A General Framework for Geo-Social Query Processing

Nikos Armenatzoglou Stavros Papadopoulos Dimitris Papadias

Department of Computer Science and Engineering
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
{nikos, stavrosp, dimitris}@cse.ust.hk

ABSTRACT

The proliferation of GPS-enabled mobile devises and the popularity of social networking have recently led to the rapid growth of *Geo-Social Networks* (GeoSNs). GeoSNs have created a fertile ground for novel location-based social interactions and advertising. These can be facilitated by GeoSN queries, which extract useful information combining both the *social relationships* and the *current location* of the users. This paper constitutes the first systematic work on GeoSN query processing. We propose a *general framework* that offers flexible data management and algorithmic design. Our architecture *segregates* the social, geographical and query processing modules. Each GeoSN query is processed via a transparent combination of *primitive queries* issued to the social and geographical modules. We demonstrate the power of our framework by introducing several "basic" and "advanced" query types, and devising various solutions for each type. Finally, we perform an exhaustive experimental evaluation with real and synthetic datasets, based on realistic implementations with both commercial software (such as MongoDB) and state-of-the-art research methods. Our results confirm the viability of our framework in typical large-scale GeoSNs.

1. INTRODUCTION

A *Geo-Social Network* (GeoSN) couples social network functionality with location-based services. Specifically, a GeoSN is a graph, where nodes represent users and edges correspond to friendship relations. Moreover, through GPS-enabled mobile devices, users publish their current geographical location to their friends, by "checking-in" at various places. The most popular GeoSN to date, Foursquare [5], accommodates over 30M users, and receives millions of check-ins per day [8]. In addition, more "traditional" social networks, such as Facebook and Twitter, have been recently augmented with check-in functionality.

This trend in geo-social networking has created opportunities for novel location-based social interactions and advertising. For instance, services like Facebook's *Nearby* and Foursquare's *Radar* return the friends that recently checked-in at close proximity to a user's current location. In addition, Foursquare has joined forces with GroupOn to provide offers from stores to nearby users [13]. We expect more advertising applications to gravitate towards exploiting both the geographical and social information of GeoSNs.

In this paper we focus on GeoSN queries that extract useful information combining both the *social relationships* and the *current* (i.e., lastly posted) *location* of the users. Despite their importance, there is a limited literature on GeoSN query processing. In industry, to the best of our knowledge, there are no white papers documenting the processing of queries such as *Radar* and *Nearby*. On the other hand, the few existing academic works [21, 39, 31, 38] focus solely on the algorithmic part, overlooking critical data management issues, namely that (i) the data storage methods greatly influence the performance of a GeoSN algorithm, and (ii) the social and geographical data may be administered by different entities.

Towards this end, we introduce a general framework for GeoSN query processing. Specifically, we propose an architecture that *segregates* the social, geographical, and query processing modules. Each GeoSN query is processed via a transparent combination of *primitive queries* issued to the social and geographical modules, which do not interact with each other. This allows separate (and thus more flexible) social and geographical data management, and permits each module to be optionally operated by a different entity. Moreover, it renders the implementation of each module orthogonal to the architecture. As such, any existing technology (e.g., cloud computing, graph databases, sophisticated spatial structures, etc.) or future advances can be easily integrated into any module.

We first address two "basic" GeoSN queries; *Range Friends* (RF), which returns the friends of a user within a given range, and *Nearest Friends* (NF), which returns the nearest friends of a user to a given location. We design various algorithms for every query type, with each algorithm utilizing a different combination of primitive queries. We show that (i) our framework provides flexibility in algorithmic design, and (ii) the algorithm efficiency heavily depends on the number and performance of the involved primitives, which in turn depends on the underlying data management scheme.

In addition, we propose a novel, more "advanced", GeoSN query type, called *Nearest Star Group* (NSG). Given a query location represented as a 2D point q and an integer m, an NSG query returns a user group of size m, which (i) forms a star subgraph of the social network, and (ii) minimizes the aggregate (Euclidean) distance of its members to q. As an example, consider that a restaurant has a 4-seat table available, and wishes to send a GroupOn-like offer (e.g., *the next group of four people who come to the restaurant will receive a 20% discount*). With $m = 4$ and q being the location of the restaurant, an NSG query would return the nearest group of 4 users to the restaurant, who are connected through a *common friend* (the center of the star). By focusing only on such groups, the GeoSN can (i) increase the effectiveness of the advertisement, avoiding overwhelming remote or socially unconnected groups with unin-

teresting offers, and (ii) minimize the cost by contacting only the star centers, who are likely to influence the other group members. We show that NSG runs in polynomial time, and introduce several implementations that comply with our framework.

Finally, we provide an exhaustive experimental evaluation of all proposed schemes, using real and synthetic datasets. In particular, we experiment with various storage implementations for both the social and geographical data, including commercial databases such as MongoDB [17] (used by Foursquare [19]), as well as state-of-the-art research schemes. We also test with two system settings; the first positions all modules at one machine, whereas the second utilizes a separate machine for each module.

Our contributions are summarized as follows:

- We propose the first general framework for GeoSN query processing, which enables flexible data management and algorithmic design.
- We introduce novel basic and advanced GeoSN queries, and devise various algorithms for solving them.
- We include thorough experiments with diverse implementations/architectures, using real and synthetic datasets.

The remainder of the paper is organized as follows. Section 2 overviews related work. Section 3 introduces our general framework. Section 4 presents our GeoSN query algorithms. Section 5 evaluates the proposed techniques with comprehensive experiments. Section 6 concludes our work.

2. RELATED WORK

We describe the relevant work on GeoSN query processing in industry and academia, and orthogonal topics on GeoSNs.

Industry. Foursquare's *Radar* and Facebook's *Nearby* return the friends who are currently in the vicinity of a user. A similar functionality is provided by Geoloqi [10], a platform for location-based services, which notifies a user when friends enter a certain range. FullCircle [9] detects groups of users (not necessarily friends) with similar interests and preferences within small distance of each other. Once a new group is found, the members are encouraged to communicate and form friendships. Hotlist [14] recommends to users various events based on their social relations (e.g., how many friends are also attending the event), preferences, and proximity to event locations. Since most of the above work is proprietary, processing algorithms are not documented.

Academic research. Huang and Liu [28] suggest a GeoSN query that returns to a user the friends that are nearby and share common interests, without providing concrete processing algorithms. [21, 39] aim at minimizing the communication cost for proximity detection among friends, by enabling users to issue location updates only when they exit certain safe regions. Liu et al. [31] propose the *k-Geo-Social Circle of Friend Query* (k-gCoFQ): given a weighted graph, a user u, and a positive integer k, the k-gCoFQ finds the group g of $k+1$ users, which (i) is connected, (ii) contains u, and (iii) minimizes the maximum distance between any two of its members (modeled as a weighted average of the Euclidean distance and a notion of social strength). [38] introduces the *Socio-Spatial Group Query* (SSGQ): given a query point q and two positive integers k, n, the SSGQ returns a group g of n users, such that (i) each user in g is socially connected with at least $(n-k)$ members of g, and (ii) the sum of distances of all members in g to q is minimized. The k-gCoFQ and SSGQ queries are NP-Hard, and their authors present approximation algorithms.

Orthogonal work. There are also other approaches that extract useful information from a GeoSN, but target at different settings to ours. In GeoFeed [23], a user receives a set of posts submitted by friends, whose geo-tagged location is within a specific area of interest. [37] predicts friendships based on past user locations. [41] recommends places and friends by taking into account the user location history. [30] performs quantitative analysis on geo-social data. [40] quantifies the influence of one user to another based on spatial and social criteria. [27, 26] process SQL-like queries on archived geo-social data. Finally, [36, 29] aim at answering GeoSN queries, while protecting user location privacy (e.g., via encryption).

Evidently, there is a narrow literature on GeoSN query processing in our setting, namely, [21, 39, 31, 38] described above. These works overlook crucial data management issues. In particular, they bundle their algorithms with specific data representations and indices, which may feature excessive costs in typical large GeoSNs. For instance, [31] employs an adjacency matrix for keeping info about the social graph, which may incur prohibitive storage overhead. [21, 38] make use of hybrid indices, incorporating both social and spatial data. Such structures may suffer from enormous maintenance costs due to high check-in rates. [38, 39] do not specify how the social graph is stored. We stress that the data representation scheme may greatly affect the performance of an algorithm. Finally, all the approaches essentially assume that all the data are owned by a single entity, and are accommodated by a single machine. In the next section we present a general framework for GeoSN query processing that overcomes these drawbacks.

3. FRAMEWORK

In Section 3.1 we describe our architecture, whereas in Section 3.2 we explain the query primitives that serve as building blocks of GeoSN query algorithms.

3.1 Architecture

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*.

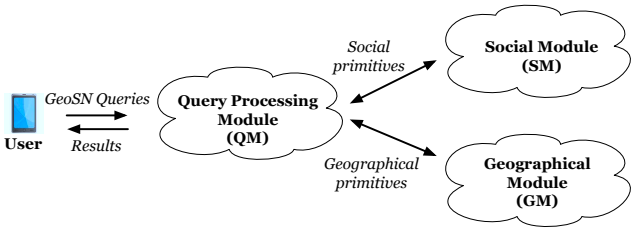


Figure 1: Proposed GeoSN architecture

The SM and GM do not interact with each other, but rather communicate only with the QM. More specifically, the QM processes a GeoSN query through an algorithm that builds upon well-defined social and geographical *primitive queries*. These primitives define a rigorous interface between QM-SM and QM-GM, respectively. The sole duties of the SM and GM (with respect to a given GeoSN

query) are to execute their corresponding primitives on their stored data. The QM eventually derives the final results by combining the outputs of the primitives, optionally exploiting auxiliary indices maintained locally.

