CHAPTER 11

Indexing and Hashing

This chapter covers indexing techniques ranging from the most basic one to highly specialized ones. Due to the extensive use of indices in database systems, this chapter constitutes an important part of a database course.

A class that has already had a course on data-structures would likely be familiar with hashing and perhaps even B^+-trees. However, this chapter is necessary reading even for those students since data structures courses typically cover indexing in main memory. Although the concepts carry over to database access methods, the details (e.g., block-sized nodes), will be new to such students.

The sections on B-trees (Sections 11.4.5) and bitmap indexing (Section 11.9) may be omitted if desired.

Exercises

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

Answer: It is preferable to use a dense index instead of a sparse index when the file is not sorted on the indexed field (such as when the index is a secondary index) or when the index file is small compared to the size of memory.

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

Answer: The clustering index is on the field which specifies the sequential order of the file. There can be only one clustering index while there can be many secondary indices.

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.

Answer: **With the structure provided by the solution to Practice Exercise 11.3a:**

a. Find records with a value of 11
   i. Search the first level index; follow the first pointer.
   ii. Search next level; follow the third pointer.
   iii. Search leaf node; follow first pointer to records with key value 11.

b. Find records with value between 7 and 17 (inclusive)
   i. Search top index; follow first pointer.
   ii. Search next level; follow second pointer.
   iii. Search third level; follow second pointer to records with key value 7, and after accessing them, return to leaf node.
   iv. Follow fourth pointer to next leaf block in the chain.
   v. Follow first pointer to records with key value 11, then return.
   vi. Follow second pointer to records with with key value 17.

With the structure provided by the solution to Practice Exercise 12.3b:

a. Find records with a value of 11
   i. Search top level; follow second pointer.
   ii. Search next level; follow second pointer to records with key value 11.

b. Find records with value between 7 and 17 (inclusive)
   i. Search top level; follow second pointer.
   ii. Search next level; follow first pointer to records with key value 7, then return.
   iii. Follow second pointer to records with key value 11, then return.
   iv. Follow third pointer to records with key value 17.

With the structure provided by the solution to Practice Exercise 12.3c:

a. Find records with a value of 11
   i. Search top level; follow second pointer.
   ii. Search next level; follow first pointer to records with key value 11.

b. Find records with value between 7 and 17 (inclusive)
   i. Search top level; follow first pointer.
   ii. Search next level; follow fourth pointer to records with key value 7, then return.
   iii. Follow eighth pointer to next leaf block in chain.
   iv. Follow first pointer to records with key value 11, then return.
   v. Follow second pointer to records with key value 17.

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?

Answer: The resultant B-tree's extended search key is unique. This results in more number of nodes. A single node (which points to mutiple records with the same key) in the original tree may correspond to multiple nodes in the result tree. Depending on how they are organized the height of the tree may increase; it might be more than that of the original tree.

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

Answer: Open hashing may place keys with the same hash function value in different buckets. Closed hashing always places such keys together in the same bucket. Thus in this case, different buckets can be of different sizes, though the implementation may be by linking together fixed size buckets using overflow chains. Deletion is difficult with open hashing as *all* the buckets may have to inspected before we can ascertain that a key value has been deleted, whereas in closed hashing only that bucket whose address is obtained by hashing the key value need be inspected. Deletions are more common in databases and hence closed hashing is more appropriate for them. For a small, static set of data lookups may be more efficient using open hashing. The symbol table of a compiler would be a good example.

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?

Answer: The causes of bucket overflow are :-

a. Our estimate of the number of records that the relation will have was too low, and hence the number of buckets allotted was not sufficient.

b. Skew in the distribution of records to buckets. This may happen either because there are many records with the same search key value, or because the the hash function chosen did not have the desirable properties of uniformity and randomness.

