Performance Report

**Reading and Writing Rates**

**Optimal block size**

Disk block size: 4096 bytes

The results from the experiment are summarized below:

|  |  |  |
| --- | --- | --- |
| **Size** | **Write\_blocks\_seq (MBPS)** | **Write\_lines (MBPS)** |
| 0.5KB | 22.6834 | 26.5146 |
| 1KB | 22.5882 |
| 4KB | 22.8988 |
| 8KB | 22.6184 |
| 16KB | 21.8792 |
| 32KB | 22.1986 |
| 1MB | 22.5014 |
| 2MB | 22.5014 |
| 4MB | 22.8536 |

For write\_blocks\_seq, all the rates are fairly close (around 22.5MBPS). From block size of 8KB onwards to 1MB, increase in block size does not contribute to better performance. This is probably because even though the time of loading into buffer and writing to binary file differs for each block size, the total time is about the same. The rates peaked at 22.8988MBPS when block size is 4KB, which matches the disk block size.

From the recorded data, write\_lines is more effective. This is because for write\_lines, we only time reading a line then writing a line, without any other manipulation to the string. With the right block size, write\_blocks\_seq should theoretically be able to outperform write\_lines, because reading and writing in blocks minimize the number of fread’s and fwrite’s. But perhaps my write\_blocks\_seq can still be optimized (eg. optimizing parsing). With the optimization, it might be able to outperform write\_lines.

**Sequential vs. random read rate**

Here are the results from read\_blocks\_seq and read\_ram\_seq:

|  |  |  |
| --- | --- | --- |
| **Size** | **Read\_blocks\_seq (MBPS)** | **Read\_ram\_seq (MBPS)** |
| 0.5KB | 281.4298 | 3586.232 |
| 1KB | 278.6954 |
| 4KB | 293.1726 |
| 8KB | 289.8928 |
| 16KB | 288.4002 |
| 32KB | 294.9954 |
| 1MB | 302.9516 |
| 2MB | 300.9178 |
| 4MB | 290.205 |

The rate of read\_blocks\_seq slowly increases with a peak of 303MBPS when block size is 1MB. However, the rate is still significantly lower than that of read\_ram\_seq. For read\_blocks\_seq, we account for the time the processor read from file into the buffer. On the other hand, the read\_ram\_seq speed is extremely fast because we don’t include the time where the processor loads the file into RAM. The only thing that is timed is list manipulation.

The ratio from class:

Taking the optimal read\_blocks\_seq speed, the ratio here is:

The ratios are different, but not too far off. There are multiple reasons that might have caused this:

* My read\_blocks\_seq can be optimized.
* The calculation in average and max might have altered the speed a little bit.

Here are the results from read\_blocks\_rand and read\_ram\_rand. They are all ran against *big.dat* with iterations .

|  |  |  |
| --- | --- | --- |
| **Size** | **Read\_blocks\_rand (MBPS)** | **Read\_ram\_rand (MBPS)** |
| 0.5KB | 1.529 | 24 |
| 1KB | 2.2458 | 48.5 |
| 4KB | 8.6514 | 182 |
| 8KB | 15.923 | 412.8142 |
| 16KB | 30.4464 | 781 |
| 32KB | 58.9012 | 1354.167 |
| 1MB | 225.4318 | 3004.625 |
| 2MB | 273.5724 | 2528.933 |
| 4MB | 302.2606 | 3191.187 |

For read\_blocks\_rand, we see a trend that as block size increases, the rate also increases. When block size increases, for each iteration, we read more records into the buffer. Then we will have to loop through the whole buffer to calculate max and average, thus the time spent will increase. However, since we are also counting more records because block size increased, the total processed bytes also increase. Base on the trend, we can assume that the amount of increase in byte size outweighs the increase in time, thus the rate increases as block size increases.

Similarly, for read\_ram\_rand, the amount of increase in byte outweighs the increase in time. Thus the rate increases as block size increases.