The segregation of SM and GM allows their administration by different entities, e.g., the SM (GM) can be maintained by a company with expertise in social networking (resp. location-based services). For instance, in UK and Japan, Facebook Places [3] cooperates with Factual [4], which provides infrastructure for location-based services. Glancee [12], a location-based service app, uses Facebook's social graph to connect nearby users. Another example is the cooperation of pure commercial social networks, e.g., Twitter or Facebook, and GeoSNs like Foursquare. A user who has both a Twitter or Facebook and a Foursquare account can post his Foursquare check-in at Twitter or Facebook [15]. Thus, if Facebook or Twitter needs the geographical information of users' check-ins to execute a GeoSN query, it obtains it from Foursquare. 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 (e.g., Agora [1]).

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. For example, due to an unexpected high rate of check-ins recently, Foursquare's system had a very long downtime. The problem was caused because their data are spread across multiple balanced database shards. When a shard is overused, a new one is added, followed by rebalancing. The rebalancing of the entire database caused the crash [6]. In a segregated system, such a crash in GM would not affect SM.

Finally, the segregation offers several other benefits. First, our 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 our architecture without extra effort.

3.2 Primitive Queries

Here we introduce the primitive social and geographical queries that are used as fundamental components in our GeoSN algorithms (presented in Section 4). These operations are supported by all typical graph and spatial data structures, and can be easily integrated with any SM and GM implementation.

We make use of the following social primitives:

- $GetFriends(u)$: Given a user u, return u's friends.
- $AreFriends(u_i, u_j)$: Given two users u_i, u_j, return $true$ if u_i, u_j are friends, and $false$ otherwise.

We also utilize the geographical primitives below. Note that every location is regarded as a (x, y) pair of coordinates in some fixed Cartesian plane. When we refer to a user's location, we mean the (x, y) coordinates associated with his/her *current* (i.e., lastly posted) check-in. We assume Euclidean space, but the general framework and query processing techniques also apply for road networks (the GM should simply integrate an indexing scheme that supports the primitives according to the network distance, e.g., [35]).

- $GetUserLocation(u)$: Given u, get u's location.
- $RangeUsers(q, r)$: Given a query point q and a real number r, return the users within distance r from q, along with their locations.
- $NearestUsers(q, k)$: Given a query point q and an integer k, return the k users nearest to q in ascending distance, along with their locations.

Observe that $RangeUsers$ and $NearestUsers$ return also the location of each user in the result. This could be very useful for the algorithms at QM that employ these primitives, while it does not affect the computational and space complexity of the result.

All the above primitives can be easily supported by the API of SM and GM. For instance, $GetFriends$ is readily implemented in the API of Facebook, whereas the API of Foursquare offers $GetUserLocation$. We do not exclude the existence of additional primitives. Nevertheless, any primitive must be treated as an *atomic* operation. To the best of our knowledge, operations that maintain *state* for future primitive invocations (e.g., an incremental version of a nearest neighbor query that extracts the *next* best result upon a new query) are not supported by any commercial GeoSN. The main reason is that maintaining state (e.g., via priority queues) for a large number of simultaneous queries is prohibitively expensive.

The efficiency of the primitives depends on the underlying storage scheme employed by SM and GM. For instance, representing the social graph by adjacency lists is preferable for $GetFriends$ (the output simply consists of the users in the list of u), whereas adjacency matrices are faster for $AreFriends$ (the output is $true$ if the bit at cell (i, j) of the matrix is 1). Similarly, although all common spatial indices support $RangeUsers$ and $NearestUsers$, they feature differences in performance.

Unfortunately, there is no unanimously accepted social or spatial storage implementation. To elaborate, Facebook uses adjacency lists stored in Memcached [16], a distributed memory caching system, whereas Foursquare uses MongoDB [17], a document-oriented database. Moreover, Twitter employs the R*-Tree spatial index [20], whereas Foursquare adopts the grid-based geohashes of MongoDB [11]. Similarly, academic research has utilized a wide variety of approaches; [27] uses adjacency lists stored in Neo4j [18] (a graph database), [31] employs an adjacency matrix, whereas [26] utilizes relational tables for storing the friendship relations; moreover, [23] indexes check-ins with a grid, [21] applies a Quad tree, while [31] exploits the R*-Tree. We stress that every GeoSN algorithm should be tailored to a specific SM and GM instantiation, selecting the combination of primitives that leads to maximum query efficiency. In the next section, we explain that a variety of GeoSN algorithms can be implemented using the described primitives.

4. QUERY PROCESSING

We study three GeoSN query types; *Range Friends* (RF) in Section 4.1, *Nearest Friends* (NF) in Section 4.2, and *Nearest Star Group* (NSG) in Section 4.3. There are various ways to process a GeoSN query using primitives. For each query type, we introduce algorithms that use different combinations of primitives, *without* requiring the existence of a sophisticated *hybrid* index at the QM.

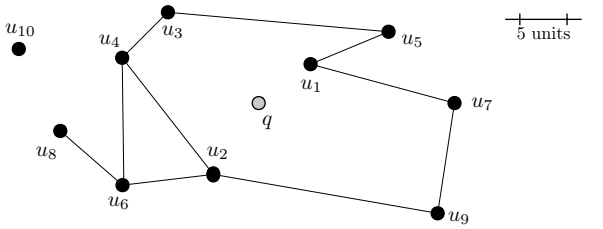
Notation. Symbols in Sans Serif typeface designate a query type, e.g., NF, sets are represented by symbols like S, $|S|$ denotes the size of S, and sets of sets appear in calligraphic form, e.g., $\mathcal{S}$.

To considerably simplify our notation throughout the section, we use symbol u to denote a user (ID, location) pair, but we do *not* use separate symbols for user ID (e.g., $u.id$) and location (e.g., $u.l$). This calls for some clarifications regarding the primitive queries explained in Section 3.2. $GetFriends$ returns only a list of user

IDs, since the social module SM does not possess any spatial data. Therefore, when we write $u_i \in F$ where $F = GetFriends(u)$, the location attribute of u_i is *null*. $GetUserLocation(u_i)$ results in filling the (*null*) location of u_i. On the other hand, when we write $u_i \in U$, where $U = RangeUsers(q, r)$, u_i encompasses *both* an ID and location.

Finally, we denote the Euclidean distance of a user u to q by $||q, u||$. Note though that, in order to execute this operator, u's location must not be *null*.

Running example. Throughout this section, we use Figure 2 as a running example, where the black points refer to the locations of 10 users $\{u_1, u_2, u_3, u_4, u_5, u_6, u_7, u_8, u_9, u_{10}\}$, the edges represent the social relations among them, and the grey point is an arbitrary query location q. The table contains the distance of each user to q.



	u_1	u_2	u_3	u_4	u_5	u_6	u_7	u_8	u_9	u_{10}
$\|\|q, u_i\|\|$	5	6.4	10	10.4	11.3	12.2	15	15.5	15.9	18.9

Figure 2: Running Example

A useful subroutine. Several algorithms presented next require the *incremental* retrieval of the "*next nearest user*" to a location q. This can be implemented via a subroutine called $NextNearestUser(q)$. However, such an operation necessitates the maintenance of *state* (e.g., the lastly retrieved nearest user, or some heap information). Recall from Section 3.2 that the primitives cannot keep state at SM and GM. Hence, $NextNearestUser(q)$ cannot be regarded as a primitive query, but rather it should be implemented at QM.

A way to do this is via repeated calls to $NearestUsers(q, k)$ as follows: we initialize $k = 0$ and, every time we need the next nearest user to q, we call $NearestUsers(q, k)$ *after incrementing* k by 1. Since the result is sorted in ascending distance to q, the next nearest user is the last entry of the list. However, this would incur considerable result overlap in successive calls. A solution to this problem is to increment k by a tunable step $s \geq 1$. Then, QM can keep the s newly retrieved users in a local *queue*, sorted in ascending distance to q. Whenever $NextNearestUser$ is invoked, it will pop the head of the queue and return it to the user. When the queue is empty, another execution of $NearestUsers(q, k)$ is necessary, after adding s to the current k.

4.1 Range Friends (RF)

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:*

$$R = \{u_i \mid AreFriends(u, u_i) \wedge ||q, u_i|| \leq r\}$$

Similar to $RangeUsers$ and $NearestUsers$, the result contains also the users' locations. For example, $RF(u_4, q, 8) = \{u_2\}$ and $RF(u_5, q, 10) = \{u_1, u_3\}$, where u_1, u_2, u_3 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 3.

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

Figure 3: Pseudocode of RF algorithms

Algorithm 1 (RF_1). This variant first extracts the set F of u's friends invoking primitive $GetFriends(u)$ in Line 1. Subsequently (Lines 2-4), for every $u_i \in F$, it retrieves its location via primitive $GetUserLocation(u_i)$, and inserts u_i in result set R if the distance between u_i and q is smaller than or equal to r. For example, $RF_1(u_4, q, 8)$ first computes $F = \{u_2, u_3, u_6\}$ and retrieves the user locations. Then, it adds only u_2 to R, since $||q, u_2|| = 6.4 \leq 8$ ($||q, u_3|| = 10 > 8$, and $||q, u_6|| = 12.2 > 8$).

Algorithm 2 (RF_2). 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, $RF_2(u_4, q, 8)$ performs $GetFriends(u_4) \cap RangeUsers(q, 8)$ = $\{u_2, u_3, u_6\} \cap \{u_1, u_2\}$ = $\{u_2\}$.

Algorithm 3 (RF_3). RF_3 calculates $U = RangeUsers(q, r)$, and then inserts $u_i \in U$ into R if $AreFriends(u, u_i) = true$. In our running example, $RF_3(u_4, q, 8)$ first executes $RangeUsers(q, 8)$ = $\{u_1, u_2\}$, and then calculates $AreFriends(u_4, u_1) = false$ and $AreFriends(u_4, u_2) = true$. Consequently, the algorithm returns $R = \{u_2\}$ as the result.

