Partial View Selection for Evolving Social Graphs

Georgia Koloniari
Technology Management Department
University of Macedonia
Thessaloniki, Greece
gkoloniari@uom.gr

Evaggelia Pitoura
Computer Science Department
University of Ioannina
Ioannina, Greece
pitoura@cs.uoi.gr

ABSTRACT

In this paper, we deal with the problem of historical query evaluation over evolving social graphs. Historical queries are queries about the social graph in the past. The straightforward way of executing such a query is by first reconstructing the whole social graph at the given time instance or interval, and then, evaluating the query on the reconstructed graph. Since social graphs are large, the cost of a complete graph snapshot reconstruction would dominate the cost of historical query execution. Given that many queries are user-centric, i.e., node-centric queries that require access only of a targeted subgraph, we propose deploying partial view instead of full snapshot construction and define conditions that determine when a partial view can be used to evaluate a query. We also propose using a cache of partial views to further reduce the query evaluation cost, and show how partial views can be extended to new views with reduced cost. Finally, we present a greedy solution for the static view selection problem and study its performance experimentally.

1. INTRODUCTION

The study of graph structures depicting real-world networks such as social networks, citation and hyperlink networks as well as biology, traffic and computer networks has received much attention recently. Our focus is particularly on social graphs, characterized by large scale and dynamic behavior, as the corresponding social networks include millions of users and change through time constantly.

Recently proposed approaches [6, 7, 11] for query evaluation in this setting mainly deal with the problems induced by the large scale of such graphs but ignore the temporal aspect. Our goal is to deal with both aspects of social graphs by supporting the evaluation of *historical queries*. That is, queries that require information about the state of the graph in the past, such as queries about the popularity (e.g., number of friends) of a user at some specific time in the past or about how this popularity changed over time.

One way to support information about the evolving social graph is by maintaining a log file recording update operations through time similarly to [16]. Any past instance of the graph can be retrieved by combining parts of the log file and the current graph. Based on this idea, the few approaches [15, 8, 9] that deal with the temporal aspect of graph structures support some form of two-phase query evaluation that involves first, retrieving the past graph snapshots required by the query and then, the actual evaluation of the query on them. In such approaches, snapshot construction induces a considerable additional cost to query evaluation.

Our goal is to reduce the cost of historical query evaluation by avoiding snapshot construction when possible. To this end, we discern between two types of queries: *global* and *targeted* [6]. Global queries measure global properties of a graph and require traversing the entire graph, while targeted queries only address specific portions (subgraphs) of the social graph. Examples in the first category include computing the diameter of the graph, degree distribution, etc, while examples in the second include measures centered around a single node v, such as its neighborhood, its K-step neighbors, induced subgraphs and other. Targeted queries play an important role in micro-level analysis of social networks [13], where measurements of the nodes (users) are evaluated such as degree centrality and local clustering coefficient. In addition, targeted queries are used as the building blocks for global query analysis, for instance, K-egonets can be used to find communities in the global graph [6].

Full graph snapshot construction for targeted queries leads to redundant computations as the queries do not access the entire graph that is constructed. Thus, instead of full snapshot construction, we propose using *partial view* construction, i.e., constructing only the subgraph targeted by a given query. In particular, we use the K-egonet of a node v as our basic unit for graph construction and query evaluation, i.e., queries are also represented as egonets. We then define a subsumption relationship between partial views, based on which we can determine when a query can be evaluated on a particular view.

We describe algorithms for constructing a partial view based on the current graph and the parts of the log file associated with the nodes included in the partial view, and also algorithms for partial view extension, so as to derive new views from existing ones with reduced cost.

Finally, we propose maintaining a cache of such partial views that used by subsequent queries can improve performance. To determine which views to cache, we study a problem of static view selection. That is, given a query workload, select the set of views to cache so as to minimize its total evaluation cost. We describe a greedy solution to the problem that is based on grouping views based on their center, and report some preliminary experimental results.

To summarize our contributions:

- We introduce partial view instead of full snapshot construction for targeted queries.
- We define subsumption conditions that determine when a partial view can be used for the evaluation of a query.
- We propose using a cache of partial views and describe a greedy solution to the static view selection problem.

Proceedings of the First International Workshop on Graph Data Management Experience and Systems (GRADES 2013), June 23, 2013, New York, NY, USA.

Table 1: Update Operations

Operation	Description
$addNode(v_i)$	adds a new node v_i in V
$addEdge(v_i, v_j)$	adds a new edge (v_i, v_j) in E
$remNode(v_i)$	deletes v_i from V
$remEdge(v_i, v_j)$	deletes edge (v_i, v_j) from E

- Finally, we present algorithms for partial view construction and view extension.

The rest of the paper is structured as follows: Section 2 describes our graph and storage model. In Section 3, we define partial views and the view selection problem. Section 4 presents a greedy solution for view selection along with algorithms for view construction and extension. Section 5 includes experimental results and Section 6 related work. We conclude in Section 7.

2. PRELIMINARIES

We model a social network as an undirected graph, $G = (V, E)$. Each graph node $v_i \in V$ corresponds to a user u_i of the social network. Edges $(v_i, v_j) \in E$ capture social relationships (i.e., friendship) between users u_i and u_j that correspond to nodes v_i and $v_j \in V$ respectively.

