_table_table _table_table _table_table

INF-2700: Database systems

Assignment 2

Magnus Lyngra

UiT id: mly004@uit.no

Git user: OpusMag

Exam id: i

October 18, 2023

1 Introduction

In this assignment we were tasked with implementing some basic elements of a database management system using a substantial amount of precode as basis for implementing some additional features. These features including extending the linear search so more than the equals operator could be used as well as creating a binary search as an alternative to the linear search. We were then tasked with comparing the two searches and comparing performance through testing and profiling.

2 Design and Implementation

[1, 2, 3] The implementation begins with extending the operators of the linear search. This is done simply by copying the code for the equals operator and making changes to the logic to suit each operator i.e. != for not equal ¡ for lesser than etc.

Then, in order to test both the linear search and the binary search, a table of sufficient size needed to be created in order to test the search implementations. Therefore, the command ”test” was added to create a table for testing in the file interpreter.c. That file is responsible for handling other database commands which allows the user to manually create a table. this new command generates a table with a field and with a for loop it creates records containing integer values from 0-999 in order. By changing the amount of times the for loop loops the amount of records in the table can be manipulated which will be useful when profiling and viewing the linear search against the binary search.

The implementations of the binary search will be explained below. It works as intended, however it causes a segmentation fault when attempting to return the table displaying the results at the end. Still, as you can see with the print statement that displays the table header, the record info is appended to it and can be displayed, so the implementation works as intended, it's just the final return that causes a segmentation fault for some unknown reason.

2.1 Binary search implementation

The second implementation is more convoluted but should be closer to the intended way to implement the binary search as described by the TA. This implementation is organized into several functions in contrast to the previous implementation which was done entirely in the binary_search function. These functions are the get_page, return_page, binary_get_record_int_val and binary_search functions.

The return_page function returns the current page in a table by taking in the name of the table and the record number. It does this by calculating the amount of records per block by using the global variables BLOCK_SIZE and PAGE_HEADER_SIZE and dividing them by the table schema length. Then calculates the record offset by adding the PAGE_HEADER_SIZE to the product of the record number modulo the amount of records per block times the table schema length. It then gets the page number by dividing the record number by the the amount of records per block. Then it set the current page to the value of the return of the get_page function which takes in the table name and the page number that was just calculated and then sets the current page position by calling in the correct function and taking in the calculated current page and record offset and then returns current page in the table.

The get_record_in_page function takes in a record object, a schema object, an offset integer, a value integer (representing the value that is being searched for), a beginning index integer, and an ending index integer. This is a recursive function that parses a page by performs a binary search on the schema to find a record that matches the given value. It first calculates the middle index of the schema using the beg(inning) and end indices and retrieves the page at the middle index using the return_page function. It then retrieves the integer value of the record at the given offset using the page_get_int_at function.

If the value matches the retrieved record value, the function sets the current position of the page to the beginning using the page_set_current_pos function and retrieves the record using the get_page_record function. The function then returns the record and prints a string indicating that the record was found. If the value is less than the retrieved record value, the function recursively calls itself with the mid index decremented by 1 as the new end index. If the value is greater than the retrieved record value, the function recursively calls itself with the mid index incremented by 1 as the new beg index. If the value does not match any record values, it prints a statement that the record is not found and returns 0.
The get_record_int_in_page function does the same as the get_record_in_page function, except it returns the int value of the record instead of the record.

The binary_search function is mostly code copied from the table_search function. However instead of the while loop there's an if statement that checks if get_record_int_in_page returns something. If it does, get_record_in_page is called and the returned record is appended to the table and the result is displayed,

3 Experiments and Results

[4, 5] In order to measure the two implementations against each other, valgrind was utilized as suggested. Since simply measuring the time each search takes was not valid, performance was measured using the percentage of cache misses and hits using valgrind with the following command: "valgrind –tool=callgrind –dump-instr=yes –simulate-cache=yes –collect-jumps=yes –collect-atstart=no –instr-atstart=no" and linear search, the following results could be observed:

	I refs	**I1 misses**	**LLi misses**
Collected	2,942,252	215	171
	I1 miss rate	**LLi miss rate**	
	0.01%	0.01%	
	D refs	**D1 misses**	**LLd misses**
	1,789,234 (1,180,661 rd + 608,573 wr)	23 (3 rd + 20 wr)	23 (3 rd + 20 wr)
	D1 miss rate		
	0.0% (0.0% + 0.0%)		
	LL refs	**LL misses**	**LL miss rate**
	238 (218 rd + 20 wr)	194 (174 rd + 20 wr)	0.0% (0.0% + 0.0%)

Table 1: Table containing 1000 records

	I refs	**I1 misses**	**LLi misses**
Collected	203,895	208	171
	I1 miss rate	**LLi miss rate**	
	0.10%	0.08%	
	D refs	**D1 misses**	**LLd misses**
	122,412 (80,236 rd + 42,176 wr)	23 (3 rd + 20 wr)	23 (3 rd + 20 wr)
	D1 miss rate		
	0.0% (0.0% + 0.0%)		
	LL refs	**LL misses**	**LL miss rate**
	231 (211 rd + 20 wr)	194 (174 rd + 20 wr)	0.1% (0.1% + 0.0%)

