b) If the tree split alternately in d dimensions, and we specified values for m of those dimensions, what fraction of the leaves would we expect to have to search?

c) How does the performance of (b) compare with a partitioned hash table?

Exercise 14.6.8: Place the data of Fig. 14.36 in a quad tree with dimensions speed and ram. Assume the range for speed is 1.00 to 5.00, and for ram it is 500 to 3500. No leaf of the quad tree should have more than two points.

Exercise 14.6.9: Repeat Exercise 14.6.8 with the addition of a third dimension, hard-disk, that ranges from 0 to 400.

! **Exercise 14.6.10:** If we are allowed to put the central point in a quadrant of a quad tree wherever we want, can we always divide a quadrant into subquadrants with an equal number of points (or as equal as possible, if the number of points in the quadrant is not divisible by 4)? Justify your answer.

! **Exercise 14.6.11:** Suppose we have a database of 1,000,000 regions, which may overlap. Nodes (blocks) of an R-tree can hold 100 regions and pointers. The region represented by any node has 100 subregions, and the overlap among these regions is such that the total area of the 100 subregions is 150% of the area of the region. If we perform a "where-am-I" query for a given point, how many blocks do we expect to retrieve?

14.7 Bitmap Indexes

Let us now turn to a type of index that is rather different from those seen so far. We begin by imagining that records of a file have permanent numbers, $1, 2, \ldots, n$. Moreover, there is some data structure for the file that lets us find the ith record easily for any i. A ***bitmap index*** for a field F is a collection of bit-vectors of length n, one for each possible value that may appear in the field F. The vector for value v has 1 in position i if the ith record has v in field F, and it has 0 there if not.

Example 14.39: Suppose a file consists of records with two fields, F and G, of type integer and string, respectively. The current file has six records, numbered 1 through 6, with the following values in order: (30, `foo`), (30, `bar`), (40, `baz`), (50, `foo`), (40, `bar`), (30, `baz`).

A bitmap index for the first field, F, would have three bit-vectors, each of length 6. The first, for value 30, is 110001, because the first, second, and sixth records have $F = 30$. The other two, for 40 and 50, respectively, are 001010 and 000100.

A bitmap index for G would also have three bit-vectors, because there are three different strings appearing there. The three bit-vectors are:

Value	Vector
`foo`	100100
`bar`	010010
`baz`	001001

In each case, 1's indicate the records in which the corresponding string appears. □

14.7.1 Motivation for Bitmap Indexes

It might at first appear that bitmap indexes require much too much space, especially when there are many different values for a field, since the total number of bits is the product of the number of records and the number of values. For example, if the field is a key, and there are n records, then n^2 bits are used among all the bit-vectors for that field. However, compression can be used to make the number of bits closer to n, independent of the number of different values, as we shall see in Section 14.7.2.

You might also suspect that there are problems managing the bitmap indexes. For example, they depend on the number of a record remaining the same throughout time. How do we find the ith record as the file adds and deletes records? Similarly, values for a field may appear or disappear. How do we find the bitmap for a value efficiently? These and related questions are discussed in Section 14.7.4.

The compensating advantage of bitmap indexes is that they allow us to answer partial-match queries very efficiently in many situations. In a sense they offer the advantages of buckets that we discussed in Example 14.7, where we found the `Movie` tuples with specified values in several attributes without first retrieving all the records that matched in each of the attributes. An example will illustrate the point.

Example 14.40: Recall Example 14.7, where we queried the `Movie` relation with the query

```
SELECT title FROM Movie
WHERE studioName = 'Disney' AND year = 2005;
```

Suppose there are bitmap indexes on both attributes `studioName` and `year`. Then we can intersect the vectors for `year` = 2005 and `studioName` = `'Disney'`; that is, we take the bitwise AND of these vectors, which will give us a vector with a 1 in position i if and only if the ith `Movie` tuple is for a movie made by Disney in 2005.

If we can retrieve tuples of `Movie` given their numbers, then we need to read only those blocks containing one or more of these tuples, just as we did in Example 14.7. To intersect the bit vectors, we must read them into memory, which requires a disk I/O for each block occupied by one of the two vectors. As mentioned, we shall later address both matters: accessing records given their numbers in Section 14.7.4 and making sure the bit-vectors do not occupy too much space in Section 14.7.2. □

Bitmap indexes can also help answer range queries. We shall consider an example next that both illustrates their use for range queries and shows in detail with short bit-vectors how the bitwise AND and OR of bit-vectors can be used to discover the answer to a query without looking at any records but the ones we want.

