inf-2700-oblig2

Sigurd Uhre

October 2022

1 Introduction

In this assignment we will familiarise us with the large precode and get as good an understanding of it as possible. A program will be implemented that generates a larger data set (table) that we will use for the various tests. We will implement relational operators that we can test on the data set and sort it. We will implement the binary search algorithm and the data-set will also be used to compare it with linear search. We will also explain a theoretical comparison with B+-tree.

2 Technical background

We can use relational operators to test queries from our generated database (table). The queries will produce a boolean answer or value when evaluated. The six most common boolean operators will be implemented in this assignment.

Linear search is a simple searching algorithm. The method traverse the list and matches each of the elements of the table with the one (or several values) selected in the search queries. Its a popular search algorithm for finding en element from an unsorted list.

In addition to the linear search we will implement binary search to search the created table. Binary search follows, like the name suggest, a dividing technique in which the table is divided into two halves, and the target(s) is compared with the middle detail of the listing. If the list is not sorted, the search first needs to sort the list and then implement the algorithm to identify the position of the targeted value. Binary search is more optimized than linear search, but the table must be sorted for the algorithm the apply ist strength.

2.1 Task 1

2.2 Task 2 - Extending the types of queries

In this task we need to add relational operators to the base program so that the integer attributes in our table is not restricted equality search.

How to run and create table:

1. Navigate to the db2700 directory
2. Run python3 pop.py to create populate.dbcmd
3. Run "./run-front < populate.dbcmd"
4. Run "./run − front"
*5. Run this queries to get started "select * from workers"*
6. Run other operators queries in the database.

Test relational operators examples:
*1. Run "select * from workers where income <= 400"*
*2. Run "select * from workers where income >= 950"*
*3. Run "select * from workers where income != 500"*
*4. Run "select * from workers where income = 500"*

*5. Run "select * from workers where income < 400"*
*6. Run "select * from workers where income > 950"*

2.3 Task 3

The binary search algorithm is only implemented to check for the equality relational operator. To distinguish between the two searches linearly search uses a single = equality operator while binary search uses two ==. The following queries can be tested to illustrate the difference.

Linearly search:
*1. select * from workers where id = 999*

Binary search
*2. select * from workers where id == 999*

3 Implementation

3.1 Task 2

In this task we added relational operators to the base program so that the integer attributes in our table is not restricted equality search. We implemented a support fro the following operators. $<, <=, >, >= and, != .$ The logic is implemented in the schema.c file under the under the table search function. This function recognise the queries operators and calls a specific function, depending on which operators are used. E.g: $int - is - more$ takes two arguments an will return the relational operator of $x > y$.

3.2 Task 3

In task 3 the goal is to implement binary search and compare its performance with linearly search. The binary search implementation find the lower and upper limit of the n elements in a sorted list. Min (lower limit) is set as the first byte zero, and max (upper limit) is found by multiplying number of records in the table with the record length. The mid value is found based on the upper and lower value calculated, this mid value is then used to correct the true mid value by setting the mid value at an modulus for the record length.

All records are stored in blocks of 512 bytes also containing a page header that occupies 20 bytes, leaving the available storage of each block to be 492 bytes. The block number is found by dividing the calculated mid value with the free bytes calculation. When the block number is found, it is used as the second argument in the $get - page$ function. The page for the specific file block is found and loaded into memory. Now that the page is loaded, the integer value is read by the offset from the block number, with the mid value modulus the free bytes in the block.

The next stage is to compare the integer value found to the value stated in the search queried. If the value recorded does not match the specified value in the search, the while loop will do a check to see if the recorded value is greater or lesser than the queried integer. Either way, once again preforming a dividing in half the portion of data blocks. Here, depending on the recorded value is lesser or greater than the value queried, the mid will be used to set a new min or max.

