MQuery: Fast Graph Query via Semantic Indexing for Mobile Context

Yuan Zhang⋆, Ning Zhang⋆, Jie Tang⋆, Jinghai Rao†, and Wenbin Tang⋆
⋆*Computer Science Department, Tsinghua University, Beijing, China*
†*Research Center, NOKIA Inc., China*
{fancyzy0526, ning.zhang.cs}@gmail.com, jietang@tsinghua.edu.cn
jinghai.rao@nokia.com, tangwb06@mails.tsinghua.edu.cn

***Abstract*—Mobile is becoming a ubiquitous platform for context-aware intelligent computing. One fundamental but usually ignored issue is how to efficiently manage (e.g., index and query) the mobile context data. To this end, we present a unified framework and have developed a toolkit, referred to as MQuery. More specifically, the mobile context data is represented in the standard RDF (Resource Description Framework) format. We propose a compressed-index method which takes less than 50% of the memory cost (of the traditional method) to index the context data. Four query interfaces have been developed for efficiently querying the context data including: instance query, neighbor query, shortest path query, and connection subgraph query. Experimental results on two real datasets demonstrate the efficiency of MQuery.**

***Keywords*-mobile social network; graph query; MQuery; SGI**

I. Introduction

Mobile devices have become one of the most important sources of one's personal information. In our daily life, people use mobile phones to send/receive messages and emails, take photos/videos, organize calendars, and manage address book. There is little doubt that mobile is becoming a ubiquitous platform for relationship-building, learning, entertainment, commerce, and social networking. At the same time, allowed by the rapid growth of flash memory, the data generated from the above activities can be stored on the mobile phone for a long period of time. With the large amount of mobile data, a lot of new applications can be developed such as user preference learning, context aware computing [3] and GPS-aware based social networking services [13][16].

Now, an important problem is how to manage the mobile context data. Since the mobile data are generated by different applications, e.g. camera, calendar, and message applications, a straightforward requirement is to link them together to support "semantic"-based search. For example, we can find relevancy between a photo and an event via the calendar if the photo is taken in the same period when the event is happening. Figure 1 illustrates the example of a "graduation ceremony" event on July 5th. Besides the photo-calendar relationship we just mentioned, the calendar entry can be associated to text messages that talk about the event. In addition, the annotated tags to a photo can be linked to a contact in the address book. In this example, "Wenchang" is the name of a contact in the address book and there is

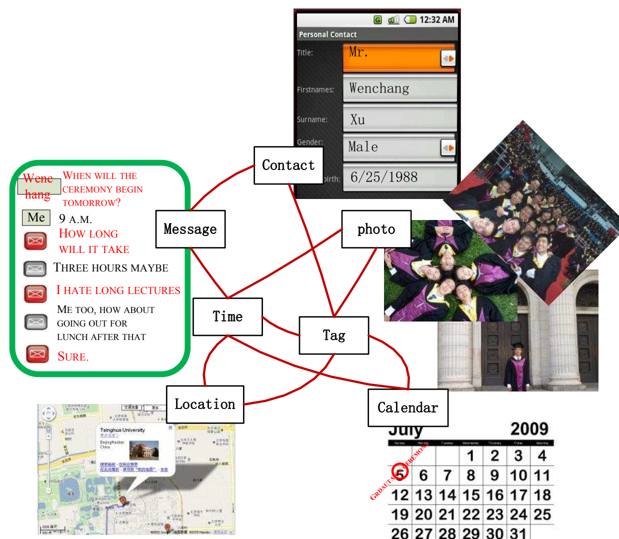

Figure 1. An example of the mobile context.

also a tag assigned to the picture taken on the event day. In some use cases, the mobile user needs to navigate from one object (e.g., a calendar event) to another object (e.g., a photo) through some common attributes. For example, the user might be willing to call "Wenchang" when he/she sees the photo. Other use cases may require efficient query mechanisms to discover the relationship between two or more objects, such as finding the common interests of two contacts based on the text messages received from them.

There are primarily two research trends for searching the mobile data. The first one follows the thread of cloud computing. Basically, all data are uploaded to a server (or cloud), then index and search are performed by the server. Finally the search results are sent back to the mobile devices. There are two limitations of the mobile cloud computing: data transfer cost and privacy. Frequent data upload/download will result in unnecessary power consumption and high communication cost; while the privacy issue restricts many important personal information to be uploaded onto the server. This leads to another research trend to perform the query/search directly on the mobile devices referred to as mobile search.

