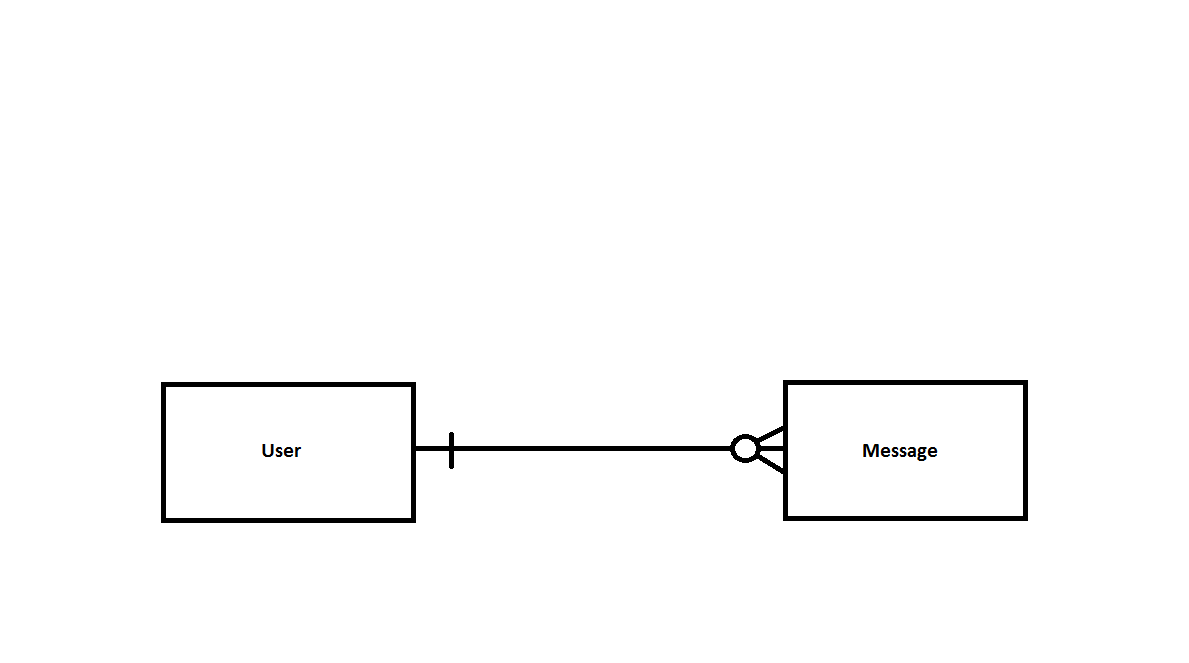
Group 6 – HCI Project #1

Please see the readme for more information about our source code and how to run it.

1.

We wanted to separate messages as a unique entity because a user could have arbitrarily many messages and because we will need to sort all messages together. We chose not to have a separate Location entity because 1) the relation between user and location is 1:1, 2) we may wish to sort or filter users by location, and 3) we did not wish to pay the computational cost of checking every existing location before creating a location when loading a source user (e.g. record\_000000.dat).



2. Table User(int id, string name, string city, string state)

Table Message(int year, int month, int day, int hour, int minute, string text, int user\_id (references User), int id)

Our reasoning here is similar to the above reasoning. We separated messages as a unique table because of the one to many relation between users and messages.

3. This task took 9.46798 minutes with our “/ruby/transform\_records\_program.rb” program.

4. Sorting all messages by a specified column took 17.7008 minutes. Sorting all users by a specified column took 9.933529 seconds. Both of these sorting processes were done with our ‘/ruby/sort\_records\_program.rb’ program. We sorted messages by both hour and minute to facilitate the querying by hour and minute in step 5. We sorted users by either id or location to facilitate queries on location and id in step 5.

5.

a) Our program ‘/ruby/query\_program.rb’ ran the first query in 1.981405 seconds and found 37 users from Nebraska.

b) Our program ‘/ruby/query\_program.rb’ ran the second query in 19.61333 seconds and found 1519 users who have messaged from 8am to 9am inclusive.

c) Our program ‘/ruby/query\_program.rb’ ran the third query in 23.75858 seconds and found 28 users from Nebraska who have messaged from 8am to 9am inclusive.

d) Our program ‘/ruby/query\_program.rb’ ran the fourth query in 27.113881 seconds and found that the user with ID 1941 was the Nebraskan with the most messages sent from 8am to 9am.

6. Let n = number of users, and m = number of messages

a) Our algorithm for this query has a worst-case time complexity of O(n). It ran quickly both because of the low complexity and because of the relatively low number of users (compared to messages).

b) Our algorithm for this query has a worst-case time complexity of O(m). It ran more slowly because of the large number of messages.

c) Our algorithm for this query has a worst-case time complexity of O(n \* m). It was slower than the above query but only slightly because of the low number of users.

d) Our algorithm for this query has a worst-case time complexity of O(n \* m). It was slower than query c) because an additional O(n) operation was performed after similar operations as performed in part c) (this additional step was necessary to find the maximum). Constant cost for each operation was also increased by having to track the running total of messages for each user.

These time differences show that worse time complexities can slow performance of algorithms, although the total number of users was low enough not to make this difference too severe.

(The B+ tree for the remaining two problems was created and used in our C# code, which can be found in the ‘BPlusTree’ folder)

7.

Generating a B+ tree for messages took 21.5 seconds for fan out of 10 and 23.4 seconds for a fan out of 200. Generating a B+ tree for Users took .4 seconds for fan out of 10 and .41 seconds for fan out of 200.

8.

Query (a) took .0046 seconds with a fan out of 10, and .0002 seconds with a fan out of 200.

Query (b) took .0646 seconds with a fan out of 10, and .0602 seconds with a fan out of 200.

Query (c) took .0618 seconds with a fan out of 10, and .0616 seconds with a fan out of 200.

Query (d) took .0074 seconds with a fan out of 10, and .0027 seconds with a fan out of 200.

We found two main things from these timings:

1. B+ trees are much faster than naïve sorts. This is in part due to their superior time complexity ( O(log(n)) time complexity for searching vs O(n) of the initial strategy).
2. Larger fan outs improve the speed of queries. This effect occurs because the tree produced ends up having fewer levels with a large fan out.