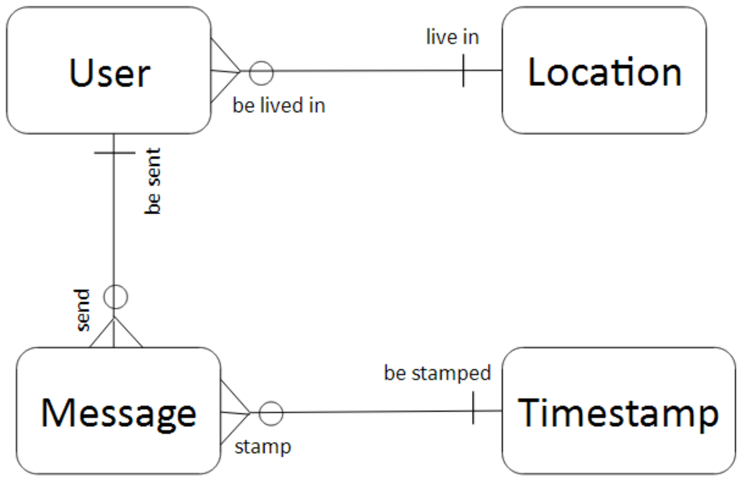
Project 1 (Part 1)

Team members:
Michael Hollman
Scott Johnson
Cassey Lottman
Darren Johnson

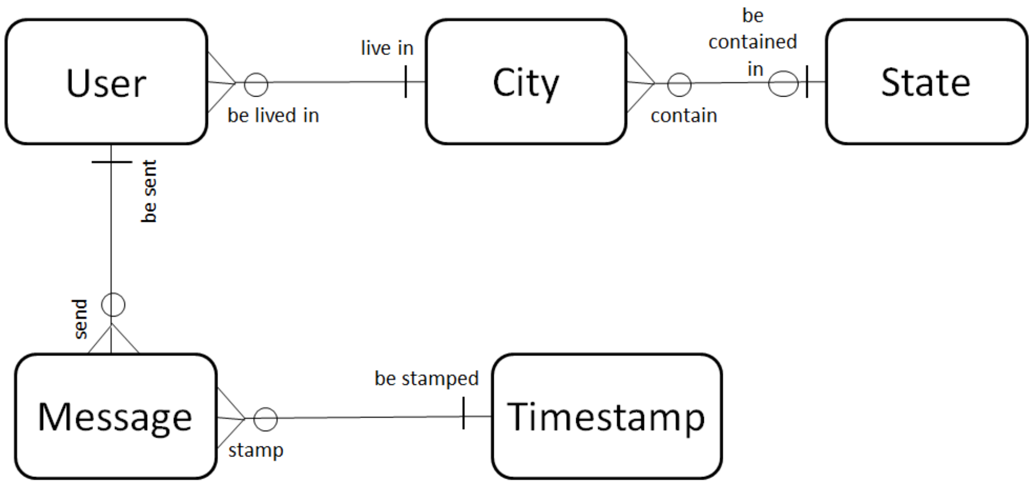
Step 1: Conceptual model

This initial entity relationship model conceptually represented the data without adding unneeded complexity. Each location can have multiple users who live in it, and each user must live in a location. Each user can have a number of sent messages associated with them, while each message has to have been sent by a user. Any given timestamp may have any number of messages associated with it, while each message must have a single timestamp representing when it was sent.