To reduce the occurrence of overflows, we can :-

a. Choose the hash function more carefully, and make better estimates of the relation size.

b. If the estimated size of the relation is n_r and number of records per block is f_r, allocate $(n_r/f_r)*(1+d)$ buckets instead of (n_r/f_r) buckets. Here d is a fudge factor, typically around 0.2. Some space is wasted: About 20 percent of the space in the buckets will be empty. But the benefit is that some of the skew is handled and the probability of overflow is reduced.

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

Answer: A range query cannot be answered efficiently using a hash index, we will have to read all the buckets. This is because key values in the range do not occupy consecutive locations in the buckets, they are distributed uniformly and randomly throughout all the buckets.

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 $10 < A < 50 \wedge 5 < B < 10$?

Answer:

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?
This query does not correspond to a range query on the search key as the condition on the first attribute if the search key is a comparison condition. It looks up records which have the value of A between 10 and 50. However, each record is likely to be in a different block, because of the ordering of records in the file, leading to many I/O operation. In the worst case, for each record, it needs to traverse the whole tree (cost is h), so the total cost is $n_1 * h$.

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.
This query can be answered by using an ordered index on the search key (A, B). For each value of A this is between 10 and 50, the system located records with B value between 5 and 10. However, each record could is likely to be in a different disk block. This amounts to executing the query based on the condition on A, this costs $n_1 * h$. Then these records are checked to see if the condition on B is satisfied. So, the total cost in the worst case is $n_1 * h$.

c. Under what conditions on n_1 and n_2 would the index be an efficient way of finding records satisfying $10 < A < 50 \wedge 5 < B < 10$.
n_1 records satisfy the first condition and n_2 records satisfy the second condition. When both the conditions are queried, n_1 records are output in the first stage. So, in the case where $n_1 = n_2$, no extra records are output in the furst stage. Otherwise, the records which

dont satisfy the second condition are also output with an additional cost of h each (worst case).

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.

Answer: There arises 2 problems in the given scenario. The first problem is names can be of variable length. The second problem is names can be long (maximum is 40 characters), leading to a low fanout and a correspondingly increased tree height. With variable-length search keys, different nodes can have different fanouts even if they are full. The fanout of nodes can be increased bu using a technique called prefix compression. With prefix compression, the entire search key value is not stored at internal nodes. Only a prefix of each search key which is sufficeint to distinguish between the key values in the subtrees that it seperates. The full name can be stored in the leaf nodes, this way we dont lose any information and also maximize the average fanout of internal 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?

Answer:

a. When a leaf page is split in a B^+-tree file organization, a number of records are moved to a new page. In such cases, all secondary indices that store pointers to the relocated records would have to be updated, even though the values in the records may not have changed.

b. Each leaf page may contain a fairly large number of records, and each of them may be in different locations on each secondary index. Thus, a leaf-page split may require tens or even hundreds of I/O operations to update all affected secondary indices, making it a very expensive operation.

c. One solution is to store the values of the primary-index search key attributes in secondary indices, in place of pointers to the indexed

records. Relocation of records because of leaf-page splits then does not require any update on any secondary index.

d. Locating a record using the secondary index now requires two steps: First we use the secondary index to find the primary index search-key values, and then we use the primary index to find the corresponding records. This approach reduces the cost of index update due to file reorganization, although it increases the cost of accesing data using a secondary index.

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*.

Answer: The existence bitmap for a relation can be calculated by taking the union (logical-or) of all the bitmaps on that attribute, including the bitmap for 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?

Answer: Note that indices must operate on the encrypted data or someone could gain access to the index to interpret the data. Otherwise, the index would have to be restricted so that only certain users could access it. To keep the data in sorted order, the index scheme would have to decrypt the data at each level in a tree. Note that hash systems would not be affected.

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.

Answer: A separate list/table as shown below can be created.
Starting address of first set of C blocks
C
Starting address of next set of C blocks
$2C$
and so on

Desired block address = Starting adress (from the table depending on the block number) + blocksize * (blocknumber % C)

For each set of C blocks, a single entry is added to the table. In this case, locating a block requires 2 steps: First we use the block number to find the actual block address, and then we can access the desired block.