Our model for capturing the evolution of the social network through time is based on the copy+log approach described in [16]. Its basic components are *graph snapshots* and a *log file* maintaining the updates on the graph.

2.1 Capturing Graph Evolution

We consider an object, node or edge, of a graph G as *valid* for the time periods for which the corresponding item (user or friendship) of the social network it represents is also valid. Each node $v_i \in V$ is valid for the time periods for which the corresponding user u_i participates in the social network represented by the graph. Similarly, each edge $(v_i, v_j) \in E$ is valid for the time periods that the corresponding users u_i and u_j are friends in the network.

DEFINITION 1 (GRAPH SNAPSHOT). *The graph snapshot of a graph G, at a time point t, is defined as the graph $SG_t = (V_t, E_t)$, where $V_t \subseteq V$ and $E_t \subseteq E$, such that $v_i \in V_t$, iff, v_i is valid at time point t and $(v_i, v_j) \in E_t$, iff, (v_i, v_j) is valid at time point t.*

Graph G captures the social network as it evolves. Any update in the social network is directly reflected on G. A graph snapshot SG_t of G can be simply viewed as an instance of G frozen at time point t, capturing the state of G at this specific time point.

We focus on the structure of the social network and consider changes such as addition/deletion of a user u_i or of a friendship relationship (edge) between two nodes in the social network. Table 1 summarizes the corresponding update operations on the graph.

Given two graph snapshots SG_{t_k} and SG_{t_l} of a graph G, we maintain in the log file the operations that, if applied to SG_{t_k}, produce SG_{t_l}.

DEFINITION 2 (GRAPH LOG). *The graph log $L_{[t_k,t_l]}$ of a graph G for a time interval $[t_k, t_l]$ is defined as a set of pairs, (op, t), such that a pair $(op, t) \in L_{[t_k,t_l]}$, iff operation op appeared in G at time point t, $t_k \leq t \leq t_l$.*

To enable us to construct snapshots of the graph in the past, the graph log needs to satisfy two requirements: it must be *complete* and *invertible*.

PROPERTY 1 (COMPLETENESS). *A graph log $L_{[t_k,t_l]}$ of a graph G is complete, if given the graph snapshot SG_{t_k}, we can derive any snapshot $SG_{t'}$, $t_k \leq t' \leq t_l$, by applying the operations ops in $L_{[t_k,t_l]}$ for which $t < t'$.*

That is, a complete log file maintains all the information necessary to construct snapshots at any time point during its time interval as long as it has access to the original graph snapshot at time t_k.

Thus, given a complete log file and the initial graph snapshot we can construct any snapshot later in time. Besides this 'forward' direction of moving through time, we are also interested in the opposite. That is, given the current snapshot to retrieve a snapshot older in time, i.e., to move 'backwards' in time. This is accomplished by invertible log files.

PROPERTY 2 (INVERTIBLE LOG). *A graph log $L_{[t_k,t_l]}$ of a graph G is invertible, if by applying on SG_{t_l} the reverse $\bar{op}$ of the operations op in $L_{[t_k,t_l]}$, for which $t > t'$, we can derive any snapshot $SG_{t'}$, $t_k \leq t' \leq t_l$.*

The log with the reverse operations is called, inverted log and denoted as $\bar{L}_{[t_k,t_l]}$. The reverse of the update operators are derived as follows: $\overline{addNode(v_i)} = remNode(v_i)$, $\overline{remNode(v_i)} = addNode(v_i)$ and so on. All operations can be inverted as long as the necessary information is maintained in the log file. In particular, to maintain a complete log that is also *invertible*, we make the following assumption. Before recording any $remNode(v_i)$, we record first $remEdge(v_i, v_j)$ operations, for each edge of v_i, annotated with the same time point as the $remNode(v_i)$ operation.

2.2 Storage Model

Given that the log file we maintain is both complete and invertible, we deduce a very space efficient storage model that requires maintaining only one graph snapshot besides the log file to permit constructing graph snapshots at any time point in the recorded time interval. In particular:

THEOREM 1. *Given the graph log $L_{[t_k,t_l]}$ of graph G that is complete and invertible, it suffices to maintain only one graph snapshot $SG_{t'}$, $t_k \leq t' \leq t_l$ to construct graph snapshots at any other time point t'', $t_k \leq t'' \leq t_l$.*

Proof. If $t'' > t'$, we construct $SG_{t''}$ by applying the ops in $L_{[t',t'']} \subseteq L_{[t_k,t_l]}$ on $SG_{t'}$. If $t'' < t'$, we construct $SG_{t''}$ by applying the ops in $\bar{L}_{[t'',t']} \subseteq \bar{L}_{[t_k,t_l]}$ on $SG_{t'}$. □

Let t_0 be the time point at which we start maintaing the log file, and t_{cur} be the current time point. Based on Theorem 1, we choose to maintain besides the log file, $L_{[t_0,t_{cur}]}$, the current graph snapshot $SG_{t_{cur}}$, as we expect queries about the recent past to be more popular.

