Practice Exercises

11.1 Indices speed query processing, but it is usually a bad idea to create indices on every attribute, and every combinations of attributes, that is a potential search keys. Explain why.

11.2 Is it possible in general to have two clustering indices on the same relation for different search keys? Explain your answer.

11.3 Construct a B$^+$-tree for the following set of key values:

$$(2, 3, 5, 7, 11, 17, 19, 23, 29, 31)$$

Assume that the tree is initially empty and values are added in ascending order. Construct B$^+$-trees for the cases where the number of pointers that will fit in one node is as follows:

a. Four

b. Six

c. Eight

11.4 For each B$^+$-tree of Practice Exercise 11.3, show the form of the tree after each of the following series of operations:

a. Insert 9.

b. Insert 10.

c. Insert 8.

d. Delete 23.

e. Delete 19.

11.5 Consider the modified redistribution scheme for B$^+$-trees described on page 501. What is the expected height of the tree as a function of n?

11.6 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 $h(x) = x \bmod 8$ and buckets can hold three records.

11.7 Show how the extendable hash structure of Practice Exercise 11.6 changes as the result of each of the following steps:

a. Delete 11.

b. Delete 31.

c. Insert 1.

d. Insert 15.

11.8 Give pseudocode for a B^+-tree function `findIterator()`, which is like the function `find()`, except that it returns an iterator object, as described in Section 11.3.2. Also give pseudocode for the iterator class, including the variables in the iterator object, and the `next()` method.

11.9 Give pseudocode for deletion of entries from 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.

11.10 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.)

11.11 Consider the *instructor* relation shown in Figure 11.1.

a. Construct a bitmap index on the attribute *salary*, dividing *salary* values into 4 ranges: below 50000, 50000 to below 60000, 60000 to below 70000, and 70000 and above.

b. Consider a query that requests all instructors in the Finance department with salary of 80000 or more. Outline the steps in answering the query, and show the final and intermediate bitmaps constructed to answer the query.

11.12 What would the occupancy of each leaf node of a B^+-tree be, if index entries are inserted in sorted order? Explain why.

11.13 Suppose you have a relation r with n_r tuples on which a secondary B^+-tree is to be constructed.

a. Give a formula for the cost of building the B^+-tree index by inserting one record at a time. Assume each block will hold an average of f entries, and that all levels of the tree above the leaf are in memory.

b. Assuming a random disk access takes 10 milliseconds, what is the cost of index construction on a relation with 10 million records?

c. Write pseudocode for bottom-up construction of a B^+-tree, which was outlined in Section 11.4.4. You can assume that a function to efficiently sort a large file is available.

11.14 Why might the leaf nodes of a B^+-tree file organization lose sequentiality?

a. Suggest how the file organization may be reorganized to restore sequentiality.

b. An alternative to reorganization is to allocate leaf pages in units of n blocks, for some reasonably large n. When the first leaf of a B$^+$-tree is allocated, only one block of an n-block unit is used, and the remaining pages are free. If a page splits, and its n-block unit has a free page, that space is used for the new page. If the n-block unit is full, another n-block unit is allocated, and the first $n/2$ leaf pages are placed in one n-block unit, and the remaining in the second n-block unit. For simplicity, assume that there are no delete operations.

   i. What is the worst case occupancy of allocated space, assuming no delete operations, after the first n-block unit is full.

   ii. Is it possible that leaf nodes allocated to an n-node block unit are not consecutive, that is, is it possible that two leaf nodes are allocated to one n-node block, but another leaf node in between the two is allocated to a different n-node block?

   iii. Under the reasonable assumption that buffer space is sufficient to store a n-page block, how many seeks would be required for a leaf-level scan of the B$^+$-tree, in the worst case? Compare this number with the worst case if leaf pages are allocated a block at a time.

   iv. The technique of redistributing values to siblings to improve space utilization is likely to be more efficient when used with the above allocation scheme for leaf blocks. Explain why.

Exercises

11.15 When is it preferable to use a dense index rather than a sparse index? Explain your answer.

11.16 What is the difference between a clustering index and a secondary index?

11.17 For each B$^+$-tree of Practice Exercise 11.3, show the steps involved in the following queries:

a. Find records with a search-key value of 11.

b. Find records with a search-key value between 7 and 17, inclusive.

