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\textsuperscript{+}-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 12.4), grid files (Section 12.9.3) and bitmap indexing (Section 12.9.4) may be omitted if desired.

**Changes from 3\textsuperscript{rd} edition:**
The description of querying on B\textsuperscript{+}-trees has been augmented with pseudo-code. The pseudo-code for insertion on B\textsuperscript{+}-trees has been simplified. The section on index definition in SQL (Section 12.8) is new to this edition, as is the coverage of bitmap indices (Section 12.9.4).

**Exercises**

12.1 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.

12.2 Since indices speed query processing, why might they not be kept on several search keys? List as many reasons as possible.

**Answer:** Reasons for not keeping several search indices include:
a. Every index requires additional CPU time and disk I/O overhead during inserts and deletions.
b. Indices on non-primary keys might have to be changed on updates, although an index on the primary key might not (this is because updates typically do not modify the primary key attributes).
c. Each extra index requires additional storage space.
d. For queries which involve conditions on several search keys, efficiency might not be bad even if only some of the keys have indices on them. Therefore database performance is improved less by adding indices when many indices already exist.

12.3 What is the difference between a primary index and a secondary index?  
Answer: The primary index is on the field which specifies the sequential order of the file. There can be only one primary index while there can be many secondary indices.

12.4 Is it possible in general to have two primary indices on the same relation for different search keys? Explain your answer.  
Answer: In general, it is not possible to have two primary indices on the same relation for different keys because the tuples in a relation would have to be stored in different order to have same values stored together. We could accomplish this by storing the relation twice and duplicating all values, but for a centralized system, this is not efficient.

12.5 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

Answer: The following were generated by inserting values into the $B^+$-tree in ascending order. A node (other than the root) was never allowed to have fewer than $\lceil n/2 \rceil$ values/pointers.

a. 

b.
12.6 For each B⁺-tree of Exercise 12.5, 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 structure 0.a:**

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 leaf node; follow first 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 key value 17.

**With structure 0.b:**

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 structure 0.c:**

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.

12.7 For each B⁺-tree of Exercise 12.5, show the form of the tree after each of the following series of operations:

b. Insert 10.
c. Insert 8.
d. Delete 23.
e. Delete 19.

Answer:
- With structure 0.a:
  
  Insert 9:

  ![Insert 9 Diagram]

  Insert 10:

  ![Insert 10 Diagram]

  Insert 8:

  ![Insert 8 Diagram]

  Delete 23:
Delete 19:

With structure 0.b:

Insert 9:

Insert 10:

Insert 8:

Delete 23:

Delete 19:

With structure 0.c:
12.8 Consider the modified redistribution scheme for B+-trees described in page 463. What is the expected height of the tree as a function of $n$?

**Answer:** If there are $K$ search-key values and $m - 1$ siblings are involved in the redistribution, the expected height of the tree is: $\log_{\lfloor (m-1)n/m \rfloor} (K)$

12.9 Repeat Exercise 12.5 for a B-tree.

**Answer:** The algorithm for insertion into a B-tree is:

- Locate the leaf node into which the new key-pointer pair should be inserted.
- If there is space remaining in that leaf node, perform the insertion at the correct location, and the task is over. Otherwise insert the key-pointer pair conceptually into the correct location in the leaf node, and then split it along the middle. The middle key-pointer pair does not go into either of the resultant nodes of the split operation. Instead it is inserted into the parent node, along with the tree pointer to the new child. If there is no space in the parent, a similar procedure is repeated.
- The deletion algorithm is:
Locate the key value to be deleted, in the B-tree.

a. If it is found in a leaf node, delete the key-pointer pair, and the record from the file. If the leaf node contains less than ⌈n/2⌉ − 1 entries as a result of this deletion, it is either merged with its siblings, or some entries are redistributed to it. Merging would imply a deletion, whereas redistribution would imply change(s) in the parent node’s entries. The deletions may ripple up to the root of the B-tree.

b. If the key value is found in an internal node of the B-tree, replace it and its record pointer by the smallest key value in the subtree immediately to its right and the corresponding record pointer. Delete the actual record in the database file. Then delete that smallest key value-pointer pair from the subtree. This deletion may cause further rippling deletions till the root of the B-tree.

Below are the B-trees we will get after insertion of the given key values. We assume that leaf and non-leaf nodes hold the same number of search key values.

12.10 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.

12.11 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 \( (n_r/f_r) \times (1 + d) \) 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.