The above algorithms have important differences. For instance, RF_2 and RF_3 necessitate a spatial index for efficient range query processing. RF_3 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, RF_2 and RF_3 are preferable to RF_1, while RF_1 is better for sparse social networks of users in the same geographic area.

4.2 Nearest Friends (NF)

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

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 $R = (u_1, \ldots, u_k)$ such that, for each $1 \leq i \leq k$:*

$$AreFriends(u, u_i) \ \wedge ||q, u_i|| \leq ||q, u_{i+1}|| \ \wedge$$
$$(\nexists\, u' : u' \notin R \wedge AreFriends(u, u') \wedge ||q, u'|| < ||q, u_k||)$$

For example, $NF(u_2, q, 2) = (u_4, u_6)$. Similar to the case of RF, the result incorporates the user locations. Figure 4 includes the pseudocode of three solution variants for NF, explained next.

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. $\quad u_i = NextNearestUser(q)$
4. $\quad$ **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. $\quad u_i = NextNearestUser(q)$
4. $\quad$ **If** $AreFriends(u, u_i)$, add u_i into R
5. **Return** R

Figure 4: Pseudocode of NF algorithms

Algorithm 1 (NF_1). NF_1 first calculates $F = GetFriends(u)$, and gets the location of every $u_i \in F$ via $GetUserLocation(u_i)$. 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, $NF_1(u_2, q, 2)$ retrieves $F = GetFriends(u_2) = \{u_4, u_6, u_9\}$, extracts the user locations via three calls to $GetUserLocation$, sorts F in ascending distance of each $u_i \in F$ to q producing ordered list (u_4, u_6, u_9). It finally returns the first 2 users in the list, i.e., $R = (u_4, u_6)$.

Algorithm 2 (NF_2). NF_2 first extracts the friends F of u through $GetFriends(u)$. It then iteratively retrieves the next nearest user u_i to q by calling $NextNearestUser$. If u_i 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, $NF_2(u_2, q, 2)$ evaluates $F = GetFriends(u_2) = \{u_4, u_6, u_9\}$. Then, it gets u_1 as the result of the first call to $NextNearestUser(q)$. Since $u_1 \notin F$, it is not added to R. Subsequently, it proceeds with retrieving users u_2-u_6 through 5 calls to $NextNearestUser(q)$, and adds only $u_4 \in F$ and $u_6 \in F$ to R. At this point $|R|$ becomes 2 and, thus, the method returns R to the user and terminates.

Algorithm 3 (NF_3). NF_3 is similar to NF_2, but instead of invoking $GetFriends$, it utilizes $AreFriends$ for checking the friendship between a user retrieved by $NextNearestUser$ and u. In our running example, $NF_3(u_2, q, 2)$ iteratively computes users u_1-u_6 via six calls to $NextNearestUser$, and performs an $AreFriends$ primitive for each of them. During this process, it adds u_4, u_6 to R (since only those are friends with u_2), 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. We experimentally evaluate them in detail in Section 5.

4.3 Nearest Star Group (NSG)

Problem formulation. NSG returns the k nearest groups of m users to a query location q, such that the users in every group are connected through a common friend. This query is based on the concept of *aggregate distance* [34]. Specifically, the aggregate distance of a set of users $U = \{u_1, \ldots, u_n\}$ to a point q is defined as $adist(q, U) = f(||q, u_1||, \ldots, ||q, u_n||)$, where f is a monotonically increasing function[1]. For example, let $U = \{u_1, u_5, u_7\}$. If $f = sum$, $adist(q, U) = ||q, u_1|| + ||q, u_5|| + ||q, u_7|| = 31.3$; if $f = max$, then $adist(q, U) = max(||q, u_1||, ||q, u_5||, ||q, u_7||) = ||q, u_7|| = 15$.

DEFINITION 1. *Given a user u and a positive integer m, a* **Star Group (SG) of a user** *u is a set that contains u and $m-1$ of his/her friends. The set of all SGs of u, given m, is defined as:*

$$\mathcal{SG}_{u,m} = \{S \cup \{u\} \mid S \subseteq GetFriends(u) \wedge |S| = m - 1\}$$

We can perceive an SG of u as a *star subgraph* of the social network, which has u as the center vertex (underlined) and the $m-1$ friends as the pendant vertices. Note that u may have *multiple* SGs. For instance, if $m = 3$, u_2 has three SGs, namely $\mathcal{SG}_{u_2,3} = \{\{\underline{u_2}, u_4, u_6\}, \{\underline{u_2}, u_4, u_9\}, \{\underline{u_2}, u_6, u_9\}\}$. On the other hand, u_1 has only one SG, i.e., $\mathcal{SG}_{u_1,3} = \{\{\underline{u_1}, u_5, u_7\}\}$.

DEFINITION 2. *Given a query location q and a positive integer m, the* **Nearest Star Group (NSG) of a user** *u is a set $NSG_{u,q,m}$, such that:*

$$NSG_{u,q,m} \in \mathcal{SG}_{u,m} \wedge (\nexists\, S' \in \mathcal{SG}_{u,m} :$$
$$adist(q, S') < adist(q, NSG_{u,q,m}))$$

$NSG_{u,q,m}$ is the m-element SG of u with the smallest aggregate distance to q. For example, considering $f = sum$ as the aggregate function, $NSG_{u_5,q,3} = \{u_1, u_3, \underline{u_5}\}$. We denote by $\mathcal{NSG}_{q,m}$ the set of all NSGs associated with q given m, e.g., $\mathcal{NSG}_{q,4} = \{NSG_{u_2,q,4}, NSG_{u_4,q,4}, NSG_{u_6,q,4}\}$ (u_2, u_4, u_6 are the only users with three friends). When there is no ambiguity on q and m, we use notation NSG_u and $\mathcal{NSG}$ instead of $NSG_{u,q,m}$ and $\mathcal{NSG}_{q,m}$, respectively.

PROBLEM 3. *Given a 2D point q and positive integers m, k, a* ***Nearest Star Group*** (NSG) *query $NSG(q, m, k)$ returns a list $R = (NSG_{u_1}, \ldots, NSG_{u_k})$ such that, for each $1 \leq i \leq k$:*

$$NSG_{u_i} \in \mathcal{NSG} \wedge adist(q, NSG_{u_i}) \leq adist(q, NSG_{u_{i+1}}) \wedge$$
$$(\nexists\, NSG_{u'} : NSG_{u'} \in \mathcal{NSG} \wedge NSG_{u'} \notin R \wedge$$
$$adist(q, NSG_{u'}) < adist(q, NSG_{u_k}))$$

As an example, consider a restaurant that wishes to advertise a table of three. For $k = 1$, $m = 3$ and $f = sum$, query $NSG(q, 3, 1)$, where q is the location of the restaurant, retrieves the group of three users, such that (i) they form a star subgraph in the social network, and (ii) their sum of distances to the restaurant is minimized. In Figure 2, $NSG(q, 3, 1) = (NSG_{u_5,q,3}) = (\{u_1, u_3, \underline{u_5}\})$. The restaurant could send the advertisement to all

[1] A function f is *monotonically increasing* iff $\forall i : x_i \geq x'_i \rightarrow f(x_1, \ldots, x_n) \geq f(x'_1, \ldots, x'_n)$.

the users, or just the center vertex u_5. The minimization of the aggregate distance of the three users to q is motivated by the fact that the offer is more likely to attract users in close proximity to q rather than remote ones.

Observe that, according to the definition of Problem 3, for $k > 1$, the SGs in R have *different* center vertices. This choice eliminates from the result groups with large intersection, i.e., centered at the same user and differing in only a few pendant users. In an advertising application, this can minimize duplicate advertisements, while maximizing the advertising coverage.

The next lemma is useful for proving the polynomial complexity of NSG, and designing our algorithms.

LEMMA 1. *It holds that* $NSG_{u,q,m} = \{u\} \cup NF(u, q, m-1)$.

PROOF. Let $S = \{u\} \cup NF(u, q, m-1) = \{u, u_1, u_2, \ldots, u_{m-1}\}$, i.e., S contains user u and his $m-1$ closest friends to q. Consider also a set $S' = \{u, u'_1, u'_2, \ldots, u'_{m-1}\}$, which is derived from S by substituting any subset of $\{u_1, \ldots, u_{m-1}\}$ of u's friends with a different set of friends. Note that $||q, u_i|| \leq ||q, u'_i||$ for all $1 \leq i \leq m-1$, by the definition of $NF(u, q, m-1)$. Then, due to the fact that f is monotonically increasing, it holds that

$$\begin{aligned} adist(q, S) &= f(||q, u||, ||q, u_1||, \ldots, ||q, u_{m-1}||) \\ &\leq f(||q, u||, ||q, u'_1||, \ldots, ||q, u'_{m-1}||) \\ &\leq adist(q, S') \end{aligned}$$

Consequently, there is no $S' \neq S$ with a smaller aggregate distance to q. By Definition 2, this means that $NSG_{u,q,m} = S = \{u\} \cup NF(u, q, m-1)$, which concludes our proof. □

THEOREM 1. NSG *runs in polynomial time.*

PROOF. Problem 3 states that the NSG result is comprised of the k user NSGs with smallest aggregate distance. Moreover, according to Lemma 1, a user NSG can be retrieved by an NF query. Hence, in order to answer NSG, we can simply compute the NSG of *every* user in the social graph (e.g., in a trivial brute-force manner), and select the k best ones. Given that NF runs in polynomial time, NSG requires polynomial computational time as well. □

Algorithmic skeleton. The brute-force solution given in Theorem 1 is clearly prohibitively expensive for large social graphs and, thus, we aim at constructing more efficient algorithms. We outline our main idea in Figure 5 in the form of abstract steps. We *iteratively* investigate users in ascending distance to q (Line 3), since the members of the result NSGs are likely to be close to q. For every retrieved user u, we (partially or fully) construct NSGs that u participates in (Line 4), appropriately updating the result R (Line 5). We terminate this process by updating and checking certain *lower bounds*; bound b_s (resp. b_{un}) contains aggregate distance information associated with the *seen* (resp. *unseen*) part of the social graph. Henceforth, we use term '*seen*' to refer to a user retrieved in Line 3 and associated with a loop iteration (we use term '*unseen*' for any other user). The goal is to stop the process (i.e., reach $b_{un} \geq b_s$) as early as possible, while guaranteeing that (i) the final result is computed by Lines 1-5, or (ii) we have sufficient information to get the final result in a *refinement* step in Line 6.