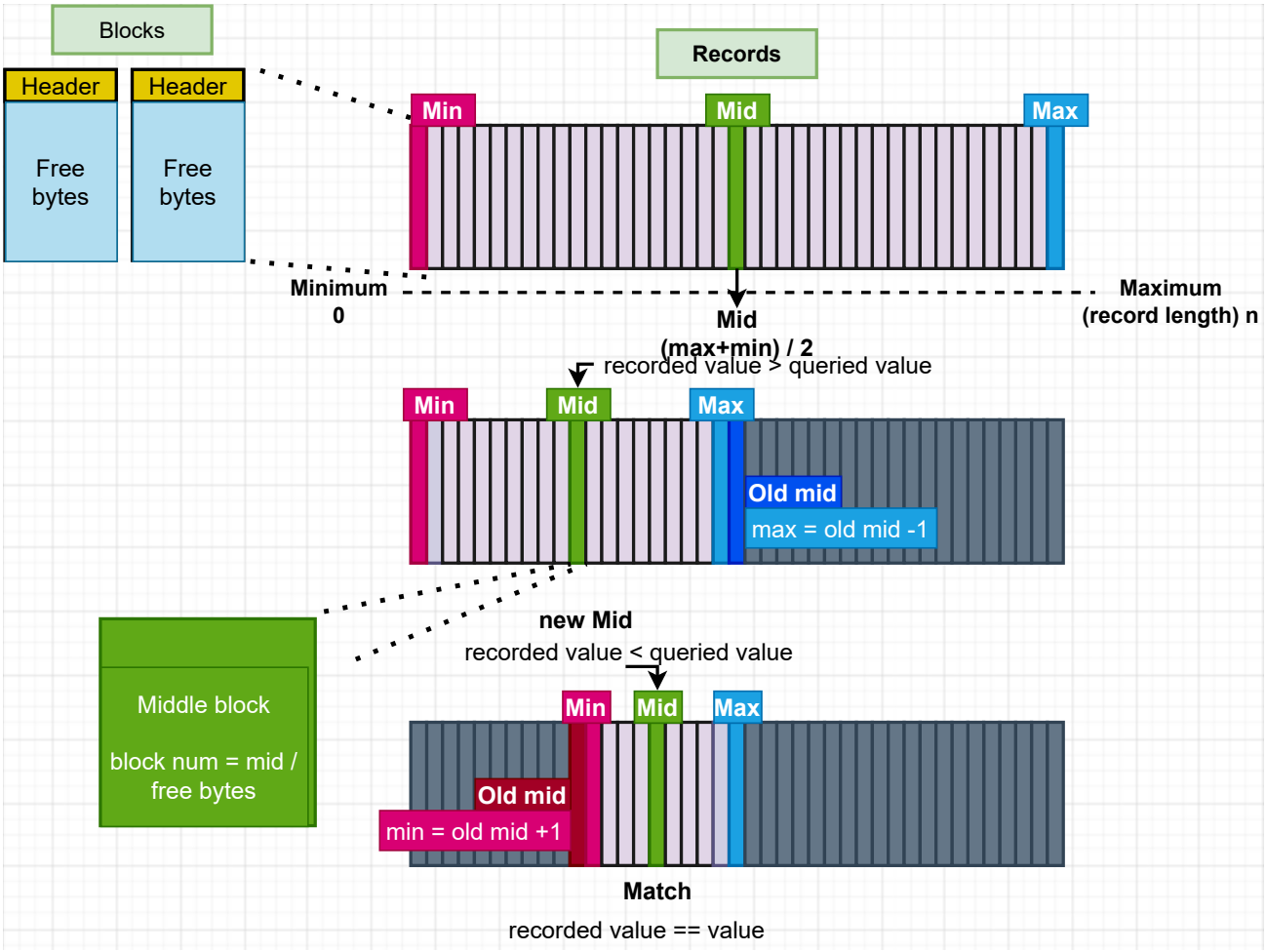

Figure 1: Binary search

If we take a look at figure 1, the recorded value is lesser than the queried value. Since this is a sorted table, the algorithm now "knows" that all values higher than mid is also higher than the queried value, they are grayed out in the next dividing. The algorithm have already tested the middle value, and therefore the next max value is set by moving one block down from the old mid ($newmax = oldmid - 1$). The same procedure is done in the third division where the queried value is grater than the recorded value. The new minimum is found by using the old mid and moving one block up ($newmin = oldmid + 1$). Once the comparison of the recorded value is equal to the value queried, there will be a match.