Example 14.41: Consider the gold-jewelry data first introduced in Example 14.27. Suppose that the twelve points of that example are records numbered from 1 to 12 as follows:

1:	(25, 60)	2:	(45, 60)	3:	(50, 75)	4:	(50, 100)
5:	(50, 120)	6:	(70, 110)	7:	(85, 140)	8:	(30, 260)
9:	(25, 400)	10:	(45, 350)	11:	(50, 275)	12:	(60, 260)

For the first component, age, there are seven different values, so the bitmap index for age consists of the following seven vectors:

25:	100000001000	30:	000000010000	45:	010000000100
50:	001110000010	60:	000000000001	70:	000001000000
85:	000000100000				

For the salary component, there are ten different values, so the salary bitmap index has the following ten bit-vectors:

60:	110000000000	75:	001000000000	100:	000100000000
110:	000001000000	120:	000010000000	140:	000000100000
260:	000000010001	275:	000000000010	350:	000000000100
400:	000000001000				

Suppose we want to find the jewelry buyers with an age in the range 45–55 and a salary in the range 100–200. We first find the bit-vectors for the age values in this range; in this example there are only two: 010000000100 and 001110000010, for 45 and 50, respectively. If we take their bitwise OR, we have a new bit-vector with 1 in position i if and only if the ith record has an age in the desired range. This bit-vector is 011110000110.

Next, we find the bit-vectors for the salaries between 100 and 200 thousand. There are four, corresponding to salaries 100, 110, 120, and 140; their bitwise OR is 000111100000.

The last step is to take the bitwise AND of the two bit-vectors we calculated by OR. That is:

$$011110000110 \text{ AND } 000111100000 = 000110000000$$

We thus find that only the fourth and fifth records, which are (50, 100) and (50, 120), are in the desired range. □

Binary Numbers Won't Serve as a Run-Length Encoding

Suppose we represented a run of i 0's followed by a 1 with the integer i in binary. Then the bit-vector 000101 consists of two runs, of lengths 3 and 1, respectively. The binary representations of these integers are 11 and 1, so the run-length encoding of 000101 is 111. However, a similar calculation shows that the bit-vector 010001 is also encoded by 111; bit-vector 010101 is a third vector encoded by 111. Thus, 111 cannot be decoded uniquely into one bit-vector.

14.7.2 Compressed Bitmaps

Suppose we have a bitmap index on field F of a file with n records, and there are m different values for field F that appear in the file. Then the number of bits in all the bit-vectors for this index is mn. If, say, blocks are 4096 bytes long, then we can fit 32,768 bits in one block, so the number of blocks needed is $mn/32768$. That number can be small compared to the number of blocks needed to hold the file itself, but the larger m is, the more space the bitmap index takes.

But if m is large, then 1's in a bit-vector will be very rare; precisely, the probability that any bit is 1 is $1/m$. If 1's are rare, then we have an opportunity to encode bit-vectors so that they take much less than n bits on the average. A common approach is called *run-length encoding*, where we represent *a run*, that is, a sequence of i 0's followed by a 1, by some suitable binary encoding of the integer i. We concatenate the codes for each run together, and that sequence of bits is the encoding of the entire bit-vector.

We might imagine that we could just represent integer i by expressing i as a binary number. However, that simple a scheme will not do, because it is not possible to break a sequence of codes apart to determine uniquely the lengths of the runs involved (see the box on "Binary Numbers Won't Serve as a Run-Length Encoding"). Thus, the encoding of integers i that represent a run length must be more complex than a simple binary representation.

We shall study one of many possible schemes for encoding. There are some better, more complex schemes that can improve on the amount of compression achieved here, by almost a factor of 2, but only when typical runs are very long. In our scheme, we first determine how many bits the binary representation of i has. This number j, which is approximately $\log_2 i$, is represented in "unary," by $j - 1$ 1's and a single 0. Then, we can follow with i in binary.[9]

Example 14.42: If $i = 13$, then $j = 4$; that is, we need 4 bits in the binary

[9]Actually, except for the case that $j = 1$ (i.e., $i = 0$ or $i = 1$), we can be sure that the binary representation of i begins with 1. Thus, we can save about one bit per number if we omit this 1 and use only the remaining $j - 1$ bits.

representation of i. Thus, the encoding for i begins with 1110. We follow with i in binary, or 1101. Thus, the encoding for 13 is 11101101.

