contains duplicates, then you would prefer SMJ.

5. **[B+Trees]** Show that the height of a B+Tree with degree d is $O(log_{\lceil d/2 \rceil} N)$, where N is the number of keys stored.

Solution:

In the worst case, all tree nodes except for root are only 50% occupied with $\lceil d/2 \rceil$ pointers to children and $\lceil d/2 \rceil - 1$ keys. Recall that the root node is special: the least number of children it can have is just 2 with a single key. This occurs when the previous root node was just split. Therefore, given a tree height i, we have a corresponding worst-case number of B$^+$-Tree nodes:

Height (i)	**Worst-Case Number of Nodes at Level** i
0 (root level)	1
1	2
2	$2\lceil d/2 \rceil$
3	$2\lceil d/2 \rceil^2$
4	$2\lceil d/2 \rceil^3$
$\vdots$	$\vdots$
h	$2\lceil d/2 \rceil^{h-1}$

Given height h, the number of keys N stored in the B$^+$-tree is expressed as,

$$N \geq 1 + 2k + 2\lceil d/2 \rceil k + 2\lceil d/2 \rceil^2 k + \ldots + 2\lceil d/2 \rceil^{h-1} k \tag{1}$$

where $k = \lceil d/2 \rceil - 1$ is the worst-case number of keys stored per node, as mentioned earlier. In Equation 1, the 1 term is the lone key stored at the root node. In the second level of the tree, there can be only 2 nodes (since in the worst case, the root node only has 2 children) and each node contains k keys. Then in the third level, there can be $2\lceil d/2 \rceil$ nodes, since both left and right nodes of the second level had $\lceil d/2 \rceil$ children, and so on, so forth.

After substituting k and a bit of re-writing, we obtain,

$$N \geq 1 + 2(\lceil d/2 \rceil - 1) \left[\sum_{i=0}^{h-1} \lceil d/2 \rceil^i \right] \tag{2}$$

The summation on the right is a *geometric series* and can be rewritten in closed-form notation,

$$= 1 + 2(\lceil d/2 \rceil - 1) \left[\frac{1 - \lceil d/2 \rceil^h}{1 - \lceil d/2 \rceil} \right] \tag{3}$$

$$= 1 + 2(\lceil d/2 \rceil - 1) \left[\frac{1 - \lceil d/2 \rceil^h}{1 - \lceil d/2 \rceil} \cdot \left(\frac{-1}{-1} \right) \right] \tag{4}$$

$$= 1 + 2(\lceil d/2 \rceil - 1) \left[\frac{\lceil d/2 \rceil^h - 1}{\lceil d/2 \rceil - 1} \right] \tag{5}$$

$$= 2\lceil d/2 \rceil^h - 1 \tag{6}$$

We now isolate h,

$$N \geq 2\lceil d/2 \rceil^h - 1 \tag{7}$$

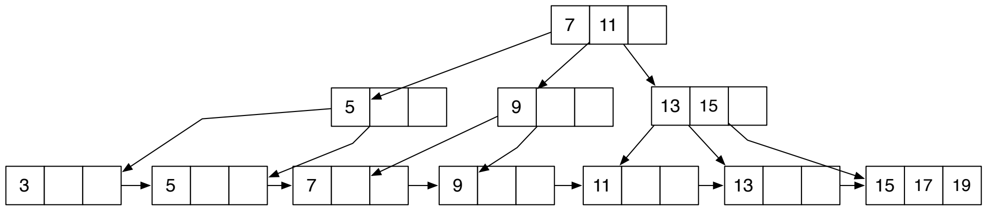
$$\frac{N+1}{2} \geq \lceil d/2 \rceil^h \tag{8}$$

$$h \leq \log_{\lceil d/2 \rceil}\left(\frac{N+1}{2}\right) \tag{9}$$

$$\therefore \quad h = O(\log_{\lceil d/2 \rceil} N) \quad \blacksquare \tag{10}$$

6. **[B+Trees]** Consider a B$^+$-Tree with $d = 4$, andyou're given a list of keys to insert: $K = (3, 5, 7, 9, 11, 13, 15, 17, 19)$. Suppose we want to try out a new splitting strategy: keep the first $\lceil d/2 \rceil - 1$ keys in the original node, and split off the rest. Show the final B+Tree if the keys in K were inserted in the given order. How does the tree differ from the one built using the splitting strategy we used in class?

   Solution: In this case, when node N splits into a sibling N', we maintain the first $\lceil d/2 \rceil - 1 = 1$ key in N. This means N' would absorb the remaining keys, plus the newly inserted key which caused the overflow, for a total of $(d-1) - (\lceil d/2 \rceil - 1) + 1 = 3$ keys.



   In this particular case, this splitting strategy built a very skewed tree (worst-case, actually), but it was only because the keys were inserted in increasing order. This is why you'd normally want to insert keys in a random order (a rule of thumb for the Binary Search Tree, too!). The splitting strategy used in class and in your homework assignment produces a full tree with only a height of 2.

7. **[Dynamic Hashing]** Repeat the above insertion sequence on an extendible-hash index. Each bucket holds only 2 tuples. Assume we use 5-bit keys, and that $h(k) = k$. For this problem, examine the **most-significant bits** at each directory.

