INF-2700: Database system

Assignment 2

Iver Mortensen
UiT mail: imo059@uit.no

October 16, 2023

1 Introduction

This report describes the extension of a simple database management system. This includes adding support for different operand types for query search and binary search on equality queries in the database. I also covers its storage and performance compared to a B+ tree, and the performance of the binary search compared to the linear search of the base system.

2 Technical background

2.1 Binary search

Binary search is algorithm used to locate a value from sorted data, without duplicates. It reduces the search space by dividing it in half by getting the middle value and comparing it to the target value, reducing the search range to the upper or lower half. This process is done to the new search range again and again until the target is found. If the search range is less than one value the target is not in the list.

2.2 B+ Tree

A B+ tree is a self-balancing tree data structure commonly used in databases and file systems. It stores the data sorted and allows for efficient data modification and search. It contains a root node that points to other nodes. These nodes can point to a new layer of nodes, increasing its depth. The last row of nodes points to leafs, which contains the actual data. All nodes contains a key representing the data it points to, resulting in the efficient search time.

3 Implementation

3.1 Extending operand types

The precode only supported equal query search. The program was extended by supporting the operands less than, less than equal, greater than, greater than equal and not equal.

When a query is sent to the program, the query is separated into tokens. To extend the operand support, a check is done on the operand token. If one of the new operands are sent in this is now handled by using this operand in the search through the database.

To test the implementation a user function was added that creates and fills a table. This functions can be called by using "make tableName", where "tableName" will be the name of the table.

3.2 Binary search

When selecting a record from a filed with the equal operand a binary search is done, instead of a linear search. The binary search implementation is fairly straight forward. The intricate part of the implementation is the accessing of a value based on an index.

To get a value we need the block where the value is stored, and how many bytes into the block the value is. To find these values, these calculations are done:

record size = number fields * size of int
records per block = (block size - header size) / record size
block number = index / records per block
record position in block = ((index % records per block) * record size) + header size

With this information the index can be converted to a value which can be used in the binary search.

3.3 B+ Tree

The current implementation stores all values after one another in memory. Adding a new values increases the storage usage by the size of one value. This results in a O(n) space complexity.
The implementation uses binary search to find a value in the database. Binary search has a time complexity of O(log n).

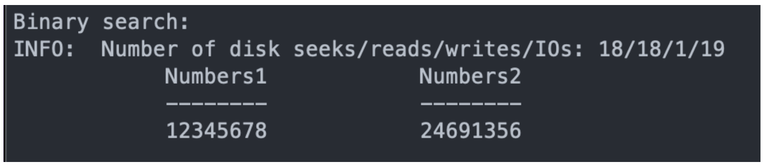
Now consider that our data is stored in a B+-tree. A B+ tree has a similar space complexity as our current implementation. In a B+ tree the data is not stored linearly, as it stores the data in leafs with pointers to each other. Adding a new value increases its storage usage by the size of one value. If the leaf it gets added to is full, a the size of the pointer to the new leaf is also added. A pointer is also added if the root or a node gets full. This increases the space usage compared to storing the data linearly. However, this does not change the space complexity, resulting in O(n).
If we look at the performance of a B+ tree we see that a search has the same time complexity of binary search at O(log n).

The current implementation using binary search and a B+ tree implementation has the same time and space complexities. The B+ tree uses slightly more storage as it has to store pointers to nodes and leafs.
The advantage of B+ tree is not shown with Big O notation. B+ trees stores data efficiently by having a high fanout [2]. This results in less I/O operations as it can read and write multiple data elements per disk access.

4 Discussion

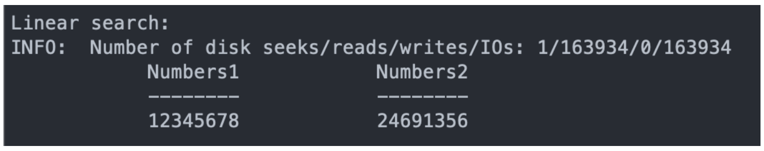
4.1 Test 1

To test the implementation of the binary search, the profiler given by the precode was used. It gives a overview of the number of disk seeks, reads, writes and IOs.
The first test was done with 2 fields with 10 million records in each field. The number being search for has the index 6,172,829, which is slightly above the midpoint. This should show the advantage of the binary search as the linear search has to traverse through a majority of the database.



```
Binary search:
INFO:  Number of disk seeks/reads/writes/IOs: 18/18/1/19
         Numbers1            Numbers2
         --------            --------
         12345678            24691356
```

