2016 17th IEEE International Conference on Mobile Data Management

# Scalable Processing of Location-based Social Networking Queries

Muhammad Aamir Saleem Xike Xie Torben Bach Pedersen
Department of Computer Science, Aalborg University, Aalborg, Denmark
{maas, xkxie, tbp}@cs.aau.dk

***Abstract*—Using GPS-enabled smart phones, social network services are enriched with location information which allows users to share geo-tagged contents with their friends. This so-called location-based social network (LBSN) data has a dual spatial and graph nature. The growing scale and importance of LBSN data necessitate a platform which (i) has both spatial and graph capabilities; (ii) supports a wide range of queries, e.g., selection, structural, and aggregate queries; (iii) supports scalable distributed processing of large data volumes.**

**In this paper, we propose such a platform, called GeoSocial-GraphX, that segregates the LBSN data into several specific graphs capturing *user-user*, *user-location*, and *location-location* relationships, and enables a wide range of LBSN queries by proposing a comprehensive set of query primitives that can be composed into more advanced queries. We implement the platform based on *GraphX*, a map-reduce infrastructure for distributed graph computation. We further improve the query performance in several ways. For social-related data, we use vertex-centric messaging operators which better address the recursive nature of graph data than traditional two-stage map-reduce. For spatial-related data, we use effective spatial partitioning and indexing methods. Experiments on both synthetic and real LBSN datasets show that GeoSocial-GraphX can process a variety of LBSN queries efficiently, scales on multicore architectures, and achieves much better performance than the state of the art competing framework, SpatialHadoop.**

## I. Introduction

With the ubiquity of location-aware mobile terminals, social network users are allowed to share geo-tagged contents with their friends. These social networks with location information are called location-based social networks (LBSN). For example, users make comments on Twitter about an event happening somewhere, upload geo-tagged pictures in Flickr, or report their "check-ins" in Foursquare. LBSN data offers richer interdependency between people and locations thus holds potential for a wide range of services.

For example, the context of a location can be deciphered by its historical visitors; the mobility behaviour of a user can be investigated by his/er historical visits. It is interesting to query for top-k popular places for a given user, and top-k frequent travellers for a set of places. Such information leverages ranking and recommendation in order to facilitate travel and social interactions [27], [19], [29]. In another example, one might need to find nearby friends for an activity, or detect a community in a local area. Similarly, one might also be interested to find a set of locations that are famous for visiting together. Such applications require considering structural information in LBSN.

Another challenge arises in the recent explosion of the amounts of LBSN data. As reported in [4], Foursquare has more than 55 million users with 6 billion check-ins. The map-reduce framework, e.g., Hadoop, is generally accepted as a solution for scalable processing of large volume datasets. However, it falls short in fully addressing the challenges in LBSN data which is of both spatial and graph nature. First, the data partitioning method used in HDFS does not consider the recursive nature of graph data and consecutive nature of spatial data. Thus, it does not efficiently support multiple random data accessing which is essential for spatial or graph queries. Second, the two-stage computation framework has limited capabilities in handling iterative algorithms required by graph (or structural) queries in LBSN.

In particular, SpatialHadoop[14] , MongoDB [9], and MD-base are map-reduce frameworks equipped with spatial features. GraphX, which is built on top of *Apache Spark* is a data-parallel computation framework supporting big graph data. None of them have capabilities to support big data of both spatial and graph nature. It is thus desirable to have a platform which embeds both spatial and graph capabilities and is able to process large volume LBSN queries that are diverse in nature.

Motivated by these challenges, this paper makes the following contributions. We consider the two major roles, *user* and *location*, in LBSN. These roles formulate three types of relationships, *user-user*, *user-location*, and *location-location*. In response to such characteristics, we segregate the LBSN data into three graphs, namely *social graph*, *activity graph*, and *spatial graph*. We present a comprehensive set of query primitives defined on these graphs. The LBSN query primitives are classified into three categories: *selection*, *aggregate*, and *structural* queries, such that general-purpose LBSN queries can be answered by the combination of one or more primitives. For example, we can retrieve a local community by combining results of a structural query on the social graph and a selection query on the spatial graph. Furthermore, we present a platform that: (i) is capable of handling large volume and distributed LBSN data; (ii) answers general purpose LBSN queries. The first objective of the platform is reached by building a data-parallel platform, GeoSocial-GraphX ( GSG *in short*). Based on that, we incorporate indexing and partitioning techniques for supporting spatial data. The second objective of the platform is reached by implementing a number of query primitives which are essential for LBSN queries on top of the platform. In order to evaluate our system, we perform experiments on three real LBSNs namely, BrightKite, Gowalla

2375-0324/16 $31.00 © 2016 IEEE
DOI 10.1109/MDM.2016.30

and FourSquare and one synthetic dataset. The experiments show that segregation of LBSN into three graphs and usage of the spatial graph for location-centric queries improve the performance up to four orders of magnitude. Furthermore, the partitioning and indexing methods for spatial data enhance the performance by 6 times on average. Experimental results also confirm the scalability of GSG on multiple cores. Moreover, GSG outperforms SpatialHadoop, a leading spatial distributed framework, in terms of efficiency, scalability, and ease of implementation, i.e., lines of code, up to 20x, 7x, and 13x, respectively.

The rest of the paper is organized as follows. Section II provides the LBSN model including definitions of the basic roles in LBSN and the segregated graphs. Section III presents the query primitives and advanced LBSN queries. Section IV covers the architecture of the proposed platform, GSG, including storage layer, operation layer, index layer, and query engine. Section V presents the experimental evaluation. Section VI presents related work, and Section VII concludes the paper and points to future work.

## II. LBSN Model

We present basic concepts in Section II-A and define the three segregated graphs in Section II-B.

### A. Preliminaries

*Definition 1:* $Users(U)$ is the set of persons that are using the LBSN. A user is identified by the unique identifier $u_i$. In Figure 1, users are $U = \{u_1, u_2, u_3, u_4, u_5, u_6, u_7, u_8\}$.

*Definition 2:* $Locations(L)$ is the set of locations. We consider a location as a circular region that covers a geographical space. A location is a four-tuple $(l_m, x_m, y_m, r_m)$, where the $l_m$ is the identifier, $x_m$ and $y_m$ represent the center of the region, and $r_m$ is the radius of the region. In Figure 1, locations are $L = \{l_1, l_2, l_3, l_4, l_5, l_6, l_7\}$.

*Definition 3:* $Activities(A)$ is the set of activities of the users. An activity is a checkin of a user including both location and time. It is given by $(u_i, l_m, t_a)$, where $u_i$ is the user identifier, $l_m$ is the location identifier, and $t_a$ is the visiting time. In Figure 1, activities are shown by dotted lines, e.g., $u_1$ has visited $l_1, l_2, l_3$, and $l_5$ at sometime.

### B. LBSN Graphs

Traditionally, LBSN data is maintained into two components on the basis of its roles: user and location. The components are termed as social component and activity component. The social component represents relationships among users, i.e., user-user. The activity component deals with users' visits at locations and captures user-location relationships. However, a location is a subject in the LBSN that has attributes such as GPS coordinates. Furthermore, the interactions of LBSN roles in the social and activity component yield the relationships between locations, i.e., location-location, such as common visitors and distances among locations.