As updates occur in G, we need to periodically update both the current snapshot and the log file. To accomplish this, we use a temporary log that records the updates on G until the next time unit. We then apply this log on the current snapshot to derive the next current snapshot. The temporary log is appended to L and the algorithm is applied anew with a new temporary log for the next time period.

3. PARTIAL VIEWS

Historical queries are queries that refer to a state of the graph in the past. The straightforward way to evaluate them proceeds in two steps. In the first step, the snapshot (or snapshots) required to evaluate the query are constructed, and in the second, the actual query evaluation takes place on them. We discern between *global* and *targeted* queries [6]. While global queries require access to the entire graph, targeted queries need to access only a specific subgraph, usually centered around one node (or a few) as such queries tend to be egocentric.

Thus, in the first step of query evaluation for targeted queries, parts of the graph that are not required are still constructed increasing the total cost of query evaluation with

redundant operations. We propose *partial view construction* instead of full snapshot construction for targeted queries, i.e., to construct only the subgraph targeted by each query and thus, reduce the total query evaluation cost.

3.1 Partial Views and View Subsumption

Before defining a partial view, let us start with some important definitions. Consider two nodes v_k and $v_l \in V$. The distance $d_t(v_k, v_l)$ between two nodes $v_k, v_l \in V$ is defined as the number of edges in the shortest path connecting the two nodes at SG_t. We denote as $N(v_k, R, t)$ the set of neighbors of node v_k at distance at most R, $R \geq 0$, in SG_t, i.e., $v_l \in N(v_k, R, t)$ iff $d_t(v_k, v_l) \leq R$.

As targeted queries tend to be centered around a specific node, such as node's v degree at time point t, we are interested in defining subgraphs centered around some node. To this end, we use egonets as the main unit for both graph construction and query representation and evaluation.

DEFINITION 3 (EGONET). *The egonet $EG(v, R, t)$ of a node v with radius R, $R \geq 0$, at time point t is the induced subgraph $EG(v, R, t) = \{V', E'\}$ of graph snapshot SG_t, such that $v' \in V'$ iff $d_t(v, v') \leq R$ and $(v'_k, v'_l) \in E'$ iff $v'_k, v'_l \in V'$ and $\exists(v'_k, v'_l)$ in SG_t.*

Using the egonet as the graph construction unit, we have:

DEFINITION 4 (PARTIAL VIEW). *A partial view PV of a graph snapshot SG_t is defined as an egonet $EG(v, R, t)$ that is determined by a center node $v \in SG_t$, a radius R and the time point t at which it is valid.*

In the rest of the paper, we use the two terms, egonet and partial view, interchangeably.

Similarly to partial views, queries are also represented as egonets.

DEFINITION 5 (QUERY EGONET). *The query egonet EG^q of a targeted query q is defined as the minimum partial view of SG_t such that the result of the evaluation of q on EG^q is equal to the result of evaluating q on the full snapshot, SG_t.*

For instance, for a query requiring the degree of a node v_q at time point t_q, it suffices to reconstruct the partial view with radius 1 around v_q at time point t_q. In the rest of the paper, we assume that targeted queries are represented by their corresponding query egonets. How these egonets are derived from a targeted query, in the general case, is out of the scope of this paper.

Given a set of partial views, we need to determine when a partial view $EG(v, R, t)$ can be used to evaluate q corresponding to some EG^q. To this end, we define a *subsumption relationship* between partial views, i.e., subsumption between two egonets.

DEFINITION 6 (VIEW SUBSUMPTION). *Given two views, $EG_1(v_1, R_1, t_1)$ and $EG_2(v_2, R_2, t_2)$, EG_1 subsumes EG_2, $EG_1 \succ EG_2$, if the result of the evaluation of any targeted query q on EG_2 is equal to the result of evaluating q on EG_1.*

For simplicity, we assume that there is at least one node $v' \in SG_t$ with $d_t(v, v') = R$ to define $EG(v, R, t)$. Otherwise, we reduce the egonet to one with a smaller radius. Note also that view subsumption is only meaningful for the same point in time, i.e., $t_1 = t_2$. The following theorem determines when one view is subsumed by another.

THEOREM 2. *Given two partial views $EG_1(v_1, R_1, t_1)$ and $EG_2(v_2, R_2, t_2)$ of a graph G, EG_1 subsumes EG_2, iff $t_1 = t_2$ and ($v_1 = v_2$ and $R_1 \geq R_2$) or ($v_1 \neq v_2$ and $d(v_1, v_2) + R_2 \leq R_1$).*

Proof. Let q be a targeted query and $EG^q = EG_2(v_2, R_2, t_2)$. For EG_1 to subsume EG_2, q needs to yield the same result when evaluated on both egonets. It suffices for EG_2 to be a subgraph of EG_1. For EG_2 to be a subgraph of EG_1 both need to be partial views of the same graph snapshot SG_t, i.e., $t_1 = t_2 = t$. If $v_1 = v_2$ and $R_1 \geq R_2$ then EG_2 is a subgraph of EG_1. If $v_1 \neq v_2$ and $d(v_1, v_2) + R_2 \leq R_1$, even though the egonets have different centers all the nodes in $N(v_2, R_2, t_2)$ are within R_1 radius of v_1 since $d(v_1, v_2) + R_2 \leq R_1$, thus, EG_2 is contained in EG_1. Therefore, $EG_1 \succ EG_2$.

