Inf-2700 assignment 2

In this assignment the goal is to extend a DBMS with new functionality. These are making it possible to use other search operators on integer fields than the basic equality search «=». These include these operators: '!=', '<', '>', '<=' and '>='. Further we must implement the ability to search for an integer using binary search instead of linear search. The binary search is limited to equality search on integers, like this «select * from table where id = 2;».

The given precode provides us with the DBMS on which we will implement our extensions for searching. The used language is C.

Design and implementation

The program uses a few different systems which we will take a look at here. First there is an interpreter which reads our input and executes the given commands. The data for a table is stored in seperate files which have to be loaded into memory before we can read the data. The loading is done using pages in and out of memory. Thus pages are essential for our program to work, to be able to read or write a table's record.

The page layer then, provides functions to work with pages. The page layer is being used by the schema layer to work on said pages.. The schema layer allows us to work with the schemas of the tables and database. It allows us to get information about the table's fields, like a field of type INT with a length of 4, and thus provides the table's schema.

The schema layer also provides functions to work with our database, like creating a new schema, removing a table, adding fields, creating a record or searching through the database.

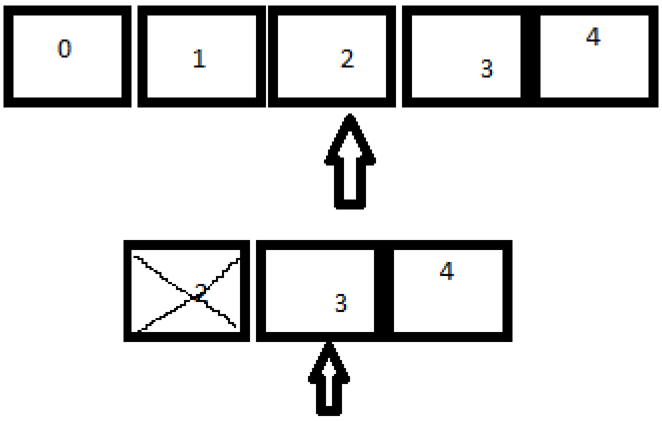
So in the schema layer is where we will implement our new functions for the DBMS. When we work with our tables we usually want to fetch some data from the database, so working with records is important. A record contains all the data for a entry in the table. For instance this is a record for an entry with fields ID and name:

ID	Name
0	Nicholas

As you can see this record contains an integer and a string. The record has a fixed size which is set on the creation of the table. The integer field ID has a length of 4 and the string field might have a set length of max 10. So the total length of this record is 14 bytes.

The length of a record is important to keep in mind when we are using the pages to fetch our data, as each page has a limit amount of space in them. Each page has the same size of 512 bytes, the first 20 bytes however are reserved by the table header, which is 20 bytes. This means that the first record in a table begins at position 20 in a page. The next record is at 20 + 14(the record length we found earlier), and so on. If a table’s records span more than one page, we have to change the page we are searching in, this is done my using the pager layer.

The design of our binary search is quite simple. We first find out how many pages the records span over and then we choose the page in the middle, inside the page itself we do a linear search to find out value. If its not in the page, we simply choose a new page depending on the values we found. We do this until we find our value, or until we try to read the same page twice(this means the value is not in the database). The binary search needs an integer field which is sorted, so our database has a field called ID, ranging from lowest to highest value.



The figure above shows the binary search at work when searching for our page which holds the wanted value. Here we are looking for the value 3, which is stored in the page marked with 3. First we search the middle page. We see that the values in this page are smaller than the one we are searching for. So this means that our value, if its in the database, must be in a higher page. So we choose the next page in the middle, which in this case is page 3, and here we find our wanted record.

To extend the searching of linear searches we first identify the operator and then we fetch a record and compare the found value using the given operator with the value we want to find.

Performance of linear search and binary search

The linear search has a worst case scenario where it uses the longest time of finding the wanted record when the record is stored at the last page at the last position or the value is not in the database. For instance if the wanted value is in record 500 it would have to search 500 times to find it. Its best case is when the first record is the one we wanted. The binary search has a worst case scenario where the value does not exist and the best case is where the value we want is in the middle(only 1 search).

By using the time command with the file «testSearch.cbcmd» we execute some selects on a table with 10.000 records.

The linear search used on average 0.046 seconds while the binary search on average used 0,018 seconds.

The linear search used a total of 21001 I/Os(Reads and seeks combined) with the harddrive, while the binary search only needed 628 I/Os. This is a big timesaving compared to the linear search. The binary search would be even faster if the searching inside each page also was done using binary search.

Another optimization that could be done is to find at which record the binary search becomes faster than the linear search and for all searches for records below that threshold we do with a linear search. This would then become a linear/binary search hybrid on pages.

Discussion

I think my implementation does a good job with what it's supposed to do. It executes equality searches on integers using binary searching and when it finds the correct page, a linear search through the page is done.

This search time could be improved as previously mentioned, by using a binary search inside a page as well.

Running 'make' will create a file called run_front. When running this file you can create tables, insert data, fetch data and so on. Writing «help» in the terminal window will post out a number of commands. To generate a table with records, set the variable gen_data in the function interpret. There is currently a bug which can cause the searches to become wonky when the same table has been generated multiple times. (Solved by deleting the table from the directory and generating the data again).

Discussion of B+ tree comparison

A B+ tree is a multilevel index for searching thorugh a database. It consists of root nodes, nonleafe nodes(or internal nodes) and leafnodes. The root nodes and non leaf nodes contain pointers to the leaf nodes, and the leaf nodes holds our records. This means that to find our record we need to search th entire height of the tree(From root, to leaf). The lowest leaf node has a pointer to the next leaf node and so on.The tree is sorted so that the smaller values are located on the left side, while the higher values is on the right side.

When we delete records in our linear database we will get empty spots, which takes alot of unneeded space, eventually the file has to be reoorganized and this has a big cost. The B+ tree avoids this overhead because it does not need to reorganize the file. Each node in the B+ tree may be half empty at minimum, which means there could be a lot of space wasted, but the speed gain is much bigger than the cost of space. Insertion and deletion is complicated, but needs few I/O operations. The worst case for the B+ tree is when all the nodes along the way needs to be split.

All in all a B+ tree organised file will be bigger than our current one because of half full(or half empty) nodes that leaves empty spaces, but it will be quicker than our current. It does not need to search through half of the pages all the time like the binary search. It simply searches a node, checks the value and rides the pointer to the next one. If the leaf node it arrives at does not contain what we are looking for, it means the value is not in the database. The binary search does bad when the value we are searching for is not in the database.

References

[1] Database system concepts, Abraham Silberschatz, Hentry F. Korth, S. Sudarshan.

[2] Slides by Wehai Yu