For LBSN queries about user-user and user-location relationships, the social component and the activity component

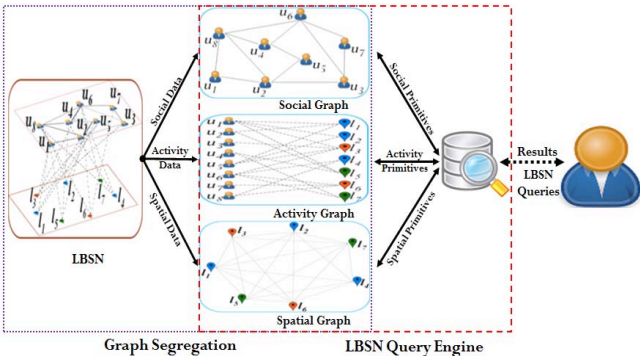



Fig. 1: LBSN Model

can be used, respectively. However, for queries about location-location relationships, it is very expensive to use the social and the activity components. An example of such queries is to find a user's best travel companions based on their common visited places. In order to process such queries in an efficient and a scalable way, it is beneficial to maintain a component that represents location-location relationships in LBSN which directly handle location oriented queries and recommendations. In this work, such a component is called *spatial graph*.

We segregate the LBSN data into three graphs, i.e., social graph, activity graph, and spatial graph as shown in Figure 1. Details of these graphs are given below.

Social relationships between users are categorized into two types: directed and undirected. For example, if users are friends on Facebook their relationship is undirected. However, Twitter users are either followers or followees, therefore, their relationship is directed. For the sake of simplicity, this study considers undirected relationships between users. However, directed relationships can be treated similarly. The social graph stores the relationships of users in the form of a graph as shown in Figure 1, where $u_1$ is a friend of $u_2$ and $u_8$.

*Definition 4:* $SocialGraph(G_S)$ is an undirected graph, where vertices represent users and edges represent relationships/ friendship between them. It is given by, $G_S = (U, F)$, where $U$ is the set of vertices that represents users, and $F$ is the set of edges representing connections between them, i.e., $F \subseteq \{U \times U\}$.

Activities of users, i.e., visits/interactions of users at locations, is the essence of LBSN. The activity graph maintains this information in the form of a bipartite graph as shown in Figure 1, where $u_1$ visits $l_1, l_2, l_3$, and $l_5$.

*Definition 5:* $ActivityGraph(G_A)$ consists of users and their Activities. It is given by $G_A = (U, L, A)$, where $U$ and $L$ are the two sets of vertices: $users$ and $locations$, respectively. $A$ is the set of edges which represents activities of users $U$ at locations $L$ at times $T$. It is given by $A \subseteq \{U \times L \times T\}$.

Locations are related to each other in several ways on the basis of their properties, e.g., distances, trajectories, and common visitors. In this study, we consider common visitors for building relationships between locations [1]. The definition

[1]We are aware of other methods to associate locations to each other, e.g., [25]. Our spatial graph extensions can support their applications with simple alterations.

of common visitors is given below.

*Definition 6:* $CommonVisitors(U_c(l_m, l_n))$ is the set of users who visited the locations $l_m$ and $l_n$. It is given by: $U_c(l_m, l_n) = \{u \in U | (u, l_m, t_a), (u, l_n, t_b) \in A\}$, where $u$ is a user that visited both locations $l_m$ and $l_n$, and $t_a$ and $t_b$ are visit times.

The locations $l_m$ and $l_n$ that have at least one[2] common visitor are called *LinkedLocations* and are represented as $l_{lk}(l_m, l_n)$. Next, the spatial graph defined on the basis of common visitors is presented.

*Definition 7:* The spatial graph represents the locations as vertices and the relationships between each other as edges. It is given by $G_L = (L, E)$, where $L$ is the set of vertices and $E$ represents the set of edges, i.e., $E \subseteq \{L \times L\}$. An edge is a pair of LinkedLocations having common visitors.

## III. LBSN Queries

This section presents the LBSN queries. We define query primitives in Section III-A and advanced queries in Section III-B.

### A. Query Primitives

Query primitives are defined as fundamental operations for processing of LBSN queries. Below, we define a comprehensive set of such query primitives for each type of graph, in a way that each segregated graph is sufficient to independently process its primitive queries. The primitives of each graph are grouped based on the type of processing performed into selection, structural, and aggregate queries, as shown in Table I. The primitive queries can further be combined to answer a wide range of general purpose LBSN queries.

The first group contains *social query primitives* that exploit the relationships among users over the social graph. Among them, FF is a selection query; FSoSD and FCF are structural queries; TCU is an aggregate query.

- $FindFriends(u_i, \sigma)$ : Given a user $u_i$ and a social separation degree $\sigma$, return the set of users that require minimum $\sigma$ steps to connect $u_i$.
- $FindSocialSeparationDegree(u_i, u_j)$ : Given the users $u_i$ and $u_j$, return the social separation degree, i.e., minimum number of hops required for connecting them.
- $FindCliqueFriends(u_i)$ : Given a user $u_i$, return the set of users that are friends with $u_i$ and as well as with each other.
- $Top-k\ ConnectedUsers(k, \sigma)$ : Given a value of separation degree $\sigma$ and a parameter $k$, return $top-k$ users based on their maximum number of connections with users in social separation degree of $\sigma$.

In the second group, we define *activity query primitives* that exploit the user and location interactions over the activity graph. Here, FUL and FLV are selection queries; FRL and FSpSD are structural queries; TNL, TV and TVL are aggregate queries.

[2]A threshold describing the minimum number of common visitors among locations to be considered as the LinkedLocations, can be utilized to control the density of the spatial graph. For example, for FourSquare dataset[21] by setting the threshold to 3, the graph density is reduced to $1.02 \times 10^{-6}$.

TABLE I: LBSN query primitives

| Graphs | Categories | Primitives | Notations |
|---|---|---|---|
| $G_S$ | Selection | FindFriends | FF |
| | Structural | FindSocialSeparationDegree | FSoSD |
| | | FindCliqueFriends | FCF |
| | Aggregate | Top-k ConnectedUsers | TCU |
| $G_A$ | Selection | FindUserLocation | FUL |
| | | FindLocationVisitors | FLV |
| | Structural | FindRangeLocations | FRL |
| | | FindSpatialSeparationDegree | FSpSD |
| | Aggregate | Top-k NearestLocations | TNL |
| | | Top-k Visitors | TV |
| | | Top-k VisitedLocations | TVL |
| $G_L$ | Selection | FindCommonVisitors | FCV |
| | | FindLinkedLocations | FLL |
| | Structural | Find LocationSeparationDegree | FLSD |
| | Aggregate | Top-k VisitedLinkedLocations | TVLL |
| | | Top-k ConnectedLinkedLocations | TCLL |

