REPORT

March 5, 2019

Lino Lastella 1001237654

CSC443H1 - W2019

Assignment 2

1 - External Sorting

Figure 1 displays three different line graphs, one for every page size.
We notice immediately that the number of pages read and written (y-axis) decreases drastically when increasing initially the number of buffers (x-axis) even by a small amount. However, any further increase of number of buffer pages does not improve efficiency as much, until no more improvement happens at all.

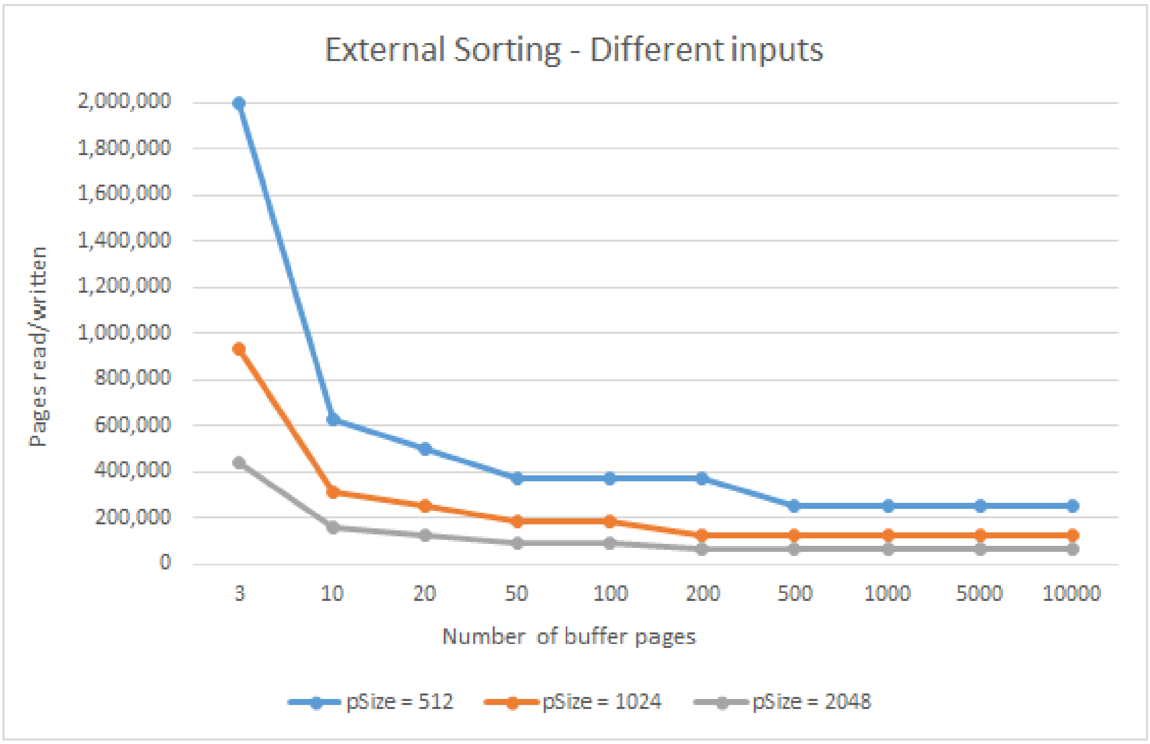

Figure 1

2 - Hash Indexing

In order to solve question 2 I adopted the following strategy:
First, I defined a bunch of classes that helped me clarify the various algorithms.
Particularly, I defined a "DataEntry" class, which is simply a data structure holding a key (string) and a row number.
Then, I defined a "Page" class. Every Page holds a collection (vector) of data entries.
Finally, the last data structure shared among all types of indexes is a "Bucket"; it holds a linked list of Pages and some bookkeeping members such as the number of overflow pages within the bucket.

Once the common classes had been defined, I moved on to create an abstract "Root" class which would function as base class for polymorphism.
The first class that inherits from Root is the static index. This index holds a vector of buckets (fixed size) and it will create overflow pages every time one of them is full.
Figure 2 shows a distribution of number of pages per bucket (for instance, there is only one bucket containing from 0 to 3 pages, but there are around 130 which contain between 12 and 15 pages)