The encoding for $i = 1$ is 01, and the encoding for $i = 0$ is 00. In each case, $j = 1$, so we begin with a single 0 and follow that 0 with the one bit that represents i. □

If we concatenate a sequence of integer codes, we can always recover the sequence of run lengths and therefore recover the original bit-vector. Suppose we have scanned some of the encoded bits, and we are now at the beginning of a sequence of bits that encodes some integer i. We scan forward to the first 0, to determine the value of j. That is, j equals the number of bits we must scan until we get to the first 0 (including that 0 in the count of bits). Once we know j, we look at the next j bits; i is the integer represented there in binary. Moreover, once we have scanned the bits representing i, we know where the next code for an integer begins, so we can repeat the process.

Example 14.43: Let us decode the sequence 11101101001011. Starting at the beginning, we find the first 0 at the 4th bit, so $j = 4$. The next 4 bits are 1101, so we determine that the first integer is 13. We are now left with 001011 to decode.

Since the first bit is 0, we know the next bit represents the next integer by itself; this integer is 0. Thus, we have decoded the sequence 13, 0, and we must decode the remaining sequence 1011.

We find the first 0 in the second position, whereupon we conclude that the final two bits represent the last integer, 3. Our entire sequence of run-lengths is thus 13, 0, 3. From these numbers, we can reconstruct the actual bit-vector, 0000000000000110001. □

Technically, every bit-vector so decoded will end in a 1, and any trailing 0's will not be recovered. Since we presumably know the number of records in the file, the additional 0's can be added. However, since 0 in a bit-vector indicates the corresponding record is not in the described set, we don't even have to know the total number of records, and can ignore the trailing 0's.

Example 14.44: Let us convert some of the bit-vectors from Example 14.42 to our run-length code. The vectors for the first three ages, 25, 30, and 45, are 100000001000, 000000010000, and 010000000100, respectively. The first of these has the run-length sequence $(0, 7)$. The code for 0 is 00, and the code for 7 is 110111. Thus, the bit-vector for age 25 becomes 00110111.

Similarly, the bit-vector for age 30 has only one run, with seven 0's. Thus, its code is 110111. The bit-vector for age 45 has two runs, $(1, 7)$. Since 1 has the code 01, and we determined that 7 has the code 110111, the code for the third bit-vector is 01110111. □

The compression in Example 14.44 is not great. However, we cannot see the true benefits when n, the number of records, is small. To appreciate the value of the encoding, suppose that $m = n$, i.e., each value for the field on which the bitmap index is constructed, occurs once. Notice that the code for a run of length i has about $2\log_2 i$ bits. If each bit-vector has a single 1, then it has a single run, and the length of that run cannot be longer than n. Thus, $2\log_2 n$ bits is an upper bound on the length of a bit-vector's code in this case.

Since there are n bit-vectors in the index, the total number of bits to represent the index is at most $2n\log_2 n$. In comparison, the uncompressed bit-vectors for this data would require n^2 bits.

14.7.3 Operating on Run-Length-Encoded Bit-Vectors

When we need to perform bitwise AND or OR on encoded bit-vectors, we have little choice but to decode them and operate on the original bit-vectors. However, we do not have to do the decoding all at once. The compression scheme we have described lets us decode one run at a time, and we can thus determine where the next 1 is in each operand bit-vector. If we are taking the OR, we can produce a 1 at that position of the output, and if we are taking the AND we produce a 1 if and only if both operands have their next 1 at the same position. The algorithms involved are complex, but an example may make the idea adequately clear.

Example 14.45: Consider the encoded bit-vectors we obtained in Example 14.44 for ages 25 and 30: 00110111 and 110111, respectively. We can decode their first runs easily; we find they are 0 and 7, respectively. That is, the first 1 of the bit-vector for 25 occurs in position 1, while the first 1 in the bit-vector for 30 occurs at position 8. We therefore generate 1 in position 1.

Next, we must decode the next run for age 25, since that bit-vector may produce another 1 before age 30's bit-vector produces a 1 at position 8. However, the next run for age 25 is 7, which says that this bit-vector next produces a 1 at position 9. We therefore generate six 0's and the 1 at position 8 that comes from the bit-vector for age 30. The 1 at position 9 from age 25's bit-vector is produced. Neither bit-vector produces any more 1's for the output. We conclude that the OR of these bit-vectors is 100000011. Technically, we must append 000, since uncompressed bit-vectors are of length twelve in this example. □

14.7.4 Managing Bitmap Indexes

We have described operations on bitmap indexes without addressing three important issues:

1. When we want to find the bit-vector for a given value, or the bit-vectors corresponding to values in a given range, how do we find these efficiently?

