A General Framework for Geo-Social Query Processing : An Experimental Evaluation

Spiros Desyllas, University of Piraeus

GeoSocial networks are systems consists of geo and social modules as presented in the work of Armenatzoglou et al.[**?**]. These modules work independently and collaborate in order to provide a framework of general Geo-Social query processing. In this paper we will try to evaluate this work by implementing a GeoSN network build upon NoSql Spatial capabilities. We examined different algorithm variations and measured the effectiveness of each in a thorough experiment. Finally we concluded by presenting which are the best algorithms for big amount of data.

1. INTRODUCTION

A Geo-Social Network [**?**] is a system architecture that we can use in order to achieve a basic Geo-Social mechanism. According to this architecture a GeoSN network consists of :

There are some primitive queries that a spatial database can perform such as:

— Social Module
— Geo Module
— Query Processing Module

The proposed architecture consists of three modules, depicted in Figure 1: a social module (SM), a geographical module (GM), and a query processing module (QM). The SM stores exclusively social data (e.g., friendship relations), whereas the GM keeps only geographical information (e.g., check-ins). The QM is responsible for receiving GeoSN queries from users, executing them, and returning the results. The users do not communicate directly with the SM and GM. The SM, GM and QM can either be three separate servers, three separate clouds, or a single system (server or cloud). However, the tasks of the three modules are segregated.

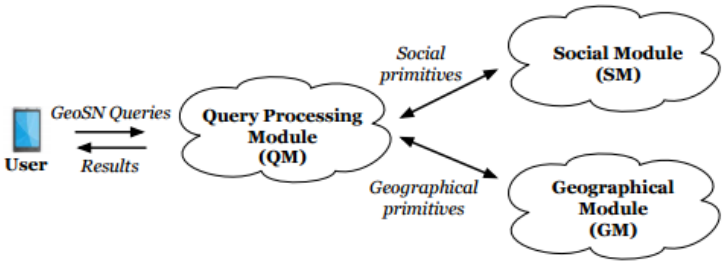

Fig. 1. Proposed GeoSN architecture

A separation like this can provides numerous benefits such as administration by different entities.The separation of QM enables third-party companies that do not own any social or geographical data to implement GeoSN queries by solely interacting with the APIs of SM and GM. In addition, separating the functionality of SM and GM renders the management of social and geographical data more flexible, because the frequent check-in updates do not burden the relatively static social structures.

Finally, the segregation offers several other benefits. First of all the studied architecture can readily integrate modifications (e.g., a new, more efficient structure) in the

implementation of SM without modifying GM, and vice versa. Second, novel GeoSN query types and algorithms can be devised, either by using a different combination of existing primitives or by implementing new ones, without the need of altering the SM and GM infrastructures. Last, social (geographical) data can be used independently for pure social (resp. geographical) queries, potentially through the same primitive operations utilized by GeoSN queries. As a result, a traditional social network can adopt this architecture without extra effort.

2. ALGORITHMS AND VARIATIONS

The following algorithms are implemented in this paper in order to achieve complex queries like RangeFriends and NearestFriends

2.1. Range Friends Algorithms

Problem formulation. Simply stated, RF returns the friends of user u that are within distance r to a location q. More formally: PROBLEM 1. Given a user u, a 2D point q and a positive real number r, a Range Friends (RF) query RF(u, q, r) returns a set R defined as follows:

$$R = ui | AreFriends(u, ui) || q, ui || r$$

Similar to RangeUsers and NearestUsers, the result contains also the users locations. For example, RF(u4, q, 8) = u2 and RF(u5, q, 10) = u1, u3, where u1, u2, u3 carry both their ID and location. This may be particularly useful for other GeoSN queries that use RF as a subroutine, while it comes with a free asymptotic cost (since the locations must be retrieved to answer the query anyway, and do not add to the space complexity of the result). We next describe three solutions for the RF query, whose pseudocode is given in Figure 2.

Input: User u, location q, radius r
Output: Result set R

/* **Algorithm 1** (RF_1) */
1. $F = GetFriends(u)$, $R = \emptyset$
2. **For** each user $u_i \in F$
3. $\quad GetUserLocation(u_i)$
4. $\quad$ **If** $||q, u_i|| \leq r$, add u_i into R
5. **Return** R

/* **Algorithm 2** (RF_2) */
1. **Return** $R = GetFriends(u) \cap RangeUsers(q, r)$

/* **Algorithm 3** (RF_3) */
1. $U = RangeUsers(q, r)$, $R = \emptyset$
2. **For** each user $u_i \in U$
3. $\quad$ **If** $AreFriends(u, u_i)$, add u_i into R
4. **Return** R

Fig. 2. Pseudocode of RF algorithms

Algorithm 1 (RF1)
This variant first extracts the set F of us friends invoking primitive $GetFriends(u)$ in Line 1. Subsequently (Lines 2-4), for every ui F, it retrieves its location via primitive $GetUserLocation(ui)$, and inserts ui in result set R if the distance between ui and q is

smaller than or equal to r. For example, $RF1(u4, q, 8)$ first computes $F = u2, u3, u6$ and retrieves the user locations. Then, it adds only u2 to R, since $||q, u2|| = 6.48(||q, u3|| = 10 > 8, and||q, u6|| = 12.2 > 8)$.