For mobile search, a fundamental issue is to design a mechanism to efficiently store and access the context data (e.g., emails, messages, photos, and events). However, few research systematically investigate this problem. Instead, it is tackled as an engineering issue and various applications on mobiles manage their data separately. Obviously, this may result in redundancy and inconsistency. A common data representation and management framework is thus needed. RDF, as a standard semantic data model, can be used to represent the mobile context. However, despite of many RDF storage and query solutions have been developed, few of them consider the mobile environment. In particular, how to achieve acceptable time and memory performance is the key to the success of using RDF as a mobile storage solution. Recently, several studies have been conducted to develop various data mining applications on mobile devices. For example, MobileMiner [18] is a platform for mining user profiles from the users' continuous moving and calling records. Works [21] try to discover the transportation mode and classical travel sequences from the GPS information collected from mobile devices. However, most of them do not emphasize the fundamental question: how to efficiently store and query the mobile context data? The problem is non-trivial and poses several unique challenges:

- *Mobile restriction.* It is necessary to consider the following restrictions for mobile devices: limited computing power, limited memory, and limited battery capacity.
- *Index compression.* Due to the memory limitation, it is infeasible to directly adopt the conventional indexing technique to the mobile context. It is important to design a method to compress the graph index while preserving the accessing efficiency.
- *Efficient query interface.* Another challenge is how to design a general and efficient query interface for the mobile context data.

To address the above challenges, we formulate and tackle the problem of querying the mobile context data, and make the following contributions. First, to address the limitation of computing power and memory capacity, we present an index compression method, which significantly improve the memory efficiency. Second, we formalize the mobile search problem and design four general query interfaces to deal with the usual query of the mobile context data and develop a practical toolkit, MQuery. Third, we apply MQuery on two real-world datasets, Kaleio Photo and Simple Context. Experimental results on the two datasets demonstrate the effectiveness and the efficiency of the MQuery toolkit.

II. Problem Formulation

To manage the various heterogenous data resident on mobile devices in a common framework, we use RDF to represent the mobile context. In general, the mobile context data can be represented as a graph $G = (V, E, W, T)$, where V is a set of nodes collected from the mobile devices, such as text messages, photos, and calendar; E is a set of edges/relationships between different nodes; W denotes the set of words that appear in the description of all nodes V; T denotes the set of types of the nodes. Each node $v \in V$ is represented by a triple $v = (id, t, D)$, where id is the index of object v, $t \in T$ is the type of v, and $D \subset W$ is a set of words (text content) associated with node v. In the rest part of the paper, we will use id_v, t_v and D_v to represent id, type and description of v respectively.

Given this, our problem can be formalized as follows:
Mobile context: Mobile context includes all the heterogenous data on mobile devices, which can be represented as the graph $G = (V, E, W, T)$.
Mobile querying task: Given a mobile context $G = (V, E, W, T)$ and a user-specific query q, the task of mobile query is to find the most relevant objects (or subgraph) from the graph G.

The above definition of mobile query is very general. In practice, the intentions of users' queries on the heterogenous graph may be different. Without loss of generality, we define four kinds of queries, i.e., Instance Query, Neighbor Query, Shortest Path Query, and Connection Subgraph Query.

- Instance Query: Given a keyword-based query $q = \{w_{q_1}, w_{q_2}, \ldots, w_{q_{n_q}}\} \subset W$ and a node type $t \in T$, which nodes $V' \subset V$ of type t are the most relevant to the query q?
- Neighbor Query: Given G, a node v_s and a set of node types $T' \subset T$, find all nodes $\{v_t\}$ of type $t \in T'$ from the graph G within the length bound l (i.e., v_s has a shortest path less than l to v_t).
- Shortest Path Query: Given G, two nodes $v_s, v_t \in V$, find the shortest path from v_s to v_t.
- Connection Subgraph Query: Given G, two nodes $v_s, v_t \in V$, find a connection subgraph G' with k nodes and containing v_s, v_t, which maximizes the value of a goodness function $H(G')$.

The instance query is similar to the document retrieval on the Web, except that we have more than one type objects. The other three queries are general for graph query. In this way, our method for dealing with the queries can be easily adapted to different applications such as context-aware recommendation and knowledge reasoning. Our formulation of the mobile query task is different from existing works on graph query. For querying on RDF data, Miller et al. [12] propose the RDQL language. However, the language is designed to answer all kinds of queries and results in a low efficiency in some queries (e.g., the above queries).

III. The Proposed Approach of MQuery

In order to address the query problem on mobile devices, we have developed a toolkit MQuery, which provides one index and four query interfaces. The input of MQuery can be any graph or text data from the file system or database on

the mobiles. As the memory of mobile devices is limited, we propose an index compression method, which significantly reduces the memory cost for storing the index of content and graph structure. Furthermore, we propose a novel connection subgraph query, which aims to find a subgraph connecting two nodes in the graph data. We propose a polynomial approximation algorithm to solve the problem.

A. Index and Compression

We can build an inverted index for the content of the nodes. For efficient query, we need load the index and the graph structure into memory. However, loading such an index and a graph into the limited memory of mobile devices is expensive. This leads us to think about the compression technology. There are several state-of-art compression methods toward this purpose, for example, Variable-Byte Coding [14], S9 [1], Rice Coding [22], and PForDelta Coding [23].

In our work, as the input data is usually a (RDF-based) graph, directly applying the existing compression methods is also infeasible. We propose an extension of S9 to compress both the index for text content and graph structure. In the experimental section, we will show that the size of the index can be significantly reduced while the time cost for compression and decompression is very limited.

The basic idea of S9 [1] is to pack as many values as possible into a 32-bit word. This is done by dividing each word into 4 status bits and 28 data bits. S9 uses nine ways to divide up the 28 data bits, such as twenty-eight 1-bit numbers, fourteen 2-bit numbers, nine 3-bit numbers (one bit unused), four 7-bit numbers, or two 14-bit numbers. Decompression can be optimized by hardcoding each of the nine cases using fixed bit masks.