2. When we have selected a set of records that answer our query, how do we retrieve those records efficiently?

3. When the data file changes by insertion or deletion of records, how do we adjust the bitmap index on a given field?

Finding Bit-Vectors

Think of each bit-vector as a record whose key is the value corresponding to this bit-vector (although the value itself does not appear in this "record"). Then any secondary index technique will take us efficiently from values to their bit-vectors.

We also need to store the bit-vectors somewhere. It is best to think of them as variable-length records, since they will generally grow as more records are added to the data file. The techniques of Section 13.7 are useful.

Finding Records

Now let us consider the second question: once we have determined that we need record k of the data file, how do we find it? Again, techniques we have seen already may be adapted. Think of the kth record as having search-key value k (although this key does not actually appear in the record). We may then create a secondary index on the data file, whose search key is the number of the record.

Handling Modifications to the Data File

There are two aspects to the problem of reflecting data-file modifications in a bitmap index.

1. Record numbers must remain fixed once assigned.

2. Changes to the data file require the bitmap index to change as well.

The consequence of point (1) is that when we delete record i, it is easiest to "retire" its number. Its space is replaced by a "tombstone" in the data file. The bitmap index must also be changed, since the bit-vector that had a 1 in position i must have that 1 changed to 0. Note that we can find the appropriate bit-vector, since we know what value record i had before deletion.

Next consider insertion of a new record. We keep track of the next available record number and assign it to the new record. Then, for each bitmap index, we must determine the value the new record has in the corresponding field and modify the bit-vector for that value by appending a 1 at the end. Technically, all the other bit-vectors in this index get a new 0 at the end, but if we are using a compression technique such as that of Section 14.7.2, then no change to the compressed values is needed.

As a special case, the new record may have a value for the indexed field that has not been seen before. In that case, we need a new bit-vector for this value, and this bit-vector and its corresponding value need to be inserted into the secondary-index structure that is used to find a bit-vector given its corresponding value.

Lastly, consider a modification to a record i of the data file that changes the value of a field that has a bitmap index, say from value v to value w. We must find the bit-vector for v and change the 1 in position i to 0. If there is a bit-vector for value w, then we change its 0 in position i to 1. If there is not yet a bit-vector for w, then we create it as discussed in the paragraph above for the case when an insertion introduces a new value.

14.7.5 Exercises for Section 14.7

Exercise 14.7.1: For the data of Fig. 14.36, show the bitmap indexes for the attributes: (a) speed (b) ram (c) hd, both in (i) uncompressed form, and (ii) compressed form using the scheme of Section 14.7.2.

Exercise 14.7.2: Using the bitmaps of Example 14.41, find the jewelry buyers with an age in the range 20–40 and a salary in the range 0–100.

Exercise 14.7.3: Consider a file of 1,000,000 records, with a field F that has m different values.

a) As a function of m, how many bytes does the bitmap index for F have?

! b) Suppose that the records numbered from 1 to 1,000,000 are given values for the field F in a round-robin fashion, so each value appears every m records. How many bytes would be consumed by a compressed index?

!! **Exercise 14.7.4:** We suggested in Section 14.7.2 that it was possible to reduce the number of bits taken to encode number i from the $2\log_2 i$ that we used in that section until it is close to $\log_2 i$. Show how to approach that limit as closely as you like, as long as i is large. *Hint*: We used a unary encoding of the length of the binary encoding that we used for i. Can you encode the length of the code in binary?

Exercise 14.7.5: Encode, using the scheme of Section 14.7.2, the following bitmaps:

a) 0110000000100000100.

b) 10000010000001001101.

c) 0001000000000010000010000.

14.8 Summary of Chapter 14

✦ *Sequential Files*: Several simple file organizations begin by sorting the data file according to some sort key and placing an index on this file.

- *Dense and Sparse Indexes*: Dense indexes have a key-pointer pair for every record in the data file, while sparse indexes have one key-pointer pair for each block of the data file.

- *Multilevel Indexes*: It is sometimes useful to put an index on the index file itself, an index file on that, and so on. Higher levels of index must be sparse.

