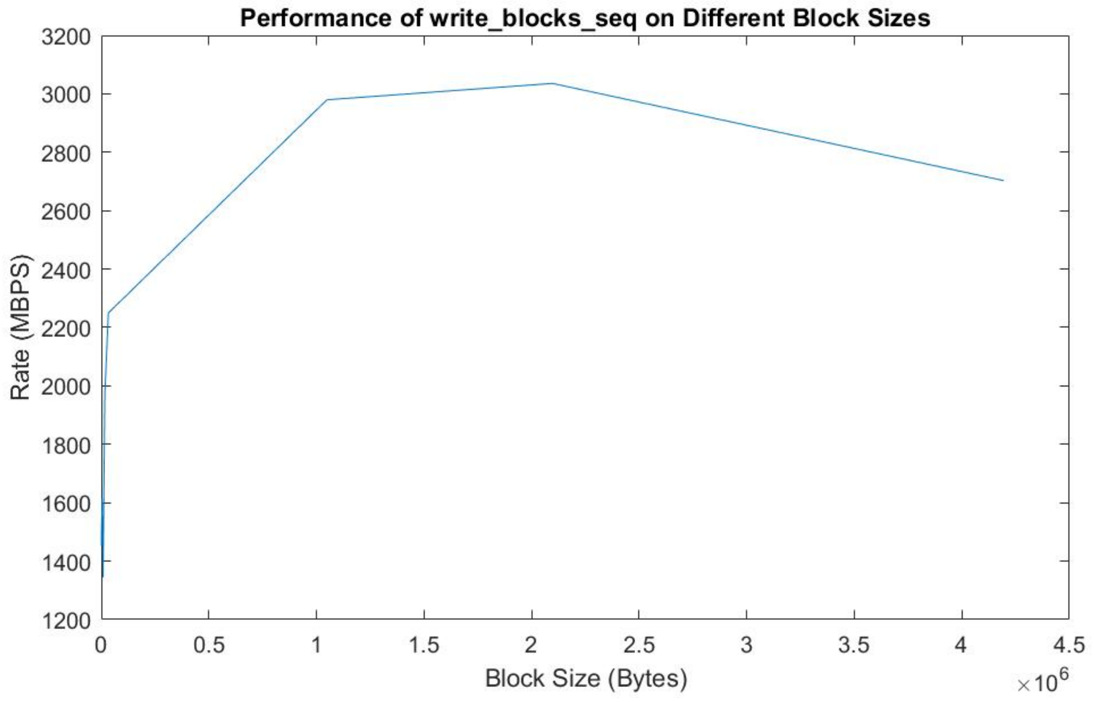
CSC443 Assignment 1 Report

Part1:

Experiment 1: Optimal Block Size

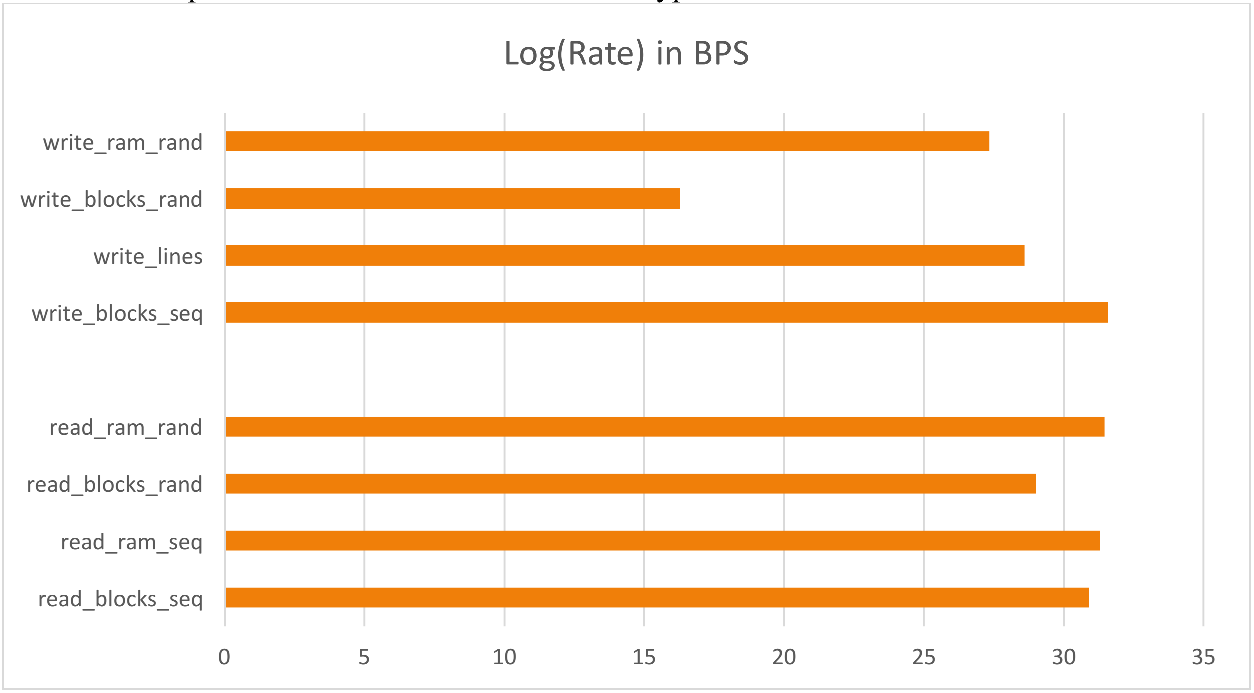
The optimal block size according to our experiment on *write_blocks_seq* is 2MB. It does not correspond to the system disk block size which is 4096. Block size of 4MB did not perform better than 2MB. The performance increases as block size increases from 512B to 2MB (except for 8KB), and decreases as block size increases from 2MB to 4MB. This is because the larger the block size, the fewer I/Os are needed to write the file. The reason why 2MB block size has better performance than 4MB block size is probably the time to write larger buffer size increases more than the time of reduced I/Os.
write_blocks_seq is more efficient than *write_lines*, because *write_lines* are doing disk I/Os for each line whereas write_blocks_seq are doing disk I/Os for each block. Therefore, *write_blocks_seq* has less disk I/Os and has better performance.
Please see the plot below.



Experiment 2: Sequential vs. Random Read Rate

Note: We performed the following experiments using SSD on CDF with block size 2MB.

Database output: Average follows = 9.731726, Maximum follows = 214381.
We used Solid State Disk as the secondary storage, it has a sequential read rate of 1930 MB/s. The RAM has a sequential read rate of 2520 MB/s. The ratio rate between SSD and RAM in our experiment is 1930/2520 = 0.766, but the ratio discussed in class is 42/358 = 0.11. Our test result does not correspond to the ratio discussed in class. They both showed RAM having a faster read rate than SSD, but the discussed ratio is much lower than ours. We think that query is one of the reason caused the difference. We implemented a query in the program of testing RAM and SSD to read records from buffer, so the running time of query was an overhead while testing reading speed. The other problem is that we only tested 5 times for each experiment. We may need more tests to get a more accurate result.
The random read program for RAM (*read_ram_rand*) has the highest read rate around 2803 MB/s, while average for sequential read for RAM (*read_ram_seq*) is about 2521 MB/s. The query program in read_ram_seq takes more time because of extra executing time of query. Therefore, the read_ram_seq should have the highest read rate without query. *read_blocks_seq* and *read_blocks_ram* have a rate of 1927 MB/s and 513 MB/s separately. In SSD, sequential read take much less time than random read. In Addition, read_ram_seq has lower read rate than read_ram_rand because of query execution. RAM always takes less time to read both randomly and sequentially than SSD. In summary, we say RAM should have faster read speed than SSD, and also sequential read take less time than random read.
Please see the performance results on all different types of reads and writes below:



EXPERIMENT 3: SEQUENTIAL VS. RANDOM WRITE RATE

In conclusion, we have learned that comparing to RAM, disk I/O operations are more expensive. Sequential reads and writes are more efficient than random reads and writes. In order to improve performance, we should consider fewer disk I/Os, reduce seek and rotation delays using sequential disk access instead of random disk access.