12.12 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:**
12.13 Show how the extendable hash structure of Exercise 12.12 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 12.12, change the third bucket to:

   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 12.12, change the last bucket to:
c. Insert 1: From the answer to 12.12, change the first bucket to:

\[
\begin{array}{c}
2 \\
7 \\
23 \\
\end{array}
\]

d. Insert 15: From the answer to 12.12, change the last bucket to:

\[
\begin{array}{c}
2 \\
7 \\
15 \\
23 \\
\end{array}
\]

12.14 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.

Answer: Let \( i \) denote the number of bits of the hash value used in the hash table. Let \( \text{BSIZE} \) denote the maximum capacity of each bucket.
delete(value $K_i$)
begin
    j = first $i$ high-order bits of $h(K_i)$;
    delete value $K_i$ 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($i_j \neq i_k$)
        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

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.

12.15 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 $i$. 

```plaintext
12.15 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 that $i$, then increase the count by 1. If the bucket if 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 $i$ 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 sufficient number of entries at the end of count array become 0.

12.16 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.

12.17 Consider a grid file in which we wish to avoid overflow buckets for performance reasons. In cases where an overflow bucket would be needed, we instead reorganize the grid file. Present an algorithm for such a reorganization.

**Answer:** Let us consider a two-dimensional grid array. When a bucket overflows, we can split the ranges corresponding to that row and column into two, in both the linear scales. Thus the linear scales will get one additional entry each, and the bucket is split into four buckets. The ranges should be split in such a way as to ensure that the four resultant buckets have nearly the same number of values.

There can be several other heuristics for deciding how to reorganize the ranges, and hence the linear scales and grid array.

12.18 Consider the account relation shown in Figure 12.25.

a. Construct a bitmap index on the attributes branch-name and balance, dividing balance values into 4 ranges: below 250, 250 to below 500, 500 to below 750, and 750 and above.

b. Consider a query that requests all accounts in Downtown with a balance of 500 or more. Outline the steps in answering the query, and show the final and intermediate bitmaps constructed to answer the query.

**Answer:** We reproduce the account relation of Figure 12.25 below.
A-217  Brighton    750
A-101  Downtown    500
A-110  Downtown    600
A-215  Mianus       700
A-102  Perryridge   400
A-201  Perryridge   900
A-218  Perryridge   700
A-222  Redwood      700
A-305  Round Hill   350

Bitmaps for branch-name

<table>
<thead>
<tr>
<th>Branch-name</th>
<th>Bitmap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brighton</td>
<td>100000000000</td>
</tr>
<tr>
<td>Downtown</td>
<td>011000000000</td>
</tr>
<tr>
<td>Mianus</td>
<td>000100000000</td>
</tr>
<tr>
<td>Perryridge</td>
<td>000001111000</td>
</tr>
<tr>
<td>Redwood</td>
<td>000000001000</td>
</tr>
<tr>
<td>Round Hill</td>
<td>000000000001</td>
</tr>
</tbody>
</table>

Bitmaps for balance

<table>
<thead>
<tr>
<th>Level</th>
<th>Bitmap</th>
</tr>
</thead>
<tbody>
<tr>
<td>L_1</td>
<td>000000000000</td>
</tr>
<tr>
<td>L_2</td>
<td>000001000110</td>
</tr>
<tr>
<td>L_3</td>
<td>011100111000</td>
</tr>
<tr>
<td>L_4</td>
<td>100001000000</td>
</tr>
</tbody>
</table>

where, level L_1 is below 250, level L_2 is from 250 to below 500, L_3 from 500 to below 750 and level L_4 is above 750.

To find all accounts in Downtown with a balance of 500 or more, we find the union of bitmaps for levels L_3 and L_4 and then intersect it with the bitmap for Downtown.

<table>
<thead>
<tr>
<th>Downtown</th>
<th>Bitmap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>011000000000</td>
</tr>
<tr>
<td>L_3</td>
<td>011100111000</td>
</tr>
<tr>
<td>L_4</td>
<td>100001000000</td>
</tr>
<tr>
<td>L_3 \cup L_4</td>
<td>111101111000</td>
</tr>
<tr>
<td>Downtown</td>
<td>011000000000</td>
</tr>
<tr>
<td>Downtown \cap (L_3 \cup L_4)</td>
<td>011000000000</td>
</tr>
</tbody>
</table>

Thus, the required tuples are A-101 and A-110.

12.19 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`.

12.20 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.