We next present three solutions that follow the general algorithmic methodology explained above. They essentially differ in the computation of NSGs, the updating of b_{un}, and the refinement of R (Lines 4-6, respectively). They also utilize different combinations of primitives. For simplicity, we explain all algorithms for $k = 1$, focusing on the case where the aggregate function is $f = sum$.

Input: Location q, positive integers m, k
Output: Result set R

1. Initialize R, b_s, b_{un}
2. **While** $b_{un} < b_s$
3. Get the next nearest user to q
4. Construct NSGs
5. Update result R and b_s, b_{un}
6. Refine R // optional step
7. **Return** R

Figure 5: Skeleton for NSG algorithms

Algorithm 1 (NSG_{eager}). The main idea is that, for every newly retrieved user u_i, the algorithm *eagerly* constructs NSG_{u_i} using any NF algorithm and Lemma 1, and computes its aggregate distance to q. If $adist(q, NSG_{u_i})$ is lower than the current best distance attained by the *seen* users, held in b_s, the process sets NSG_{u_i} as the current result R and b_s to $adist(q, NSG_{u_i})$. Next, it updates a bound on the smallest aggregate distance that can achieved by an *unseen* u_i, stored in b_{un}. Specifically, b_{un} is the sum of distances of (i) the currently investigated u_i to q, and (ii) the $m-1$ *nearest seen* users to q. The intuition behind b_{un} is that, in the *best* case, the aggregate distance of NSG_{u_t} of *any* unseen user u_t is $||q, u_i||$ (as if u_t were at the same distance to q as the last user u_i) plus the $m-1$ smallest possible distances to q (as if u_t were friends with the $m-1$ nearest users to q). If there are fewer than $m-1$ users seen so far ($i < m-1$), NSG_{eager} adds $(m-i-1) \cdot ||q, u_i||$, simulating the missing $m-i-1$ users as being at the same distance to q as u_i. The procedure stops when $b_{un} \geq b_s$, as the unseen users cannot have a better NSG. No refinement step is required. Figure 6 presents the pseudocode of NSG_{eager}. We prove the correctness of the algorithm in the long version of this paper [22].

Input: Location q, positive integer m
Output: Result set R

1. Initialize $b_s = \infty, b_{un} = 0, R = \emptyset, i = 1$
2. **While** $b_{un} < b_s$
3. $u_i = NextNearestUser(q)$
4. $NSG_{u_i} = \{u_i\} \cup NF(u_i, q, m-1)$
5. **If** $|NSG_{u_i}| = m \wedge adist(q, NSG_{u_i}) < b_s$
6. $R = (NSG_{u_i}), b_s = adist(q, NSG_{u_i})$
7. $b_{un} = ||q, u_i|| + \sum_{(1 \leq j \leq i) \wedge (j < m)} ||q, u_j||$
8. **If** $i < m-1, b_{un} += (m-i-1) \cdot ||q, u_i||$
9. $i++$
10. **Return** R

Figure 6: Pseudocode of NSG_{eager}

Figure 7 walks through all the steps of call $NSG_{eager}(q, 3, 1)$. The algorithm concludes in 7 iterations, where iteration i retrieves and investigates user u_i. A dashed circle around a user u indicates that the algorithm constructs NSG_u, and the number next to this circle is the aggregate distance of NSG_u. Bold edges constitute the current result R, where the underlined user is its center. The values of b_s, b_{un} appear at the bottom of each iteration. At iteration 1, NSG_{eager} retrieves the nearest user of q, u_1, and computes $NSG_{u_1} = \{\underline{u_1}, u_5, u_7\}$ and $adist(q, NSG_{u_1}) = 31.3$. It then sets $R = (NSG_{u_1})$ as a candidate result, and $b_s = 31.3$ as the smallest aggregate distance found so far. Moreover, it calculates $b_{un} = (||q, u_1|| + \sum_{j=1}^{1} ||q, u_j||) + 1 \cdot ||q, u_1|| = 15$ (the sum in the parenthesis is from Line 7, whereas the second factor is from Line 8). This is the minimum distance that can be obtained by any unseen user, assuming his distance and that of his friends is equal to $||q, u_1||$. Since $b_{un} < b_s$, the algorithm proceeds to iteration 2, where it retrieves the second nearest user u_2, and com-

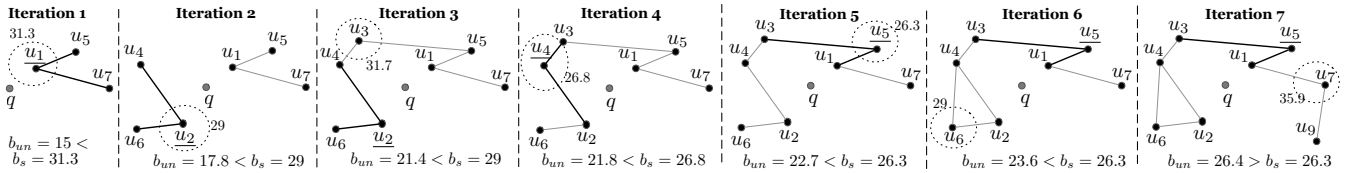


Figure 7: Example of NSG_{eager}

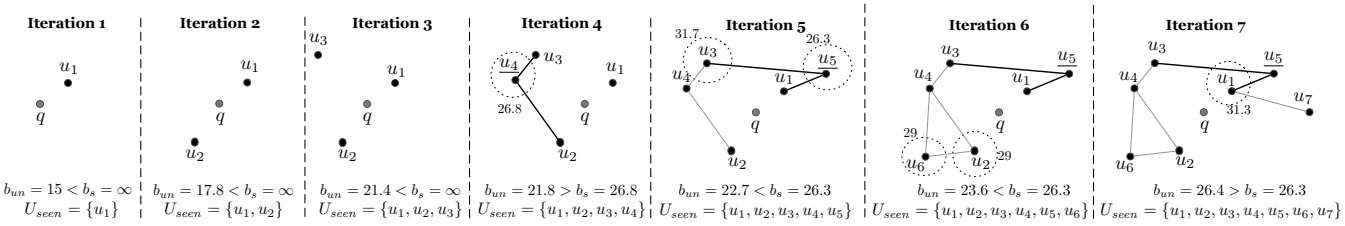


Figure 9: Example of NSG_{lazy}

putes $NSG_{u_2} = \{\underline{u_2}, u_4, u_6\}$, $adist(q, NSG_{u_2}) = 29 < b_s$. Therefore, it updates b_s to 29, and sets $R = (NSG_{u_2})$ and $b_{un} = (||q, u_2|| + \sum_{j=1}^{2} ||q, u_j||) = 17.8$. The algorithm proceeds similarly until iteration 7, where $b_{un} = (||q, u_7|| + \sum_{j=1}^{2} ||q, u_j||) = 26.4 > b_s = 26.3$ and, hence, the procedure concludes outputting NSG_{u_5} as the result. The NSG of any user that has not been seen must have aggregate distance at least 26.4, even if he is a friend with the two users closest to q (u_1,u_2).

Algorithm 2 (NSG_{lazy}). This approach follows a similar algorithmic framework to NSG_{eager}. The main difference is that, when the next nearest user u_i to q is retrieved, NSG_{lazy} does *not* construct the *entire* NSG_{u_i}, but *lazily* builds the NSG from the users seen so far, kept in a list U_{seen} sorted in ascending distance to q. Moreover, u_i contributes to the creation of the NSGs of the users in U_{seen}. This implies that the algorithm maintains *partial NSGs* for the users in U_{seen}. When an NSG is completed, we update R and b_s similarly to NSG_{eager}. For simplicity, we use the same notation for the partial and complete NSG of user u (i.e., NSG_u); we simply check $|NSG_u|$ against m to verify whether it is complete or partial. Finally, b_{un} is updated in the same manner as in NSG_{eager}. Figure 8 illustrates the pseudocode of NSG_{lazy}, whereas [22] includes its correctness proof.