Algorithm 2 (RF2)
This algorithm gets the friends of u through $GetFriends(u)$, and executes $RangeUsers(q, r)$ to get the users that are within distance r to q. Finally, it performs an intersection between these two sets, which yields the result R. For instance, $RF2(u4, q, 8)$ performs $GetFriends(u4)RangeUsers(q, 8) = u2, u3, u6u1, u2 = u2$.

Algorithm 3 (RF3)
RF3 calculates $U = RangeUsers(q, r)$, and then inserts $uiUintoRifAreFriends(u, ui) = true$. In our running example, $RF3(u4, q, 8)$ first executes $RangeUsers(q, 8) = u1, u2$, and then calculates $AreFriends(u4, u1) = false$ and $AreFriends(u4, u2) = true$. Consequently, the algorithm returns $R = u2$ as the result.

The above algorithms have important differences. For instance, RF2 and RF3 necessitate a spatial index for efficient range query processing. RF3 could also benefit from an adjacency matrix implementation (because it invokes AreFriends numerous times). In addition, as we demonstrate in our experiments, the machine architecture (centralized or distributed) has a significant impact on their relative performance. Finally, the data and query parameters are also vital in determining the best algorithm, e.g., if there are few users within a range, RF2 and RF3 are preferable to RF1, while RF1 is better for sparse social networks of users in the same geographic area.

2.2. Nearest Friends Algorithms

Problem formulation. NF returns the k friends of user u that are closest to location q in ascending distance. Formally:

Input: User u, location q, positive integer k
Output: Result set R

/* **Algorithm 1** (NF_1) */
1. $F = GetFriends(u)$, $R = \emptyset$
2. **For** each user $u_i \in F$, compute $GetUserLocation(u_i)$
3. Sort F in ascending order of $||q, u_i||$
4. Insert the first k entries of F into R
5. **Return** R

/* **Algorithm 2** (NF_2) */
1. $F = GetFriends(u)$, $R = \emptyset$
2. **While** $|R| < k$
3. $u_i = NextNearestUser(q)$
4. **If** $u_i \in F$, add u_i into R
5. **Return** R

/* **Algorithm 3** (NF_3) */
1. $R = \emptyset$
2. **While** $|R| < k$
3. $u_i = NextNearestUser(q)$
4. **If** $AreFriends(u, u_i)$, add u_i into R
5. **Return** R

Fig. 3. Pseudocode of NF algorithms

PROBLEM 2. Given a user u, a 2D point q and a positive integer k, a Nearest Friends (NF) query NF(u, q, k) returns a list

For example, NF(u2, q, 2) = (u4, u6). Similar to the case of RF, the result incorporates the user locations. Figure 3 includes the pseudocode of three solution variants for NF, explained next

Algorithm 1 (NF1) NF1 first calculates F = GetFriends(u), and gets the location of every ui F via GetUserLocation(ui). Subsequently, it sorts F in ascending distance of each user therein to q, and inserts the first k entries of F into R. In our example, NF1(u2, q, 2) retrieves F = GetFriends(u2) = u4, u6, u9, extracts the user locations via three calls to GetUserLocation, sorts F in ascending distance of each ui F to q producing ordered list (u4, u6, u9). It finally returns the first 2 users in the list, i.e., R = (u4, u6).

Algorithm 2 (NF2) NF2 first extracts the friends F of u through GetFriends(u). It then iteratively retrieves the next nearest user ui to q by calling NextNearestUser . If ui is in F, it is added to R. When the size of R becomes equal to k, we are certain that we have evaluated the correct result. In Figure 2, NF2(u2, q, 2) evaluates F = GetFriends(u2) = u4, u6, u9. Then, it gets u1 as the result of the first call to NextNearestUser (q). Since u1 6 F, it is not added to R. Subsequently, it proceeds with retrieving users u2- u6 through 5 calls to NextNearestUser (q), and adds only u4 F and u6 F to R. At this point —R— becomes 2 and, thus, the method returns R to the user and terminates.