If EG_1 subsumes EG_2, then EG_2 is a subgraph of EG_1 and both are subgraphs of the same graph snapshot SG_t, i.e. $t = t_1 = t_2$. We may discern two cases, either $v_1 = v_2$ or $v_1 \neq v_2$. Let us first assume $v_1 = v_2$ and $R_2 > R_1$. Then $\exists v'' \in EG_2$ and $v'' \notin EG_1$, then EG_2 is not a subgraph of EG_1. Thus, if it holds $v_1 = v_2$, it also holds $R_2 \leq R_1$. Let us now assume $v_1 \neq v_2$ and $d(v_1, v_2) + R_2 > R_1$. Similarly, we determine that EG_2 is not a subgraph of EG_1. Thus, $d(v_1, v_2) + R_2 > R_1$. □

3.2 View Selection

The use of partial views instead of full graph snapshots reduces the overall query evaluation cost as partial views have lower construction cost than full snapshots. To further reduce the cost of the first step of query evaluation, that is of partial or full snapshot construction, one can choose to materialize (cache) a set of such snapshots or partial views to be used by subsequent queries. While full snapshots require a considerable space overhead for their materialization, and thus such a solution is not practical, partial views require significantly less space as we consider most targeted queries to require egonets of a rather small radius. Therefore, we propose maintaining a cache of partial views so as to further facilitate query evaluation.

Let $Q = \{q_1, q_2, \ldots, q_n\}$ be a set of targeted queries and $EG^{q_1}, EG^{q_2}, \ldots, EG^{q_n}$ be their corresponding query egonets. Our goal is to determine which egonets to cache to achieve the greatest possible reduction in the evaluation cost of Q. Let $cost(q_i)$ be the evaluation cost of query q_i. The cost consists of two parts: the construction cost for EG^{q_i} and the cost of processing q_i on EG^{q_i}. Then, the total evaluation cost of the set Q is defined as: $cost(Q) = \sum_{q_i \in Q} cost(q_i)$.

DEFINITION 7 (STATIC VIEW SELECTION PROBLEM). *Given the current graph snapshot SG_{cur}, log file $L_{[t_0, t_{cur}]}$ and a set of targeted queries $Q = \{q_1, q_2, \ldots, q_n\}$, select from the set of corresponding query egonets $EG^Q = \{EG^{q_1}, EG^{q_2}, \ldots, EG^{q_n}\}$, a set $C \subseteq EG^Q$ of K egonets, $0 < K < n$, such that, given that C is materialized, the total evaluation cost of Q is minimized.*

The reduction in $cost(Q)$ is achieved by reducing the construction cost of the egonets. Therefore, the basic idea is to select for caching the egonets that can be exploited by the largest number of queries. To this end, in addition to the partial view construction algorithms, we also define view extension algorithms that derive new views from existing ones with cost lower than the cost required for constructing them from scratch. These algorithms enable the cached views to be used for reducing the cost for more queries, and thus further reduce the total evaluation cost of Q.

4. SOLVING THE PROBLEM

Next, we describe the components necessary to solve the static view selection problem, along with the view construction and extension methods.

4.1 View Construction

Given the current graph snapshot SG_{cur} and the log file $L_{[t_0, t_{cur}]}$, to construct a full snapshot SG_t, $t_0 \leq t \leq t_{cur}$, based on Theor. 1, we apply $\bar{L}_{[t, t_{cur}]} \subseteq \bar{L}_{[t_0, t_{cur}]}$ on SG_{cur}.

To construct a partial view $EG(v, R, t)$, which is a subgraph of SG_t, we do not need to apply all the updates in the log, but only the updates that concern nodes that appear in the partial view.

Algorithm 1 *Hop1Neighbors*

Input: $v, SG_{cur}, \pi_v L_{[t,t_{cur}]}$
Output: $N(v,1,t)$
1: Retrieve $N(v,1,t_{cur})$ from SG_{cur}
2: Apply $\pi_v \bar{L}_{[t,t_{cur}]}$ on $N(v,1,t_{cur})$ to get $N(v,1,t)$
3: **return** $N(v,1,t)$

Algorithm 2 *EgoNet1*

Input: $v, SG_{cur}, L_{[t,t_{cur}]}$
Output: $EG(v,1,t)$
1: $N(v,1,t)$:=$Hop1Neighbors(v, SG_{cur}, \pi_v L_{[t,t_{cur}]})$
2: Initialize $EG(v,1,t)$ to $N(v,1,t)$
3: **for all** $v_k, v_l \in N(v,1,t)$ **do**
4: Retrieve $\pi_{v_k} L_{[t,t_{cur}]}$ or $\pi_{v_l} L_{[t,t_{cur}]}$
5: **if** $(v_k, v_l) \in SG_t$ **then**
6: Add the edge (v_k, v_l) to $EG(v,1,t)$
7: **end if**
8: **end for**
9: **return** $EG(v,1,t)$