Once again, read\_ram\_rand speed is extremely fast because we don’t include the time where the processor loads the file into RAM. The only thing timed is list manipulation. We can see that for any block size, read\_ram\_rand outperforms read\_blocks\_rand.

Here is a plot of the different read’s performance (I used the fastest rate of each read):

In conclusion, here are my observations with regards to read speed:

* Reading from memory is significantly faster than from secondary storage.
* Sequential read is almost always faster than random read (with the exception of read in blocks with block size 2MB)
* While Jacob’s paper shows that sequential read from SSD is faster than random read from RAM, this is not the case from my experiments. This can be caused by multiple difference (eg. my program can be further optimized, how to time read\_ram\_rand etc).
* The speed of each read from my experiments can be summarized as:  
  read\_ram\_seq > read\_ram\_rand > read\_blocks\_seq > read\_blocks\_rand

**Sequential vs. random write**

Here are the results from write\_blocks\_rand and write\_ram\_rand. They are ran against *records.dat* with iterations .

|  |  |
| --- | --- |
| **Write\_blocks\_rand (MBPS)** | **Write\_ram\_rand (MBPS)** |
| 0.0218 | 33.06 |

Once again, we can observe the pattern that writing from RAM is faster than writing with blocks. While the speed write\_ram\_rand, write\_blocks\_seq, write\_lines are similar, write\_blocks\_rand is significantly slower. This is possibly because in write\_blocks\_rand, we need to do fseek twice and fread, fwrite once respectively; while in write\_ram\_rand, we can easily locate a record with list notation.

**Reading and Writing Rate Summary**

Here is a summary of the performance of all the functions:

In conclusion, there are multiple things I learned from doing this assignment:

* Finding the optimal block size is important because it will change the access speed.
* Different reading/writing functions might have different optimal block.
* For some functions, increasing block size means there will be less I/O and faster access speed.
* For most cases, reading/writing from RAM is significantly faster than reading/writing in blocks.
* While it is faster to read/write from RAM, it is very expensive to have a big RAM. Thus eventually we need to read/write in blocks.
* Sequential access is almost always better random access. Thus one of the goals of database management system is to maximize sequential access and minimize random access. This can include:
  + put related blocks close to another (cylinder group)
  + pre-fetch block(page) if necessary (double buffering)
  + use disk scheduling algorithms to minimize time of disk I/O

**A1.2 – 2PMMS**

**Timing disk\_sort:**

I ran the disk\_sort program on the original “Arizona State University Twitter Data Set”, and used the timing command to time evaluate the performance. I used the optimal block size of 4096 bytes. I ran this part of the experiment 5 times, and took the average of the results. Here is the performance of my disk\_sort program:

(Average) Total elapsed time: 1 min 27 sec.

(Average) Maximum resident set size: 207604 KB

Note that the maximum resident set size is fairly close to the allocated memory of 200MB (=200000KB).

**Performance and RAM (buffer) Size**

I performed this part of the experiment on the concatenated input file which is more than 1.5 times larger than the total main memory on the machine. I tried memory size of 1/2, 1/4, 1/8, 1/16, 1/32, and 1/64 of the original 200MB, and the program cannot perform the two-pass algorithm when it reaches 1/128 of the original 200MB.

However, since it is discovered that qsort actually uses more memory than the allocated memory, we are instructed to split the partition into two. For example, when we allocate 200MB to RAM, each sublist/run would actually have size of 100MB instead of 200MB. Thus the two-pass algorithm would not work when we get to 1/64 of the original 200MB.

Once again, I ran each memory size multiple times and took the average of the results, and here are the results:

|  |  |  |  |
| --- | --- | --- | --- |
| **Memory size** | **Fraction of original 200MB:** | **Total elapsed time (mm:ss)** | **Max. resident set size (KB)** |
| 200MB | 1 | 3:14 | 207704 |
| 100MB | ½ | 3:18 | 105272 |
| 50MB | ¼ | 3:33 | 53844 |
| 25MB | 1/8 | 4:18 | 28372 |
| 12.5MB | 1/16 | 4:22 | 15496 |
| 6.25MB | 1/32 | 5:54 | 9084 |

Notice that for the max resident set size, all the results are fairly closed to the allocated RAM memory; the max resident set size are all around 2MB away from the allocated memory.