Algorithm 3 (NF3) NF3 is similar to NF2, but instead of invoking GetFriends, it utilizes AreFriends for checking the friendship between a user retrieved by NextNearestUser and u. In our running example, NF3(u2, q, 2) iteratively computes users u1-u6 via six calls to NextNearestUser , and performs an AreFriends primitive for each of them. During this process, it adds u4, u6 to R (since only those are friends with u2), and concludes (—R— = 2). Similar to the RF algorithms, the relative performance of the NF solutions depends on the implementation, existing indexes, machine architecture, data distribution, and query parameters

3. MODULES IMPLEMENTATION

After examining the algorithm variations for these two type of queryes (RF and NF) we proceed to the actual implementation of the modules. For the implementation we used Ruby[**?**] programming language for coding the api of the modules and the middleware of the database and MongoDB[**?**] in order to store and index data. We used MongoDB indexes in order to store social and geo data. For the social module we used MongoDB's single field to index the UserId and for the geo module we used GEO2D (2d Sphere) to index user location. We used GeoJSON[**?**] to store user location as it is supported by MongoDB geospatial indexes. With the use of GEO2D index the Geo Module can perform the following basic geospatial queries as presented by Theodoridis et al. [**?**]

— Point Location Query
— Range Query
— Join Query
— Nearest Neighbor Query

3.1. Social Module

The social module consists of a ruby class file which provide the following methods: $getFriends(userid)$ and $areFriends(userid_1, userid_2)$. These two methods implement the primitive queries and mechanics for the social module. The $getFriends(userid)$ takes a userId as input and returns all the friends of the given user id in the form of

a JSON [**?**] array. The $areFriends(userid_1, userid_2)$ method takes the ids of two users and returns $True$ if the two users are friends and false if not

3.2. Geo Module

The geo module consists of a ruby class file which provide the following methods: $nearestUsers(q, k)$, $getUserLocation(u)$ and $rangeUsers(q, r)$. These methods implement the primitive queries and mechanics for the geo module. nearestUsers takes a query point q and an integer k as input and returns the k users nearest to q in in ascending distance, along with their locations in a form of a JSON[**?**] array. getUserLocation takes user id as an input and returns the user's location in the form a pair coordinate (latitude and longitude). Finally rangeUsers takes a query point q and distance in meters r as input and returns the users within distance r from q, along with their locations, again in a form of JSON[**?**] array.

3.3. Query Processing Modules

The query processing module is a ruby class file which actually uses the above implemented modules in order to achieve more complex queries based on the primitive ones of the previous mentioned modules. This module initially makes a connection to the social and geo module in order to be able to used them. We stored geo and social data in MongoDB as already mentioned so this module uses Mongod Ruby driver in order to connect to MongoDB. Some portions of the driver are implemented in c++ for performance reasons. The methods implemented in this module are : rangeFriends and nearestFriends. We developed 3 different variations for each method based on the algorithms that we presented in the section above. Method rangeFriends takes a user u , a 2D point q and a distance in meters r as input and returns all friends within the given range as a JSON array. Method nearestFriends takes User u and a positive integer k as input and returns a JSON array consisting friend's Id and friend's distance from q in ascending order.

4. EVALUATION AND RUNNING EXAMPLES

In order to initially provide synthetic data sets to modules we implemented a seed ruby program that randomly created users and location based on random latitude and longitude. Each user has the two previous and two next created as friends. We then run a warm up script in order to perform some queries to MongoDB in order to warm up caches. The first test run included 100 users and the second run test included 10000 users. We wanted to test algorithm variations with a small and a big data set to test if their performance was influenced by the size of data. All the tests war run using an AMD A6 Dual Core Cpu clocked at 4.1Ghz and 4GB Ram. The datasets were stored in a regural HDD drive and MongoDB was set up in a Linux Ubuntu desktop distribution.

5. SMALL DATASET MEASUREMENTS

In the following tables we can see the test results for the small dataset of the 100 users for the RangeFriends and Nearest Friends query processing

Range Friends (RF) - 100 Records

Aglorithm Variation	Execution Time (ms)
A	2.33
B	9.64
C	52.42

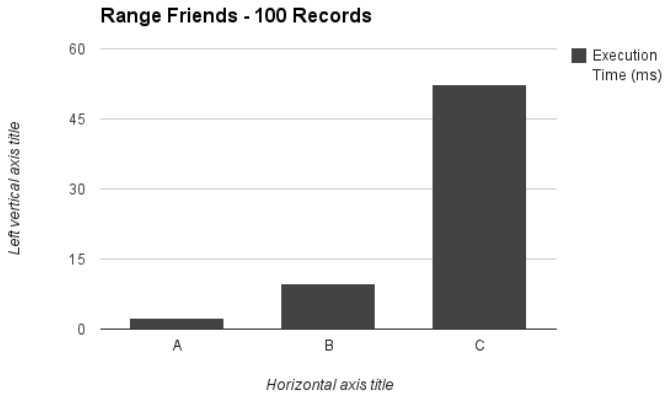

Fig. 4. Range Friends (RF) - 100 Records

Nearest Friends (NF) - 100 Records

Aglorithm Variation	Execution Time (ms)
A	1.34
B	3.73
C	47.11

6. BIG DATASET MEASUREMENTS

In the following tables we can see the test results for the big dataset of the 1000 users for the RangeFriends and Nearest Friends query processing

