INF-2700: Database Systems

Assignment 2

Jørgen Pettersen

UiT id: jpe130@uit.no

Git user: JorgenWP

October 17, 2023

1 Introduction

In this assignment we have been tasked to implement some basic elements of a database management system (DBMS). This firstly consists of extending the base program such that queries on integer attributes are not restricted to equality search. Secondly, the base program only support linear search in order to find record. Our task is to extend the DBMS to also support binary search on an integer field. A sub task also requires implementing a method for generating a table of an arbitrary size, in order to support testing of the program.

2 Technical Background

2.1 Linear search

The linear search algorithm, also known as sequential search, is one of the simplest and most intuitive searching techniques in computer science. It is used to locate a specific item within a list or array by inspecting each element one by one until a match is found or the entire list has been searched. The time complexity of the linear search algorithm is O(n), where 'n' represents the number of elements in the list. This means that in the worst-case scenario, the algorithm may need to examine every element in the list before finding a match [1].

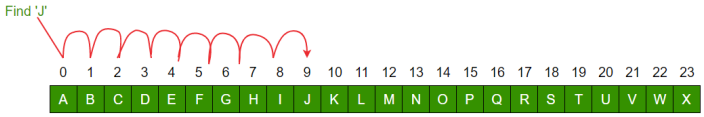
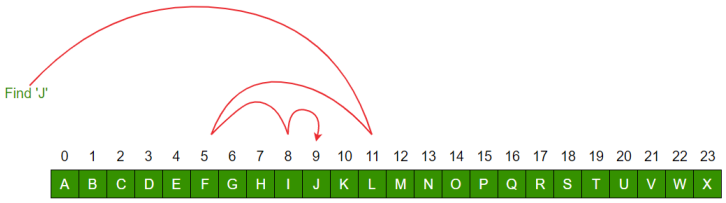


Figure 1: Linear search algorithm [3]

2.2 Binary search

The binary search algorithm is a significantly more efficient search technique used to locate a specific item within a sorted list or array. Unlike the linear search, binary search leverages the property of sorted data to reduce the search space exponentially. The binary search algorithm has a time complexity of O(log n), where 'n' is the number of elements in the list [2].

The algorithm start with a sorted list and two defined pointers, one at the beginning (low) and one at the end (high) of the list. The middle index is then calculated as (low + high) / 2, before comparing the item at this index with the target item. If they match, the search is successful, and the algorithm returns the index of the matching item. If the target item is smaller than the middle item, discard the upper half of the list and set 'high' to 'mid - 1'. If the target item is larger, discard the lower half of the list and set 'low' to 'mid + 1'. This is then repeated by using the new middle index until a match is found or 'low' exceeds 'high'.



[3]

Figure 2: Binary search algorithm [3]

3 Design and Implementation

The program consists of multiple files implementing the database management system. In "schema.c" we find the code for searching a table in the database. This is where the implementation of the two tasks will be done. Furthermore, the "interpeter.c" implements the frontend of the program. Here is where the functionality for generating a table of an arbitrary size will be located.

3.1 Task 2 - Extending the types of queries

Task 2 of the assignment requires adding support for multiple type of queries. In the given base program queries are restricted to equality search on integer attributes. This means that the program needs support for searches using the following symbols <, <=, >, >=, !=. This problem was solved simply by adding a new function where the operand of the query is checked in order to assign the respective compare function. The function uses a string compare function to find out which operand was typed in by the user and returns the corresponding compare function. These compare functions was defined in the file with the purpose of returning true or false depending on whether or not the condition (set by the operand) is true.

3.2 Task 3 - Binary search

The next task requires extending the base program to support binary search on an integer field. The search should also be restricted to equality queries and assume that there is no duplicate of rows with the searched attribute. To do this, we need to get the basics of a binary search. Therefore, we first find the upper and lower limit of the table getting searched. The lower limit (low) is set as the first byte, while the end of the list (high) is calculated by multiplying the number of records in the table with the size of a single record. This is followed by calculating the block size, which represents the size of a block in the database minus the page header size. This defines the actual space available for storing records within a block. Then the amount of free bytes is calculated by finding the remainder of the block size when divided by the length of a record. This represents the free space within the block not used by complete records.

