Exercises on file organizations part 2 with solutions

Data management

A.Y. 2018 – 2019

Maurizio Lenzerini

Problem 1

Consider the relation

PROVINCE(name, area, president, population)

with 200.000 tuples, and such that 20 tuples fits in each page. Relation PROVINCE is static (essentially no insertions, no deletions), and it will be frequently queried to compute all provinces whose area is in a given range.

1. Which is the method you would choose to represent the relation PROVINCE?
2. Explain why.
3. On the basis of the selected method, compute the average cost (in terms of page accesses) of the query

   select ***

   from *PROVINCE*

   where *area = <constant>*

   assuming that in the average the number of provinces with the same value for the attribute area is 18.

Solution 1

Range queries ---> clustering tree index

Clustering tree index + no insertion/deletion ---> ISAM

File sorted on area + ISAM index (with search key “area”) is the method for representing the relation.

We choose alternative (2) for the index. As for the cost of the search operation, we know that the number of pages of PROVINCE is B = 200.000/20 = 10.000, and 200.000/18=11112 is the number of different areas that we find in the relation. Since the index is clustering, it can be sparse, and we can decide to have one data entry in the leaves for each page in the data file. How many data entries fit in one page? If 20 tuples with 4 attributes fit in one page, we assume that 40 data entries fit in one leaf page (we do not even need a pointer to the next leaf because in general the leaves are allocated sequentially in ISAM). Therefore, we need 11.112/40=278 leaves. To compute the cost of the operation, we know that we have to count the page accesses for locating the data entry, plus 2 page accesses (in the average) for the data file (since we have 18 results in the average, and 20 records per page, in the worst case such 18 records will be distributed in 2 pages). Therefore, the cost of the operation is: ($\log_F 278$ + 2), where F is the fan-out of the tree. To estimate F, we simply count the number of index entries per page, which is again 40, because each index entry has the same form of a data entry. Therefore the cost is ($\log_{40} 278$ + 2) = 2 + 2 = 4.

Problem 2

Consider the relation WRITING(autcod, bookcod, booktitle, …), containing information on books written by authors. We know that the authors are 67.000, and each author has written 10 books in the average. Also, we know that a non clustering B$^+$-tree index using alternative 2 is available on WRITING with search key <autcod>, and such that in each leaf we have space for 100 data entries.

1. Tell what is the average cost (in terms of page accesses) of the query

   select *

   from *WRITING*

   where *autocod = <constant>*

   assuming that the query evaluation algorithm uses the B$^+$-tree index.
1. Explain the method used to answer question 1.
2. If the size of each data entry is 10 bytes, which is the size (in bytes) of the space occupied by all the leaves of the tree index?

Solution 2

Since the index is non clustering, it cannot be sparse, which means that for each value of autcod we have as many data entries in the index as the number of books written by that author (10 in the average). Therefore, we have 67.000 $\times$ 10 = 670.000 data entries, and since in each page we have room for 100 data entries, taking into account the fact that each leaf page is 67% full, we conclude that the index has 10.000 leaves.

After we reach the right leaf by traversing the tree, we still have one page access to count, because the 10 data entries (in the average) to consider might not fit in one page, and then the 10 access to the data records, because the index is non clustering. Since each page has room for 100 data entries, we conclude that it has room for 100 index entries too, and we can compute the fan-out as the average between the minimum number and the maximum number of index entries in each node: (50 + 100)/2 = 75. The cost of the operation is therefore $\log_{75}10.000 + 1 + 10 = 3 + 1 + 10 = 14$ page accesses.

Since the size of each data entry is 10 bytes, 100 slots for the data entries corresponds to 1.000 bytes. Hence, the size of the space occupied by all the leaves of the tree index is 1.000 bytes multiplied 10.000 pages, i.e., 10.000.000 bytes.

Problem 3

Consider the relation RENT(code,person,city,cost) that stores information about rents of bicycles, with the code of the bicycle, the person who rented the bicycle, the city of the rent, and the cost of the rent. The relation has 2.000.000 tuples, stored in 200.000 pages, and has 10.000 different values in the attribute cost. We assume that all fields in every record and every pointer have the same length, independently of the attribute. There is a dense, non-clustering B+-tree index on RENT with search key cost, using alternative 2. Consider the query

> ***select*** *code, person, city*
>
> ***from*** *RENT*
>
> ***where*** *cost = <constant>*

that asks for code, person, and city of all rents with a given cost, and tell how many page accesses we need for computing the answer to the query.

Solution 3

Since the relation has 2.000.000 records stored in 200.000 pages, we infer that 10 tuples fit in one pages. Every tuple has 4 fields, and therefore 40 values fit in one page, which means that each leaf page has room for 20 data entries. Since every leaf is full at 67% of occupancy, we have 13 data entries in each leaf. This means that we need 2.000.000/13 = 153.846 leaf pages for the tree index.

Now, 20 is also the maximum number of index entries in every page. It follows that the number of index entries in each node is between 10 and 20, and therefore we can assume the average (i.e., 15) as the fan out of the tree.

So, we need $\log_{15} 153.846 = 5$ page accesses to get to a leaf when we search for a given value of cost. After that, we have to analyze all the leaves with the given value of "cost". Since there are 10.000 different values of “cost”, and we have 2.000.000 records, we know that the average number of records with the same value of "cost" is 200. Since 200/13 = 15, we can conclude that we have to access other 16 leaves.
Finally, for each of the 200 data entries that we analyze, we have to access the corresponding record in the data file, and therefore we have to consider further 200 page accesses.