Input: Location q, positive integer m
Output: Result set R

1. Initialize $b_s = \infty$, $b_{un} = 0$, $R = \emptyset$, $i = 1$, $U_{seen} = \emptyset$
2. **While** $b_{un} < b_s$
3. $u_i = NextNearestUser(q)$
4. $F = GetFriends(u_i)$
5. **For** all $u \in U_{seen} \cap F$ // in asc. dist. to q
6. **If** $|NSG_{u_i}| < m$
7. add u to NSG_{u_i}
8. **If** $|NSG_{u_i}| = m \wedge adist(q, NSG_{u_i}) < b_s$
9. $R = (NSG_{u_i})$, $b_s = adist(q, NSG_{u_i})$
10. **If** $|NSG_u| < m$
11. add u_i to NSG_u
12. **If** $|NSG_u| = m \wedge adist(q, NSG_u) < b_s$
13. $R = (NSG_u)$, $b_s = adist(q, NSG_u)$
14. add u_i to U_{seen} // U_{seen} is sorted in asc. dist. to q
15. $b_{un} = ||q, u_i|| + \sum_{(1 \le j \le i) \wedge (j < m)} ||q, u_j||$
16. **If** $i < m - 1$, $b_{un} += (m - i - 1) \cdot ||q, u_i||$
17. $i++$
18. **Return** R

Figure 8: Pseudocode of NSG_{lazy}

Figure 9 depicts a detailed example of the algorithm, for call $NSG_{lazy}(q, 3, 1)$. We follow the same notation as in Figure 7, adding set U_{seen} for easy reference. The retrieval of u_1-u_3 in the first three iterations, respectively, does not cause the construction of any complete or partial NSG, but updates b_{un}. At iteration 4, the algorithm fetches u_4, who has two friends $u_2, u_3 \in U_{seen}$ that have been already seen. Therefore, it completes NSG_{u_4} and sets $R = (NSG_{u_4})$, $b_s = 26.8$. In addition, the procedure uses u_4 to construct two *partial* NSGs, namely NSG_{u_2} and NSG_{u_3}, which contain u_4 as the only pendant vertex. Iteration 5 causes the construction of NSG_{u_5}, and u_5 contributes to the *completion* of NSG_{u_3}. The process continues updating R, b_s and b_{un}, and concludes at iteration 7 ($b_{un} > b_s$) returning NSG_{u_5} as the result. Comparing with Figure 7, the algorithm performs the same number of iterations, because it uses exactly the same b_s, b_{un} bounds. However, as opposed to NSG_{eager}, NSG_{lazy} avoids building complete NSGs for all seen users.

Algorithm 3 (NSG^*_{eager}). This approach is an *optimization* of NSG_{eager} with two important differences: (i) after it retrieves the next nearest user u_i to q, NSG^*_{eager} attempts to construct NSG_{u_i} using only u_i's friends in range b_s (the current smallest aggregate distance) to q. This is because, if a friend of u_i is outside range b_s, NSG_{u_i} is guaranteed to be worse than the current best. (ii) NSG^*_{eager} increases bound b_{un} more *aggressively* to terminate faster. Specifically, b_{un} here is the best distance that can be achieved by an unseen user without any seen friends, i.e., $b_{un} = m \cdot ||q, u_j||$ where u_j is the lastly seen user. This looser bound ensures that, upon termination of the loop, an unseen user *that is not friends with a seen user* cannot yield a better NSG than the current best. However, this does not guarantee that the best result has been found. In particular, a *refinement step* is needed to check the case of *unseen users in range b_s with seen friends*. The intuition is that the extra cost of the refinement step may be lower than the savings attained from the earlier termination.

Figure 10 contains the pseudocode of NSG^*_{eager}. The algorithm needs an *initialization* step (Lines 2-3), in order to calculate b_s for the first time before entering the while loop. It next proceeds as in NSG_{eager}, namely it retrieves the next nearest user u_i to q (starting from u_1). However, in contrast to NSG_{eager}, in Line 7 NSG^*_{eager} constructs (potentially *partial*) NSG_{u_i} using the friends of u_i (retrieved in Line 6) that are within b_s distance to q. If NSG_{u_i} is complete, R and b_s are properly updated similar to NSG_{eager} (Lines 8-9). The b_{un} bound is set in Line 15. Lines 18-22 constitute the refinement step, which is facilitated by the information gathered in

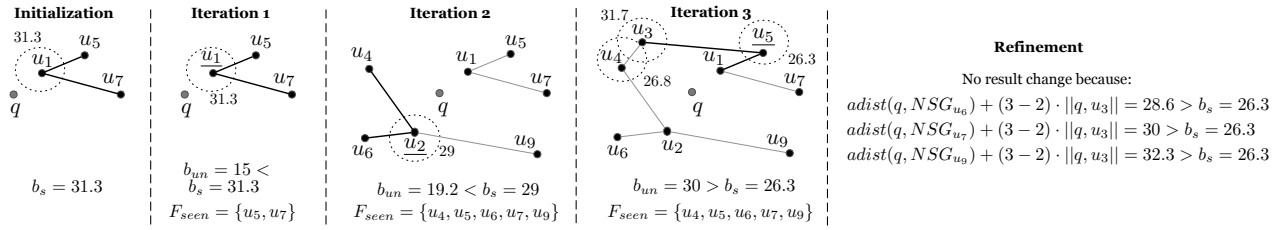


Figure 11: Example of NSG^*_{eager}

Input: Location q, positive integer m
Output: Result set R

1. Initialize $b_s = \infty$, $b_{un} = 0$, $R = \emptyset$, $F_{seen} = \emptyset$, $i = 1$
2. $u_1 = NearestUsers(q, 1)$, $NSG_{u_1} = \{u_1\} \cup NF(q, u_1, m-1)$
3. $b_s = adist(q, NSG_{u_1})$
4. **While** $b_{un} < b_s$
5. $u_i = NextNearestUser(q)$
6. $F = RF(u_i, q, b_s)$
7. $NSG_{u_i} = \{u_i\} \cup$ the $m-1$ nearest users to q in F
8. **If** $|NSG_{u_i}| = m \wedge adist(q, NSG_{u_i}) < b_s$
9. $R = (NSG_{u_i})$, $b_s = adist(q, NSG_{u_i})$
10. **For** all $u \in F$
11. **If** $u \notin F_{seen}$, add u to F_{seen}
12. **If** $|NSG_u| < m$, add u_i to NSG_u
13. **If** $|NSG_u| = m \wedge adist(q, NSG_u) < b_s$
14. $R = (NSG_u)$, $b_s = adist(q, NSG_u)$
15. $b_{un} = m \cdot ||q, u_i||$
16. $i++$
17. $i = i - 1$ // so that u_i is the lastly seen user
18. **For** all $u \in F_{seen} \wedge |NSG_u| < m$
19. **If** $adist(q, NSG_u) + (m - |NSG_u|) \cdot ||q, u_i|| < b_s$
20. $NSG_u = NF(u, q, m-1)$
21. **If** $adist(q, NSG_u) < b_s$
22. $R = (NSG_u)$, $b_s = adist(q, NSG_u)$
23. **Return** R

Figure 10: Pseudocode of NSG^*_{eager}

Lines 10-14. Specifically, all the retrieved *friends* of the *seen* users are put in a list F_{seen}. The algorithm lazily constructs their NSGs as new users u_i arrive in Line 5. If such an NSG is completed, R and b_s may be updated (Lines 13-14). Finally, the refinement step completes the partial NSGs of F_{seen} in case the condition in Line 19 holds, updating R, b_s if necessary. We prove the correctness of NSG^*_{eager} in [22].

Figure 11 illustrates an example for call $NSG^*_{eager}(q, 3, 1)$, complying with the notation of Figures 7 and 9, and adding F_{seen} for easy reference. The initialization step builds NSG_{u_1} and sets $b_s = adist(q, NSG_{u_1}) = 31.3$. At iteration 1, the algorithm sets $R = (NSG_{u_1})$ and $b_{un} = 3 \cdot ||q, u_1|| = 15$, preserves b_s, and adds u_5, u_7 into F_{seen}. It also creates partial NSGs for the users in F_{seen} using u_1. At iteration 2, NSG^*_{eager} constructs NSG_{u_2}, which becomes the current best result giving a new $b_s = 29$. The friends of u_2 (u_4, u_6, u_9) join F_{seen}, and initialize their partial NSGs. The value of b_{un} becomes 19.2; observe that it increases faster than in the case of NSG_{eager} and NSG_{lazy}. At iteration 3, the procedure retrieves u_3, creates NSG_{u_3}, and completes the NSGs of $u_4, u_5 \in F_{seen}$. The best among them is NSG_{u_5} that updates b_s to 26.3. Bound b_{un} becomes $30 > b_s$ and, hence, the algorithm terminates. Next, the refinement step does not need to complete the partial NSGs of $u_6, u_7, u_9 \in F_{seen}$, since the condition of Line 19 is not satisfied. The algorithm returns NSG_{u_5} as the result. This example demonstrates the potential superiority of NSG^*_{eager} versus NSG_{eager} and NSG_{lazy}; it performs fewer iterations, while the refinement step does not incur any additional overhead.

The performance of the three described NSG solutions depends on the number of seen users, as well as the number and efficiency of the primitive queries involved. In the next section we include their thorough experimental comparison. Some final remarks concern the case of $k > 1$ and the usage of aggregate function $f = max$. The modifications of our pseudocodes for $k > 1$ are straightforward: we simply maintain a k-element list R, which contains the k best NSGs at all times (sorted in ascending aggregate distance to q), and b_s now constitutes the k^{th} smallest distance. Moreover, NSG^*_{eager} (Figure 10) constructs k NSGs in Line 2, properly setting b_s in Line 3. Finally, the case of $f = max$ is supported *only* for NSG_{eager} and NSG_{lazy}. The sole alteration concerns setting $b_{un} = ||q, u_i||$ in every loop iteration. Correctness is proved in a very similar fashion to the case of $f = sum$. We omit the details due to space constraints.

5. EXPERIMENTS

Section 5.1 describes our experimental setup, whereas Section 5.2 presents our results.

5.1 Setup

Storage schemes. We employed two different data storage approaches for the social (SM) and geographical (GM) module: a *disk-based* (with cache), and a *memory-based*. All schemes were implemented in C++, under Linux Ubuntu.

The disk-based approach uses MongoDB [17], a popular commercial *document-oriented database*. MongoDB keeps the information in documents in the hard disk. At the SM, we store the social graph as a set of such documents. Every document corresponds to a user, and carries his ID and a *sorted* list of his friends' IDs (i.e., an adjacency list). All documents are indexed with a B$^+$-Tree on user ID. *GetFriends* retrieves the user document and returns the friend list. *AreFriends* entails reading a user document and checking his friend list. A social update (i.e., the insertion or deletion of a graph edge between two users) involves reading *two* user documents, altering their friend lists, and writing the *entire* two lists back in their documents kept in the disk. At the GM, we create a document for each user, which contains his ID and coordinates. These documents are indexed with a B$^+$-Tree on user ID, which enables fast answering of *GetUserLocation*. *RangeUsers* and *NearestUsers* are readily supported in MongoDB by built-in functions. To answer such queries, MongoDB exploits efficient spatial indexing techniques, such as *grids* and *geohashing* [11]. A location update entails both a document update and an index update. MongoDB does not have a caching component. Instead, it uses Linux's caching mechanism.