- $FindUserLocation(u_i, t_a, t_b)$ : Given a user $u_i$ and a time interval $[t_a, t_b]$, return the set of locations that $u_i$ visited between $t_a$ and $t_b$.
- $FindLocationVisitors(l_m, t_a, t_b)$ : Given a location $l_m$, and a time interval $[t_a, t_b]$, return the set of users that visited $l_m$ between $t_a$ and $t_b$.
- $FindRangeLocations(l_m, d_r)$ : Given a location $l_m$ and a radius $d_r$, return the set of locations within the circular region centered at $l_m$ with radius $d_r$.
- $FindSpatialSeparationDegree(u_i, u_j, t_a, t_b)$ : Given the users $u_i$ and $u_j$, and a time interval $[t_a, t_b]$, return the minimum number of hops required for connecting $u_i$ with $u_j$ based on their visited locations.
- $Top-k\ NearestLocations(l_m, k)$ : Given a location $l_m$, and a parameter $k$, return the set of k locations in the ascending order of distances from $l_m$.
- $Top-k\ Visitors(k, t_a, t_b)$ : Given a parameter $k$, and a time interval $[t_a, t_b]$, return $top-k$ users based on their maximum number of activities during $t_a$ and $t_b$.
- $Top-k\ VisitedLocations(k, t_a, t_b)$ : Given a parameter $k$, and a time interval $[t_a, t_b]$, return $top-k$ locations with maximum number of visits between $t_a$ and $t_b$.

The third group defines the *spatial query primitives* that exploit the relationships among locations, i.e., spatial graph. Here, FCV and FLL are selection queries; FLSD is a structural query; TVLL and TCLL are aggregate queries.

- $FindCommonVisitors(l_m, l_n)$ : Given the locations $l_m$ and $l_n$, return the common visitors of $l_m$ and $l_n$, if there are any.
- $FindLinkedLocations(l_m, \lambda)$ : Given a location $l_m$ and a degree of separation $\lambda$ , return the set of locations that require no more than $\lambda$ steps to connect $l_m$.
- $FindLocationSeperationDegree(l_m, l_n)$ : Given the locations $l_m$ and $l_n$, return the minimum number of steps required for connecting them.
- $Top-k\ VisitedLinkedLocations(l_m, k)$ : Given a location $l_m$, a set of its $LinkedLocations$ and a parameter $k$, return $top-k\ LinkedLocations$ of $l_m$ based on their maximum number of $CommonVisitors$.
- $Top-k\ ConnectedLinkedLocations(k)$ : Given a parameter $k$, return $top-k$ locations based on their

maximum vertex degree, i.e., connections with other locations.

### B. Advanced LBSN Queries

Advanced LBSN queries utilize the combinations of the query primitives for processing. These queries are divided into four groups on the basis of these primitives, i.e., the queries that are composed of social and activity, social and spatial, activity and spatial, and social, activity and spatial query primitives, respectively. Due to limited space, we provide only two groups of these queries.

In the first group, we define the queries that combine the social and the activity query primitives.

- $RangeFriends(u_i, \sigma, l_m, t_a, t_b, d_r)$ : Given a user $u_i$, a social separation degree $\sigma$, a location $l_m$, a time interval $[t_a, t_b]$ and a radius $d_r$, return the set of friends of $u_i$ in social separation degree of $\sigma$ whose locations during $[t_a, t_b]$ are within the circular region centered at $l_m$ with radius $d_r$.
- $Top-k\ NearestFriends(u_i, \sigma, l_m, t_a, t_b, k, U, A,)$ : Given a user $u_i$, a social separation degree $\sigma$, a location $l_m$, a time interval $[t_a, t_b]$ and a parameter $k$, return the set of $k$ users that are friends of $u_i$ in social separation degree of $\sigma$, in the ascending order of distances between their current locations and $l_m$ during $[t_a, t_b]$.
- $Top-k\ NearestStarGroup(\omega, l_m, t_a, t_b, k)$ : Given a number of users $\omega$, a location $l_m$, a time interval $[t_a, t_b]$ and a parameter $k$, return the $k$ groups of $\omega$ users each, in ascending order of aggregate distance of the user's locations among each other and with $l_m$ during $[t_a, t_b]$ such that users of each group are connected by at least one common friend [18].

In the second group, we define the advanced LBSN queries that combine the social and the spatial query primitives.

- $Top-k\ EndorsedLinkedLocation(u_i, l_m)$ : Given a user $u_i$, a location $l_m$ and parameter $k$, return the $k$ $LinkedLocations$ of $l_m$ that have maximum number of the user's friends as $Common\ Visitors$ in between.
- $Top-k\ CommonVisitorFriends(U)$ : Given a user $u_i$, return the $k$ friends of $u_i$ based on their maximum number of appearances as $CommonVisitors$ with $u_i$.

## IV. GSG Architecture

GSG[3] is a graph-parallel platform that supports distributed processing of LBSN queries on large volume data. The architecture of GSG is composed of four parts, i.e., storage layer, operation layer, index layer and query engine, as shown in Figure 2.

### A. Storage layer

This layer is responsible for providing the data structures and mechanisms for storage of LBSN graphs in a distributed way.

The underlying framework of GSG, GraphX extends the Spark's[13] distributed data scheme, RDD[28]. RDD is an

[3]GSG code at: https://github.com/masaamir/GeoSocial-GraphX.git

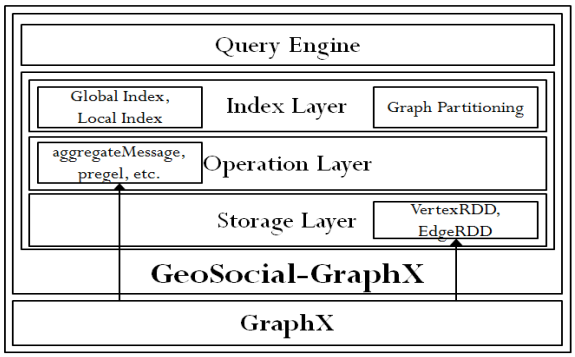



Fig. 2: Architecture of GeoSocial-GrapX

immutable, persistent and parallel data structure that supports distributed operations on it. In GraphX, VertexRDD and EdgeRDD[23] extend RDD to support augmented attributes for vertices and edges, respectively, and are abstractly combined to construct the graphs. GSG utilizes this data scheme for storage of the LBSN graphs.

In LBSN graphs, there are two types of vertices: users and locations. A user vertex maintains a user identifier and a vertex type. A location vertex keeps the location id as the identifier and corresponding location information (e.g., latitude, longitude, and radius) as attributes. Furthermore, the LBSN graphs posses three types of edges with their corresponding unique attributes. An edge of the social graph maintains the ids of users as source and destination vertices, and their corresponding relationship, i.e., friends as an attribute. An edge of the activity graph is composed of ids of a user as a source vertex, a location as a destination vertex, and visit time of user at the location as an edge attribute. An edge of the spatial graph consists of location ids as source and destination vertices, and ids of common visitors as edge attribute. Figure 3 (a) shows an example of the storage of the spatial graph.