Algorithm 3 *EgoNetR*

Input: $v, G_{cur}, L_{[t,t_{cur}]}$
Output: $EG(v,R,t)$
1: $EG(v,1,t) := EgoNet1(v, SG_{cur}, L_{[t,t_{cur}]})$
2: $EG(v,R,t) := EG(v,1,t)$, $Added := \emptyset$, $i := 1$
3: **while** $i < R$ **do**
4: **for** $v_k \in EG(v,R,t)$: $d(v_k,v) = i$ and $v_k \notin Added$ **do**
5: $EG(v_k,1,t) := EgoNet1(v_k, G_{cur}, L_{[t,t_{cur}]})$
6: Add all objects in $EG(v_k,1,t)$ to $EG(v,R,t)$
7: Add v_k to $Added$
8: **end for**
9: $i++$
10: **end while**
11: **return** $EG(v,R,t)$

Definition 8 (Log Projection on node v). *The log projection on node v, $\pi_v L_{[t_k,t_l]}$, is the set of (op,t) pairs $((op,t) \in L_{[t_k,t_l]})$ such that: $(op,t) \in \pi_v L_{[t_k,t_l]}$, iff $(op = addNode(v)$ or $op = remNode(v))$ or $((op = addEdge(v_i,v_j)$ or $op = remEdge(v_i,v_j))$ and $(v_i = v$ or $v_j = v))$.*

Given $SG_{t_{cur}}$ and $L_{[t_0,t_{cur}]}$, Alg. 3 presents the iterative procedure that constructs $EG(v,R,t)$. Starting from the center v of the egonet, which we may consider as an egonet $EG(v,0,t)$, at each iteration i $(1 \le i < R)$, the egonet's radius (i) is extended by 1 by applying Alg. 2 on all nodes at distance i from v. Egonets of increasing radius are progressively produced until the target $EG(v,R,t)$ is derived.

Given node v, SG_{cur}, and the log projection on v, $\pi_v L_{[t,t_{cur}]}$, Alg. 2 constructs $EG(v,1,t)$. The algorithm uses Alg. 1 which constructs $N(v,1,t)$ and then, establishes the edges between the nodes in $N(v,1,t)$ to produce $EG(v,1,t)$.

We will prove the correctness of Alg. 3 using the following two lemmas.

Lemma 1. *An object o, node or edge, $\in SG_t$, (if $o \in G_{cur}$ and $\nexists(add(o) \in L_{(t,t_{cur}]})$) or $(\exists rem(o) \in L_{[t,t_{cur})})$.*

Proof. If $o \in G_{cur}$, o is deleted from SG_t, iff $\exists add(o) \in L$. Thus, if $\nexists(add(o) \in L)$, $o \in SG_t$. If $o \notin G_{cur}$ and $o \in G_t$, o must have been deleted after t, thus, $\exists rem(o) \in L$. □

Lemma 2. *Given $EG(v,i,t)$ it suffices to apply algorithm $EgoNet1(v_k, G_{cur}, L_{[t,t_{cur}]})$ $\forall v_k$, $v_k \in N(v,i,t)$ and $d(v,v_k) = i$, to construct $EG(v,i+1,t)$.*

Proof. Let $U = \bigcup_{v_k \in N(v,i,t) \text{and } d(v,v_k)=i} EG(v_k,1,t)$. Then, $EG(v,i+1,t) = EG(v,i,t) \bigcup U$, as U defines the subgraph formed by the nodes at distance $i+1$ of v and if we apply its union with the egonet with radius i, we get the egonet with radius $i+1$. □

Algorithm 4 *EgoNetT*

Input: $EG1(v,R,t), L_{[t_0,t_{cur}]}, t', G_{cur}$
Output: $EG2(v,R,t')$
1: $EG2 := \emptyset$, $Added := \emptyset$, $N(v,R,t') := \emptyset$, $i := 0$
2: **while** $i <= R$ **do**
3: **for all** $v_k \in EG1$: $d(v_k,v) = i$ and $v_k \notin Added$ **do**
4: Retrieve $N(v_k,1,t)$ from $EG1$
5: **if** $t' > t$ **then**
6: Apply $\pi_v L_{[t,t']}$ on $N(v,1,t)$ to get $N(v,1,t')$
7: **else**
8: Apply $\pi_v \bar{L}_{[t',t]}$ on $N(v,1,t)$ to get $N(v,1,t')$
9: **end if**
10: Add v_k to $Added$
11: $N(v,R,t') := N(v,R,t') \bigcup N(v,1,t')$
12: **end for**
13: $i++$
14: **end while**
15: **for all** $v_k, v_l \in N(v,R,t')$ **do**
16: **if** $(v_k, v_l) \in SG'_t$ **then**
17: Add edge (v_k, v_l) to $EG2$
18: **end if**
19: **end for**
20: **return** $EG2$

We are now ready to prove the correctness of Alg. 3.

