CHAPTER 24

Advanced Indexing Techniques

Practice Exercises

24.1 Both LSM trees and buffer trees (described in Section 14.8.2) offer benefits to write-intensive workloads, compared to normal B^+-trees, and buffer trees offer potentially better lookup performance. Yet LSM trees are more frequently used in Big Data settings. What is the most important reason for this preference?

Answer:
The biggest difference is that buffer trees require more random I/O for insert operations as compared to LSM trees, whereas LSM trees perform more sequential I/O operations. For Big Data settings where data is resident on magnetic disks, random I/O is significantly more expensive than sequential I/O, and thus LSM trees are preferred.

24.2 Consider the optimized technique for counting the number of bits that are set in a bitmap. What are the tradeoffs in choosing a smaller versus a larger array size, keeping cache size in mind?

Answer:
A larger array requires fewer add operations, but if you increase the size of the array beyond the cache size, there will be an increased overhead for array element access. Thus, the array should be sized to fit in cache. With L1 cache sizes of a few megabytes, an array indexed by 16 to 24 bytes would work well.

24.3 Suppose you want to store line segments in an R-tree. If a line segment is not parallel to the axes, the bounding box for it can be large, containing a large empty area.

- Describe the effect on performance of having large bounding boxes on queries that ask for line segments intersecting a given region.
- Briefly describe a technique to improve performance for such queries and give an example of its benefit. Hint: You can divide segments into smaller pieces.

Answer:
FILL

24.4 Give a search algorithm on an R-tree for efficiently finding the nearest neighbor to a given query point.

Answer:
Idea: Priortize search of children nodes based on the distance of the bounding box fom the query point (the distance is 0 if the bounding box contains the query point). Maintain a priority queue of nodes based on the distance. When you find an answer at distance d and also all nodes whose bounding boxes are at distance d or closer have been explored, the search can stop since any unexplored items are at distance greater than d.

24.5 Give a recursive procedure to efficiently compute the spatial join of two relations with R-tree indices. (Hint: Use bounding boxes to check if leaf entries under a pair of internal nodes may intersect.)

Answer:
Every pair of bounding boxes at each level that intersect need to be explored further; when exploring a pair, all intersecting child pairs are generated and explored further, unless they are leaves in which case the answer can be generated.

24.6 Suppose that we are using extendable hashing on a file that contains records with the following search-key values:

2, 3, 5, 7, 11, 17, 19, 23, 29, 31

Show the extendable hash structure for this file if the hash function is $h(x) = x$ mod 8 and buckets can hold three records.

Answer:
The extendable hash structure is shown in **??**

24.7 Show how the extendable hash structure of Exercise 24.6 changes as the result of each of the following steps:

a. Delete 11.

b. Delete 31.

c. Insert 1.

d. Insert 15.

Answer:

a. Delete 11: From the answer to Exercise 24.6, change the third bucket to:

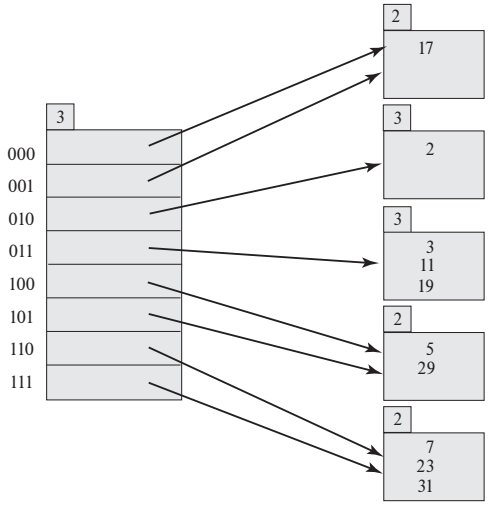
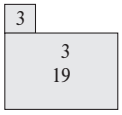
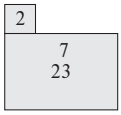


Figure 24.101 The extendable hash structure for Exercise 24.6

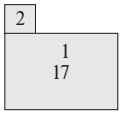


At this stage, it is possible to coalesce the second and third buckets. Then it is enough if the bucket address table has just four entries instead of eight. For the purpose of this answer, we do not do the coalescing.

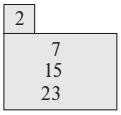
b. Delete 31: From the answer to Exercise 24.6, change the last bucket to:



c. Insert 1: From the answer to Exercise 24.6, change the first bucket to:



d. Insert 15: From the answer to Exercise 24.6, change the last bucket to:



24.8 Give pseudocode for deletion of entries fromAVi an extendable hash structure, including details of when and how to coalesce buckets. Do not bother about reducing the size of the bucket address table.

Answer:

Let i denote the number of bits of the hash value used in the hash table. Let **bsize** denote the maximum capacity of each bucket. The pseudocode is shown in **??**.

Note that we can only merge two buckets at a time. The common hash prefix of the resultant bucket will have length one less than the two buckets merged. Hence we look at the buddy bucket of bucket j differing from it only at the last bit. If the common hash prefix of this bucket is not i_j, then this implies that the buddy bucket has been further split and merge is not possible.

When merge is successful, further merging may be possible, which is handled by a recursive call to *coalesce* at the end of the function.

24.9 Suggest an efficient way to test if the bucket address table in extendable hashing can be reduced in size by storing an extra count with the bucket address table. Give details of how the count should be maintained when buckets are split, coalesced, or deleted. (*Note*: Reducing the size of the bucket address table is an expensive operation, and subsequent inserts may cause the table to grow again. Therefore, it is best not to reduce the size as soon as it is possible to do so, but instead do it only if the number of index entries becomes small compared to the bucket-address-table size.)

Answer:

If the hash table is currently using i bits of the hash value, then maintain a count of buckets for which the length of common hash prefix is exactly i.

Consider a bucket j with length of common hash prefix i_j. If the bucket is being split, and i_j is equal to i, then reset the count to 1. If the bucket is being split and i_j is one less than i, then increase the count by 1. It the bucket is being coalesced, and i_j is equal to i, then decrease the count by 1. If the count becomes 0, then the bucket address table can be reduced in size at that point.

However, note that if the bucket address table is not reduced at that point, then the count has no significance afterwards. If we want to postpone the reduction, we have to keep an array of counts, i.e., a count for each value of common hash prefix. The array has to be updated in a similar fashion. The bucket address table can be reduced if the i^{th} entry of the array is 0, where

```
delete(value K_l)
begin
    j = first i high-order bits of h(K_l);
    delete value K_l from bucket j;
    coalesce(bucket j);
end

coalesce(bucket j)
begin
    i_j = bits used in bucket j;
    k = any bucket with first (i_j − 1) bits same as that
        of bucket j while the bit i_j is reversed;
    i_k = bits used in bucket k;
    if(i_j ≠ i_k)
        return; /* buckets cannot be merged */
    if(entries in j + entries in k > bsize)
        return; /* buckets cannot be merged */
    move entries of bucket k into bucket j;

    decrease the value of i_j by 1;
    make all the bucket-address-table entries,
    which pointed to bucket k, point to j;

    coalesce(bucket j);
end
```

Figure 24.102 Pseudocode for delition of Exercise 24.8.

i is the number of bits the table is using. Since bucket table reduction is an expensive operation, it is not always advisable to reduce the table. It should be reduced only when a sufficient number of entries at the end of the count array become 0.