In order to store the graphs, both VertexRDDs and EdgeRDDs are partitioned and distributed among nodes, where a node represents a computational unit (e.g., a computer). The vertex cut approach is utilized to partition the graph. This approach helps to reduce more communication and storage overhead as compared to the edge cutting approach [22]. Vertices are kept on the same node with their connecting edges. Figure 3 (b) shows an example of the distributed spatial graph. The graph is split into two partitions, each of which corresponds to a node. Here, vertices 2 and 4 are cut in order to split the graph. These vertices are replicated and stored on both nodes. Next, we provide the algorithm for the construction of the spatial graph. The construction for the social and activity graphs are similar. They are thus omitted due to page limits.

In Algorithm 1, users' check-ins data are utilized to extract two kinds of information, i.e., check-ins that contain the visits of users at their corresponding locations (line 6) and vertices, i.e., locations (line 8). The users are grouped on the basis of their visited locations to extract edge attributes, i.e., common visitors (line 12). These groups are then used to create the list of edges (lines 14-21). The extraction of check-ins, vertices, and grouping of users are performed by map-reduce operations. However, the construction of edges utilizes concurrent

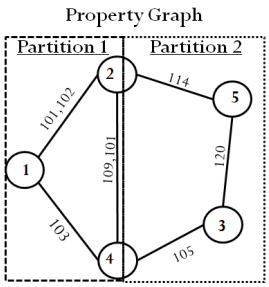
VertexRDD

| Id | Property | Routing |
|---|---|---|
| 1 | (hotel, 0.3) | 1 |
| 2 | (cinema,0.1) | 1,2 |
| 3 | (school,0.6) | 2 |
| 4 | (park,0.7) | 1,2 |
| 5 | (station,0.3) | 2 |

EdgeRDD

| Src-Id | Dest-Id | Property |
|---|---|---|
| 1 | 2 | 101,102 |
| 1 | 4 | 103 |
| 2 | 4 | 109,101 |
| 3 | 4 | 105 |
| 2 | 5 | 114 |
| 3 | 5 | 120 |

(a) vertexRDD and EdgeRDD




(b) property graph

Fig. 3: Data storage for the spatial graph

**Algorithm 1:** ConstructSpatialGraph(UserCheck-ins): $G_L$

```
1  begin
2      Check − ins = ∅
3      edges = ∅
4      vertices = ∅
5      foreach (u_1, loc_1) ∈ UserCheck − ins do
                          // loc_1 contains the attributes of location
6          Check − ins+ = (u_1, loc_1)
7          if l_1 ∉ Location then
8              vertices+ = new Location(loc_1, "location")
9          end
10     end
11     foreach loc_1 ∈ vertices do
12         UsersGroupByLocation+ = Check − ins.groupBy(loc_1)
           // provide visitors of each location
13     end
14     parallel foreach (loc_1, U_1) ∈ UsersGroupByLocation AND
       count <= size(UsersGroupsByLocations)/2 do
15         count + +
16         parallel foreach (loc_2, U_2) ∈ UsersGroupByLocation do
17             if loc_1 ≠ loc_2 then
18                 edges+ = Edge(loc_1, loc_2, (U_1 ∩ U_2))
19             end
20         end
21     end
22     G_L = createGraph(vertices, edges)
23     return spatial graph
24 end
```

programming to exploit maximum computational resources in the cluster. We use scala based concurrent programming model: akka [1] for parallel operations, i.e., nested for loops (lines 14-21 and 16-20). Further, the graph is constructed by utilizing vertices and edges (line 22).

### *B. Operation layer*

The role of this component is to provide the operators that are applied on the graphs for processing of LBSN queries. These operators are grouped into two types, i.e., core operators such as *mapEdges, triplets, and aggregateMessage*, and advanced operators such as *pregel, and connectedComponents*. The core operators transform the structures and properties of input graphs on the basis of user-defined functions. The advanced operators are the optimized variants of core operators that provide advanced graph algorithms. We selectively discuss two core and one advanced operators which we believe best reflect the nature of the operators of corresponding groups.

- **triplet** is a core graph operator that combines edges along with attributes of their neighbouring vertices.
- **aggregateMessage** is a core aggregate operator that aggregates values from neighbouring edges and vertices of each vertex.
- **pregel** is a bulk-synchronous parallel graph operator that recursively use a vertex-centric approach for traversing the graph.

*triplet* logically combines VertexRDD and EdgeRDD, and provides an augmented EdgeRDD. The augmented EdgeRDD contains the attributes of edges and well as attributes of their source and destination vertices. In order to avoid confusion, we call each record of augmented EdgeRDD as *triplet tuple*. *triplet* is utilized when attributes of both vertices and edges are needed. Furthermore, it is utilized for other operations such as *aggregateMessage* and *pregel*.

*aggregateMessage* performs aggregate operations using local communication, i.e., messaging among neighbouring vertices. This operation is considered suitable for LBSN queries that require aggregated information of neighbouring vertices, e.g., FF and FUL. The traditional map-reduce structure falls short in addressing such local interactions that make it expensive to establish communication among vertices. On the other hand, *aggregateMessage* establishes this communication by two sub-functions: *sendMsg* and *mergeMsg*. Following example provides the usage of *aggregateMessage* for processing of a LBSN query.

Consider the query FF for all vertices of the social graph. In order to process it, first, the augmented EdgeRDD is created by using *triplet*. Then, *sendMsg* concurrently sends the ids of source vertices to their destination vertices for each triplet tuple. *mergeMsg* runs at every user vertex and combine all the messages received. Thus, every user vertex ends up with the ids of directly connected vertices, i.e., friends.

*pregel* utilizes vertex-centric approach similar to *aggregateMessage* to communicate among neighbouring vertices. However, it operates recursively in a series of iterative steps, called super-steps. Compared with map-reduce, the parallel communication among neighbouring vertices, and optimized storage, and un-persistence of intermediate results are utilized to efficiently process the query by the *pregel* operator.

For example, consider a query that requires to find degrees of a given user with all other users in the social graph. we use *pregel* and implement the single source shortest path algorithm to process this query in the super-step fashion. In order to process this query, initially, every vertex maintains its degree value from the source vertex that is 1 for directly connected edges and infinity for non-directly connected edges. In each super-step, vertices communicate and receive the updated values of their neighbouring vertices from the previous super-step. On the basis of this information, vertices compute the minimum value to connect the source vertex. Vertices update their values only if the newly computed value is smaller than their current values. These intermediate results are maintained in the memory in an optimized way and used for next super-step. The iterations are continued until there remain no vertices to be updated. At this moment, the values of vertices represent their degrees from the source vertex.

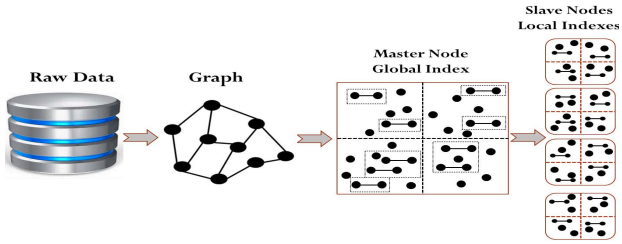



Fig. 4: Indexing mechanism for the spatial graph: k-d tree and quadtree are used for the global and local indexes, respectively.