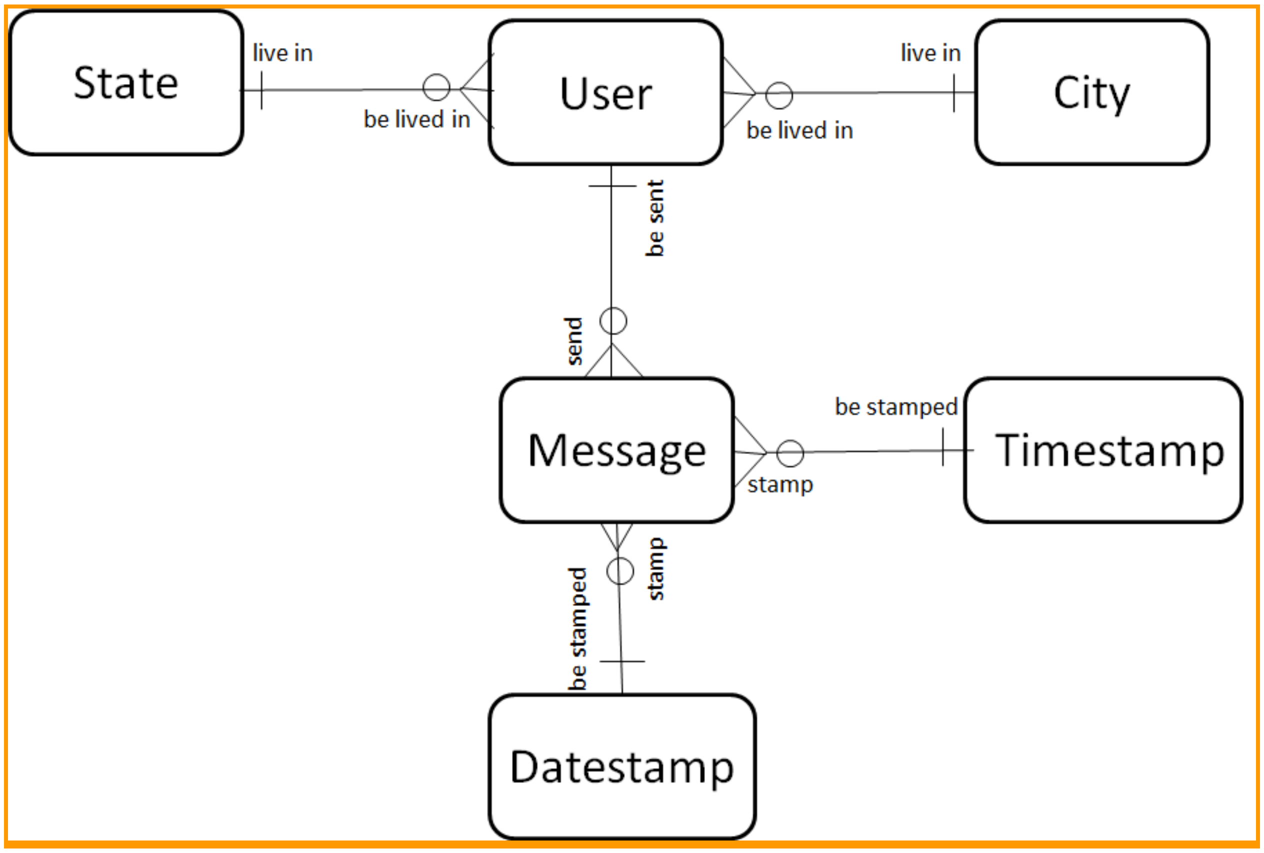


Slightly better model:

This model breaks out the “state” part of location into a separate table, helping to reduce repeating data in the location table. However, upon inspecting the data we realized that not all locations had a matching state with them, and that the additional complexity this would add to our queries was significant enough to make it not really worth it. Additionally, if the user lives in a state but the city is not provided, we could lose some data in this model because state is not tied to a user.



Final, Accepted Model:
The final model breaks out the “state” part of location into a separate table as well as the splitting the timestamp into a date (containing year, month and day) and a timestamp (containing hour and minute). These changes reduce repeating data in the location table and timestamp tables. Upon inspecting the data we realized that not all locations had a matching state with them. However, at times we may want to search by state, so it would be more efficient to split city and state to make searching easy and to eliminate more repeated data.



Step 2: Logical Modeling

First normal form:
USER (<u>UserID</u>, UserName, LocationID)
LOCATION(<u>LocationID</u>, City, State)
TIMESTAMP(<u>TimeStampID</u>, Year, Month, Day, Hour, Minute)
MESSAGE (<u>Text ID</u>, UserID, TimeStampID, Text)

This is the relational notation version of our first ER model. It attempts to minimize the amount of repeated data and more adequately allows for one-to-many relationships.

This model eliminates several repeating groups:

- One user could have multiple messages but we don't know how many, so we put messages in their own table with a foreign key that links back to the User who sent it.
- Locations also repeat - we put location in a separate table so we only have to save the location once even if there are 5000 users from New York City, New York, and provided a foreign key in User that links to the Location.
- Timestamps also can repeat, as multiple messages could be sent at the same time by different users. We thus split timestamp into a separate table, and set a foreign key in message to link to timestamp.

- Primary key - we made IDs the primary key for all the tables because using a natural key could raise a number of issues, such as certain potential natural keys not being unique, not being persistent (a user's name might change after a marriage, for example), or may not be applicable to all data members in the table. (Not all locations will have both a city and state, depending on the country of origin.)
- We split location into city and state inside the table, as it made the logic slightly more intuitive to find users from a specific state.

Third normal form (Final model):
USER (UserID, Name, CityID, StateID)
CITY (CityID, Name, StateID)
STATE(StateID, Name)
TIMESTAMP(TimeStampID, Hour, Minute)
DATESTAMPID(DateStampID, Year, Month, Day)
MESSAGE (MessageID, UserID, TimeStampID, DateStampID, Text)

In its third normal form, very little data is repeated and one-to-many relationships are represented by foreign keys to different tables to avoid repeating groups within a table row.

This model eliminates repeating groups.
Locations are separated into city and state to further reduce repeated data and to make it more efficient to sort/search by state.
Timestamps also can repeat, as multiple messages could be sent at the same time by different users. Therefore timestamps go in their own table. Times will be separated by time and date, in order to further eliminate repeating groups. This will also help optimize our queries, since for this assignment we want to query by time, and don't really care about the date.

For the primary keys, we could have used Name as the PK of State. However, all of our other tables use ID as a surrogate key. We decided to keep using id to maintain consistency and to save space. (With about 50 states, we can have a 2 digit ID number, which will save space versus a long string when we use state's id as a foreign key in other tables.