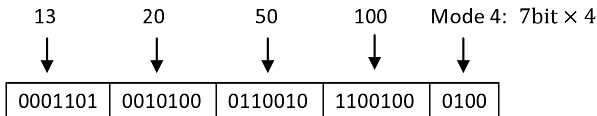

Figure 2. An example of S9 (Simple9 coding).

Figure 2 shows an example of compressing four numbers 13, 20, 50 and 100. Because the maximum number 100 is less than 2^7 but larger than 2^5, S9 will choose mode 4: 7bit×4, using four 7bits number. We can see that the storage size for the four numbers is decreased from four 32-bit integers to only 32-bit integer.

In the inverted index for the content information, each word w_i is associated with a sequence of node ids $id_{v_1}, id_{v_2}, \ldots, id_{v_{N(w_i)}}$, which indicates word w_i appears in $D_{v_1}, D_{v_2}, \ldots, D_{v_{N(w_i)}}$. Before compression, storing these ids needs $N(w_i)$ integers. After packing them using S9, the memory cost will decrease significantly. [19] and [15] further introduce an approach, called d-gap, to obtain a higher compression ratio.

d-gap This approach sorts the sequence to guarantee $id_{v_1} < id_{v_2} < \ldots < id_{v_{N(w_i)}}$. Then the original id sequence is replaced by $id_{v_1}, id_{v_2} - id_{v_1}, \ldots, id_{v_{N(w_i)}} - id_{v_{N(w_i)-1}}$ and we compress the new d-gap sequence. Because the average value of the sequence becomes smaller, it is expected that a 32-bit integer can store more values, thus achieving a higher compression ratio.

SGI To index and compress the graph, the situation would be a bit different. Generally, the graph structure is stored as adjacency lists, in which each node v is associated with a list of neighboring nodes $id_{v_1}, id_{v_2}, \ldots, id_{v_{d(v)}}$ where $d(v)$ is the degree of v. For consistent purpose, we refer to the adjacency lists as graph index and the problem is still how to compress the index. Similarly we can use S9 or d-gap to compress the graph index; however, we see that different assignments of the id to nodes on the graph will result in very different compression ratios of the index. For example, if the five numbers are all smaller than 20, then we can use a $5bit \times 5$ mode, thus one 32-bit integer is enough; while if a number is 100 and the other four numbers are smaller than 10, we have to use a $7bit \times 4$ mode, which finally requires two 32-bit integers and leads to an inevitable waste of some memory. By revisiting this problem, we find that carefully re-allocating id to nodes on the graph to make the ids in an adjacency list as close as possible can result in a better compression performance.

Based on these considerations, we propose a method, called **SGI** (S9 for Graph Index), for indexing graph data and compressing the graph index. The basic idea is to find an id assignment such that for each node the difference of adjacent nodes' id is as small as possible. This is because a small difference can easily achieve a higher compression ratio. Specifically, we first re-allocate the id of v from id_v to id'_v satisfying that $id'_{v_i} < id'_{v_{i+1}}$. Then we compress the d-gap sequence $id'_{v_1}, id'_{v_2} - id'_{v_1}, \ldots, id'_{v_{d(v)}} - id'_{v_{d(v)-1}}$ using S9. The sum of the sequence is

$$sum(v) = id'_{v_1} + \sum_{i=2}^{d(v)} (id'_{v_i} - id'_{v_{i-1}}) = id'_{v_{d(v)}} \quad (1)$$

Our goal is then to minimize $sum(v)/d(v)$ for each node, thus $\sum_{v \in V} sum(v)/d(v) = \sum_{v \in V} id'_{v_{d(v)}}/d(v)$ for all nodes, where $id'_{v_{d(v)}}$ is the maximum neighbors' id of v.

This problem itself is an intractable problem. We propose an approximation algorithm to re-allocate the ids of nodes. Our main idea is as follows: given a graph comprised of n nodes, we always allocate larger id to the node with smaller *weight*. The weight of a node is defined as follows:

$$weight(v) = \sum_{u \in neigh(v)} \frac{1}{d(u)} \quad (2)$$

where $neigh(v)$ represents the neighbors of node v. $weight(v)$ can be used to evaluate the influence of id'_v on $\sum_{v \in V} sum(v)/d(v)$. For each neighbor u, if the maximum neighbor id of u is id'_v, then $sum(u)/d(u) = id'_v/d(u)$.