The total number of page accesses is:

$$\log_{15} 153.846 + 16 + 200 = 5 + 16 + 200 = 221$$

Problem 4

Consider the relation Professor(lastname, firstname, dateofbirth, cityofbirth, university, salary), which stores information about professors. The most frequent queries posed to such relations are:

1. ***select distinct*** *lastname, firstname* ***from*** *Professor* ***order by*** *lastname*
2. ***select*** *cityofbirth, university, salary*

   from *Professor*

   where *lastname = c1 and firstname = c2 and dateofbirth = c3*

We know that relation Professor has 15.000.000 tuples, the size of every memory page is 18.000 bytes, the size of each field (relation attribute, or pointer) is 30 bytes, and there are in the average 5 professors with the same combination of last name, first name, and date of birth.

Tell which method would you choose for representing the relation in secondary storage, taking into account that your goal is to execute the above two queries efficiently. Also, for each query, tell which method would you choose for executing the query, and how many page accesses would be needed for computing the answer to the query, given the chosen method.

Solution 4

We define a clustering tree-based index on <lastname, firstname, dateofbirth>, dense, without duplicates, using alternative 2, so that the index supports an index-only scan for the first query (note that if the index were sparse, we could not use an index-only scan for the query) and supports the second query efficiently.

Let us compute the number of leaves in the index. The number of pages in the leaves will be the number of tuples of the relation divided by 5 (we divide by 5 because there are no duplicates in the index) and then divided by the number of data entries in one page. The number of tuples in the relation is 15.000.000. Taking into account the size of each page and the 67% occupancy rule, we have that in every leaf we use 12.000 bytes. Since each data entry occupies 4 * 30 B = 120 bytes, we have that 12.000/120 = 100 is the maximum number of data entries in each leaf. Therefore the number of leaves is 15.000.000/(5 $\times$ 100) = 30.000. We also know that each node of the tree contains at most 150 index entries (18.000 / (4*30)), so we can assume that the fan out is (150/2 + 150)/2 = 112.

Solution 4

Query 1:

We perform the index-only scan. The cost is 30.000 page accesses. Note that, taking into account the size of the various fields of relation Professor, we know that every page contains at most 100 tuples (18.000 / (6 * 30)) of the relation Professor, and therefore the minimum number of pages needed to store the relation is 15.000.000/100 = 150.000. Therefore, the index-only scan is indeed advantageous.

Query 2:

We use the index and spend $\log_{112}$ 30.000 = 3 page accesses for getting to the leaf, and then we spend another two page accesses at most, to retrieve from the relation all the requested data for all the professors (5) in the answer. So, the total number of page accesses is 5.

Problem 5

Consider the relation APARTMENT(code,year,cost, area), and the following relevant queries (ordered from the most significant one to the least significant one):

1. Compute the information on the apartment with a given code
2. Compute the apartments in a given interval of costs
3. Compute the apartments in a given interval of areas

We have 1.000.000 apartments (10 records fit in one page), uniformly distributed over 100 areas.

1. Which structures would you choose for representing the relation? Why?
2. Which is the cost of query 3 in terms of number of page accesses, if the range contains 5 values?

Solution 5

I would represent the relation with:

- a sorted file D for the data records (sorted according to the attribute “cost”),
- a hash-based index on search key “code” using alternative 2, for supporting query N.1,
- a sparse clustering B+-tree index using alternative 2 on search key “cost” to support the range query N.2 (only clustering indexes support range queries very well),
- a dense non clustering B+-tree index using alternative 2 on search key “area” to support query N. 3.

As for query N.3, it is not clear whether the index on search key “area” might be of any help. Indeed, such index has to be non clustering (because only one clustering index per relation can be defined), and we know that in the non clustering index case the number of page accesses could be equal to the number of qualifying records. However, in this case we know that the 1.000.000 apartments are uniformly distributed over 100 areas, so that we have an average of 10.000 apartments per area. Since the query refers to a range of 5 values, we have an average of 50.000 qualifying records per query, which means an average of 50.000 data entries to analyze in the leaves. How many data entries fit in one page? Since 10 records, each one of 4 attributes, fits in one page, we can assume that 20 data entries (each one constituted by 2 value. the value of the search key, and the record id) fits in one page.

Solution 5

This means that we need 2.500 pages to store the data entries that are relevant of one execution of query N.3. Since the leaves are 67% full, the number of pages that we will access in one execution of query N.3 will be 3.800. So, the cost of query N. 3 would be:

1. the cost of searching for one area (since the data entries to store are 1.000.000, and 20 data entries fit in one page, taking into account the 67% occupation rule, we have 15 data entries per page, which means that we have 50.000 leaves, and, considering a fan-out of 15, that is the average between 20/2 and 20, the cost is $\log_{15} 50.000 = 4$), plus
2. 3.800 pages in the leaves, plus
3. 50.000 (if the naive algorithm, with no sorting of qualifying records, is used)

The total is 53.804, which is certainly advantageous with respect to scanning the relation (requiring 1.000.000/10 = 100.000 page accesses). So, in this case the index is useful.

To give an example of a case where the index is not useful, notice that if the query referred to a range of 15 values we would have 150.000 qualifying records per query, and the cost of the query would be 153.804 (if the naive algorithm, with no sorting of qualifying records, is used), which is worse than scanning the relation. To sum up, since the index can be useful in some cases, we decide to build the index, leaving the responsibility to the query engine to decide, for a particular execution of the query, if using it or not.