11.18 The solution presented in Section 11.3.4 to deal with nonunique search keys added an extra attribute to the search key. What effect could this change have on the height of the B$^+$-tree?

11.19 Explain the distinction between closed and open hashing. Discuss the relative merits of each technique in database applications.

11.20 What are the causes of bucket overflow in a hash file organization? What can be done to reduce the occurrence of bucket overflows?

11.21 Why is a hash structure not the best choice for a search key on which range queries are likely?

11.22 Suppose there is a relation $r(A, B, C)$, with a B$^+$-tree index with search key (A, B).

a. What is the worst-case cost of finding records satisfying $10 < A < 50$ using this index, in terms of the number of records retrieved n_1 and the height h of the tree?

b. What is the worst-case cost of finding records satisfying $10 < A < 50 \wedge 5 < B < 10$ using this index, in terms of the number of records n_2 that satisfy this selection, as well as n_1 and h defined above?

c. Under what conditions on n_1 and n_2 would the index be an efficient way of finding records satisfying $10 < A < 50 \wedge 5 < B < 10$?

11.23 Suppose you have to create a B$^+$-tree index on a large number of names, where the maximum size of a name may be quite large (say 40 characters) and the average name is itself large (say 10 characters). Explain how prefix compression can be used to maximize the average fanout of nonleaf nodes.

11.24 Suppose a relation is stored in a B$^+$-tree file organization. Suppose secondary indices stored record identifiers that are pointers to records on disk.

a. What would be the effect on the secondary indices if a node split happens in the file organization?

b. What would be the cost of updating all affected records in a secondary index?

c. How does using the search key of the file organization as a logical record identifier solve this problem?

d. What is the extra cost due to the use of such logical record identifiers?

11.25 Show how to compute existence bitmaps from other bitmaps. Make sure that your technique works even in the presence of null values, by using a bitmap for the value *null*.

11.26 How does data encryption affect index schemes? In particular, how might it affect schemes that attempt to store data in sorted order?

11.27 Our description of static hashing assumes that a large contiguous stretch of disk blocks can be allocated to a static hash table. Suppose you can allocate only C contiguous blocks. Suggest how to implement the hash table, if it can be much larger than C blocks. Access to a block should still be efficient.

Bibliographical Notes

Discussions of the basic data structures in indexing and hashing can be found in Cormen et al. [1990]. B-tree indices were first introduced in Bayer [1972] and Bayer and McCreight [1972]. B^+-trees are discussed in Comer [1979], Bayer and Unterauer [1977], and Knuth [1973]. The bibliographical notes in Chapter 15 provide references to research on allowing concurrent accesses and updates on B^+-trees. Gray and Reuter [1993] provide a good description of issues in the implementation of B^+-trees.

Several alternative tree and treelike search structures have been proposed. **Tries** are trees whose structure is based on the "digits" of keys (for example, a dictionary thumb index, which has one entry for each letter). Such trees may not be balanced in the sense that B^+-trees are. Tries are discussed by Ramesh et al. [1989], Orenstein [1982], Litwin [1981], and Fredkin [1960]. Related work includes the digital B-trees of Lomet [1981].

Knuth [1973] analyzes a large number of different hashing techniques. Several dynamic hashing schemes exist. Extendable hashing was introduced by Fagin et al. [1979]. Linear hashing was introduced by Litwin [1978] and Litwin [1980]. A performance comparison with extendable hashing is given by Rathi et al. [1990]. An alternative given by Ramakrishna and Larson [1989] allows retrieval in a single disk access at the price of a high overhead for a small fraction of database modifications. Partitioned hashing is an extension of hashing to multiple attributes, and is covered in Rivest [1976], Burkhard [1976], and Burkhard [1979].

Vitter [2001] provides an extensive survey of external-memory data structures and algorithms.

Bitmap indices, and variants called **bit-sliced indices** and **projection indices**, are described in O'Neil and Quass [1997]. They were first introduced in the IBM Model 204 file manager on the AS 400 platform. They provide very large speedups on certain types of queries, and are today implemented on most database systems. Research on bitmap indices includes Wu and Buchmann [1998], Chan and Ioannidis [1998], Chan and Ioannidis [1999], and Johnson [1999].