Algorithm 1: Re-allocate id of nodes

Input: A data graph $G = (V, E, W, T)$;
Output: Node id id'_v for each $v \in V$;

1.1 $\forall v \in V$ calculate and insert its weight $weight(v)$ into the priority queue;
1.2 $S \leftarrow V$;
1.3 **for** $i \leftarrow 1$ *to* n **do**
1.4   Extract minimum $weight(v)$ from the queue and let $id'_v = n - i + 1$;
1.5 $S \leftarrow S \setminus \{v\}$;
1.6   **foreach** $u \in S, \exists w \in V, (v, w) \in E, (w, u) \in E$ **do**
1.7   Update $weight(u) \leftarrow weight(u) - 1/d(w)$;
1.8   **end**
1.9 **end**

For example, id n should be given to the node v with the smallest weight. We greedily minimize the influence of id'_v and thus minimize the sum of $sum(v)/d(v)$. In summary, our algorithm consists of two steps (1) sort the nodes according to their weight. (2) given the sorted nodes in terms of weight: $weight(v_1) \leq weight(v_2) \ldots \leq weight(v_n)$, we assign their ids as: $id'_{v_1} = n, id'_{v_2} = n - 1, \ldots, id'_{v_n} = 1$. The time complexity of this algorithm is $O(m)$.

This algorithm can be further improved by an updated greedy strategy that is elaborated in Algorithm 1. Different from the previous one, $weight(v)$ is updated during the execution of the new algorithm. In our algorithm, the id is assigned in decreasing order. If we have assigned i to v, for each neighbor w of v, the maximum neighbors' id of w is no less than i. Let S be the of nodes which not assigned an id yet. Thus for each w's neighbor u and $u \in S$, its weight can be decreased by $1/d(w)$ because w's maximum neighbors' id can not be the id of u. We use priority queue to maintain the value of $weight(v)$ and get the minimum $weight(v)$. Finally, the time complexity is $O(n + m \log n)$.

B. Instance Query

Based on the built index, we can design various query functions. The first, also the simplest query interface is called instance query, the goal of which is to find nodes that are mostly relevant to a given keyword-based query based on the content information.

Formally, given a graph $G = (V, E, W, T)$ and a query $q = \{w_{q1}, w_{q2}, \ldots, w_{qn_q}\} \subset W$, which is a set of keywords, we use a classical information retrieval method[9] to calculate the relevance of each node v to the query q by:

$$s_v(q) = \frac{\sum\limits_{w_i \in q, w_i \in D_v} tf_{D_v}(w_i) tf_q(w_i) idf^2(w_i)}{\sqrt{\sum\limits_{w_i \in D_v} [tf_{D_v}(w_i) idf(w_i)]^2} \sqrt{\sum\limits_{w_i \in q} [tf_q(w_i) idf(w_i)]^2}} \quad (3)$$

where $tf_{D_v}(w_i)$ is the term frequency of word w_i in D_v and $tf_q(w_i)$ is that in the query; $idf(w_i)$, the inverse term frequency of word w_i, is defined as $idf(w_i) = \log \frac{|V|+1}{N(w_i)+1}$, here $N(w_i)$ is the number of nodes whose descriptions contain the word w_i. After building the index, calculation of the relevance scores can be done in a complexity of $O(\sum_{w_i \in q} N(w_i))$, where $N(w_i)$ is what in the definition of $idf(w_i)$.

With the instance query interface, we can develop some basic keyword-based search applications, such as finding text messages containing specific keywords.

C. Neighbor Query

Given a specific node v_s, a set of node types $T' \subset T$ and a length bound l, the goal of neighbor query is to find a set of nodes $V_t = \{v_t \in V | d(v_s, v_t) < l\}$ of type $t \in T'$, where $d(v_s, v_t)$ denotes the shortest paths between v_s and v_t, and l is a length bound (usually set as 6, following the concept of small-world theory). Basically, this interface is designed to find neighbors of certain types for a given node, which is a common requirement in recommendation and navigation scenarios, for example, finding the most relevant photos to a friend (via connections of events or messages).

To deal with this problem, we employ a heap-based Dijkstra algorithm [4] based on the built index. Two differences from the original Dijkstra algorithm lies in (1) we use a heap to store and sort the traversed path and (2) the algorithm terminates when the minimum distance exceeds a bound l. A special case is that, when all weights are 1, we use Breadth-First-Search instead of Dijkstra, because in this case BFS can achieve a lower complexity.

D. Shortest Path Query

The objective is to find the shortest path between two nodes v_s and v_t in a graph $G = (V, E, W, T)$. Finding shortest path is especially useful in revealing how close two nodes are in the graph. In the graph, we assume the weights of all edges are positive. Then we can use a heap-based Dijkstra algorithm to find the shortest path. More accurately, in our implementation, we use a bi-directional Dijkstra to make it faster. We provide an approximate algorithm to deal with really large graphs. The algorithm generates a small candidate subgraph [6] first and then performs Dijkstra search on the subgraph. We carefully grow the neighbors around the two nodes. Initially the candidate subgraph is empty and each time a node is added into it. The process terminates when some terminal condition is satisfied. Suppose the set of nodes expanded from s is V_s and the set of nodes expanded from t is V_t. We set a connectivity threshold θ and the stopping condition is $|V_s \cap V_t| \geq \theta$.