Theorem 3. *Algorithm 3 is correct.*

Proof. It suffices to prove that at each iteration i, applying Alg. 2, in lines 4-8 of Alg. 3, correctly derives $EG(v,i,t)$ from $EG(v,i-1,t)$, which is true according to Lem. 2, if Alg. 2 is correct. Algorithm 2 correctly constructs $EG(v,1,t)$ by adding to $N(v,1,t)$ only the edges that belong to $EG(v,1,t)$ and none other based on Lem. 1, if Alg. 1 is correct. Finally, based on Lem. 1, by checking $\pi_v L_{[t,t_{cur}]}$, Alg. 1 correctly constructs $N(v,1,t)$. □

View Extension. Given a partial view $EG(v,R,t)$ an interesting issue is how the view can be used to derive a new view $EG'(v',R',t')$ more efficiently than constructing it using Alg. 3. We consider extension between views with the same center, and discern the following four cases.
Case I ($v = v'$ and $t = t'$ and $R > R'$): Based on Theor. 2, $EG \succ EG'$. Thus, we do not need to construct EG', as any query for EG' can be evaluated on EG.
Case II ($v = v'$ and $t = t'$ and $R < R'$): Based on Lem. 2, to construct EG', we progressively extend $EG(v,R,t)$ to $EG(v,R+1,t)$, ..., $EG(v,R',t)$ by applying Alg. 2.
Case III ($v = v'$ and $t \neq t'$ and $R \ge R'$): Based on Theor. 2, we do not need to alter R, but only t to t'. According to Theor. 1, if $t > t'$, we need to apply backward construction with $\bar{L}_{[t',t]}$, while if $t < t'$, forward construction with $L_{[t',t]}$ is deployed. Algorithm 4 presents the procedure.

To prove the correctness of Alg. 4, we extend Lem. 1 based on Theor. 1 to check for an object in $L_{[t,t']} \subseteq L_{[t,t_{cur}]}$. The proof is similar to Lem. 1.
Case IV ($v = v'$ and $t \neq t'$ and $R < R'$): We need to extend both to a new radius and a new time. We apply Alg. 4 to move to time t', and then, extend its radius similar to Case II. The steps can be also applied in reverse order.

Cost Evaluation. Next, we provide an estimation for the evaluation cost of a historical query. This is split into: the construction cost of the snapshot or partial view on which q is evaluated ($cost_{con}$), and the cost for processing q ($cost_{pr}$).

$$cost(q) = \begin{cases} cost_{con}(SG_{t_q}) + cost_{pr}(q), & \text{if } q \text{ global} \\ cost_{con}(EG^q) + cost_{pr}(q), & \text{if } q \text{ targeted} \end{cases} \quad (1)$$

We ignore the processing cost in the measure as it depends on the type of query. The construction cost is proportional to the number of updates applied to construct the snapshot or view. We assume that all updates have the same cost. Let $S(L_{[t,t']})$ be the size, i.e., the number of updates in $L_{[t,t']}$, $t < t'$. It holds that $S(L_{[t,t']}) = S(\bar{L}[t,t'])$.

Given G_{cur}, $L_{[t_0,t_{cur}]}$ and t, $t_0 \leq t \leq t_{cur}$, for global queries, to construct a snapshot SG_t, we apply $\bar{L}_{[t,t_{cur}]} \subseteq \bar{L}_[t_0,t_{cur}]$ on G_{cur}. Thus, $cost_{con}(SG_{t_q}) = S(L_{[t,t_{cur}]})$.

For targeted queries, for an egonet EG^q, the updates in the log file projections on all the nodes that belong to the view at any time between $[t_q, t_{cur}]$ are applied.

$$cost_{con}(EG^q) = \sum_{t_i=t_q}^{t_{cur}} \sum_{v_j \in N(v_q,R_q,t_i)} S(\pi_{v_j} L_{[t,t_{cur}]}) \qquad (2)$$

For view extension, given $EG(v,R,t)$ and t' (without loss of generality let $t < t'$) to derive $EG(v,R,t')$, the term $S(\pi_{v_j} L_{[t,t_{cur}]})$ in Eq. 2 is replaced by $S(\pi_{v_j} L_{[t,t']})$, where $S(\pi_{v_j} L_{[t,t_{cur}]}) \geq S(\pi_{v_j} L_{[t,t']})$. Thus, compared to view construction, the cost is reduced.

4.2 Greedy Selection

Given the current graph snapshot SG_{cur}, log file $L_{[t_0,t_{cur}]}$, set of queries $Q = \{q_1, q_2, \ldots, q_n\}$ and corresponding query egonets $EG^Q = \{EG^{q_1}, EG^{q_2}, \ldots, EG^{q_n}\}$, the goal of static view selection is to determine a set $C \subseteq EG^Q$ of size K, such that the total cost of Q, $cost(Q)$, is minimized. Given that its query egonet is cached, the processing cost of each query q ($cost_{pr}(q)$) is fixed. Therefore, the goal is to reduce the construction cost of Q ($cost_{con}(Q)$).

The optimal solution is derived by an exhaustive algorithm that considers all possible $C \subseteq EG^Q$ of size K and computes the construction cost of Q for each such set. The set C yielding the smallest construction cost is selected.