Part2:

EXPERIMENT:

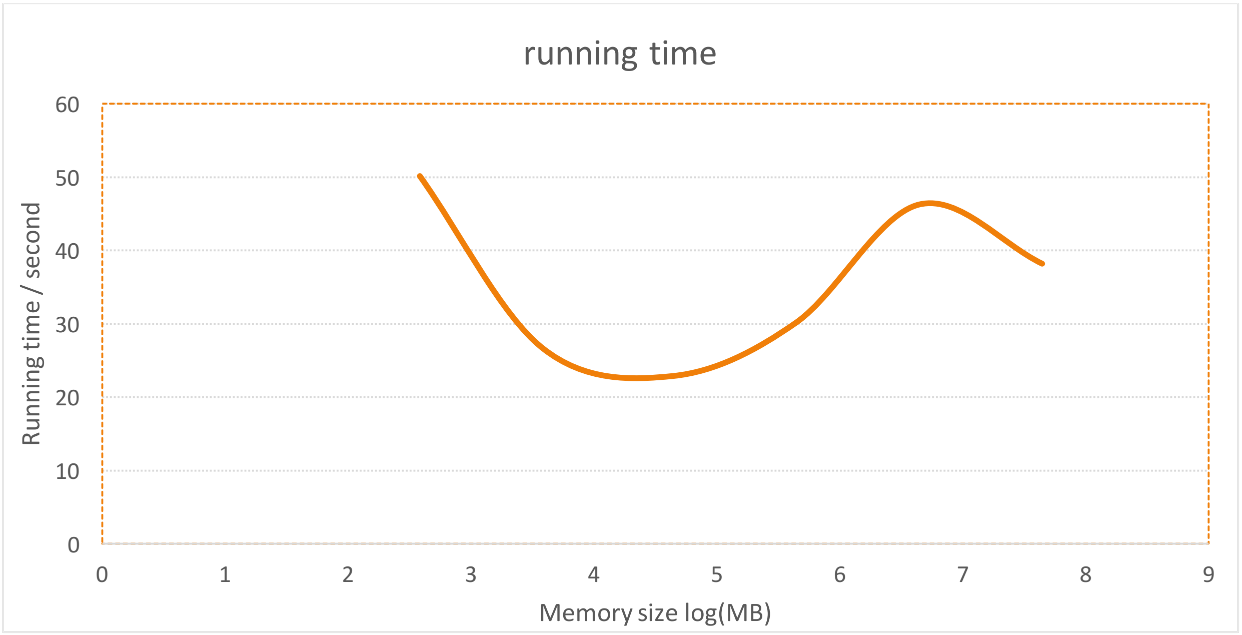
Test result for 2PMMS:

Memory size (MB)	Elapsed time(m:ss)	Maximum resident set size(KB)
200	38.19	206224
100	46.29	103920
50	30.16	50708
25	22.88	26136
12	26.65	13812
6	50.14	5600

Black pass 2PMMS

Red only pass Phase I

Only Memory size of 200 MB and 100 MB can perform 2PMMS, others only only accomplish phase I because our optimal block size is 2MB and memory size is too small for merging in phase II.



Test result for Unix sort:

User time (seconds): 115.44
System time (seconds): 3.73
Percent of CPU this job got: 141%
Elapsed (wall clock) time (h:mm:ss or m:ss): 1:24.46

```
Maximum resident set size (kbytes): 3365516
```

Question:

1.Is there any difference in performance in your experiments? Explain why there is a difference or why there is no difference.