E. Connection Subgraph Query

We propose a novel query, called connection subgraph query for the graph data. It aims to identify a subgraph that connects two input nodes in a graph. In many cases, a single (shortest) path from v_s to v_t only offers limited information. Ideally, it is desirable to identify a subgraph that encodes the

Algorithm 2: Connection Subgraph Query

Input: A data graph $G = (V, E)$, two nodes v_s and v_t, bound k

Output: Subgraph $G' = (V', E')$ with k nodes which approximately maximizes $H(G')$

2.1 $G' \leftarrow \emptyset$;
2.2 **while** $|V'| \leq k$ **do**
2.3 Find the augmenting path P which minimizes the number of new nodes added in G';
2.4 Increment units of flow through P;
2.5 $G' \leftarrow G' \cup P$;
2.6 **end**

most "significant" connection information between the two nodes, with some given constraints.

Formally, we can define the query as follows. The connection subgraph query is to find a connection subgraph $G' \subset G$ containing k nodes including v_s, v_t, which maximizes the value of a goodness function $H(G')$. Two natural measures of the goodness would be the shortest distance and the maximum flow. In our work, we choose the latter and the definition follows the network flow theory. The graph can be regarded as a flow network, where the weight of each edge is formalized as the capacity of the edge. We define the function $H(G')$ as follows,

$$H(G') = \max\{|f|\} = \max\{\sum_{v \in V} f_{v_s v}\} \quad (4)$$

where f_{uv} is the flow on edge (u, v) which satisfies the flow constraint.

Thus, our goal is to find such G' to maximize $H(G')$. The problem is in NPC, which can be proved by making a reduction to the Set Cover problem, a classical NPC problem. We design a greedy algorithm based on EdmondsKarp maximum flow algorithm [5] to achieve an approximate but high quality solution. The framework of our algorithm is shown in Algorithm 2. First let $G' = (V', E')$ be empty and in each iteration, we find the augmenting path which minimizes the number of its nodes not in V'. The idea behind this is that we want $|V'|$ to grow as slowly as possible in order to maximize the times of incrementing the units of flow. The augmenting path can be found as follows: in each iteration of the EdmondsKarp algorithm, it finds the shortest augmenting path on the residual graph $G_f = (V_f, E_f)$ and the length of the path is the number of edges. The weight of each edge can be viewed as 1. In our work, we change the weights of the edges. $\forall (u, v) \in E_f$, $w(u, v) = 0$ if $v \in V'$, otherwise $w(u, v) = 1$. In this way, the length of a path is equivalent to the number of its nodes which are not in V'. Thus the shortest path after changing the weights satisfies the requirement in Algorithm 2.

IV. Application

The graph query interfaces can help with many applications. Here we illustrate two real applications in Nokia China Research Center.

A. Kaleido Photo

Kaleido Photo is a project aiming to help users manage and share their photos taken by a mobile phone. In Kaleido Photo, users can navigate to other objects, e.g. the calendar, the address book and the SMS from a photo based on the relevance of their meta data. To support this, Kaleido Photo uses a richer meta data schema for JPEG photos, and many information like photo owner's phone number and user-generated tags can be associated with a photo. Here we describe three tasks required by Kaleido Photo, and see how our query interfaces can help.

Task 1 Photo clustering: When a user builds an album with title and description information, Instance Query can be used to discover all photos related to the album. For example, the result photos may have a tag that is specified in the album title, or is taken in the same location/time period.

Task 2 Relation searching: If the user is interested in a photo, he might want to know who took this photo and the relationship between the owner and himself. Here Shortest Path Query can be used to find out the path.

Task 3 Photo recommendation: It finds photos related to (e.g., taken by) a user's friends and recommend them to the user. Here we can use Neighbor Query to find the photos whose distance from the friend is less than 3.

B. Nokia Simple Context

Nokia Simple Context (https://simplecontext.com/eb2/) is another project in Nokia Research Center. The project tries to provide a common framework for collecting, storing and sharing mobile context data (e.g., SMS, call log, tags, GPS, GSM, and calendar). We apply the proposed query interfaces to support four interesting search scenarios.

Task 1 The first one is to find related short messages, calendar entries and notes via Instance Query.

Task 2 Given two nodes v_1 and v_2 of type *contact*, find the paths from v_1 to v_2 using Connection Subgraph Query. It can mine the relations between two persons.

Task 3 Given the name of a person, find the easiest way to contact this person. It is particularly useful if this person is not in one's address book but mentioned in a text message from another one, so the message sender who probably knows the person is suggested. This can be done by a combination of Instance Query and Neighbor Query.

Task 4 Find information of all participants of a meeting stored as a calendar entry. This can be done by Neighbor Query.

V. Experimental Results

We conduct various experiments to validate both computational and memory efficiency of MQuery on different datasets. All data sets and codes are publicly available.[1]

[1] http://arnetminer.org/mobilequery/

A. Experimental Setup

Dataset We apply the MQuery toolkit on two real-world datasets: a Kaleido Photo dataset and a Nokia Simple Context dataset, both provided by Nokia Research Center(NRC).