The binary search itself is executed using a loop that runs until a match is found or "low" exceeds "high". It starts off by calculating the middle value based on the upper and lower limit previously found. Modulo is then used to ensure that the middle is aligned with record boundaries, making sure that its pointing to the beginning of a record. Further, Within the loop, it calculates the current block number and offset where the record is located in the page based on the middle offset. Then the page containing the block is loaded into memory using this block number. The position of the record is then calculated by adding the page header size to the offset of the record in the page. This makes it possible to retrieve the integer value of the record.

Having retrieved a value, the next step of the algorithm is to compare it to the desired value. If a match is found it sets the current position of the page to the offset where the value was found and retrieves the record into the target record parameter. If the comparison indicates that the fetched value is smaller than the target value, it adjusts the range by updating the minimum offset and recalculating the middle accordingly. And otherwise, if the fetched value is greater than the target value, the maximum limit is updated before recalculation the middle value.

4 Experiments and Results

The database management system has been tested by utilizing the profiling capability provided by the base program. This has been done by running the program and manually running query's to a table. The profiling results was then displayed when doing query's. In order to test the program on relatively large tables, support for generating a table of an arbitrary size was added. This was implemented such that the records in the table are ordered on a given integer field, without any duplicate. Testing of the program therefore consisted of doing queries on this field.

4.1 Task 2

The task of extending the base program such that queries on integer attributes are not restricted to equality search, seems to work as intended. There was added support for all the following operands <, <=, >, >=, !=. Testing showed some flaws when using the greater or less than operators where the output sometimes would not include all rows. However, this is likely because of some error with the precode, since running the query multiple times would result in the correct output.

4.2 Task 3

The task of adding support for binary search on integer values was also successfully implemented. The searches had the desired outcome and in a much more cost effective way than the linear search. To test the difference in performance between the linear and binary search, a large sorted table containing 1 000 000 records, each with 3 fields, was utilized. It was also noticed that the linear search algorithm considered duplicates in the table, rather than only searching for unique values. Therefore, the tests as shown in figure 3 and 4 below, was conducted when the linear algorithm was modified to only look for unique identifiers.

The linear search algorithm can have a very good best case, O(1), if the value is present in the first index. However, the worst and average case is be very bad, O(n). As we see in figure 3 the further down the list the element is located, the longer time and more disk seeks, reads and IOs is needed. The seeks here are so low that it does not show up in the diagram.

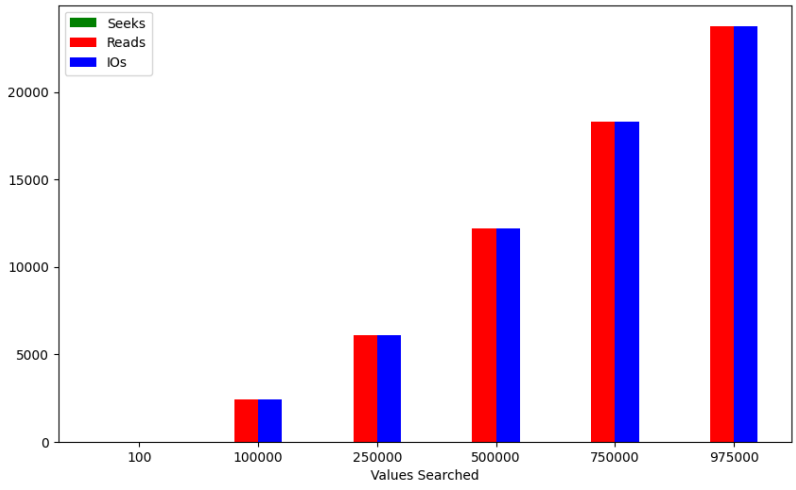


Figure 3: Linear search profiling

When it comes to the binary search, the profiling shows us a much more even result for the different values. As seen in figure 4, the number of seeks, reads and IOs does not exceed more than 17 for a table of this size. This is because of the superior time complexity for binary searches, which has a average and worst case of only O(log n) [2]. This makes binary search a lot faster than linear search, especially for large tables.

