Assignment 2

Database Systems

Contents

1 Introduction

In this assignment I was tasked with understanding, modifying, testing and benchmarking a database system. The source code has sparse commenting, making the actual code difficult to understand. There was a html file tree supplied, describing the design, but there were few details concerning the actual implementation.

I have chosen to assume the third part of the assignment entails implementing an algorithm, that – like the linear search algotithm – performs its search directly on the database, without generating metadata files.

The benchmark should examine the disk operations, rather than the actual runtime. This is understandable, since most student's computers these days likely use SSDs, in which seek time is almost negligible. Most large databases still use hard disk drives, and as such a large number of seeks would severely impact performance.

1.1 Requirements

1. Understand and familiarise oneself with the precode (I've chosen to let my solution of the assignment itself be a demonstration of my understanding the codebase)
2. extend the functionality of the 'where' clause to support more standard integer operations
3. Find and replace the current linear search with a binary one, for equality searches on integer fields.
4. Compare results to a theoretical B^+-treeimplementation (covered in 4.1)

1.2 Technical background

For this assignment, I have implemented the testing file in python 3.10, so one thing you need to know to read the implementation.

- Python match statement
  Match statements in python are like switch/case statements in other languages. They take one argument on the first line, and have multiple outcomes (cases) on the next indent level. The chosen case is executed without fallthrough, so no 'break' statements are needed.
- Python maps
  A map in Python is akin to a list of keyword arguments. That is, arguments whose names are explicitly stated in a function call. Python dictionary objects are the default maps.

2 Design and Implementation

The design of the solution is divideed into two parts; the C implementation injected into the pre-code, and the python program. I will refer to the C code as the binary search algorithm, and the python program as the testing program.

2.1 Page binary search algorithm

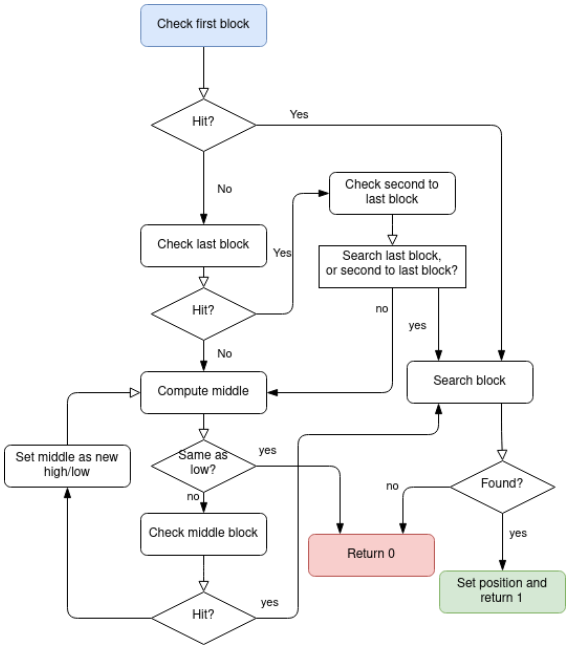

Figure 1: Binary search execution

The binary search algorithm first searches on block level. Figure 1 provides a high-level view of the execution flow. What the 'check block' and 'search block' boxes mean is further explained in 2.1.1 and 2.1.2, respectively, but for now I'll treat them as black boxes.
First the algorithm checks the first and last blocks separately, as they are edge cases. If the reference value is found, the position of the database is set, and the function returns 1. Except when the value is found at the first record of the last block, in which case, the second to last block is checked. If the value is again found at the first record of the block, the binary search algorithm continues executing normally.
The normal loop of the algorithm checks the middle block, and does one of five things;

- If the value is found at the beginning of the block, the previous block is checked in the same way. If the block is being checked as a result of this process, in order to avoid a linear search, the block being checked in the parent function is set as the new high.
- If the value is found at the end of a block, the position is returned immedeately.
- If the reference value is found to be between the highest and lowest values, the algorithm launches a search on the block. If found, the position of the talbe is set to the first found occurence, and the function returns 1. Otherwise it returns 0.

- If the reference is above the range of the block, the block is set to the new low. If the low point remains unchanged, the function returns 0.
- If the reference is below the range of the block, the block is set to the new high.