Table I
DETAIL INFORMATION ABOUT THE DATASETS. $\sum_{v \in V} |D_v|$ IS THE NUMBER OF WORDS OF ALL DESCRIPTIONS

Dataset	$\|\mathbf{V}\|$	$\|\mathbf{E}\|$	$\|\mathbf{W}\|$	$\sum_{\mathbf{v} \in \mathbf{V}} \|\mathbf{D_v}\|$
Photo	5,729	5,832	786	23,861
Context	16,354	25,544	3,837	117,223

Table I shows statistics of the two datasets. The Photo dataset consists of 5,672 photos uploaded by 57 users. Each photo is associated with several tags and comments created by the users and each user has a name and an email address. The nodes set V of graph $G = (V, E, W, T)$ includes both the user nodes and the photo nodes. If a photo p is uploaded by user u, then we create an edge between nodes v_p and v_u. Each user can also add the other users in his/her friend list. We represent each friendship as an edge in the graph G. We set weights of all edges as one. Moreover, $\sum_{v \in V} |D_v|$ is the total number of words in all descriptions. From Table I, we can see that on average, each node has about 4 words in its description and each word appears in about 31 descriptions of nodes.

The second dataset, Simple Context, consists of 6 types of data: 5,942 SMS, 4193 call logs, 6 GPSs, 5,000 GSMs, 215 user-generated tags, and 53 calendar entries. Each data is represented as a node in the graph G with a specific type. We use the content of SMS, tag and calendar as the description of the corresponding nodes and the latitudes and longitudes information of GPS and GSM as the description of the location nodes. To build the relationship between the nodes, we build another two types of nodes: 724 phone number nodes and 221 time nodes. Each piece of SMS or call log has a phone number which indicates an edge between the phone number nodes and the SMS nodes. An edge exists between a time node v_t and a data node if the data is generated during the time t. On average, each node has about 7 words as its description and each word appears in about 30 description of nodes.

Evaluation Measures To evaluate the performance of MQuery, we consider the memory cost and the time cost. In MQuery, the memory cost can be defined as,

$$M_{total} = M_{index} + M_{graph} + M_{other} \tag{5}$$

where M_{index} is the memory used to store the inverted index and M_{graph} is the memory used to store the graph, including the weights of edges; M_{other} is the rest part of memory usage including linking the library, some constant data, code data and other things which have no relations with the size of data. In our experiment, we focus on evaluating M_{index}

Table II
THE OVERALL MEMORY COST(KB) ON TWO DATASETS. M_{index} AND M_{graph} IS THE MEMORY USED FOR STORING INVERTED INDEX AND GRAPH. M_{other} IS THE REST PART OF MEMORY USAGE.

	Photo dataset		Context dataset	
	Original	Compressed	Original	Compressed
M_{index}	384	152	1612	604
M_{graph}	208	187	864	542
M_{other}	1,976	2,037	1,772	1,881
M_{total}	2,568	2,376	4,248	3,028

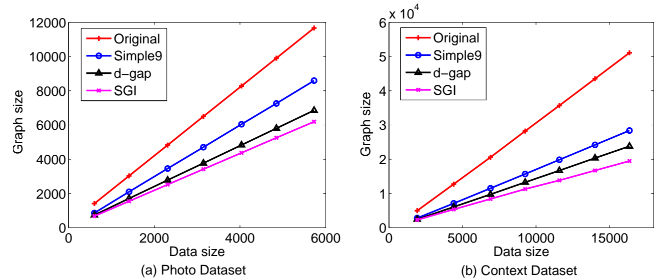

Figure 3. Performance of graph index compression on two datasets. Graph size is the number of 32-bit integers needed to store the graph index, not including weights of edges

and M_{graph} because they are related to the size of data while we also provide the result of M_{total} and M_{other}.

The time cost of MQuery is defined as:

$$T_{total} = T_{initialize} + T_{query} + T_{output} \tag{6}$$

where $T_{initialize}$ is the time cost of loading data into the memory and build the compressed inverted index and graph index. T_{query} is the time cost that MQuery needs to process a query and it is the main part we concern. In the following experiment result, the time cost is T_{query} unless mentioned especially. T_{output} is the time cost that MQuery needs to output the results and retrieve some extra information about the results from the database if necessary.

In all the experiments, the time cost of each query is the average cost by executing the same query for 10,000 times.

Experimental Environment All experiments are performed on a Nokia N900 (600MHz, 256M RAM) smartphone with Maemo linux operating system. The program is written in C/C++ compiled with gcc/g++.

B. Performance of SGI Compression

We first evaluate the compression performance of the proposed method, SGI. Given the photo dataset and the context dataset, we randomly pick out $10\%, 25\%, 40\% \ldots 100\%$ of the data to test the performance. We compare the graph size without (w/o) compression and with compression by three different methods: Simple9, d-gap and our proposed SGI. Graph size is measured by the number of 32-bit integers needed to store the graph index (adjacency lists of the graph), not including weights of edges.