4 Results

4.1 Task 2

The relational operators seems to work as intended. It is recommended to test these out yourself to exercise their functionality. In the figure below, a <= operator is illustrated. This operator checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true [1]. The other operators can be tested in the same way like explained above.

```
db2700> select * from workers where income <= 360
INFO:  Number of disk seeks/reads/writes/IOs: 5/19/1/20
                 id              income          department
                 --              ------          ----------
                102                 358                   9
                259                 357                   3
                337                 357                  22
                363                 351                  19
                624                 350                   7
                683                 353                  15
                696                 353                  11
                713                 352                  10
                861                 354                   7
                909                 359                  26
                977                 354                  13
                983                 357                  13
```

Figure 2: Relational operator <=

4.2 Task 3

We compare the linear search with the binary search. The table is sorted by the workers id, and we need to search for this attribute in order to illustrate the differences between the search algorithms since the table must be sorted fro the binary search to be optimized.

```
db2700> select * from workers where id = 999
INFO:  Number of disk seeks/reads/writes/IOs: 1/25/0/25
                 id              income          department
                 --              ------          ----------
                999                 369                  10

db2700> select * from workers where id == 999
INFO:  Number of disk seeks/reads/writes/IOs: 3/2/1/3
                 id              income          department
                 --              ------          ----------
                999                 369                  10

db2700>
```

Figure 3: Worst-case linear search (top) compared to binary search.

The figure above illustrates a worst case scenario for the linear search where the method needs to traverse the complete table in order to find the last id. Here, the number of disk seeks/reads/writes and IOs have the clearest differences for this id table. The linearly search perform several more disk reads and IOs compared to the binary search.

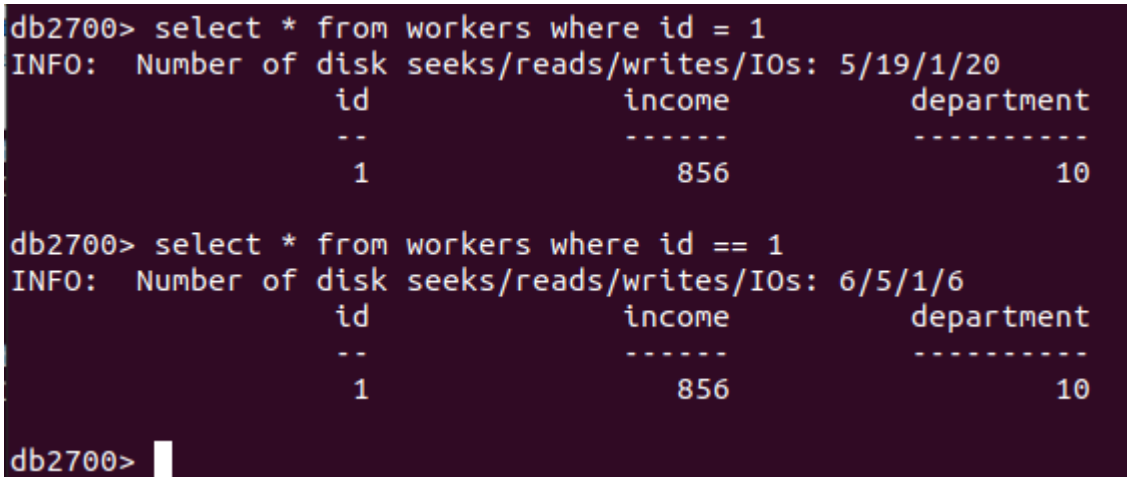

Figure 4: Best-case linear search (top) compared to binary search.

The figure above display the best case linear search and compare it to the binary search. The differences in performance between the two searches is marginally smaller, but the binary search still preform significantly less disk reads and IOs.

