Weekly exercises IN3020 and IN4020
Week 5

May 29, 2022

1 Review

Questions adapted from [1] and mandatory assigment 1 2021.

1. List up the data structures for indices that we have seen and explain their respective advanteages and disadvantages, as well as their uses

   Solution:

   - B+-tree - exact and interval searches are fast
   - B-tree (default in postgres), same as B+-trees, only that inner nodes have pointers
   - hash-based index - fewer blocks than B-trees, but only efficient for exact searches no intervals
   - (GiST for complex data types)
   - multidimensional trees - for multiple attributes
   - grid files - like hash
   - bitmap indexes

2. Can you think of techniques other than an unordered overflow block that can be used to make insertion in an ordrered file efficient?

   Solution: Possible to slide into adjacent blocks

3. What are the differences among primary, secondary, and clustering indexes?

   Solution:

   - Primary index: Indexing field is key and sort value is unique

- Clustering index: data is physically sorted on disk. Any number of tuples for a search-key value
- Secondary index: No actual sorting, only index is sorted. Can be used in addition to other indexes.

4. Why can we have at most one primary or clustering index on a file, but several secondary indexes?

   Solution: We can only sort the data on one attribute

5. How does multilevel indexing improve the efficiency of searching an index file?

   Solution: We reduce the part of the index that we search thus reducing the number of blocks we must read from the disk. No need if all records of the index fits in one block.

6. What is the order n of a B-tree? What do internal and leaf nodes of a B-tree contain?

   Solution: Each node has at least $\frac{n+1}{2}$ and at most n search keys and $n+1$ pointers. All leaves are at the same level. It has inner nodes and leaf nodes where leaf nodes have only pointers to data.

7. What is the order n of a B+-tree? What do internal and leaf nodes of a B+-tree contain.

   Solution: Each node has at least $\frac{n+1}{2}$ and at most n search keys and $n+1$ pointers. Inner nodes: all pointers are to subtrees. Leaf nodes: Pointers to data and one next pointer.

8. Explain what alternative choices exist for accessing a file based on multiple search keys.

   Solution: Multidimensional indexes:

   - Tree-like indexes
   - Hash-like indexes: grid files
   - Bitmap indexes

2 PostgreSQL

1. Create a table `person` in your Postgres database with the following attributes

   - `personid` INTEGER
   - `lastname` VARCHAR

- `firstname` VARCHAR
- `gender` CHAR

2. Download this dataset `https://www.uio.no/studier/emner/matnat/ifi/IN3020/v22/weekly-exercises/person.csv` (it is a dump of the person table from the movie database) about persons and copy the data into your table:
   `=>\copy person from person.csv`
3. Turn on timing (`=>\timing on`)
4. Insert a person `(007, Bond, James, M)` into the table. How long did it take?
5. Update the lastname of person 007. How long does it take?
6. Delete the person `(007, Bond, James, M)` from the table. How long did it take?
7. How long does it take to find the person with `personid` 12345?
8. How long does it take to find the person with `firstname = 'Will'` and lastname = 'Smith'?
9. Create an index on `personid` and do exercise 7 again. What is the time now? Use: `create index pid_index on person(personid);` to create the index.
10. Create another index on firstname and last name and do exercise 8 again. Use: `create index name_index on person(firstname, lastname);` to create this index What is the time now?
11. Is there a time difference if we have one index for both first and lastname or if we have two indexes, one for firstname and one for lastname? Use `drop index name_index;` to remove the index on both first and lastname before you create the two separate indexes.
12. Insert, update and delete a person from the table. How long does this take with the two indexes?

    Solution: We don't have any sorting of the data itself that need to be updated. Indexes are easy to update when the nodes are not full.
13. Do you see some obvious benefits and/or drawbacks by using indexes?
14. To some extent, PostgreSQL lets the user choose what type of index you want to create. Try using different types of indexes and observe how this affects the query. (see `https://www.postgresqltutorial.com/postgresql-indexes/postgresql-index-types/`).

15. You can cluster (sort) the table by using `cluster person using pid_index`. Does this improve runtime? Why do you think so? In what cases do you think clustering improves performance?

    Solution: Yes, when you are fetching many rows. If you want all person with personid < 10000, you can probably see a difference between clustered and non clustered index.

3 Calculations

3.1

Suppose a block holds either up to five records, or up to 20 key-pointer pairs. As a function of n, the number of records, how many blocks do we need to hold:

1. A data file
2. A dense index
3. A sparse index

Solution:

1. Data file: $n/5$
2. A dense index: $n/20$
3. A sparse index: $n/100$

3.2

Suppose that a block can hold either up to 20 records or up to 99 keys and 100 pointers. Also assume that an average B+-tree node is 70% full; i.e., it has 69 keys and 70 pointers. We can use B+-trees as part of several different structures. For each structure described below, determine:

- The total number of blocks needed for a 100,000-record file
- The average number of disk I/Os (i.e., block reads) needed to retrieve a record given its search key

1. The data file is a sequential file, sorted on the search key, with 20 records per block. The B+-tree is a dense index

   Solution:
   100 records, average 70% = 69 keys and 70 pointers

   - $100000/70 \approx 1429$ leaf nodes. The level above this needs $1429/70 \approx 21$ nodes. The level above has only the root node. Thus in total: $100000/20 + 1429 + 21 + 1 = 6451$ blocks

- To retrieve one block, we access the block with the root node, one block in the second and in the third level before we access the block containing the record. In total 4 disk I/Os.