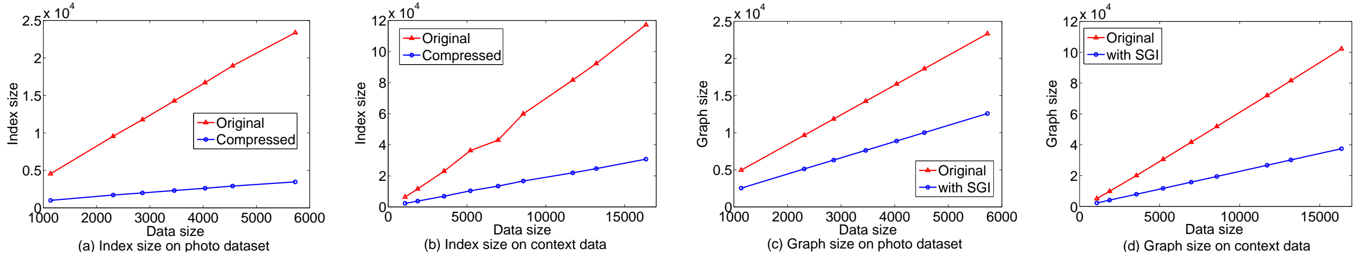

Figure 4. Memory cost of the inverted index and graph storing. Specially, index size is the number of 32-bit integers needed to store the inverted index and graph size is the the number of 32-bit integers needed to store both the graph index and weights on the edges.

Figure 3 shows the original graph size and the graph size after different compression methods. We see that our SGI can achieve a higher compression ratio on the graph size. Here, we define the compression ratio as $CS = \frac{\textit{graph size w/o compression}}{\textit{graph size with compression}}$. Then on average, for the photo dataset, we obtain $CS_{S9} = 1.4218$, $CS_{dgap} = 1.7516$, while $CS_{SGI} = 1.9301$. Clearly, we see that for compressing the graphical data, SGI significantly outperforms (35.75% and 10.19% reduction respecitively) the two baseline methods (S9 and d-gap). For the context dataset, SGI also achieves a better improvement (37.92% and 16.82%) than the two baseline methods (S9 and d-gap).

C. Memory Cost of MQuery

We test the memory cost of the inverted index and the graph storing on the two datasets. To accurately evaluate the performance, we do not simply conduct experiments on the whole datasets. Instead, we randomly pick out some of data from the datasets and test on different data size, as in Section V-B. The memory cost of inverted index is measured by index size and the memory cost of graph storing is measured by the graph size. The index size is measured by the number of 32-bit integers that are needed to record the whole inverted index. The graph size is measured by the number of 32-bit integers needed to store the graph index and the weights of edges.

We use S9 with d-gap to compress the inverted index and use SGI to compress the graph. Figure 4 shows the index size and graph size with and without compression. It shows that our method achieves a significant reduction on the memory cost. Averagely, the compression ratio is about 3. Moreover, we see a trend that the compression ratio grows with the data size, which indicates that we may have a better compression performance when dataset becomes larger. Table II shows the memory cost for two datasets. For the context dataset, the total memory compressed by SGI is decreased by about 30% compared with the original memory cost. Also, since M_{other} accounts for a smaller part for larger dataset, we can save more memory with SGI for large dataset.

D. Time Cost of MQuery

We conduct another experiment to evaluate the time cost of MQuery. First, we test the time cost of compression with SGI on the two datasets. Table III shows the time cost of initialization $T_{initialize}$, including storing graph and building index. From the table, SGI takes about 300 ms more to initialize for photo dataset and about 1000 ms more for context dataset compared with the original one. This extra time cost is mainly for compression, but it needs to initialize one time.

Table III
TIME COST(MS) OF INITIALIZATION, INSTANCE QUERY AND SHORTEST PATH QUERY ON TWO DATASETS. IT'S USED TO EVALUATE THE TIME COST OF COMPRESSION AND DECOMPRESSION

	Photo dataset		Context dataset	
	Original	Compressed	Original	Compressed
Initialization	760	1080	1010	2030
Instance Query	0.053	0.071	0.621	0.640
Shortest Path Query	1.78	2.36	54.2	79.7

Table IV
SUMMARY OF TIME COST(MS) OF MQUERY ON TWO DATASETS

Query Interfaces	Photo dataset	Context dataset
Instance Query	0.473	0.561
Neighbor Query(Dijkstra)	28.84	58.72
Neighbor Query(BFS)	1.15	2.62
Shortest Path Query(Dijkstra)	15.83	18.79
Shortest Path Query(BFS)	1.12	2.72
Connection Subgraph Query	5.90	18.79

Second, we evaluate the time cost of decompression with SGI. We conduct Instance Query to see the time cost of decompressing the inverted index and execute Shortest Path Query to test the time cost of decompressing the graph index. From Table III, we can find that the additional time needed for decompressing both the inverted index and the graph index accounts for a little part of the total time cost. Therefore, our SGI algorithm achieves a good performance on both memory and computational efficiency.

Finally, Table IV shows that each query interface is time efficient. Specially, for Neighbor Query and Shortest Path Query, we test the performance using different implementations, Dijkstra and BFS. In this experiment, we randomly select different queries for the four different query interfaces and calculate their average time cost. This eliminates the influence of the case speciality and we get time costs of average performance. From the table, we see that all query interfaces can perform with a very low time cost, which confirms the necessity and success of this method.

Table V
TIME(MS)/MEMORY(KB) COST OF APPLICATIONS ON TWO DATASETS

	Photo dataset	Context dataset
Task 1	0.085/2,540	0.735/4,108
Task 2	0.971/2,580	2.462/4,120
Task 3	0.337/2,584	2.433/4,024
Task 4	-	7.814/4,188

E. Performance of Application

Table V shows the time and memory cost of the different tasks in the two applications described in Section IV. All tasks can be done in less than 10ms and some of them can even be done in 1ms. The memory cost M_{total} is also acceptable. Therefore, MQuery can support these applications very well in terms of both efficiency and effectiveness.