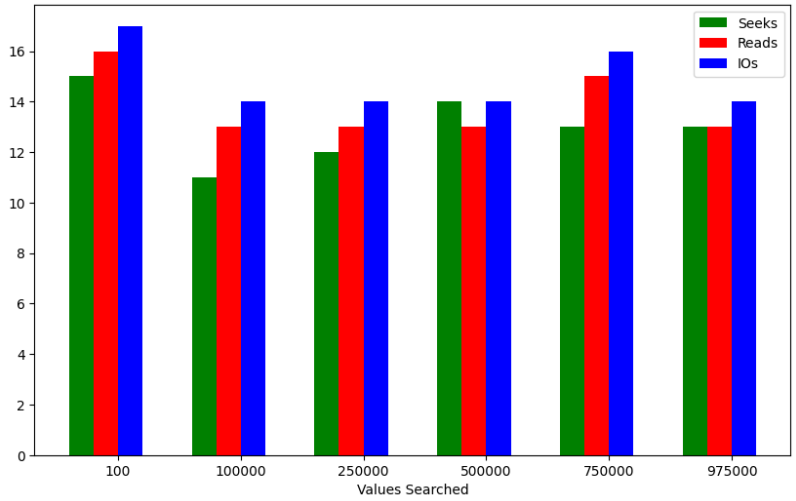


Figure 4: Binary search profiling

Binary search does however also has some drawbacks. One of these are that they require sorted data. Linear search on the other hand can be used irrespective of whether the data is sorted or not. Secondly, binary search also requires that the data structure being searched be stored in contiguous memory locations and that the data must be able to be ordered. The binary search is regardless a much better option when it comes to searching databases where its time complexity makes searching large databases efficient. This is where the linear search comes short.

5 Discussion

5.1 Task 4 - Comparison with B^+-tree

When comparing the storage size and query performance between the binary search solution and a B^+-tree organized file for a database, several factors come into play. In the case of binary search, the storage size primarily depends on the data layout and the number of records in the database. Binary search involves a sequential storage of records, without the need for additional data structures for indexing. The total storage size is given by number of records times the record length. On the other side, B^+-tree organized files require additional storage space to maintain the index structure. The index structure consists of keys and pointers to data records. The size of the index is determined by the number of entries (keys and pointers) and the structure of the B^+-tree. They are also balanced, and the height of the tree is relatively small, resulting in a smaller index size compared to the data size. This also leads to the B^+-tree requiring less disk accesses. In addition, data records are stored separately and may require additional space for node management and metadata.

When it comes to query performance, binary search is good at quickly locating a specific item within a dataset. It offers a time complexity of O(log n), making it efficient for large databases, especially when data is stored sequentially on disk. Binary search is primarily designed for equality searches, which means it's good for finding a single item or determining if a value exists in the dataset. However, it's not well-suited for more complex queries, such as range searches or insertion and deletion operations. B^+-tree organized files are designed for efficient data retrieval across a wide range of queries. They do also provide excellent performance for equality searches but do also perform for more complex operations. As said B^+-trees maintain a balanced tree structure, ensuring that the height of the tree remains relatively small. This property results in consistent performance regardless of the dataset size, making them suitable for large databases.

In the end the choice between binary search and B^+-tree organized files depends on the specific requirements of the database and the trade-off between storage size and query performance. Binary search on one hand is good for situations where the dataset is relatively small and sorted. It also minimizes storage overhead and offers excellent performance for equality searches. On the other side B^+-tree organized files are ideal for applications that demand efficient data retrieval across a variety of query types. They perform well in scenarios where the datasets is relatively big and require consistent performance.

6 Conclusion

In the end, the implementation for adding support for extended types of queries in addition to adding a binary search algorithm, was completed successfully. The database management system (DBMS) has been given new functionality that fulfills the requirements of the assignment.

References

[1] GeeksforGeeks. (2023, June 2). Linear Search. In GeeksforGeeks. Retrieved October 16, 2023, from https://www.geeksforgeeks.org/linear-search

[2] GeeksforGeeks. (2023, July 26). Binary Search. In GeeksforGeeks. Retrieved October 163, 2023, from https://www.geeksforgeeks.org/binary-search

[3] GeeksforGeeks. (2023, August 16). Linear Search vs Binary Search. In GeeksforGeeks. Retrieved October 16, 2023, from https://www.geeksforgeeks.org/linear-search-vs-binary-search

[4] GeeksforGeeks. (2023, September 6). Introduction of B^+ Tree. In GeeksforGeeks. October 17, 2023, from https://www.geeksforgeeks.org/introduction-of-b-tree