The elapsed time is different for different memory size. Any memory size less than or equal to 50MB which is too small for 2PMMS. Memory size with 200MB has the least elapsed time. Although only 200 MB and 100 MB satisfy 2PMMS, it shows a tendency that the elapsed time increases as memory size decreases. The reason is that large memory size means that less sub arrays to be sorted separately in main memory, so it takes less time in phase I, and in phase II less file open streams needed to read each sub list file. Memory size 200MB has the most maximum resident set 201MB, and 100MB only takes half of it.

2.Which program is faster: your implementation or Unix `sort`? Which one uses less memory? Explain the difference (or the lack of difference) in performance. If there is a difference - what in your opinion could explain it?

My implementation is faster than Unix sort. Unix sort use 3286MB memory while my implementation only uses 201MB memory for 200MB allocated memory size. The elapsed time for Unix sort is 1:24.46 which is longer than my implementation. User time of Unix sort is 115 seconds, and my implementation takes around 10 seconds. In my implementation read binary file directly, while Unix sort read input line by line and extract sort keys from each line of input. Unix sort takes much time to read the inputs and then convert them into binary format.

Conclusion:

In 2PMMS, we perform 4 I/Os for per each block, so the running time depends on the memory size which can hold large unsorted array. More memory means faster sorting sub arrays in main memory. Unix sort take more memory and more time than 2PMMS.

Part 3:

STEP 1: EXPRESSING QUERIES

1.Producing the list and the count of user ID pairs which represent "true friends".

2.Producing the list of top 10 "celebrities", sorted by (in-degree - out-degree)-difference in descending order (from more famous to less famous).

Description:

1. Select the user ID pairs which they followed by each other, and count the number of pairs.
2. Select the top 10 users which they have the largest difference between in-degree and out-degree, sort these 10 users in descending order of difference.

Relational Algebra:

R(uid1, uid2) := Twitter(uid1, uid2)

1. $(\sigma_{R1.uid1 = R2.uid2 \text{ and } R1.uid2 = R2.uid1} (\rho_{R1}(\sigma_{uid1 < uid2} R) \times \rho_{R2}(\sigma_{uid2 < uid1} R)$
2. Outdegree (uid, count) := $\gamma_{uid1 \to uid, COUNT(uid2) \to count}\ R$

   Indegree (uid, count) := $\gamma_{uid2 \to uid, COUNT(uid1) \to count}\ R$

   $\tau_{diff}\ \rho_{(uid, indegree, outdegree, diff)} (\Pi_{Indegree.uid, Indegree.count, Outdegree.count, Indegree.count-Outdegree.count}$

   $\sigma_{Indegree.uid=Outdegree.uid}$ (Indegree × Outdegree))

SQL:

1.

```
SELECT COUNT(*)
FROM
(SElECT T.uid1, T.uid2
FROM
        (SELECT * FROM R WHERE uid1 < uid2) as T,
        (SELECT * FROM R WHERE uid2 < uid1) as S
        WHERE T.uid1 = S.uid2 AND T.uid2 = S.uid1);
```

2.

```
CREATE VIEW outdegree as SELECT uid1 as uid, count(uid2) as count FROM R GROUP BY uid1;
CREATE VIEW indegree as SELECT uid2 as uid, count(uid1) as count FROM R GROUP BY uid2;
```

```
SELECT indegree.uid, indegree.count as indegree, outdegree.count as outdegree, (indegree.count -
outdegree.count) as diff
FROM indegree, outdegree
WHERE indegree.uid = outdegree.uid
ORDER BY diff DESC
LIMIT 10;
```

STEP 2: IMPLEMENTING QUERIES

Implementation:

Our test block size is 2MB

1. First, sort edges dataset by uid1 and uid2 respectively, and then we create two relations that R is sorted with ascending order uid1, uid2, and S is sorted with ascending order uid2, uid1. Second, split 200MB available main memory into two input buffers (98MB) for each relation and one output buffer (4MB). Load R and S into main memory, and find intersections by zigzag join. If R.uid1 < S.uid2 , move pointer to next tuple in R, if R.uid1 > S.uid2, move pointer to next tuple in S. When R.uid1 = S.uid2, compare R.uid2 and S.uid1, if R.uid2 < S.uid1, move pointer to next tuple in R; if R.uid2 > R.uid1, move pointer to next tuple in S. Keep looping R and S until find R.uid1 = S.uid2 and R.uid2 == S.uid1, save (R.uid1, R.uid2) into output buffer.