Table 2: Table containing 10000 records

	I refs	I1 misses	LLi misses
Collected	321,824	218	127
	I1 miss rate	**LLi miss rate**	
	0.07%	0.04%	
	D refs	**D1 misses**	**LLd misses**
	181,450 (117,229 rd + 64,221 wr)	91 (10 rd + 81 wr)	85 (4 rd + 81 wr)
	D1 miss rate		
	0.1% (0.0% + 0.1%)		
	LL refs	**LL misses**	**LL miss rate**
	309 (228 rd + 81 wr)	212 (131 rd + 81 wr)	0.0% (0.0% + 0.1%)

Table 3: Binary search 1000 records

	I refs	I1 misses	LLi misses
Collected	322,498	220	158
	I1 miss rate	**LLi miss rate**	
	0.07%	0.05%	
	D refs	**D1 misses**	**LLd misses**
	181,724 (117,376 rd + 64,348 wr)	88 (5 rd + 83 wr)	87 (4 rd + 83 wr)
	D1 miss rate		
	0.0% (0.0% + 0.1%)		
	LL refs	**LL misses**	**LL miss rate**
	308 (225 rd + 83 wr)	245 (162 rd + 83 wr)	0.0% (0.0% + 0.1%)

Table 4: Binary search containing 10000 records

When looking at the two first tables which are for the linear search you can see that miss rate is minimal when it's on a 1000 records, however when the size is increased to 10000 records, the miss rate increases quite substantially. As for binary search, the miss rate with a record size of 1000 is substantially higher, however it barely increases when the record size is increased to 10000. Since the time complexity of a linear search is O(n) where n in this case is the number of records in the table, this makes sense. As for binary search it's time complexity is log2(n) because for each iteration, the size is possible scope of the search is halved. It is therefore far less impacted by an increase of records than a linear search is.
Using the built in profiler the results for the binary search is: Number of disk seeks/reads/writes/IOs: 6/8/81/89 and the results for the linear search is: Number of disk seeks/reads/writes/IOs: 2/244/81/325. Based on this you can see that the linear search does a lot more I/O operations compared to the binary search, so the binary search is more efficient.

4 Discussion

[6, 7] The most challenging part of this assignment was undoubtedly to get a good grasp of the precode and figure out how all the parts fit together and how to successfully add the required extra features. The assignment text itself is unclear as to how it wants you to implement a binary search, it does not clearly state that you must make use of already existing functions and global variables in order to complete it correctly. Luckily the TA cleared up this confusion and explained that it should be accomplished by calculating what page the record you were searching for was in, then parsing through that page recursively in order to find the record with the value you were searching for. Implementing this proved to be challenging, but the implementation functions as intended, except that it results in a segmentation fault when attempting to return the table. However, as explained earlier, the table can be displayed and it contains the relevant record, so the implementation works.
Now as for binary search in comparison to linear search, it's obvious that the larger the database, the better binary search will perform compared to a linear search. When comparing binary search against a B+-Tree, a B+-Tree offers greater predictability in performance as the size of the database is largely irrelevant to how it performs. This contrasts binary search which is better suited for larger databases. The time complexity for a binary search is log2(n) while the time complexity of a B+-Tree is O(t logt n). Comparing the actual file size between the implementation created in task 3 and if the file was organized as a B+-Tree will be purely

theoretical. In the table created in task 3, the values are stored in an pages which contains blocks that contains records which contains the individual values. A B+ tree stores data pointers at its leaf nodes which are different from than internal nodes. Leaf nodes contain an entry for every value of the search field, along with a data pointer to the record or block that contains the record. The file size of a B+ Tree organized file would likely be larger than the file size for the solution created in task 3 because of how a B+ Tree file stores its data.

5 Conclusion

In conclusion, the required extra features have been implemented according to the task requirements, however there is a bug with a segmentation fault at the very end, but the implementation itself does work regardless of that.

6 Sources

References

[1] Page and block Difference between page and block in OS. Retrieved 17:34, October 10th, 2023, from `https://www.geeksforgeeks.org/difference-between-page-and-block-in-operating-system/`

[2] Binary Search Algorithm Binary Search Algorithm. Retrieved 19:00, October 12th, 2023, from `https://en.wikipedia.org/wiki/Binary_search_algorithm`

[3] Binary Search Binary Search. Retrieved 19:48, October 12th, 2023, from `https://www.geeksforgeeks.org/binary-search/`

[4] Profiling with Valgrind Profiling with Valgrind. Retrieved 10:24, October 14th, 2023, from `https://developer.mantidproject.org/ProfilingWithValgrind.html`

[5] Interpreting Cachegrind Output How Do You Interpret Cachegrind Output For Caching Misses. Retrieved 14:23, October 14th, 2023, from `https://stackoverflow.com/questions/20172216/how-do-you-interpret-cachegrind-output-for-caching-misses`

[6] Introduction of B-Tree Introduction of B-Tree. Retrieved 16:58, October 15th, 2023, from `https://www.geeksforgeeks.org/introduction-of-b-tree/`

[7] Binary Tree vs B-Tree Difference Between Binary Tree and B-Tree. Retrieved 19:37, October 15th, 2023, from `https://www.geeksforgeeks.org/difference-between-binary-tree-and-b-tree/`