Note: We have listed a few alternative models at the end of this document.

Step 3: Preliminary Physical Modelling (1)
Note: all following timing results were run on OS X 10.9.
See readme.txt in the submitted files for a brief explanation of the various submitted source files.

Total Process time for creating tables: 208.460437 seconds

Number of files created:

Users:	40000
Cities:	18986

States:	53
Messages:	1992168
Timestamps:	1440
Datestamps:	336
Total:	2052983

Step 4: Preliminary Physical Modelling (2)
Sorting Tables:

Table	**Sorted By**	**Time**
Users	Id	0 (sorted upon creation)
Cities	StateId (numeric), then Name (alpha)	6.518334 seconds
States	Name (alphabetically)	0.041493 seconds
Messages	Id	0 (sorted upon creation)
Timestamps	Hour, then Minute (numerically)	0.473892 seconds
Datestamps	Year, month, then day (numerically)	0.139414 seconds

All of our sorting algorithms use C's built-in quicksort algorithm, which is O(nlogn). However, our Users and Messages were already sorted by Id and named as such, so we didn't have to re-sort them.

Users were left sorted by Id because the potential advantage of sorting them by something such as state was negligible for the queries we had to conduct.

Cities were sorted by StateId, but this decision was fairly arbitrary since none of the 4 queries depend on city. An alternative sorting would be alphabetically by name.

States were sorted by name, since this allowed us to binary search for any given state by name.

Messages were left unsorted, which meant they were naturally sorted by id. Theoretically, messages could be sorted by time, which could potentially speed up queries B, C, and D, but it made little sense to us to sort approximately 1.9 million files by timestamp.

Timestamps and datestamps were both sorted numerically so that they could be binary searched.

Step 5: Queries

Query	**Complexity**	**Result**	**Processing time**
A	O(log(S) + U)	594 users	9.935137 seconds
B	O(log(S) + U + T + M)	30863 users	381.170013 seconds
C	O(log(S) + U + T + M)	457 users	375.375885 seconds
D	O(log(S) + U + T + M)	UserId: 21547 (9 messages)	374.926666 seconds

Part 6: Complexity Comparison
Query a is O(log(S) + U) where U is the total number of users. Log(S) accounts for the binary search through states, where S is the number of states, to get the ID of Nebraska. Then it loops through each user.

Query b is O(log(S) + U + T + M) where U is the total number of users, T is the total number of distinct timestamps, and M is the total number of messages. This is because it loops through U times to initialize one array and then loops through T times to initialize another array. Then it loops approximately log(T) times for the binary search of Timestamp but in the worst case (all times are between 8 and 9) it loops through T times (however, this goes away because it is less than a constant multiple times T). Then it loops through each message. Log(S) accounts for the binary search through states.

Query c is O(log(S) + U + T + M) where U is the total number of users, T is the total number of distinct timestamps, and M is the total number of messages. This is because it loops through T times to initialize an array. Then it loops through each user. Then it loops approximately log(T) times for the binary search of Timestamp but in the worst case (all times are between 8 and 9) it loops through T times (however, this goes away because it is less than a constant multiple times T). Then it loops through each message. Log(S) accounts for the binary search through states.

Query d is O(log(S) + U + T + M) where U is the total number of users, T is the total number of distinct timestamps, and M is the total number of messages. This is because it is the same as query c but with the added calculation of checking to see if the user of a valid message is the new max. However this is less than a constant multiple times (U + T + M) so the time complexity stays the same.

You would expect that the first query would take less time and the last three queries would take about the same time. This is what actually happened, but the fact that the last three queries were so much slower than the first shows how expensive IO is for reading the files from the message table. There were considerably more messages and timestamps than there were users, which explains why the times for queries B, C, and D were an order of magnitude greater than the time for query A.

Step 07

In all cases it took longer to run the queries in Project 1 than it did in Assignment 1, with the last three queries taking significantly longer.

Query	Assignment 1	Project 1
A	4.403710 Seconds	9.935137 seconds
B	4.348595 Seconds	381.170013 seconds
C	4.121144 Seconds	375.375885 seconds
D	4.492406 Seconds	374.926666 seconds

The largest change between Assignment 1 and Project 1 was caused by the increased number of File I/O operations that had to be done due to our breaking the data out into numerous separate tables as opposed to the single table structure we started with. Though the way we constructed the data was more logical and eliminated repeating groups, it also resulted in vastly more files that had to be iterated through for queries 2-4 (which required iterating through each message, as opposed to iterating through each user file, with there being around 50 times more message files than user files).

Breaking things into separate tables helps when it reduces the number of files that have to be opened for queries, but in the case of this project we broke each piece of data into a separate file, which drastically increased the amount of file I/O that had to be performed, reducing the performance of our queries. In stage 2 we will implement a better way of organizing the data that may result in the performance increases that we would eventually expect to see from breaking things into a more logical tabular form. .