### C. Index layer

Query primitives that involve spatial attributes (i.e., those on the spatial graph and the activity graph) can be further optimized by indexing techniques. With an index, the mapping phase of map-reduce operators benefits from the efficient locating of data. The indexing scheme for GSG is composed of two parts: *global index* and *local index*.

**Global index** partitions the data across cluster nodes. It is maintained on the master node. The purpose of the global index is to locate the edges of nearby locations on the same node.

We provide two types of global indexes: grid and k-d tree. Grid based indexes divide the geographical space into equal-sized partitions and distribute data accordingly. k-d tree partitions the space in a more adaptive manner. Thus, it is preferred for skewed data.

The computation of the global index is performed by the following steps. (1) Default non-partitioned graph is created and augmented EdgeRDD using *triplet* is obtained. (2) Minimum bounding rectangles (MBRs) are constructed for triplet tuples of the augmented EdgeRDD based on user-defined partitioning strategy, i.e., grid or k-d tree. On the basis of these MBRs, the triplet tuples are mapped and distributed among nodes. (3) Edges are extracted from the augmented EdgeRDD on corresponding nodes. The vertices are also transferred to the nodes that store their connecting edges. (4) The memories used for storing augmented EdgeRDD and non-partitioned graph are released. (5) The k-d tree or grid is maintained at the master node as the global index.

**Local index** For each node, a local index to efficiently retrieve the local data is stored. We provide two indexing methods: a two-dimensional method, i.e., quadtree and a three-dimensional method, i.e., octree, for the spatial graph and the activity graph, respectively. The quadtree is used for the spatial graph because its triplet tuples have only spatial attributes: latitude and longitude. The octree is deployed for the activity graph because triplet tuples in the activity graph are of both spatial and temporal attributes. The maximum number of triplet tuples in a tree node is controlled by a given capacity. When the capacity of a tree node is reached, it splits into sub-tree nodes. It splits into 4 sub-tree nodes for a quadtree and 8 sub-tree nodes for an octree. Thus, the height of a quadtree is more than an octree. The overall process of computation of index is shown in Figure 4.

In order to retrieve a data item, the global index is used to find the node on which the required data resides. Then, the local index is used to efficiently fetch the required data item.

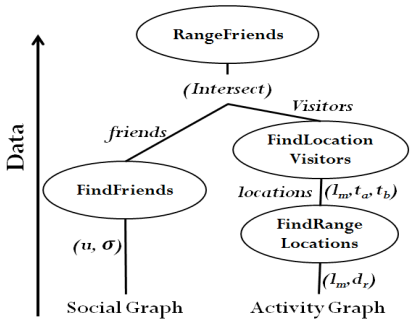



Fig. 5: Query plan for RangeFriends$(u, \sigma, l_m, t_a, t_b, d_r)$

### D. Query Engine

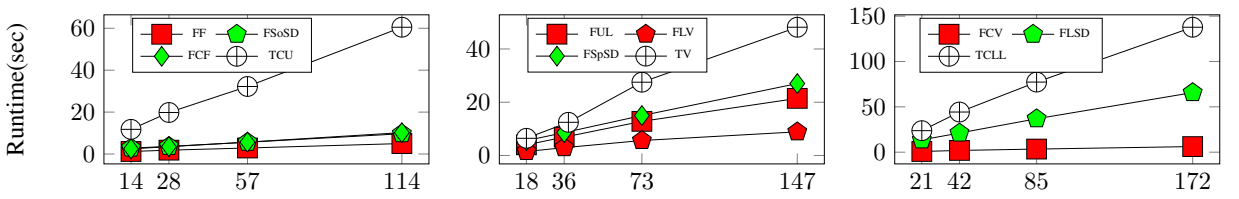
This component provides the implementation of all the query primitives and advanced LBSN queries mentioned in Section III. It exposes a programmable API that provides the opportunity for building general purpose LBSN queries. In this section, we provide the mechanisms to process advanced LBSN queries with the help of a sample query plan.

Advanced queries are segregated into query primitives and processed on corresponding LBSN graphs. There are two kinds of benefits that are achieved by opting this way of query processing. First, data based optimizations are performed on corresponding graphs and utilized for efficient processing of data that can be difficult for hybrid graph structures, e.g., spatial data oriented indexing or graph partitioning strategies for the activity and spatial graph. Second, efficiency is achieved by executing basic primitives in parallel on their respective graphs. Distributed processing of a job may not require all the resources of clusters, i.e., mappers and reducers. Thus, concurrent threads utilize the available resources to efficiently process the queries. The following example provides the processing of an advanced LBSN query: RangeFriends that is defined in Section III-B. The overall query plan is shown in Figure 5. The query is segregated into three primitives: FF, FRL, and FLV. FF is processed on the social graph. Simultaneously, FRL and FLV are processed on the activity graph. We exploit akka programming model [1] for concurrent programming. An independent thread is run for processing of each primitive, i.e., FF and FRL. The output of FRL is utilized for FLV. Then, the intermediate results of FF and FLV are combined by utilizing distributed operations on the cluster to get the final output.

## V. Experiments

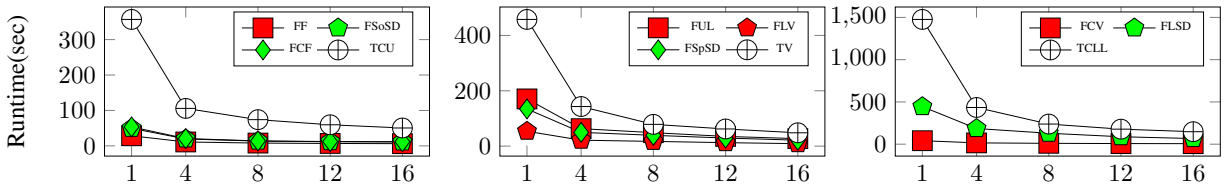
### A. Setup

We conduct our experiments on a multi-processor machine with 4 AMD Opteron 6376 processors with 2.3GH, having 8 cores each. The machine has 512 GB RAM. To simulate the distributed environment, we create five virtual nodes with 4 cores and 100GB of RAM each. Among them, one is master and four are slaves/workers. This is used as a default setting for the cluster of GSG and SH. The OS is Ubuntu 14.04 LTS. GSG is implemented based on GraphX 1.3.1. Moreover,




(a)-No. of edges in $G_S$(in Million) (b)-No. of edges in $G_A$(in Million) (c)-No. of edges in $G_L$(in Million)

Fig. 6: Scalability of the GSG with respect to data size




(a)-No. of cores ($G_S$) (b)-No. of cores ($G_A$) (c)-No. of cores ($G_L$)

Fig. 7: Scalability of the GSG with respect to number of cores

SpatialHadoop v2.2 with Apache Hadoop 1.2.1 is used for comparison. For concurrent programming, we exploit Akka 2.3.6. The experiments are conducted in Scala 2.10.4.