- *Secondary Indexes*: An index on a search key K can be created even if the data file is not sorted by K. Such an index must be dense.

- *Inverted Indexes*: The relation between documents and the words they contain is often represented by an index structure with word-pointer pairs. The pointer goes to a place in a "bucket" file where is found a list of pointers to places where that word occurs.

- *B-trees*: These structures are essentially multilevel indexes, with graceful growth capabilities. Blocks with n keys and $n + 1$ pointers are organized in a tree, with the leaves pointing to records. All nonroot blocks are between half-full and completely full at all times.

- *Hash Tables*: We can create hash tables out of blocks in secondary memory, much as we can create main-memory hash tables. A hash function maps search-key values to buckets, effectively partitioning the records of a data file into many small groups (the buckets). Buckets are represented by a block and possible overflow blocks.

- *Extensible Hashing*: This method allows the number of buckets to double whenever any bucket has too many records. It uses an array of pointers to blocks that represent the buckets. To avoid having too many blocks, several buckets can be represented by the same block.

- *Linear Hashing*: This method grows the number of buckets by 1 each time the ratio of records to buckets exceeds a threshold. Since the population of a single bucket cannot cause the table to expand, overflow blocks for buckets are needed in some situations.

- *Queries Needing Multidimensional Indexes*: The sorts of queries that need to be supported on multidimensional data include partial-match (all points with specified values in a subset of the dimensions), range queries (all points within a range in each dimension), nearest-neighbor (closest point to a given point), and where-am-I (region or regions containing a given point).

- *Executing Nearest-Neighbor Queries*: Many data structures allow nearest-neighbor queries to be executed by performing a range query around the target point, and expanding the range if there is no point in that range. We must be careful, because finding a point within a rectangular range may not rule out the possibility of a closer point outside that rectangle.

- ✦ *Grid Files*: The grid file slices the space of points in each of the dimensions. The grid lines can be spaced differently, and there can be different numbers of lines for each dimension. Grid files support range queries, partial-match queries, and nearest-neighbor queries well, as long as data is fairly uniform in distribution.

- ✦ *Partitioned Hash Tables*: A partitioned hash function constructs some bits of the bucket number from each dimension. They support partial-match queries well, and are not dependent on the data being uniformly distributed.

- ✦ *Multiple-Key Indexes*: A simple multidimensional structure has a root that is an index on one attribute, leading to a collection of indexes on a second attribute, which can lead to indexes on a third attribute, and so on. They are useful for range and nearest-neighbor queries.

- ✦ kd-*Trees*: These trees are like binary search trees, but they branch on different attributes at different levels. They support partial-match, range, and nearest-neighbor queries well. Some careful packing of tree nodes into blocks must be done to make the structure suitable for secondary-storage operations.

- ✦ *Quad Trees*: The quad tree divides a multidimensional cube into quadrants, and recursively divides the quadrants the same way if they have too many points. They support partial-match, range, and nearest-neighbor queries.

- ✦ *R-Trees*: This form of tree normally represents a collection of regions by grouping them into a hierarchy of larger regions. It helps with where-am-I queries and, if the atomic regions are actually points, will support the other types of queries studied in this chapter, as well.

- ✦ *Bitmap Indexes*: Multidimensional queries are supported by a form of index that orders the points or records and represents the positions of the records with a given value in an attribute by a bit vector. These indexes support range, nearest-neighbor, and partial-match queries.

- ✦ *Compressed Bitmaps*: In order to save space, the bitmap indexes, which tend to consist of vectors with very few 1's, are compressed by using a run-length encoding.

14.9 References for Chapter 14

The B-tree was the original idea of Bayer and McCreight [2]. Unlike the B+ tree described here, this formulation had pointers to records at the interior nodes as well as at the leaves. [8] is a survey of B-tree varieties.

Hashing as a data structure goes back to Peterson [19]. Extensible hashing was developed by [9], while linear hashing is from [15]. The book by Knuth [14] contains much information on data structures, including techniques for selecting hash functions and designing hash tables, as well as a number of ideas concerning B-tree variants. The B+ tree formulation (without key values at interior nodes) appeared in the 1973 edition of [14].

Secondary indexes and other techniques for retrieval of documents are covered by [23]. Also, [10] and [1] are surveys of index methods for text documents.

The *kd*-tree is from [4]. Modifications suitable for secondary storage appeared in [5] and [21]. Partitioned hashing and its use in partial-match retieval is from [20] and [7]. However, the design idea from Exercise 14.5.6 is from [22].