VI. Related Work

The ubiquitous platform of mobile devices attract considerable interest from the research community. In [18], a demo called MobileMiner, is presented to show how data mining techniques can help in mobile communication data analysis. In [20] and [2], the authors study the characteristics of search queries submitted from mobile devices using various applications on Yahoo!. In [8], the authors present a log-based comparison of search patterns among computers, smart mobile phones and conventional ones. Kamvar et al. [7] provide an overview of the trends in mobile search. In [11], the authors make use of the predefined categories for proper Web image handing and develop an automatic Web image classification method to solve the problem of poor input interfaces on mobile devices.

Mobile users want to access and manipulate information and services specific to certain situation. In order to manage the mobile context, a precise definition of shared interfaces is required. In [17], the authors present an overview of the Mobile Ontology and highlight its advantages by defining such a semantic model. Korpipaa et al. [10] propose a uniform mobile terminal software framework that provides systematic methods for acquiring and processing useful context information from a user's surroundings. Recently, with the pervasiveness of GPS-enabled devices, a few researches also study the spatial mining problem on mobiles. In [21], the authors propose an approach based on supervised learning to infer transportation mode from raw GPS data.

VII. Conclusion and Future Work

In this paper, we study the problem of semantic indexing for mobile context data and present a unified framework called MQuery to index and query the mobile data. We propose an efficient index compression method for graph data and develop four interfaces for querying them. Experimental results on two different real world data sets show that the index compression greatly decreases the memory cost and also clearly outperforms (+10%-37%) existing compression methods. Experiments also demonstrate that MQuery can perform different kinds of queries efficiently.

Acknowledgment

The work is supported by a research award from Nokia China Research Center. Jie Tang is also supported by Natural Science Foundation of China (No. 60703059), Chinese National Key Foundation Research (No. 60933013), National High-tech R&D Program (No. 2009AA01Z138).

References

[1] V. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Inf. Retrieval*, 8(1):151–166, 2005.
[2] R. Baeza-yates, G. Dupret, and J. Velasco. A study of mobile search queries in japan. In *WWW'07*, 2007.
[3] L. Capra, W. Emmerich, and C. Mascolo. Carisma: Context-aware reflective middleware system for mobile applications. *IEEE Trans. Software Eng.*
[4] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269271, 1959.
[5] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*.
[6] C. Faloutsos, K. S. McCurley, and A. Tomkins. Fast discovery of connection subgraphs. In *KDD'04*.
[7] M. Kamvar and S. Baluja. Deciphering trends in mobile search. *IEEE Computer*.
[8] M. Kamvar, M. Kellar, R. Patel, and Y. Xu. Computers and iphones and mobile phones, oh my!: a logs-based comparison of search users on different devices. In *WWW'09*.
[9] W. A. G. Kenneth W. Church. Inverse document frequency (idf): A measure of deviations from poisson. In *Proceedings of the Third Workshop on Very Large Corpora*.
[10] P. Korpipaa, J. Mantyjarvi, J. Kela, H. Keranen, and E.-J. Malm. Managing context information in mobile devices. In *Pervasive Computing*.
[11] T. Maekawa, T. Hara, and S. Nishio. Image classification for mobile web browsing. In *WWW'06*.
[12] L. Miller, A. Seaborne, and A. Reggiori. Three implementations of squishql, a simple rdf query language. In *ISWC'02*, pages 423–435, 2002.
[13] M.-H. Park, J.-H. Hong, and S.-B. Cho. Location-based recommendation system using bayesian user's preference model in mobile devices. In *UIC'07*.
[14] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *SIGIR'02*, pages 222–229, 2002.
[15] F. Silvestri, S. Orlando, and R. Perego. Assigning identifiers to documents to enhance the clustering property of fulltext indexes. In *SIGIR'04*, pages 305–312, 2004.
[16] R. Simon and P. Frohlich. A mobile application framework for the geospatial web. In *WWW'07*.
[17] C. Villalonga, M. Strohbach, N. Snoeck, M. Sutterer, M. Belaunde, E. Kovacs, A. V. Zhdanova, L. W. Goix, and O. Droegehorn. Mobile ontology: Towards a standardized semantic model for the mobile domain. In *ICSOC Workshops*, pages 248–257, 2007.
[18] T. Wang, B. Yang, J. Gao, D. Yang, S. Tang, H. Wu, K. Liu, and J. Pei. Mobileminer: a real world case study of data mining in mobile communication. In *SIGMOD '09*.
[19] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *WWW'09*.
[20] J. Yi, F. Maghoul, and J. O. Pedersen. Deciphering mobile search patterns: a study of yahoo! mobile search queries. In *WWW '08*.
[21] Y. Zheng, L. Liu, L. Wang, and X. Xie. Learning transportation mode from raw gps data for geographic applications on the web. In *WWW'08*.
[22] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6, 2006.
[23] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *ICDE'06*, page 59, 2006.