TABLE II: Statistics of Datasets (in Million)

| LBSN | Vertices | | Edges | | |
|---|---|---|---|---|---|
| | Users | Locations | $G_S$ | $G_A$ | $G_L$ |
| BrightKite | 0.06 | 0.77 | 0.2 | 4.49 | 0.65 |
| Gowalla | 0.2 | 1.28 | 0.95 | 6.44 | 0.15 |
| FourSquare | 0.02 | 0.85 | 0.12 | 2.07 | 0.39 |
| Synthetic | 31.18 | 50.97 | 114.64 | 147.34 | 172.02 |

**Datasets.** We utilize both real and synthetic datasets. Three real LBSN datasets are used: Brightkite [20], Gowalla [20] and FourSquare [21]. The number of edges in the spatial graph is determined by the minimum number of *CommonVisitors*. Here, in order to capture all the common visitors among locations, we set its value to 1. To test the scalability of our proposals, we create a synthetic dataset of size 25GB by iteratively scaling up the graphs of the Brightkite dataset. In order to create this synthetic dataset, in each iteration, first nodes and edges are replicated then offset with new ids thus a new dataset is fetched. The vertices in both the original and replicated datasets are connected by random connections. At each iteration, 1% of the vertices in the original graph are randomly chosen to connect with those in the replicated graph. The iterations are continued until required dataset is retrieved. The details about datasets are shown in Table II.

### B. Results

**Scalability:** The first experiment tests the scalability of query processing. Few queries from each primitive groups of the graphs, i.e., selection, aggregate, and structural query primitives, that best reflect the behavior of the corresponding groups are considered for evaluation. We evaluate the scalability of GSG by performing two types of experiments.

First, we analyze the query execution time by keeping the number of cores constant and varying the size of the dataset. The purpose is to analyze the performance of GSG when the dataset grows. Figure 6 presents the results. Here, it is observed that the scalability for all types of queries is very close to linear. This shows that GSG scales well when data grows. All the queries show the same pattern, however, aggregate queries scale a bit less than the rest, since they require the processing of the whole graph.

Second, we fix the dataset size and vary the number of cores. The purpose of this experiment is to test how GSG scales with the amount of computational resources. The results are shown in Figure 7. Here, as the number of cores grows from 1 to 16 cores, we can see the maximum speed-up is 9.8x. The savings in execution time are better for fewer nodes, i.e., we see a speed-up up to 3.4x for 1 to 4 cores. However, afterwords the gain in speed-up decreases gradually, i.e., from 4 to 16 cores the speed-ups are 6.2x, 8.3x, and 9.8x, respectively. The reasons for the smaller gains when adding many cores is that the ratio of communication to computation cost grows. This is consistent with the observations in [26]. Further, it can also be observed that aggregate queries show better speed-up. Because they require more processing and thus decrease the effect of communication.

**SpatialHadoop Comparison:** We compare the efficiency, scalability, and ease of implementation of GSG with the leading Big Data system, SpatialHadoop (SH). SH is a map-reduce based system, however, GSG's underlying framework uses memory based computation. Thus, to make a fair comparison, we do not count the I/O cost. We consider queries from all types of query primitives, i.e., FF from the selection, FCF, FLSD and FRL from the structural, and TNL from the aggregate query primitives. We use FF to evaluate the vertex-centric approach of the operator *aggregateMessage (AM)* in GSG. Furthermore, it is implemented with the different number of degrees, i.e., $\sigma$ (1-3) to reflect the performance with respect to different iterations. To simulate a real workload, we execute multiple queries concurrently, with random input parameters. The results of all the datasets provide similar trends, however,

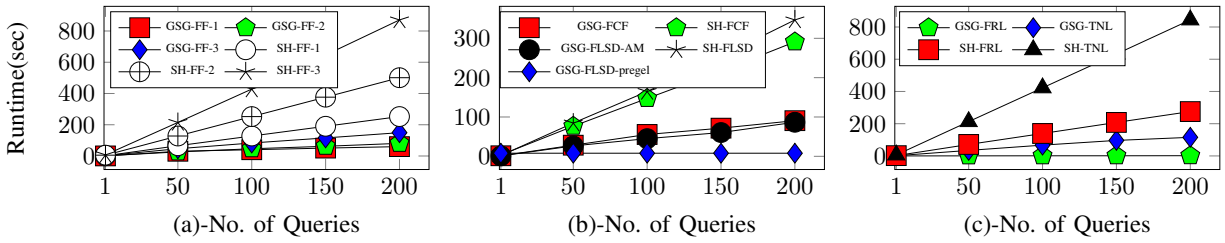



Fig. 8: Comparison of computation time of GeoSocial-GraphX and SpatialHadoop (BrightKite)

TABLE III: Comparison of computation time (in seconds) of spatial primitives on activity graph and spatial graph

| Data set | Graph | Time | Query | Activity Graph | | | | Spatial Graph | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | 1 | 5 | 10 | 15 | 1 | 5 | 10 | 15 |
| Brighkite | $G_A$ | 43.7 | FCV | 7.5 | 35.4 | 71.3 | 109.1 | 0.3 | 1.6 | 3.1 | 3.9 |
| | | | FLL | 31125.6 | 155625.9 | 311254.9 | 466881.9 | 2.4 | 11.9 | 21.5 | 32.6 |
| | | | TCLL | 31127.5 | 155620.9 | 311253.4 | 466880.5 | 2.4 | 7.7 | 14.7 | 21.3 |
| | $G_L$ | 31127.12 | TVLL | 31125.4 | 155622.8 | 311249.7 | 466871.5 | 1.7 | 6.3 | 12.2 | 17.8 |
| | | | FLSD | 31127.2 | 155632.6 | 311264.6 | 466863.8 | 3.4 | 11.6 | 22.3 | 30.5 |
| Gowalla | $G_A$ | 59.4 | FCV | 12.8 | 48.1 | 95.9 | 127.1 | 0.06 | 0.2 | 3.1 | 4.9 |
| | | | FLL | 52180.2 | 104360.5 | 156541 | 208721.5 | 0.4 | 0.9 | 2.4 | 3.7 |
| | | | TCLL | 52180.2 | 104360.7 | 156541.2 | 208721.7 | 0.3 | 1.02 | 2.3 | 3.8 |
| | $G_L$ | 52183.1 | TVLL | 52180.2 | 104360.7 | 156541.2 | 208721.6 | 0.2 | 0.6 | 1.52 | 2.67 |
| | | | FLSD | 52184.2 | 104372.1 | 156563.6 | 208757.4 | 0.39 | 1.0 | 2.8 | 4.1 |
| FourSquare | $G_A$ | 52.19 | FCV | 8.7 | 37.8 | 75.9 | 108.9 | 0.5 | 1.1 | 2.1 | 2.9 |
| | | | FLL | 24078.6 | 48157.3 | 72235.9 | 96314.7 | 2.9 | 12.3 | 21.9 | 30.4 |
| | | | TCLL | 24078.6 | 48157.3 | 72236.1 | 96314.8 | 2.6 | 8.4 | 17.1 | 25.1 |
| | $G_L$ | 24079.2 | TVLL | 24078.5 | 48157.1 | 72235.8 | 96314.4 | 1.3 | 5.5 | 10.6 | 16.1 |
| | | | FLSD | 24080.6 | 48163.1 | 72247.8 | 96333.7 | 2.3 | 10.4 | 20.1 | 30.9 |

due to page limits, the results of Gowalla and FourSquare are omitted. Figure 8 (a) shows the results for the Brightkite dataset where GSG outperforms SH on average by 2.5 times for a single query and this time difference increases up to 4 times for 3 degrees. Moreover, when going from 1 to 200 concurrent queries, GSG scales 1.5 times better than SH for 1-degree and 3.5 times better for 3-degrees. GSG's better performance is due to its vertex-centric approach as compared to SH's map-reduce based methods.