2. The same as (a), but the data file consists of records in no particular order, packed 20 to a block

   Solution: Same answers as (a)

3. The same as (A), but the B+-tree is a sparse index

   Solution: $100000/20 = 5000$ blocks that need an index/pointer pair. $5000/70 \approx 72$. All 72 will fit in the root node. Total $5000+72+1 = 5073$ blocks and 3 disk I/Os.

4. The data file is a sequential file, and the B+-tree is a sparse index, but each primary block of the data file has one overflow block. On average, the primary block is full, and the overflow block is half full. However, records are in no particular order within a primary block and its overflow block.

   Solution: Primary block and overflow block have on average 30 records in total. $100000/30 \approx 3334$ double blocks. Which gives $3334/70 \approx 48$ leaf nodes. Total $3334 * 2 + 48 + 1 = 6717$ blocks and 31/3 disk I/Os. (On average we need to check the overflow block 1/3 of the times).

5. Instead of the B+-tree leaves having pointers to data records, the B+-tree leaves hold the records themselves. A block can hold ten records, but on average, a leaf block is 70% full; i.e., there are 7 records per leaf block.

   Solution: 7 records per leaf node, total $100000/7 = 14286$ leaf nodes. Level above needs $14286/70 \approx 204$ blocks, level above needs $204/70 \approx 3$ blocs. We also need a root node. In total $14286 + 204 + 3 + 1 = 14494$ blocks and 4 disk I/Os.

3.3

Exercise Adapted from [1].
Consider a disk with block size $B = 512$ bytes. A block pointer is $P = 6$ bytes long, and a record pointer is $PR = 7$ bytes long. A file has $r = 30000$ EMPLOYEE records of fixed length. Each record has the following fields:

- `Name` (30 bytes),
- `Ssn` (9 bytes)
- `Department_code` (9 bytes)
- `Address` (40 bytes)

- `Phone` (10 bytes)
- `Birth_date` (8 bytes)
- `Sex` (1 byte)
- `Job_code` (4 bytes)
- `Salary` (4 bytes, real number).

An additional byte is used as a deletion marker.

1. Calculate the record size R in bytes

   Solution: Record length $R = (30+9+9+40+9+8+1+4+4)+1 = 115$ bytes.

2. Suppose the file is ordered by the key field `Ssn` and we want to construct a sparse primary index on `Ssn`. Calculate (a) the number of first-level index entries we need; (b) the number of first-level index blocks; (c) the number of level needed if we make it into a multi-level index; (d) the total number of blocks required by the multi-level index; (e) the number of block accesses needed to search for and retrieve a record from the file given its SSN value (using the primary index).

   Solution: Index record size R = SSN + P9 + 6 = 15 bytes

   (a) We can have floor(512/115) = 4 records per block. Number of first level entries = number of file blocks = ceililng(30000/4) = 7500 entries.

   (b) First, we need to find the number of indexes in a block. That is floor(512/15) = 34 index entries per block. Number of first level index entries: = ceiling(7500/34) = 221 blocks.

   (c) The number of levels: Lowest-level index entries = 221, above this level we need ceiling(221/34) = 7 nodes. In addition, we need a root node. Hence the index has 3 levels.

   (d) Total number of blocks for the index: 221 + 7 + 1 = 229 blocks

   (e) Number of block accesses: 4

3. Suppose the file is not ordered by `Ssn` and we want to construct a secondary index on `Ssn`. Repeat the previous exercises for the secondary index and compare it with the primary index.

   Solution:

   (a) Number of first-level index entries = number of file records = 30000.

(b) Number of first level index entires: = ceiling(30000/34) = 883 blocks

(c) Number of levels: second level ceiling(883/34) = 26 + one root node. Hence the index has 3 levels.

(d) Total number of blocks for the index: $883 + 26 + 1 = 918$ blocks.

(e) Number of block accesses: 4.

4. Suppose that the file is ordered by `Department_code` and we want to construct a clustering index on `Department_code` that uses block anchors (every new value of `Department_code` starts at the beginning of a new block). Assume there are 1.000 distinct values of `Department_code` and that EMPLOYEE records are evenly distributed among these values. Calculate (a) the number of first-level index entries; (b) the number of first-level index blocks; (c) the total number of blocks required by the multi-level index (d) the number of levels needed to make a multi-level index; (e) the number of block accesses needed to search for and retrieve all records in the file having a specific `Department_code` value using the clustering index (assume that multiple blocks in a cluster are either contiguous or linked by pointers).

   Solution:

   (a) Number of first-level index entries r 1 = number of distinct `Department_code` values = 1000 entries

   (b) One index entry has size `Department_code` + P = 9 + 6 = 15 bytes. Number of entries per block = 512/15 ≈ 34. Number of first-level index blocks = ceiling(1000/34) = 30.

   (c) Calculate the number of level: Number of secondary-level entries = 30. ceiling(30/34) = 1. Since the second level has only one block it is the root. Thus we have 2 levels.

   (d) The number of blocks is $30 + 1 = 31$.

   (e) The number of block accesses to search for the first block in the cluster of blocks = $1 + 2 = 3$. We have ≈ 30000/1000 = 30 records. 115 bytes/record gives 4 records per block. *cieling*(30/4) = 8 blocks. Thus we need 2 + 8 blocks to access all. (We don't need to access the first twice).

4 Online exercises

Take the 3-minute test on `https://use-the-index-luke.com/3-minute-test/postgresql`. It has 5 exercises. The solutions are explained after you try

them! And you can find may examples in the book and online.

References

[1] Ramez Elmasri and Sham Navathe. *Fundamentals of database systems*. Vol. 7. Pearson, 2014.