2.1.1 Page checking algorithm

The page checking algorithm (figure 2) works by comparing the reference value to the first and last values. A number of return values are possible, because different actions should be taken depending on the results of the check, it's not as simple as the binary search performed on one page.

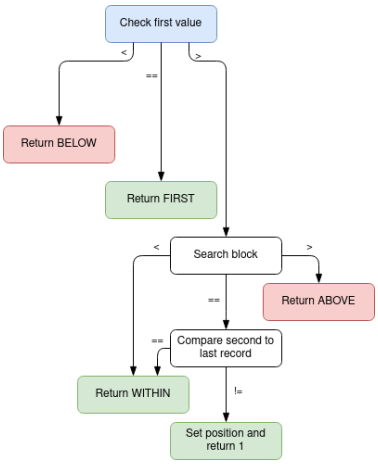

Figure 2: Binary search execution

2.1.2 Record binary search algorithm

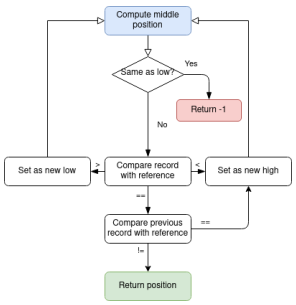

Figure 3: Binary search execution

The record search algorithm (figure 3) is similar to the parent search algorithm, but with a few key differences;

- The first and last records are not checked, as they are evaluated in parent function.

- Upon finding the record, the function returns the position of the record within the block.
- If the record isn't found, the function returns -1 instead

2.2 Testing program

I chose to make an external program to test the database management system, as this more accurately simulates normal use of the database system front. I won't go too much into detail about the specifics of the program, since it's only peripheral to the assignment.

The main function of the program is generating a file with instructions for the database front. First a random schema is created, with fields selected from a pre-made list. Then instructions are created to create a table with the schema, and to insert records into the table, with one ascending integer field. The field does not ascend strictly, and not continuously, simulating a real, messy database with records representing real, messy data.

2.2.1 Flags

I have designed the testing program to run with a few flags, to ease the task of testing and benchmarking the main program. Run the program with the flag '–help' for more information.

2.2.2 Classes

There are two classes in the testing program, the schema class and the field class. The program utilises these classes in order to generate random records for the database.

The field class is initiated with a name, type and some options detailing its contents. It remembers its own name, byte length, and how to generate a new random entry based on its specifications.

The schema class takes a list of field maps, and uses them to generate a list of fields. The schema remembers its total byte length (sum of the lengths of each field), how to generate a new random record, and the history of all random records it has generated.

3 Results

I've tested my implementation with random queries on databases with records numbering in the hundreds of thousands, and it's performed flawlessly, apart from the fact that the program seems to write all its output to 'stderr', rather than 'stdout'. I'll owe this up to the pre-code, and assume that it's not a problem for me to solve.

The benchmark is – as specified in the assignment – measured in pager operations. I have excluded the 3 seek, read and write operations that are only related to the metadata, as they are constant between the two different operations.

These are the results from performing one single query on a database of size 20 000.

	Binary		Linear	
	Seeks	Reads	Seeks	Reads
Total	105,0	101,0	10	42 646,0
Average	10,5	10,1	1	4 264,6
Min	8,0	7,0	1	164,0
Max	11,0	11,0	1	999,0

4 Discussion

The results make two things clear. That the binary search algorithm uses far fewer reads than the linear search – but also, the performance of the binary search algorithm suffers greatly without proper indexing. The number of reads is the same (most of the time) as the number of seeks.

This is a massive detriment to the performance of the algorithm, and considering how even a non-clostering integer key index can be done with only two integer values. Two integer values (8 bytes) is shorter than any database schema that can't easily be sorted, and thus requires way fewer pages to store than most databases it indexes.

4.1 B^+-tree

The largest advantage of the B^+-treeis that there's no need to keep the database sorted – apart from clustering reasons – since the leaf level of the tree keeps an ordered index of all the records. For this reason, a B^+-treeimplementation would in this assignment, like with indexing, allow for searching all integer field non-linearly.

The one advantage my implementation has as compared to the B^+-treeimplementation, is that no extra metadata is generated, whilst the B^+-treerequires several blocks of additional metadata to work.