As for the total elapsed time, theoretically it should not depend on the number of runs K. This is seen in the first 3 cases of the experiment, the total elapsed time for 200MB, 100MB, and 50MB are all fairly close (~3:20ish). However, one can observe an increasing trend in the total elapsed time as memory size increases. I suspect that this is due to the structure of the program disk\_sort. In phase 1 of disk\_sort, my program will write the sorted sublists into different .dat files. Those sublists would be opened (and closed) every time the corresponding input buffer needs to be refilled. When the RAM memory size is small, there will be more sublist (smaller.dat files) and more run in phase 1. Thus in phase 2, the input buffer would be smaller as well. In other words, more fopen and fclose are called, thus decreasing the time efficiency of the program.

In the first 3 test cases, the total elapsed time is around the same because the number of sublists is still small. However when it does to memorize size of 25MB, there would be 160+ .dat files, and each time we have to fseek and read a relatively small number of elements into the input buffer. Thus this explains why there is a different in performance.

**Performance against Unix Sort**

This is the result when I tried to time the unix sort:

|  |  |  |
| --- | --- | --- |
| **Sort on original data** | **Total elapsed time** | **Max. resident set size (KB)** |
| My disk\_sort | 1:27 | 207640 |
| Unix sort | 2:08 | 345920 |

From the result, one can observe that the unix merge sort is slightly slower than the 2PMMS. After doing some research on the Unix sort command, I found that Unix uses “External R-Way merge” to sort through large data, which is similar to 2PMMS. This explains why the total elapsed time is still very close. The difference might come from instantaneous changes in CPU or because of the different in input file (disk\_sort takes in a ~600MB .dat file while unix sort takes in a 1GB+ .csv file). Unix sort might have to do more processing.

As for maximum resident set size, there is a difference of around 100MB in the amount of memory used. Resident set size is the amount of memory that belongs to a process that is held in the RAM. This can possible be due to the fact that we have full specific control over memory usage in our 2PMMS program, however when we run the unix sort, we have no control over how much RAM is used. Another reason is possibly due to the difference in input format (same reason as the time difference). The unix sort takes in anything in general while my disk\_sort program takes in something very specific.

3. Degree distribution:

Maximum out-degree (uid1): 214381

Maximum in-degree (uid2): 564512

Below is the two histograms, one for out-degree, and one for in-degree. The histogram has a strong indication of the power-law distribution. When calculating the relationship between the two variables, I took two random points from the data.

For the out-degree function, I used the two points (0, 6.737) and (3.307, 0). Function describing the out-degree relationship:

For the in-degree function, I used the two points (0, 6.577) and (3.301, 0). Function describing the in-degree relationship:

Summary of what I learned:

Through this part of the assignment, I got more familiar with the 2PMMS algorithm. I think often times it’s easily to listen to information in lecture, but implementing them is a totally different story. The assignment definitely filled in the details of 2PMMS that we didn’t not learn in class, such as using a heap for phase 2, or how the input buffers are refilled each time. It is also an interesting experiement to compare 2PMMS with the unix built-in sort. It’s amazing to see my implementation is actually faster than the built-in sort (for this specific file).

Another thing that I learned is the power-law relationship. This is a new relationship between 2 variables that I have never seen in any of my undergrad courses. It is interesting to see how a lot of man-made phenomena approximately follow this power-law relationship, and this can be shown from the twitter data in this assignment.

Questions to ask:

* max resident set size comparison for unix sort vs disk sort
* power law. do we need to find the function explicitly?
* separate report???? one for performance one for one for twitter data