As the exhaustive approach is exponential to the size of Q, we proceed with a baseline greedy algorithm. The greedy algorithm iteratively selects at each step to add to C the egonet with the greatest construction cost among all available egonets. While more efficient, the baseline greedy algorithm ignores that by materializing one partial view, the construction of other views may also be influenced as view extension with reduced cost is possible. Such view extension is possible only between views that have the same center.

Based on this assumption, we proceed on extending the baseline greedy approach to a *two-phase greedy* algorithm. The algorithm first groups the egonets according to their center. At each iteration, in each group, the egonet with the greatest construction cost is selected. Then, given the selected egonet, the total construction cost of the group is re-evaluated and the benefit from materializing the egonet computed. After all groups have computed this benefit, the group with the greatest benefit is selected, all costs are updated and the egonet selected from the group is added to C and removed from the group. The algorithm proceeds in the next iteration until K egonets are selected (Alg. 5).

Algorithm 5 Two-Phase Greedy Selection

Input: $G_{cur}, L_{[t_0,t_{cur}]}, Q = \{q_1, q_2, \ldots, q_n\}, EG^Q = \{EG^{q_1}, EG^{q_2}, \ldots, EG^{q_n}\}, K$
Output: C
1: $C := \emptyset$
2: Estimate the total construction cost for Q
3: Group the egonets in EG^Q according to their center
4: **for** l=0 to K **do**
5: **for all** Groups GR_j of egonets **do**
6: Select the EG^{q_i} with $max(cost_{con}(EG^{q_i}))$
7: Given EG^{q_i}, estimate the cost for GR_j
8: **end for**
9: Select from all EG^{q_i} of all GR_j, the one that achieves the greatest cost reduction, $EG^{q_w} \in GR_w$
10: Update $cost_{con}(EG^{q_i})$ for all egonets in GR_w
11: Remove EG^{q_w} from EG^Q and added it to C
12: **end for**
13: **return** C

5. EXPERIMENTAL EVALUATION

In our experiments, we first study the properties of egonets, and then, evaluate the greedy selection approach. We use a real data set from [18]. The data set describes the structure, user-to-user links, from the Facebook New Orleans network. To derive our log, we parsed the data and maintained only the links that were accompanied by a timestamp indicating the time of their establishment. The current graph was produced by creating all edges in the log.

Egonets Properties. We consider 10000 nodes that form more than 100000 edges. We measure the size of the egonets of radius $R = 1$ and $R = 2$ for all nodes. In Fig.1(left) the y-axis is the number of nodes that have an egonet up to size x as indicated in the x-axis. For most nodes the size of egonet with $R = 1$ is between 1 to 30, while there is a small number of nodes with egonet of size > 100. For $R = 2$, there is a number of nodes (≈ 500) with egonets that include almost all nodes in the graph. This is a well known property of social networks, where a small number of nodes are very well-connected, while most have much smaller degrees [1]. Thus, for such well-connected nodes with very large egonets, we will consider special policies in future work.

Figure 1(center) shows how the updates are distributed over time. The dataset used contains only *add edge* operations and we modified it to also include an *add node* when a node is encountered for the first time. Such append-only behavior is usual in social networks, in which density increases over time following a power-low distribution, while their diameter shrinks [10]. We observe that at first there is a small peak at the number of *add node* updates, but as time progresses their rate stabilizes, while the add edge update remain the dominant type of updates.

View Selection comparison. We evaluate the *two-phase* greedy selection algorithm by measuring the construction cost for a given query workload. We compare our approach to a baseline *greedy* and a *random* approach. We also include the cost when *no-cache* is built. We use 5000 nodes, and a workload of 100 queries. The cache size is 10, and for the queries, $R = 1$ and t and v are uniformly dispersed. We vary the overlap among the queries from 0 where no groups are formed, to 70, where 70% of the queries have the same center with at least one other query (no identical queries are allowed). Figure 1(right) shows that the two-phase approach outperforms all others when there is overlap among the queries. For no overlap, the baseline greedy approach achieves the same performance with the two-phase one, since the fact that it ignores the dependencies in the cost between views of the same group, is in this case irrelevant. As the overlap among the queries increases, all costs are reduced as even with random selection some useful views are materialized. Still, the two-phase approach has the greatest benefits. For 70% overlap, two-phase incurs almost half the cost of the greedy approach, while random is almost three times worse.

6. RELATED WORK

There is a large body of work on temporal data management including relational databases (see, for example [14] and [16] for excellent surveys on the topic), RDF (e.g., [5]) and XML documents (e.g., [12], [2]). In [12], a sequence

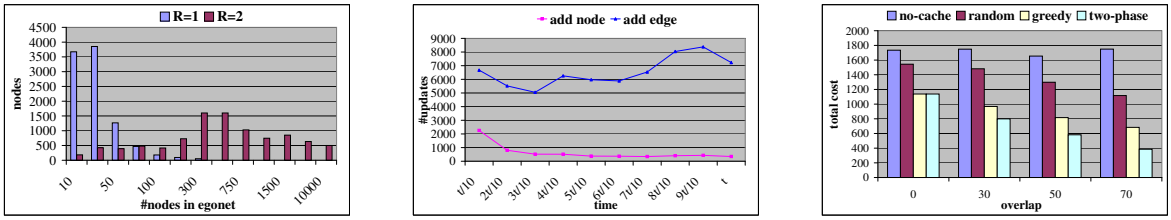

Figure 1: (left) Size of egonets, (center) updates over time and (right) view selection.