In the memory-based approach, we store the social graph at SM in a main memory *hash table*, where the key is the user ID and the value is a *sorted* list of his friends' IDs. Primitive *GetFriends* is a simple lookup in the table, whereas *AreFriends* involves a table lookup and a binary search over a friend list. A social update involves finding the *two* users in the hash table, and altering their

friend lists. At the GM, we store each user (represented by his ID and coordinates) in a regular 300×300 *grid*. We also use a hash table on the user ID, to facilitate fast execution of *GetUserLocation*. We perform *RangeUsers* in a straightforward manner using the grid, whereas we adopt the CPM [33] algorithm for answering *NearestUsers*. A location update needs a hash table lookup (to find the user and change his location field), the deletion of the user from the old grid cell, and his insertion to the new grid cell.

Machine architecture. We conducted our experiments using both a *centralized* and *distributed* architecture. The centralized scenario positions the three modules (SM, GM and QM) at a single server. The distributed setting utilizes a separate server for each module, where the QM machine communicates with the SM and GM machines via a 100Mbps Ethernet network. Each server is an Intel Core 2 Duo, with a 2.33GHz CPU and 4GB RAM.

Datasets. We used both real and synthetic datasets. We derived the *real* dataset from Foursquare and Twitter as follows. We first gathered 12,652 users that posted a Foursquare check-in as a twit on their Twitter account on the *same* day (May 30th, 2012) in New York City. The check-in coordinates were spread in an overall area of 1,112 km^2. This user information was maintained at the GM. We then extracted their friends from Twitter (where the average number of friends was 437), yielding a Twitter subgraph of 2,220,627 users (note though that these additional users did not have any location data). This social graph was kept at the SM.

We also created five different *synthetic* datasets, containing 1-5 million users, respectively, simulating scenarios in a big city. We used the Barabási-Albert preference model [24] for generating the edges, setting the average number of friends to 100 (note that this number is currently 190 in Facebook [2], and was 5-8 in 2011 in Foursquare [7]). The distribution of the number of friends in this model follows a power law, which is in tangent with the current trend in popular social networks [32]. We assigned locations to *all* users respecting the following two principles: (i) two friends are more likely to check-in at nearby places, and (ii) the distribution of the distance between two friends follows a power law [25]. More specifically, starting from a random user at a random location, we traversed the entire graph in a BFS fashion and assigned locations to the users based on their Euclidean distances to their friends, which were randomly derived from the distribution of [25]. The resulting check-in coordinates were spread in an overall area of 7,853 km^2. From the above datasets, the social graphs were kept at the SM, whereas the location data were stored at the GM.

Parameters. Table 1 summarizes the system parameters with their ranges, where r is the radius (in km) in RF, s is the increment step in $NextNearestUser$, k is the result size in NF and NSG, m is the group size in NSG, and N is the synthetic dataset size.

Table 1: System parameters and their ranges

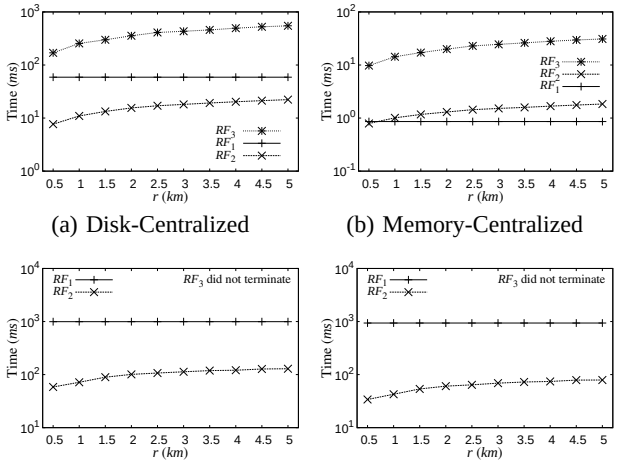
Parameter	r	s	k	m	N
Range	0.5-5 km	1K-5K	1-10	2-7	1M-5M

Evaluation methodology. Since the two machine architectures are orthogonal to the two storage schemes described above, we explored in total four different scenarios; namely, *Disk-Centralized*, *Memory-Centralized*, *Disk-Distributed*, and *Memory-Distributed*. We assessed the algorithms with respect to their *total query response time*, i.e., the time elapsed from the instant a query is issued to its result retrieval. The reported time is the average over 100 random queries. For the disk-based case, we performed *cache warm-up* by issuing 50 random primitive queries.

5.2 Results

Range Friends (RF). Recall that RF_1 involves an execution of $GetFriends$ and subsequent calls of $GetUserLocation$ for the retrieved friends. RF_2 performs an intersection of the results of $GetFriends$ and $RangeUsers$. RF_3 executes $RangeUsers$ and performs subsequent calls to $AreFriends$.

Figure 12 assesses the query time (in ms) of our three RF algorithms as a function of radius r (in km) for our real dataset. Figure 12(a) shows the disk-based centralized scenario. The performance of RF_1 is unaffected by r, because its primitives are independent of r. In contrast, the query time of RF_2 and RF_3 increases with r, because they both call $RangeUsers$ whose processing time rises with r. RF_2 exhibits the best performance (7.67-22.25 ms), outperforming RF_1 by almost up to one order of magnitude (58.94 ms), and RF_3 by more than two orders of magnitude (169.46-547.73 ms). The main reason is that RF_2 entails only two relatively inexpensive primitive operations. On the contrary, RF_1 invokes $GetUserLocation$ for every friend (i.e., 437 times on the average), resulting in a high I/O cost. RF_3 has the worst performance, as it executes $AreFriends$ for every user in the range, which may include up to thousands of users.



(a) Disk-Centralized (b) Memory-Centralized

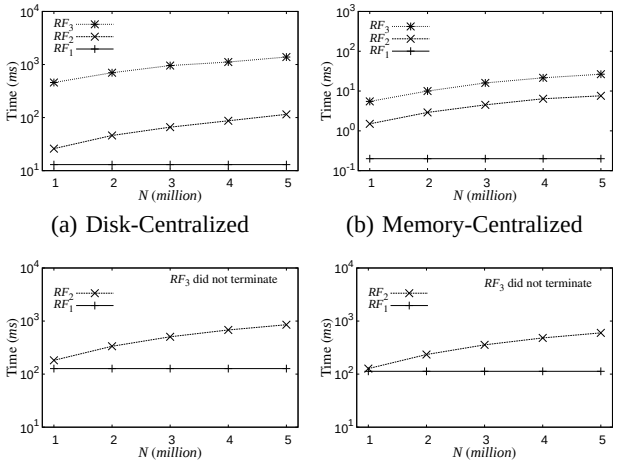
(c) Disk-Distributed (d) Memory-Distributed

Figure 12: Query time for RF vs. r (Real dataset)

Figure 12(b) depicts our results for the memory-based centralized case. RF_2 and RF_3 follow the same trends as in Figure 12(a), but their times are lower because the primitives are faster in main memory. The most important observation is that RF_1 now becomes the best algorithm. The reason is that $GetUserLocation$ does not incur any I/O cost, which was the dominant overhead in the disk-based case. As such, RF_1 becomes about twice faster than RF_2. Figures 12(c) and 12(d) demonstrate the query time in the disk-based and memory-based distributed scenario, respectively. RF_3 does not terminate in a reasonable time frame, due to the numerous $AreFriends$ calls. RF_2 outperforms RF_1 by up to one order of magnitude in both settings, maintaining its query time below 130 ms. The major reason is that RF_1 pays the considerable network delay for performing the multiple $GetUserLocation$ calls, which is the dominant cost in both the disk- and memory-based scenarios.

In Figure 13 we assess the scalability of the RF algorithms using the synthetic datasets. We vary the dataset size N, fixing the radius to $r = 2.5\ km$. RF_1 is practically unaffected by N; for all N, $GetFriends$ retrieves the same number of friends (since the average number of friends in all datasets is fixed to 100), whereas

$GetUserLocation$ is efficiently handled by a B$^+$-Tree whose height is minimally influenced by the increase in N. The costs of RF_2 and RF_3 rise with N; larger datasets are denser and, thus, $RangeUsers$ retrieves more users. Contrary to the case of the real dataset, here RF_1 is always superior to RF_2 and RF_3, since the smaller average number of friends (100 vs. 437) makes $GetFriends$ considerably cheaper, and leads to much fewer $GetUserLocation$ invocations. Even for the case of $N = 5$M in the disk-based distributed scenario, the overhead of RF_1 is in the order of 100 ms.



(a) Disk-Centralized (b) Memory-Centralized

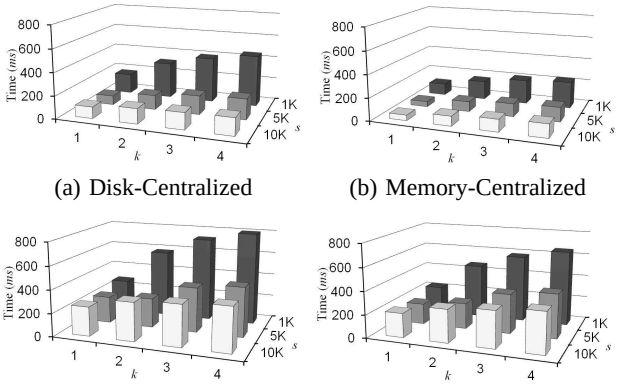
(c) Disk-Distributed (d) Memory-Distributed

Figure 13: Query time for RF vs. N (Synthetic datasets)

Nearest Friends (NF). Recall that NF_1 invokes $GetFriends$ followed by multiple $GetUserLocation$ operations. NF_2 executes $GetFriends$, and calls $NextNearestUser$ multiple times. NF_3 involves multiple $NextNearestUser$ and $AreFriends$ calls. We first include a discussion on the increment step s of $NextNearestUser$, which considerably affects the performance of NF_2 and NF_3.