Figure 8 (b) shows the results of structural queries, i.e., FCF and FLSD. GSG outperforms SH by 2.5x and scales 1.3 times better than SH from 1 to 200 queries. GSG uses the operators *AM* and *pregel* while SH utilizes traditional map-reduce methods for the implementation of FLSD. Vertex-centric approach by *AM* shows a significant improvement as compared to SH and outperforms it by 2x. The computation time of *pregel* is 4 times slower as compared to *AM* for a single query. The reason is that *pregel* computes the social separation degree of a given vertex with all vertices of the social graph. However, it retains the query time for multiple queries and outperforms *AM* by 9x for 50 queries. The speed-up increases with the number of queries. Thus, we conclude that *pregel* is beneficial for the queries that require a traversal of the whole graph or multiple concurrent queries. Furthermore, it also shows better performance in processing higher degrees operations. However, *AM* is effective for a single query that requires processing up-to few degrees of vertices.

SH is designed for efficient processing of spatial data. Thus, we utilize its corresponding features to compare the results with GSG. We implement aggregate spatial queries, i.e., FRL and TNL to compare the results. These queries are implemented by considering the locations as *Point* in SH. Furthermore, indexes for both SH and GSG are used to compare the computation time. The corresponding results are depicted in Figure 8 (c). GSG computes FRL 20 times faster and scales 7 times better up to 200 queries as compared to SH. Similarly, for FNL, GSG outperforms SH by 4x and scales 1.5 times better. Results for the Gowalla and FourSquare datasets are similar and not shown due to space constraints.

GSG exposes a programmable API for building general purpose queries. We analyze the ease of implementation of queries in GSG as compared to SH. We calculate and compare the Lines of Code (LoC), required to implement queries for both systems. The queries, FF, FCF, FLSD, FRL and TNL take 3 LoC with GSG, but requires 31, 39, 34, 15, and 12 LoC with SH, respectively, i.e., 4-13 times more LoC as compared to GSG.

**Spatial Graph:** This set of experiments shows the significance of GSG's materialized spatial graph in processing of the spatial primitives. These primitives can also be processed on the activity graph, i.e., by traversing the user-location data. However, the activity graph based solution has to repeatedly retrieve a large number of check-ins for query processing. The spatial graph based solution alleviates the problem by reusing the intensive computation during the graph construction. We implement the query primitives as listed in row $G_L$ of Table I to compare the results. These primitives are implemented on the activity graph as well as on the spatial graph. We compare the evaluation time for the query processing on corresponding graphs as shown in Table III.

It can be observed that the query execution time of a single query(FLL) on the spatial graph for Brightkite is 13,000

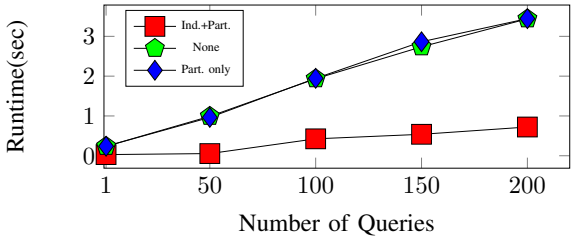



Fig. 9: Improvement of the GSG with partitioning and indexing for the query: FRL. (FourSquare)

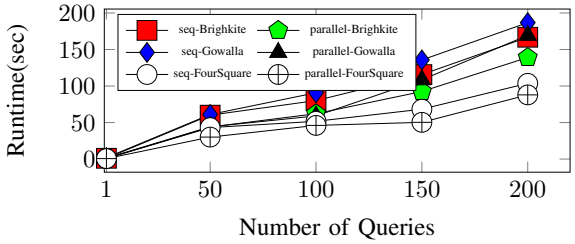



Fig. 10: Optimization for an advanced LBSN query: Range-Friends. (BrightKite)

times less when using the activity graph. This time difference increases up to 14,000 times for 15 queries. Among the queries examined, FCV is an outlier. For FCV the speed-up increases from 25 times to 28 times for one to 15 queries. The reason is that its processing requires less traversal of the activity graph as compared to the other queries. The speed-up is more than four orders of magnitude for Gowalla and three orders of magnitude for the FourSquare. The reason for less speed-up for FourSquare is high number of edges in the spatial graph as compared to the Gowalla. These results confirm the importance of the spatial graph for processing of the spatial primitives. Moreover, it can be observed from the Table III that the construction time of the spatial graph is significantly higher than that of the activity graph, i.e., 7000 times in the Brightkite. Furthermore, materialization costs for Brightkite, Gowalla, and FourSquare datasets are 10 MB, 5 MB, and 8 MB, respectively, which is negligible in comparison to other storage. Taking this in combination, we observe that the seemingly high cost of computing the spatial graph is offset already *after a single query*

**GSG Optimization:** We evaluate the optimizations of GSG by using partitioning and indexing methods. We select a query from the activity graph primitives: FRL that best reflects the spatial nature of LBSN queries. We compare its execution time in three conditions, i.e., non-indexed, partitioned only, and combined partitioned and indexed. We use the k-d tree for partitioning of data among nodes. Furthermore, we use the k-d tree for the global index and octree as the local index. Figure 9 provides the results for this experiment. Here, it can be observed that the query execution time for partitioned data is similar to the time of non-partitioned and non-indexed data. The reason is that by partitioning the data, we locate the data items of nearby places on the same node. However, for FRL, the whole dataset is traversed to find the results because the locations of required data points are not determined. Thus, only partitioning is not enough to improve the query execution time. On the other hand, by providing indexes with partitioned data, the nodes that contain the required data are identified and only relevant data is traversed. Thus, for combined partitioning and indexing method, a speed-up of 7x is achieved for FourSquare. Here, the speed-up remains approximately same for an increasing number of concurrent queries. In the case of BrightKite and Gowalla, we get a speed-up of around 5x and 6x, respectively. However, due to limited space, results are not shown here. Furthermore, the computation time for multiple queries decreases significantly. The reason is that the usage of concurrent programming maximizes the computational resources.

We further evaluate the optimization for processing of advanced LBSN queries. We select an advanced query: Range-Friends for the evaluation. The overall query plan is shown in Figure 5. The query is decomposed into 3 query primitives, i.e., FF, FRL, and FLV. The query is implemented by two ways. First, we compute the query primitives in a sequential way. Second, the query primitives are computed by using the Akka based concurrent programming model. We further run the experiments for multiple queries. These queries are also processed in parallel. Figure 10 presents the results on BrightKite dataset, where seq represents sequential traversal. Here, it can be observed that we achieve a speed-up of 1.4x for a single query. The dependency of FLV on FRL causes the decrease in parallelism and therefore, the speed-up. On increasing the number of concurrent queries, the speed-up reaches to 1.2x. The reason is that concurrent threads of multiple queries acquire computational resources that cause the decrease in speed-up for parallel computation of query primitives. Similar trends are observed from results on Gowalla and FourSquare datasets. The results are omitted due to page limits.

