Performance Report

**JOIN**

I ran both SQLite and C++ implementation of the query in my laptop. For the C++ implementation of the queries, I ran it with memory size of 200MB and block size of 4KB.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Result from SQLite (sec) | Result from C++ implementation | |
| Time (sec) | Max. Resident set size |
| Find true friend | 1367 | 1054 | 207400 |
| Find top 10 celebrities | 406 | 48 | 207484 |

For the first query, I implemented it using BNLJ. Originally, I just implemented it by allocating M-1 blocks for the outer relation, and 1 block for the inner relation. Then for each tuple in the inner relation, I will scan the M-1 blocks sequentially. However after a while, I discovered that this implementation is too slow. Instead, I decided to first sort the relation, then read in M-1 blocks (that is already in sorted order). Then for each tuple in the inner relation, I will do a binary search on the outer M-1 block on the search key. If the key matches a tuple in the outer relation, I will scan sequentially forward and backward to look for a tuple that match both field.

I know this implementation is still not ideal, because the runtime of BNLJ is not linear time. However, the implementation is still faster than the SQLite runtime. This is perhaps due to SQLite being a BDMS that is not very good at optimizing queries (after all its open-source). Or perhaps SQLite optimize queries in a more general setting. However when I am implementing the query in C++, I can optimize it specifically for this query.

For the second query, I used an (un-optimized version of) SMJ. First, I took the sorting function from A1.2 and sorted the files according to uid1 and uid2 respectively (need to sort according to uid1 because the original file is not sorted). While sorting, I kept count of how many people each user is following or being followed. Once the sorting is done, I would also out two files that list the in degree and out degree of each user respectively. Then I will do a join on the in degree and out degree files. Since we know that there won’t be duplicate, I just need to scan through both file linearly. When the userID from the in degree file and the out degree file matches, I will check if the difference is big enough to be put on the top-10 list. If the user only exist in the in degree file, I will patch the out-degree with 0.

From the data obtained, we know that my C++ implementation is significantly faster than the SQLite implementation. This is probably due to SMJ being linear time. I suspect that SQLite probably use a very basic joining method (such as BNLJ).

Here is a plot that summarizes the results:

**Breif summary on all performance result**

Here is a list of summary on the performance on all 3 parts of A1:

* When doing I/O on a big data file, it’s always faster to read/write it sequentially than randomly
* When reading a data file in blocks, the performance would be affected based on the block size. In my experiment, the block size is optimal when it’s 4KB, which is also the default block size in my system
* My C++ implementation of 2PMMS is faster than the Unix sort.
* My C++ implementation of the two join queries is also faster than the SQLite execution.

**Conclusion**

As I look back and reflect on the pervious parts of the report, here are the things that I learned when it comes to working with a large dataset:

* When it comes to handling big data file, often times it wouldn’t fit into the RAM. Thus we need to somehow transfer data back and forth in between RAM and disk while achieving what we wanted to do (eg. sort or join).
  + Part 2 of the assignment really solidified my understanding of the 2PMMS program. I realized that ordinary sort (eg quick sort) would not have worked here because of the size of the data set. Thus we need to sort chunks of the file then merge them back together.
  + Part 3 of the assignment solidified my understanding of how join works in a DBMS level. Even though I did not get to implement/go through all the different join methods, it’s definitely gave a chance for me to look into how we can use sorting or hashing to implement joining.
* Often times the pre-existing functions (eg. unix sort or SQLite join) might be good at executing general programs, but my C++ implementations can exceed their performance.
* RAM size theoretically should not matter when dealing with large file (as long as it passes the memory requirement). However, in the physical implementation, there might be a slight performance drawback when the RAM size is too small (see part 2 results).
* Block size matters! Based on the different block size, the performance of the reading and writing would change.