Figure 14 plots the query time of NF_2 versus s and result size k (the case of NF_3 is similar and, thus, omitted), for the real dataset. Observe that NF_2 is affected by the *combination* of s and k. In most settings, a higher s leads to better performance. This suggests that executing *fewer* $NearestUsers$ primitives at the expense of retrieving "redundant" users is preferable to multiple calls with a higher result overlap. However, for small k (i.e., 1 and 2), the value of s that leads to the lowest query time (5K) lies between the two extremes. The reason is that the NF result is relatively close to q for a small k and, hence, a moderate s leads to minimizing both the "redundant" users and the number of primitive calls. The benefits are more pronounced in the distributed setting (Figures 14(c) and 14(d)), because the "redundant" users inflict both processing and network overhead. In the remaining experiments, we fine-tuned s to its optimal value for every setting and algorithm.

Figure 15 evaluates the three NF algorithms when varying k for the real dataset. NF_1 is independent of k. On the other hand, the number of calls to $NextNearestUser$ in NF_2 and NF_3 rises with k and, thus, performance deteriorates. In the centralized scenario, NF_1 exhibits the best performance. The reason is that, in NF_2 and NF_3, (i) numerous users are investigated, and (ii) a friendship test is performed for every retrieved user. NF_3 is worse than NF_2 because its numerous calls to $AreFriends$ outweigh the one-time cost of $GetFriends$ in NF_2. The costs of NF_2 and NF_3 in Figure 14(b) are close, because the overhead of $AreFriends$ is smaller

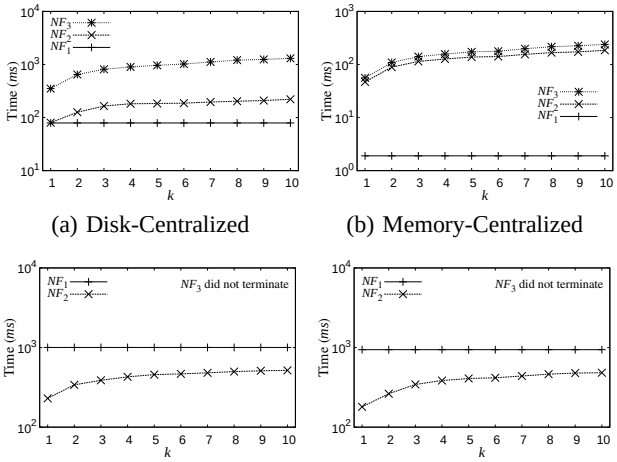


(a) Disk-Centralized (b) Memory-Centralized

(c) Disk-Distributed (d) Memory-Distributed

Figure 14: Query time for NF_2 vs. s and k (Real dataset)

than that in the disk-based case. In the distributed case, NF_2 becomes the best algorithm; NF_1 executes $GetUserLocation$ multiple times which impose great network delay, whereas NF_2 performs fewer primitive invocations due to the step-wise prefetching of $NextNearestUser$. NF_3 does not terminate within a reasonable time frame, due to its numerous $AreFriend$ queries.



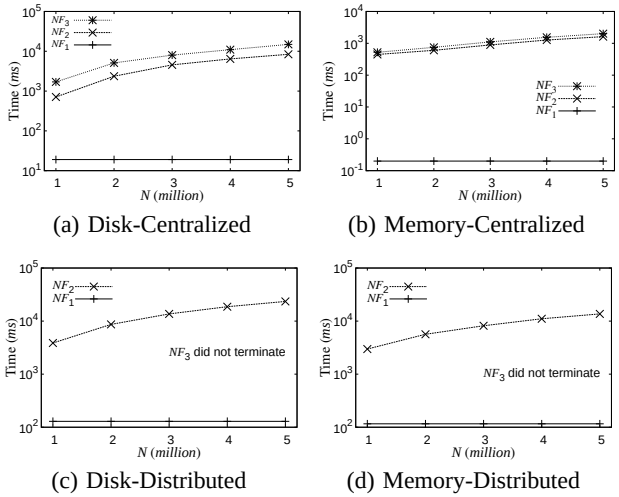
(a) Disk-Centralized (b) Memory-Centralized

(c) Disk-Distributed (d) Memory-Distributed

Figure 15: Query time for NF vs. k (Real dataset)

Figure 16 assesses the scalability of the NF algorithms on the synthetic datasets, varying the dataset size N and setting $k = 5$. NF_1 is unaffected by N, since, similar to RF_1, $GetFriends$ always retrieves a fixed number of users (100 on the average). On the other hand, the costs of NF_2 and NF_3 increase with N; all areas become denser and, hence, $NextNearestUser$ investigates a much larger number of users until it finds the k nearest friends (e.g., $\sim$ 150K users when $N = 3$M). Consequently, NF_1 is the best algorithm in all settings, with overhead in the order of 100 ms even in the most demanding setting (Figure 16(c)).

Nearest Star Group (NSG). Recall that, NSG_{eager} builds the complete NSGs for users around the query location, until no unseen user can lead to a better result. NSG_{lazy} builds partial NSGs involving only users seen so far. NSG^*_{eager} involves a more aggressive bound than the previous algorithms, in order to reach an earlier termination. Routines NF (in NSG_{eager}) and RF (in NSG^*_{eager}) were implemented using the fastest algorithms for each setting. We also applied $f = sum$ as the aggregate function.

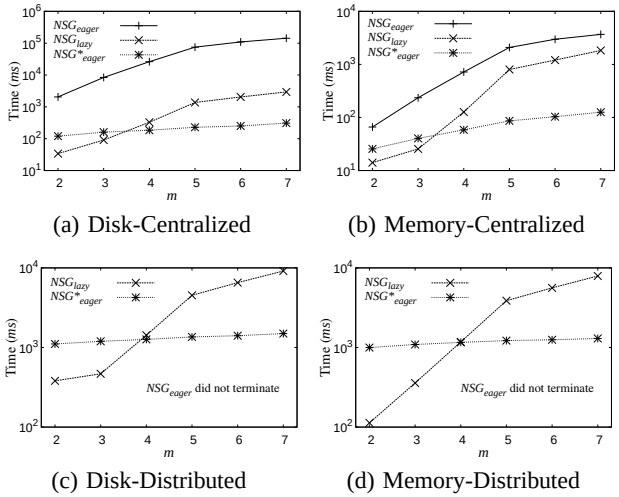


(a) Disk-Centralized (b) Memory-Centralized

(c) Disk-Distributed (d) Memory-Distributed

Figure 16: Query time for NF vs. N (Synthetic datasets)

Figure 17 depicts the performance of the NSG algorithms for the real dataset, as a function of the group size m ($k = 1$). The cost of every algorithm increases with m, as more users are seen to find the NSG result. NSG_{eager} exhibits the worst performance (Figures 17(c) and 17(d) omit its costs, as it did not terminate within reasonable time); it involves expensive NF calls, whereas NSG_{lazy} and NSG^*_{eager} invoke the cheaper $GetFriends$ and RF, respectively. In most settings, NSG^*_{eager} is the best algorithm, whereas its query time is always smaller than 1.2 sec. The largest performance gain over NSG_{lazy} reaches up to more than one order of magnitude (Figure 17(b)). The main reason is that NSG^*_{eager} explores up to one order of magnitude fewer users in the while loop than NSG_{lazy} (118 vs. 3,539), while its refinement step is very fast. Nevertheless, NSG_{lazy} is superior for $m \leq 3$ ($m \leq 4$) in the centralized (distributed) case; it runs fewer primitives per iteration, while its number of iterations is comparable to that in NSG^*_{eager}.

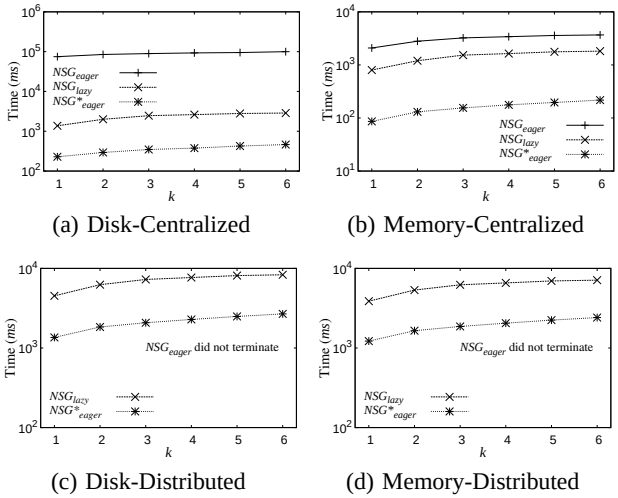


(a) Disk-Centralized (b) Memory-Centralized

(c) Disk-Distributed (d) Memory-Distributed

Figure 17: Query time for NSG vs. m (Real dataset)

Figure 18 assesses the performance of the NSG algorithms for the real dataset, when varying k and setting $m = 5$. For groups of 5 users, NSG^*_{eager} is the best algorithm for the reasons explained in Figure 17. The cost of every method naturally increases with k, since more users must be investigated to construct the k best NSGs comprising the result. However, the query time of NSG^*_{eager} is below 3 sec even for $k = 6$ in all scenarios.