## VI. Related Work

In this section, we present the existing work in two parts, i.e., data processing systems in terms of their appropriateness for processing of LBSN data (graph and spatial data), and data management strategies for LBSN data.

**Big Data Systems:** First, We provide the analysis of well known open source graph database systems. Neo4j[10] is a single tier graph database. It maintains data in graph formats thus, efficiently process corresponding operations. Apache Giraph[2] and GraphLab[6] are general purposed distributed graph processing systems. Giraph is an analogous system of the proprietary Pregel. Pregel and Giraph exploit Bulk Synchronous Parallel model while GraphLab implements an asynchronous method. These systems work on a vertex-centric approach, thus suitable for iterative graph algorithms and graph analytics. There are several other graph database systems that lies in same category such as VertexDB[16], HypergraphDB[8], flockdb[3], and Trinity[15]. GraphX[7] is a graph parallel platform that is built on top of the distributed tabular data framework, Spark [13] with the optimization for

graph processing. Thus, it efficiently supports both tabular and graph based data processing. However, none of these systems explicitly contribute towards spatial operations.

Next, we analyze well known open source spatial data processing systems. PostGIS [12] is a spatial database extension for PostgreSQL. It is a RDBMS based system with the support of processing at a single machine. Neo4j-Spatial[11] is a single tier graph database system with the support of spatial operations. Esri provides tools and APIs [5] for spatial data processing with distributed computing framework Hadoop, i.e., spatial framework for Hadoop and geoprocessing tools for Hadoop. Further, SpatialHadoop[14] is also a distributed system, especially designed for spatial data that optimizes the spatial queries by using map-reduce based spatial indexes. However, none of them can handle large volume spatial graph data efficiently.

We choose GraphX as the underlying framework for building our GSG platform because it has better support for efficient and scalable graph processing. The GraphX data structures and operators are given in Section IV along with the corresponding changes and extensions in GSG.

**LBSN Data Management:** This section narrates the overview of existing work in perspective of LBSN data management. In [24], authors maintain the LBSN data in an adjacency matrix. That may cause storage and computation overhead in large LBSNs. In [17], authors aggregate the data and exploit hybrid indexes for data access that may require extensive cost due to the rapid increase of check-ins. In [18], authors segregate the LBSN data into social and geographical (activity) components. This framework works efficiently for the social and activity-centric queries. However, for location-centric queries they need to recursively traverse geographical component that causes the huge computation overhead. Thus, a data management mechanism that could efficiently maintain all the types of queries is required. Further, it should be scalable enough to deal with huge data of LBSN.

Our proposed GSG framework provides a data management mechanism to deal with these issues. We segregate the LBSN data into three graphs namely, social graph, activity graph and spatial graph. Furthermore, it is capable of dealing with LBSN queries with graph and spatial nature on large volumes of data.

## VII. Conclusion and Future Work

In this paper, we present a platform, GeoSocial-GraphX (GSG) for distributed processing of LBSN queries on large data volumes. GSG segregates the LBSN data into three graphs, namely social graph, activity graph, and spatial graph, to support scalable and efficient processing. A comprehensive set of query primitives is defined on these graphs. The primitives can be combined to answer a wide range of general purpose LBSN queries. GSG also features distributed storage mechanisms, vertex-centric operators, and spatial indexing and partitioning techniques that are used to further optimize the processing and scalability of LBSN queries.

The experiments on real and synthetic datasets show that GSG scales well on multicore architecture and for increasing dataset sizes. The partitioning and indexing methods improve the performance up to 6 times. Furthermore, segregation of LBSN and usage of the spatial graph improves the processing of location-centric queries upto four orders of magnitude. GSG also outperforms the competing system, SH in terms of efficiency, scalability and LoC by up to 20X, 7X and 13x, respectively.

In the future, we plan to consider updates in LBSN graphs. The graphs will be updated using batch updates to optimize the update process. Furthermore, GSG will be enriched with more LBSN queries with corresponding optimization methods for query specific data management and processing techniques.

## Acknowledgments

This research has been funded in part by the European Commission through the Erasmus Mundus Joint Doctorate "Information Technologies for Business Intelligence - Doctoral College" (IT4BI-DC).

## References

[1] akka. http://akka.io/.
[2] Apache giraph. http://giraph.apache.org/.
[3] flockdb. https://github.com/twitter/flockdb.
[4] Foursquare. https://foursquare.com/about.
[5] Gis tools for hadoop. http://esri.github.io/gis-tools-for-hadoop/.
[6] Graphlab. https://graphlab.org.
[7] Graphx. http://spark.apache.org/graphx/.
[8] Hypergraphdb. http://www.hypergraphdb.org/.
[9] mongodb. https://www.mongodb.org/.
[10] Neo4j,. http://neo4j.com/.
[11] Neo4j-spatial. https://github.com/neo4j-contrib/spatial.
[12] Postgis. http://postgis.net/.
[13] Spark. https://spark.apache.org/.
[14] Spatialhadoop. http://spatialhadoop.cs.umn.edu/.
[15] Trinity. http://research.microsoft.com/en-us/projects/trinity/.
[16] Vertexdb. http://www.dekorte.com/projects/opensource/vertexdb/.
[17] A. Amir, A. Efrat, J. Myllymaki, L. Palaniappan, and K. Wampler. Buddy tracking - efficient proximity detection among mobile friends. In *INFOCOM*, 2004.
[18] N. Armenatzoglou, S. Papadopoulos, and D. Papadias. A general framework for geo-social query processing. In *PVLDB*, 2013.
[19] J. Bao, Y. Zheng, and M. F. Mokbel. Location-based and preference-aware recommendation using sparse geo-social networking data. In *GIS*, 2012.
[20] E. Cho, S. A. Myers, and J. Leskovec. Friendship and mobility: User movement in location-based social networks. In *KDD*, 2011.
[21] H. Gao, J. Tang, and H. Liu. Exploring social-historical ties on location-based social networks. In *AAAI*, 2012.
[22] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.
[23] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.
[24] W. Liu, W. Sun, C. Chen, Y. Huang, Y. Jing, and K. Chen. Circle of friend query in geo-social networks. In *DASFAA*, 2012.
[25] J. Shi, N. Mamoulis, D. Wu, and D. W. Cheung. Density-based place clustering in geo-social networks. In *SIGMOD*, 2014.
[26] R. S. Xin, D. Crankshaw, A. Dave, F. M. J. Gonzalez, Joseph E, and I. Stoica. Graphx: Unifying data-parallel and graph-parallel analytics. In *AAAI*, 2012.
[27] M. Ye, P. Yin, and W.-C. Lee. Location recommendation for location-based social networks. In *GIS*, 2010.
[28] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.
[29] Y. Zheng, L. Zhang, X. Xie, and W.-Y. Ma. Mining interesting locations and travel sequences from gps trajectories. In *WWW*, 2009.