   Solution: As a clarification, we use 5-bit keys because the largest key value to be inserted was 19, i.e., we need at least $\lceil log_2 19 \rceil$ bits to represent 19. The first two keys (3 and 5) are inserted trivially. The next key 7 overflows the lone block, and will cause a split at the local directory, incrementing $i_0 = 1$. Because the local depth is now greater than the global depth ($i_0 > i$), we propagate the split to the global directory, then we attempt to re-insert 7. However, the most significant i bits of 7 still hashes to a bucket that is full, causing another split! Because the local depth is again greater than the global depth, we again double the global directory, and re-insert 7. Once again, the most significant 2 bits of 7 *still* hashes it to a bucket that is full, causing *yet another split at the local and global directories!* Finally, 7 is inserted, followed by 9 and 11. Inserting 13 will cause a split at the local level, but it does not propagate to the global level (why?). Afterwards, 15, 17, and 19 can be inserted without further splits.

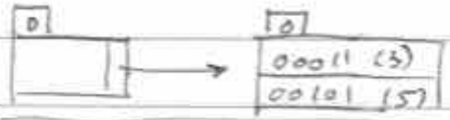
   This question demonstrates the "growing pains" I referred to in class. When starting fresh, expect a lot of splits right away (which take time and doubles the amount of space occupied by the global directory!). However, after a good amount of splits, things tend to stabilize: notice that all remaining keys after 7 were inserted without further expansion of the global directory.

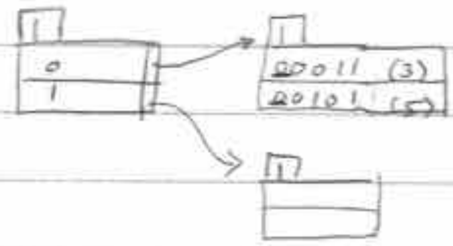
Insert Order

3 (00011)

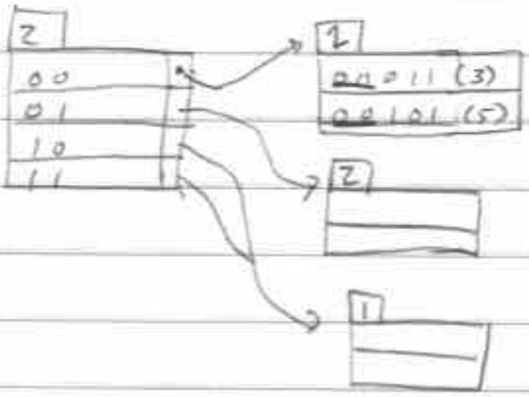
5 (00101)

7 (00111)





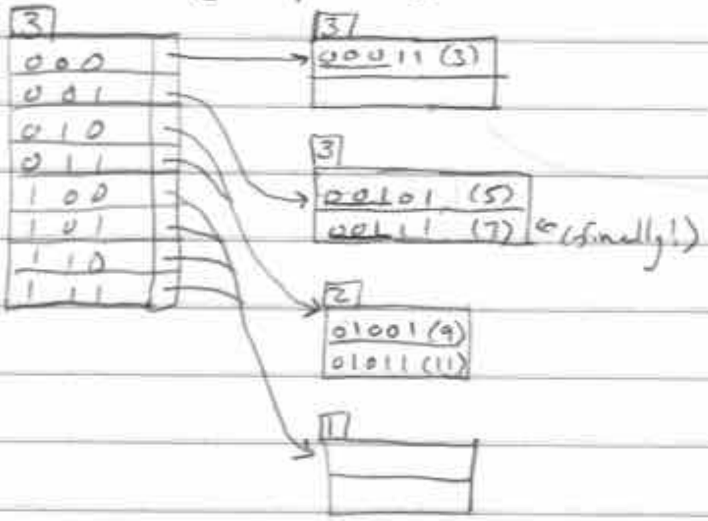
(tricky! Split Again!)



(real tricky! Split Again!)

9 (01001)

11 (01011)

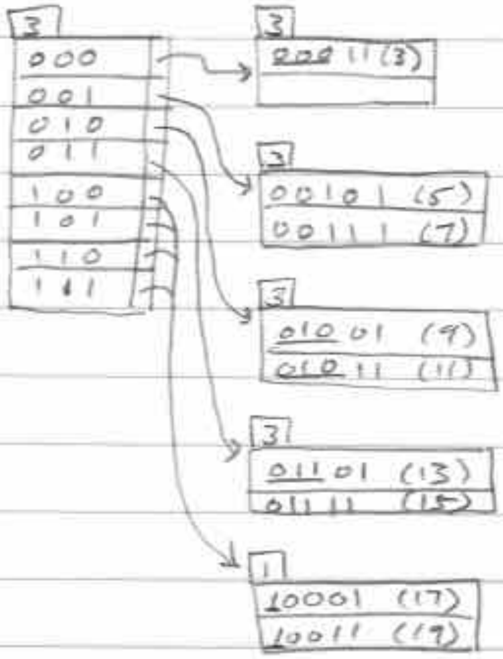


13 (01101)

15 (01111)

17 (10001)

19 (10011)



3
000
001
010
011
100
101
110
111
3
00011 (3)
3
00101 (5)
00111 (7)
3
01001 (9)
01011 (11)
3
01101 (13)
01111 (15)
1
10001 (17)
10011 (19)