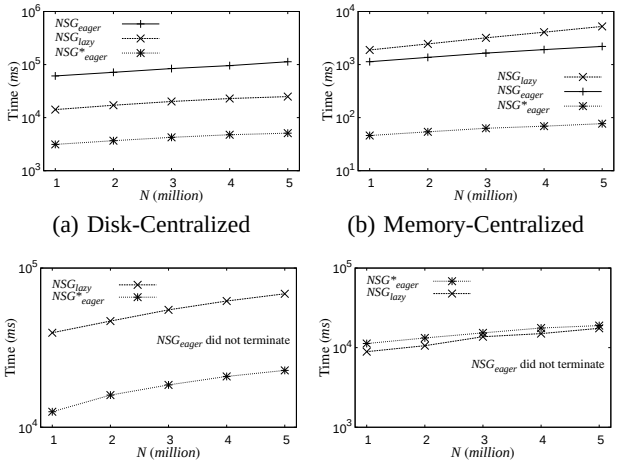


(a) Disk-Centralized (b) Memory-Centralized

(c) Disk-Distributed (d) Memory-Distributed

Figure 18: Query time for NSG vs. k (Real dataset)

Figure 19 depicts the scalability results of the NSG algorithms on the synthetic datasets, when varying the dataset size N, and setting $k = 3$, $m = 5$. The cost of every algorithm increases with N. The reason is that all areas become denser and, therefore, more users must be investigated before discovering the result. NSG^*_{eager} is superior to NSG_{eager} and NSG_{lazy} in Figures 19(a)-19(c), for the same reasons as in the real dataset case. However, in Figure 19(d), NSG_{lazy} becomes marginally better than NSG^*_{eager}. The reason is that the cost of $GetFriends$ in the main-memory setting is lower than that in the disk case. Therefore, the gains of NSG^*_{eager} due to the fewer explored users becomes much less pronounced. Finally, despite the complexity of the NSG query and our relatively weak machines, the best algorithms yield query response times in the order of a few seconds, even in the most challenging settings.



(a) Disk-Centralized (b) Memory-Centralized

(c) Disk-Distributed (d) Memory-Distributed

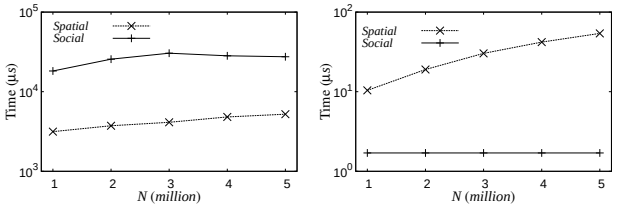
Figure 19: Query time for NSG vs. N (Synthetic datasets)

Updates. In each experiment, we performed 100,000 social updates, and an equal number of spatial updates. A social update was a friendship creation between two users, whereas a spatial update

was a check-in that altered the current location of a user to a new one (see Section 5.1 for the implementation details). We report the average update times for the real and synthetic datasets, focusing only on the centralized setting (the times are the same in the distributed case, as each update occurs *locally* at SM or GM).

For the real dataset, in the disk-based scenario, a social update takes $2.3\ ms$ and a spatial update $0.17\ ms$. Observe that a social update is considerably more expensive than a spatial update, because it entails reading two long friend lists (each of average size 437) from the disk, whereas a spatial update involves changing a single entry in the effective spatial indices of MongoDB. Note that MongoDB employs Linux's cache, which significantly reduces both read and write costs. For the memory-based case, a social update requires $3.57\ \mu s$, and a spatial update $5.4\ \mu s$. Here, a social update is cheaper than a spatial one; the social cost is now very low (since the I/O cost is eliminated), whereas our grid may contain a long ID list in the user's old cell, which must be traversed to locate his ID and delete it.

Figure 20 illustrates the update time when varying the dataset size N for the synthetic datasets. In general, the cost of spatial updates in both the disk- and memory-based setting increases with N, since the dataset cardinality affects the size and performance of the spatial indices. On the other hand, the social updates are rather minimally impacted by N, since their cost is mainly influenced by the average number of friends. The observed fluctuation in Figure 20(a) is due to the ID lookups during the updates and the randomness of the datasets. Finally, a spatial (social) update is more costly than a social (spatial) update in the memory-based (disk-based) scenario for the same reasons as in the real dataset scenario.



(a) Disk-Centralized (b) Memory-Centralized

Figure 20: Update time vs. N (Synthetic datasets)

Summary. We derive three main conclusions from our empirical analysis: (i) the best algorithm for each query type depends on the setting (data management method and system architecture), (ii) the performance of our algorithms is excellent even under the most challenging scenarios (e.g., for NSG queries, in the disk-based case, for large m and k), and scales well with the dataset size, and (iii) our implementations can handle hundreds of updates per second even for large dataset sizes ($N = 5$M).

6. CONCLUSION

In this paper we conducted the first systematic study of GeoSN query processing. In particular, we introduced a general framework that segregates the social, geographical and query processing modules, enabling flexible data management and algorithmic design. A GeoSN query is processed via a combination of primitive operations issued to the social and geographical modules. We also introduced novel GeoSN queries, and designed various solutions based on different sets of primitives. Finally, we performed an exhaustive experimental evaluation on real and synthetic datasets with realistic implementations, which confirmed the viability of our framework and the practicality of our GeoSN queries and algorithms.

7. REFERENCES

[1] Agora. http://agora-app.heroku.com/.
[2] Facebook anatomy. https://www.facebook.com/notes/facebook-data-team/anatomy-of-facebook/10150388519243859/.
[3] Facebook Places. https://www.facebook.com/about/location/.
[4] Factual. http://www.factual.com/.
[5] Foursquare. http://www.foursquare.com/.
[6] Foursquare downtime problem. http://blog.foursquare.com/2010/10/05/so-that-was-a-bummer/.
[7] Foursquare friendships. http://vimeo.com/22641902.
[8] Foursquare statistics. https://foursquare.com/about/.
[9] Fullcircle. http://www.fullcircle.net/.
[10] Geoloqi. http://www.geoloqi.com/.
[11] Geospatial indexes in MongoDB. http://docs.mongodb.org/manual/core/geospatial-indexes/.
[12] Glancee. http://www.glancee.com/.
[13] GroupOn Now! deals available on Foursquare. https://blog.groupon.com/cities/groupon-now-deals-available-in-foursquare/.
[14] Hotlist. http://www.hotlist.com/.
[15] Linking Foursquare with Facebook and Twitter. http://support.foursquare.com/entries/21738953-linking-foursquare-with-facebook-and-twitter/.
[16] Memcached. http://memcached.org/.
[17] Mongodb. http://www.mongodb.org/.
[18] Neo4j. http://neo4j.org/.
[19] Scaling MongoDB at Foursquare. http://www.10gen.com/presentations/mongonyc-2012-scaling-mongodb-foursquare.
[20] Twitter: Real-time Geo. http://www.slideshare.net/raffikrikorian/rtgeo-where-20-2011.
[21] A. Amir, A. Efrat, J. Myllymaki, L. Palaniappan, and K. Wampler. Buddy tracking - efficient proximity detection among mobile friends. *Pervasive and Mobile Computing*, 3(5):489 – 511, 2007.
[22] N. Armenatzoglou, S. Papadopoulos, and D. Papadias. A general framework for geo-social query processing. Full version of this paper, available online at http://www.cse.ust.hk/~nikos/geosns/GeoSNs-long.pdf, 2013.
[23] J. Bao, M. F. Mokbel, and C.-Y. Chow. GeoFeed: A location aware news feed system. In *ICDE*, 2012.
[24] A. L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.
[25] E. Cho, S. A. Myers, and J. Leskovec. Friendship and mobility: user movement in location-based social networks. In *SIGKDD*, 2011.
[26] Y. Doytsher, B. Galon, and Y. Kanza. Querying geo-social data by bridging spatial networks and social networks. In *LBSN*, 2010.
[27] Y. Doytsher, B. Galon, and Y. Kanza. Managing socio-spatial data as large graphs. In *WWW*, 2012.
[28] Q. Huang and Y. Liu. On geo-social network services. In *Geoinformatics*, 2009.
[29] A. Khoshgozaran and C. Shahabi. Private buddy search: Enabling private spatial queries in social networks. In *CSE*, 2009.
[30] N. Li and G. Chen. Analysis of a location-based social network. In *CSE*, 2009.
[31] W. Liu, W. Sun, C. Chen, Y. Huang, Y. Jing, and K. Chen. Circle of friend query in geo-social networks. In *DASFAA*, 2012.
[32] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *SIGCOMM*, 2007.
[33] K. Mouratidis, D. Papadias, and M. Hadjieleftheriou. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *SIGMOD*, 2005.
[34] D. Papadias, Y. Tao, K. Mouratidis, and C. Hui. Aggregate nearest neighbor queries in spatial databases. *ACM Transactions on Database Systems (TODS)*, 30(2):529–576, 2005.
[35] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In *VLDB*, 2003.
[36] C. Ruiz Vicente, D. Freni, C. Bettini, and C. S. Jensen. Location-related privacy in geo-social networks. *IEEE Internet Computing*, 15(3):20–27, May 2011.
[37] S. Scellato, C. Mascolo, M. Musolesi, and V. Latora. Distance matters: Geo-social metrics for online social networks. In *WOSN*, 2010.
[38] D.-N. Yang, C.-Y. Shen, W.-C. Lee, and M.-S. Chen. On socio-spatial group query for location-based social networks. In *SIGKDD*, 2012.
[39] M. L. Yiu, L. H. U, S. Šaltenis, and K. Tzoumas. Efficient proximity detection among mobile users via self-tuning policies. In *PVLDB*, 2010.
[40] C. Zhang, L. Shou, K. Chen, G. Chen, and Y. Bei. Evaluating geo-social influence in location-based social networks. In *CIKM*, 2012.
[41] Y. Zheng, L. Zhang, Z. Ma, X. Xie, and W. Ma. Recommending friends and locations based on individual location history. *ACM Transactions on the Web (TWEB)*, 5(1):5, 2011.