4.3 Task 4

A B+ -tree is mostly used for dynamic indexing on multiple levels. The B+ -tree stores data pointers only on what we call leaf nodes. B+ trees have two types of nodes, internal nodes and leaf nodes. The internal nodes only point to other nodes in the tree while leaf nodes point to data in the database using data pointers which make search more accurate and faster. They can store a large number of key values a height of 3,4 or 5 so that every lookup of the large number of keys will require 3, 4 or 5 disk accesses, which is not a lot disc accesses for possibly billions of rows.

For storage size, B+ trees store a significant number of keys in each node in addition to pointers and often having a large number of leaf nodes. A single node often fits into a cache, meaning that the key can be found within the node or determine which leaf node to read next, all this is done without going back to disk, so the access time is actually very small [2]. The B+ trees contains pointers, nodes and keys and will require more storage than a binary search tree. This storage will grow linearly larger with the n -number of records.

Both binary search trees an B+ trees have a average performance og *O(log n)*. The first and foremost advantage for B+ trees compared to binary search trees is that the B+ tree interacts well with cashes. Nodes in binary search tree will be placed in a random order in memory, every time a pointer i followed the process will pull in a new block of memory to the processor cache, which compared to accessing memory already in the cache is significantly slower [2]. For B+ trees, every lookup will result in a disk accesses, the number of disk accesses to fetch a value is proportional to the height of the tree. The shallower the tree, the less number of disk accesses [3].

Linear search from task 3 needs to iterate trough the complete sorted table with a worst-case scenario of *O(n)*, meaning that the the performance will result in a large number of disk reads and IOs increasing linearly with the number of n records. The binary search tree start with n elements in search space but will for every division (cutting n in half) reduce the search space to $n/2$ and then $n/4$ in the next division until the element is found. This method might give us better results in terms of number of memory accesses, but B+ trees will perform better in terms of run time of the memory accesses [2].

The performance comes down to if time complexity or space complexity is emphasized. B-trees work well and often provide substantial benefits when used for in-memory storage. If we emphasize time over disk accesses, B+ trees will performer better even in a "small" database of 1K records. Binary search will however

do fewer disk accesses compared to B+ tree search.

5 Discussion and conclusion

Big-O notation is a standard to show/illustrate the efficiency of an algorithm in its worst-case scenario relative to the (n)-input size. Like explained earlier, the linear search will iterate from the beginning of the table to the end regardless of how big the table is and what value it is looking for. The common running time for the linear search is O(n). If we look at figure 3 and 4, we can see that the number of disk reads and IOs is high on both scenarios for linear search. If the table is larger, the number of disk reads an IOs will increase accordingly.

Binary search have a time complexity of O(log n) notation, and will outperform linear search increasingly with larger n-tables in a sorted table. The main drawback of the binary search algorithm is that it needs a sorted list to be applied, without a sorted list the algorithm logic will not work. When used in a sorted list we can see that the performance is better compared to the linear search (figure 3 and 4). The binary search algorithm execute more disk seeks but has considerably less disk reads and IOs.

B-trees work well and often provide substantial benefits when used for in-memory storage. If we emphasize time over disk accesses, B+ trees will performer better even in a "small" database of 1K records. Binary search will however do fewer disk accesses compared to B+ tree search.

6 References

1. C - Operators [Internet]. Tutorialspoint.com. 2022 [cited 6 October 2022]. Available from: https://www.tutorialspoint.com operators.htm

2. BSTs? A, Jain P. Advantage of B+ trees over BSTs? [Internet]. Stack Overflow. 2022 [cited 7 October 2022]. Available from: https://stackoverflow.com/questions/15485220/advantage-of-b-trees-over-bsts

3. Why does "b tree" or" b+ tree" used for database index rather than binary search tree? [Internet]. Quora. 2022 [cited 7 October 2022]. Available from: https://www.quora.com/Why-does-b-tree-or-b+-tree-used-for-database-index-rather-than-binary-search-tree