2. First sort edges dataset by uid1 and uid2 respectively, and then create R with (userid, indegree) by sroted_uid1 files and create S with (userid, outdegree) by sorted_uid2 file. Next, merge R and S with same user id and calculate the difference between indegree and outdegree, output the top 10 celebrities who has the largest difference.

STEP 3: REPORTS

Test Result:

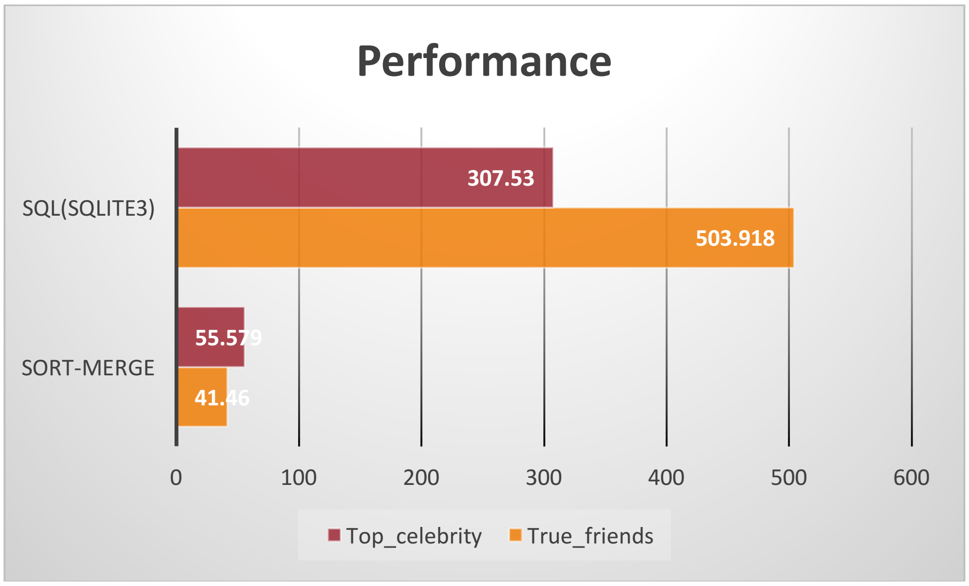
Total number of friends: 21776094
Total number of distinct users: 11316811

Top 10 Celebrities			
User Id	In-degree	Out-Degree	Difference
5994113	564512	292	564220
7496	350885	6035	344850
1349110	341963	1472	340491

1629776	172231	2120	170111
8121005	155967	34	155933
2041453	152689	620	152069
797152	118826	74	118752
6623784	116002	183	115819
645019	107914	275	107639
3403	102877	2946	97931

Performance:

	True Friends (s)	Top celebrities(s)
SQL (sqlite3)	503.918	307.530
Implementation(Sort-Merge)	41.460	55.579



Conclusion:

Our implementation take much less time than Sqlite3 no matter for computation the true friends or top celebrities. We use sort-merge to compute the query, sort the relations first and then use zigzag to merge tuples. The possible reason why Sqlite3 SQL take almost 10x running time is that it compares every tuple pair for two relations, and this process takes a lot of loop. Our implementation tries to limit numbers of I/Os, so we have less I/Os than SQL in Sqlite3. Less

I/Os mean less time. Different RDBM system may have different implementation for SQL, so it is important to know what kind of structure and data we are dealing with before we choose the appropriate RDBM system.