Grid files first appeared in [16] and the quad tree in [11]. The R-tree is from [13], and two extensions [24] and [3] are well known.

The bitmap index has an interesting history. There was a company called Nucleus, founded by Ted Glaser, that patented the idea and developed a DBMS in which the bitmap index was both the index structure and the data representation. The company failed in the late 1980's, but the idea has recently been incorporated into several major commercial database systems. The first published work on the subject was [17]. [18] is a recent expansion of the idea.

There are a number of surveys of multidimensional storage structures. One of the earliest is [6]. More recent surveys are found in [25] and [12]. The former also includes surveys of several other important database topics.

1. R. Baeza-Yates, "Integrating contents and structure in text retrieval," *SIGMOD Record* **25**:1 (1996), pp. 67–79.

2. R. Bayer and E. M. McCreight, "Organization and maintenance of large ordered indexes," *Acta Informatica* **1**:3 (1972), pp. 173–189.

3. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: an efficient and robust access method for points and rectangles," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1990), pp. 322–331.

4. J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Comm. ACM* **18**:9 (1975), pp. 509–517.

5. J. L. Bentley, "Multidimensional binary search trees in database applications," *IEEE Trans. on Software Engineering* **SE-5**:4 (1979), pp. 333-340.

6. J. L. Bentley and J. H. Friedman, "Data structures for range searching," *Computing Surveys* **13**:3 (1979), pp. 397–409.

7. W. A. Burkhard, "Hashing and trie algorithms for partial match retrieval," *ACM Trans. on Database Systems* **1**:2 (1976), pp. 175–187.

8. D. Comer, "The ubiquitous B-tree," *Computing Surveys* **11**:2 (1979), pp. 121–137.

9. R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible hashing — a fast access method for dynamic files," *ACM Trans. on Database Systems* **4**:3 (1979), pp. 315–344.

10. C. Faloutsos, "Access methods for text," *Computing Surveys* **17**:1 (1985), pp. 49–74.

11. R. A. Finkel and J. L. Bentley, "Quad trees, a data structure for retrieval on composite keys," *Acta Informatica* **4**:1 (1974), pp. 1–9.

12. V. Gaede and O. Gunther, "Multidimensional access methods," *Computing Surveys* **30**:2 (1998), pp. 170–231.

13. A. Guttman, "R-trees: a dynamic index structure for spatial searching," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1984), pp. 47–57.

14. D. E. Knuth, *The Art of Computer Programming, Vol. III, Sorting and Searching, Second Edition*, Addison-Wesley, Reading MA, 1998.

15. W. Litwin, "Linear hashing: a new tool for file and table addressing," *Intl. Conf. on Very Large Databases*, pp. 212–223, 1980.

16. J. Nievergelt, H. Hinterberger, and K. Sevcik, "The grid file: an adaptable, symmetric, multikey file structure," *ACM Trans. on Database Systems* **9**:1 (1984), pp. 38–71.

17. P. O'Neil, "Model 204 architecture and performance," *Proc. Second Intl. Workshop on High Performance Transaction Systems*, Springer-Verlag, Berlin, 1987.

18. P. O'Neil and D. Quass, "Improved query performance with variant indexes," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1997), pp. 38–49.

19. W. W. Peterson, "Addressing for random access storage," *IBM J. Research and Development* **1**:2 (1957), pp. 130–146.

20. R. L. Rivest, "Partial match retrieval algorithms," *SIAM J. Computing* **5**:1 (1976), pp. 19–50.

21. J. T. Robinson, "The K-D-B-tree: a search structure for large multidimensional dynamic indexes," *Proc. ACM SIGMOD Intl. Conf. on Mamagement of Data* (1981), pp. 10–18.

22. J. B. Rothnie Jr. and T. Lozano, "Attribute based file organization in a paged memory environment, *Comm. ACM* **17**:2 (1974), pp. 63–69.

23. G. Salton, *Introduction to Modern Information Retrieval*, McGraw-Hill, New York, 1983.

24. T. K. Sellis, N. Roussopoulos, and C. Faloutsos, "The R+-tree: a dynamic index for multidimensional objects," *Intl. Conf. on Very Large Databases*, pp. 507–518, 1987.

25. C. Zaniolo, S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian, and R. Zicari, *Advanced Database Systems*, Morgan-Kaufmann, San Francisco, 1997.