of versions of XML documents is collected from the web. The difference between two consecutive versions is represented by complete deltas based on persistent identifiers of the XML nodes, while only the current version of the document is maintained. To avoid the overhead of applying deltas to retrieve old versions, in [2], all versions of XML data is merged into one hierarchy where an element appearing in multiple versions is stored only once along with a timestamp. Temporal reasoning is incorporated into RDF in [5], yielding temporal RDF graphs. Its semantics include temporal entailment and a syntax for adding temporal labels into standard RDF graphs. In our work, we adopt invertible log files to track changes through time ([16] [12]) and adapt them appropriately for large-scale graphs.

View selection and semantic caching have also been widely studied ([4],[17]) for relational databases. Views for graphs are studied in [3], in the context of query processing under schema mappings. The authors consider query processing under GLAV mappings, which map queries over a source schema to queries over the target schema, and extend their work to graphs using regular path expressions as their query model. Our work is different as we consider the temporal aspect of graphs and a different query model.

To deal with the large-scale of graphs, general graph management systems [6, 11] that work in parallel and distributed settings have been proposed. GBASE [7] uses a common underlying primitive of several graph mining operations, which is shown to be a generalized form of matrix-vector multiplication, while Pregel [11] uses a sequence of supersteps applied in parallel, by each node executing the same user-defined function, and separated by global synchronization points. We plan to explore how to deploy our approach on top of such systems to construct partial views in parallel.

The problem of supporting historical queries for graphs has only been addressed recently [15, 8, 9]. [9] presents a general framework for managing temporal information based on logs, called graph deltas, and explores different ways for using deltas to construct graph snapshots. Based on the similarity in a sequence of graphs produced by a graph evolving through time, in [15], graph representatives are computed by clustering similar graphs and differences from these representatives are stored. In [8], snapshot retrieval queries traverse a hierarchical index, the DeltaGraph, which compactly records the changes in a graph, and apply the updates recorded at its edges. Similarly to our cache, a GraphPool of cached snapshots is also used. Our approach supports, in addition to full snapshot construction, also partial view construction with reduced cost for targeted queries.

7. CONCLUSIONS & FUTURE WORK

In this paper, we support historical queries on graphs representing evolving social networks. We use a log for tracking changes on the graph, and aim at reducing the cost of the graph construction phase that precedes query evaluation, by using partial instead of full snapshot construction. We show when a partial view can be used for query evaluation and define algorithms for view construction and extension. We also propose using a cache of partial views and define a greedy solution for static view selection.

We plan to enhance view extension by including extension between views with different centers and show how views can be combined, i.e., by taking their union. We also intend to study variations of the view selection problem, such as a budgeted version with a limited cache size and a dynamic version equipped with cache replacement policies.

ACKNOWLEDGEMENTS

Research co-financed by the ESF and Greek national funds through the Operational Program "Education and Lifelong Learning" of NSRF-Research Funding Program:Thales:Cloud9.

8. REFERENCES

[1] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science Mag.*, 286:509–512, 1999.
[2] P. Buneman, S. Khanna, K. Tajima, and W. C. Tan. Archiving scientific data. *ACM TODS*, 29:2–42, 2004.
[3] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. Query processing under glav mappings for relational and graph databases. *PVLDB*, 6(2):61–72, 2012.
[4] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *22nd VLDB*, 1996.
[5] C. Gutierrez, C. A. Hurtado, and A. Vaisman. Introducing time into rdf. *IEEE TKDE*, 19(2):207–218, 2007.
[6] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. Gbase: a scalable and general graph management system. In *KDD*, 2011.
[7] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: mining peta-scale graphs. *Knowl. Inf. Syst*, 27(2):303–325, 2011.
[8] U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. In *ICDE*, 2013.
[9] G. Koloniari, D. Souravlias, and E. Pitoura. On graph deltas for historical queries. In *1st WOSS*, 2012.
[10] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *KDD*, 2005.
[11] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.
[12] A. Marian, S. Abiteboul, G. Cobena, and L. Mignet. Change-centric management of versions in an xml warehouse. In *27th VLDB*, 2001.
[13] W. E. Moustafa, A. Deshpande, and L. Getoor. Ego-centric graph pattern census. In *ICDE*, 2012.
[14] G. Özsoyoglu and R. T. Snodgrass. Temporal and real-time databases: A survey. *IEEE TKDE*, 7(4):513–532, 1995.
[15] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng. On querying historical evolving graph sequences. *PVLDB*, 4(11):726–737, 2011.
[16] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Comput. Surv.*, 31(2):158–221, 1999.
[17] D. Theodoratos and T. Sellis. Data warehouse configuration. In *23rd VLDB*, 1997.
[18] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi. On the evolution of user interaction in facebook. In *WOSN*, 2009.