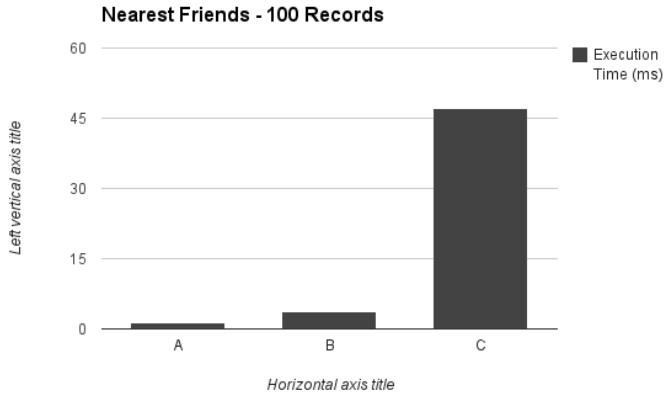

Fig. 5. Nearest Friends (NF) - 100 Records

Range Friends (RF) - 10000 Records

Aglorithm Variation	Execution Time (ms)
A	8.37
B	298.43
C	4529.02

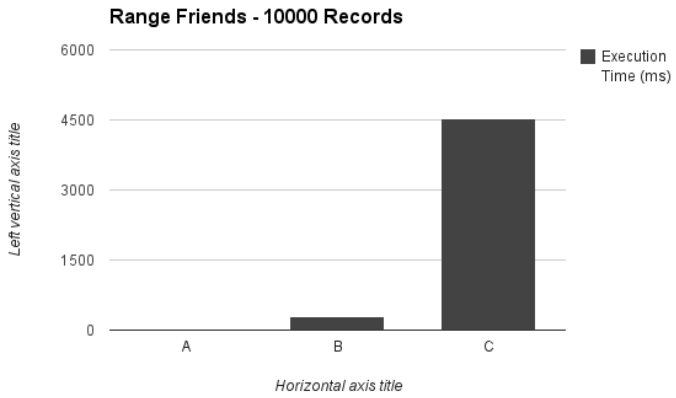

Fig. 6. Range Friends (RF) - 10000 Records

Nearest Friends (NF) - 10000 Records

Aglorithm Variation	Execution Time (ms)
A	8.2
B	389.62
C	4856.17

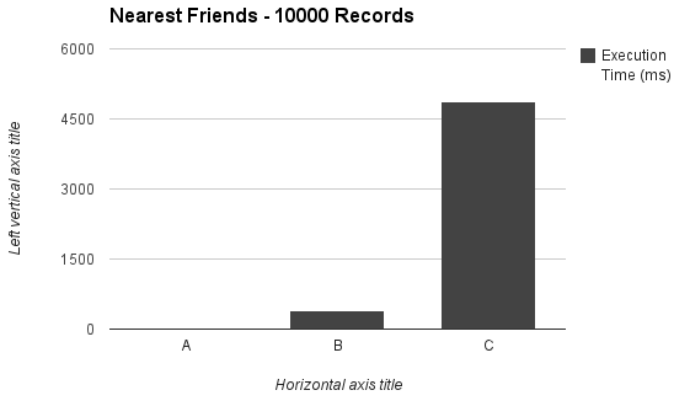

Fig. 7. Nearest Friends (NF) - 10000 Records

7. CONCLUSIONS

In this paper, we examined and experimented with a implementation of a Geo-Social network query processing system based on MongoDB geospatial indexes and programmed with Ruby. We evaluated 3 algorithm variations for each GeoSN query based on primitives queries implemented by primitive modules and evaluated each algorithm implementation by measuring it's process time against small and big amount of data. The queries that we experimented on were RangeFriends (RF) and NearestFriends (NF), but a similar work needs to be done for GeoSN query NearestStarGroup (NSG). We used NoSql spatial capabilities but this experiment could be run on a classic relational DMBS such us PostgreSQL.

REFERENCES

Geojson. http://geojson.org/.

Json. http://json.org/.

Mongodb. http://www.mongodb.org/.

Ruby programming language. https://www.ruby-lang.org/en/.

N. Armenatzoglou, S. Papadopoulos, and D. Papadias. A general framework for geo-social query processing. *Proc. VLDB Endow.*, 6(10):913–924, Aug. 2013.

A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest pair queries in spatial databases. *SIGMOD Rec.*, 29(2):189–200, May 2000.