Figure 1: Binary search with 10 million entries.



```
Linear search:
INFO:  Number of disk seeks/reads/writes/IOs: 1/163934/0/163934
         Numbers1            Numbers2
         --------            --------
         12345678            24691356
```

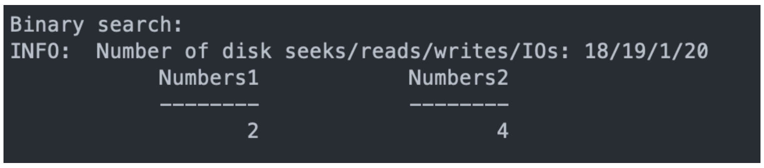
Figure 2: Linear search with 10 million entries.

From the results shown in Figure 1 and Figure 2 we can see the advantage of the binary search. The linear search has to check every value before it finds the requested value. The binary search can jump back and forward, closing in on the requested value, not having to check every value on the way. This results in the binary search having drastically less reads.

It should be noted that jumping around in memory the way binary search does results in a lot of cache misses [1]. In some database sizes, this can result in the linear search being faster, as it checks values next to each other in memory, reducing cache misses.

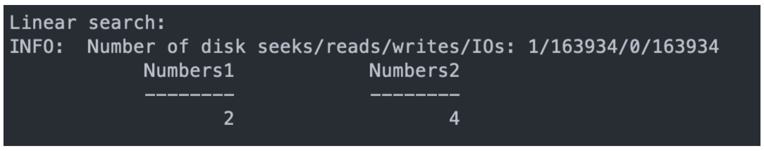
4.2 Test 2

A second way linear search can be quicker is if the requested value is early in the searched data. To give the linear search an advantage I tried accessing the first index of the database.



```
Binary search:
INFO:  Number of disk seeks/reads/writes/IOs: 18/19/1/20
         Numbers1            Numbers2
         --------            --------
                2                   4
```

Figure 3: Binary search on first value with 10 million entries.



```
Linear search:
INFO:  Number of disk seeks/reads/writes/IOs: 1/163934/0/163934
         Numbers1            Numbers2
         --------            --------
                2                   4
```

Figure 4: Linear search on first value with 10 million entries.

We can see in figure 3 that the binary search has to do extra steps to find the value at the first index, compared to last test in Figure 1. This increase in reads is because the first index is at the bottom of a full binary tree, requiring the most steps to get to.
Despite of this advantage, we can see that the linear search result in Figure 4 has the same inefficiency as the last test in Figure 2.

Both linear search tests gave the same result. I believe this is because the linear search traverses through every record of the database, regardless if it found the corresponding value. This is caused by how the precode is implemented. The precode does not assume that the database is sorted and only contains one entry of each value. This means it does not stop searching if it finds the requested value, as there could be duplicates.
The binary search only works on a sorted database with no duplicates and therefore makes these assumptions.

4.3 Test conclusion

The binary search is more efficient on a large, sorted database, without duplicates. The linear search is less efficient under these circumstances, however it is more flexible and can work on an unsorted database with duplicates.

4.4 Binary search

When converting an index to a value in memory, a few calculations are made. Some of these calculations only needs to be done once for each search, however they are done each time a index is converted. They are done each time to make the code and theory easier to understand. The code is not made to be super efficient. The combination of the compiler and caching probably negates this ”inefficient” code anyway.

5 Conclusion

In conclusion, the implementation was successful in achieving the extended operands for query search and a binary search for equality queries. The efficiency of search was notably improved by the binary search, reducing the read/writes to disk. For a more efficient and dynamic database storage, a B+ tree data structure could be implemented.

References

[1] Binary search is a pathological case for caches (July 30th, 2012). Retrieved 12:00, Okt 13th, 2023, from `https://pvk.ca/Blog/2012/07/30/binary-search-is-a-pathological-case-for-caches/`

[2] B+ tree (15 October 2023, 21:14 (UTC)). Retrieved 10:00, Okt 16th, 2023, from `https://en.wikipedia.org/wiki/B%2B_tree`