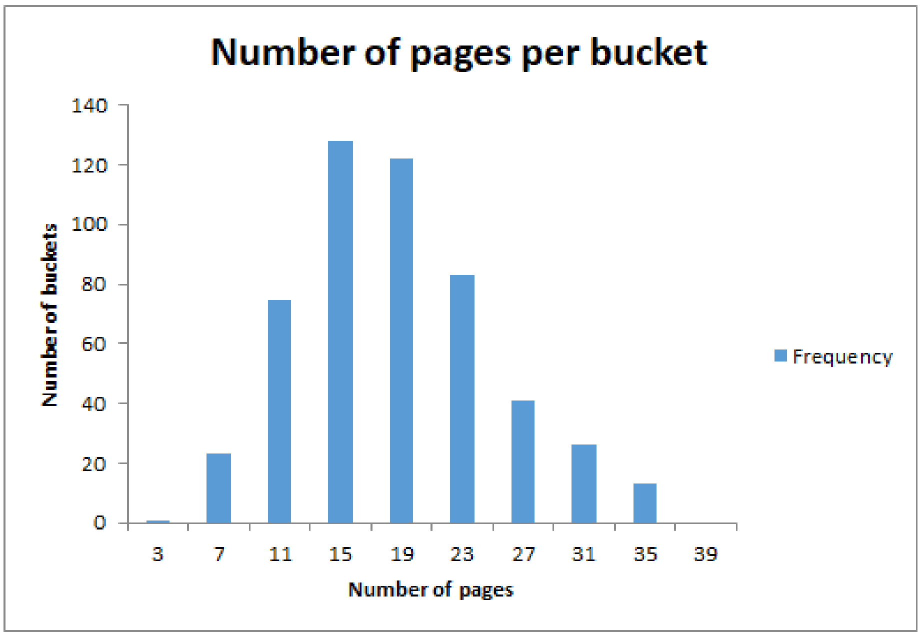

Figure 2

Next up is extendible hashing.
For this class I defined another type of bucket, called "ExtBucket", which only differs from the regular bucket by keeping track of local depths.
However, the index is quite different than before. The ExtendibleHashing::Index holds a vector of buckets, a directory (vector of numbers) and a number representing the global depth.
This index only starts using overflow pages once the underlying bucket contains only all duplicates. Additionally, I found myself forced to start using overflow pages once global depth grew larger than 25, since my computer started to run out of memory (allowed by the instructor on Piazza).
This led to the consequence of having many pointers pointing to the same bucket and most of the buckets containing at most 3 pages.
I did not make an histogram for this part since it would only show a huge bar for the number of buckets containing less than 4 pages, outgrowing all the other bars (can be deduced from the run output).

Finally, I moved onto implementing linear hashing.
In this case I did not even use a directory, since from the assignment requirements we were guaranteed to have an initial number of buckets which is a power of two.
This type of index improved drastically the performance issues encountered in extendible hashing. However, we notice immediately from Figure 3 that the index contains a very large amount of empty buckets.
Luckily, the overall improvement is still significant and the majority of the buckets only contain one or two pages (when they're not empty).

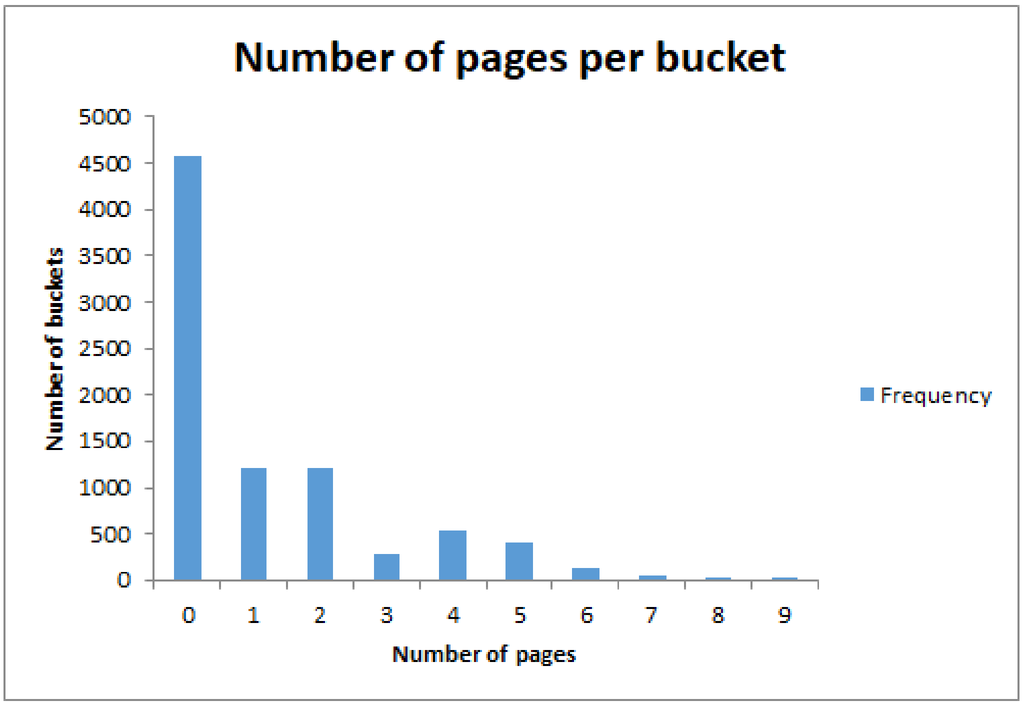

Figure 3

For all three types of index I used a similar binary file format when writing to disk. They all share a common header, i.e. 10 bytes representing respectively the type of the index, the field the index is based on, the page size and the number of primary buckets. Immediately following the header I inserted the main index block.
This block consists of sequential allocation of all primary buckets, and at the end of all such buckets a pointer to the beginning of the overflow pages block for that bucket (if any). Overflow pages for every bucket are also sequential, and the end of the overflow pages block is marked with a sequence of ASCII(31) ('0') for every bucket.

For static index, the main index block follows immediately the header, since there is no need to store any additional information.
For linear index, the main index block is only preceded by 2 bytes representing the number of bits to read from the hashed value of a queried key.
For extendible index, before I could write the main index block, I had to write the directory. So in addition to the same 2 bytes as in linear hashing, I allocated a sequential block of numbers (4 bytes each) representing the directory. Once that was done, the main block index was inserted as in the previous two types of indexes.

3 - Query by Index

When reading the binary index file created in the previous section, a similar strategy across all types of indexes was adopted.
First I read the common header, and for the cases of extendible hashing and linear hashing, I did some more manipulation to locate the primary bucket.
Then, I used the pointer at the end of the primary index page to locate the start of the overflow pages block, and seeked there.
Finally, I read all overflow pages until I found the end-of-overflow-pages-block marker.
This process allowed me to collect all row numbers for the given queried value.
Once I knew all the row numbers it was only left to open the main database file and seek to every row, reading one page at the time (I assumed that every record was in a different page, since we were told not to bother handling the cases of rows appearing on the same page).

Final note: all the screenshots for every part of this assignment are in the compressed folder "screenshots.zip".