COSI: Cloud Oriented Subgraph Identification in Massive Social Networks

Matthias Bröcheler
University of Maryland
A.V. Williams Building
College Park, MD 20742, USA
matthias@cs.umd.edu

Andrea Pugliese
Università della Calabria
Via P. Bucci, 41/C
Rende, Italy
apugliese@deis.unical.it

V.S. Subrahmanian
University of Maryland
A.V. Williams Building
College Park, MD 20742, USA
vs@cs.umd.edu

***Abstract*—Subgraph matching is a key operation on graph data. Social network (SN) providers may want to find all subgraphs within their social network that "match" certain query graph patterns. Unfortunately, subgraph matching is NP-complete, making its application to massive SNs a major challenge. Past work has shown how to implement subgraph matching on a single processor when the graph has 10-25M edges. In this paper, we show how to use cloud computing in conjunction with such existing single processor methods to efficiently match complex subgraphs on graphs as large as 778M edges. A cloud consists of one "master" compute node and k "slave" compute nodes. We first develop a probabilistic method to estimate probabilities that a vertex will be retrieved by a random query and that a pair of vertices will be successively retrieved by a random query. We use these probability estimates to define edge weights in an SN and to compute minimal edge cuts to partition the graph amongst k slave nodes. We develop algorithms for both master and slave nodes that try to minimize communication overhead. The resulting COSI system can answer complex queries over real-world SN data containing over 778M edges very efficiently.**

I. Introduction

During the last decade, there has been viral growth in social networks. FaceBook, Flickr, Twitter, YouTube and Blogger, all implement social networks. Both SN owners and SN users are interested in a variety of queries that involve subgraph matching. For example, consider the small social network shown in Fig. 1. Users of such a network might ask queries such as:

(Q1) Find all vertices $?v_1, ?v_2, ?v_3, ?p$ such that $?v_1$ works at the University of Maryland and $?v_1$ is a faculty member and $?v_2$ is an Italian university and $?v_3$ is a faculty member at $?v_2$ who is a friend of $?v_1$ and $?v_3$ has commented on a posting (or paper) $?p$ by $?v_1$. This query corresponds to a subgraph as shown in Fig. 2 – it might be used by a University President to find existing interactions between his faculty and those in Italy (e.g., just before he goes for a meeting with the Italian embassy). When this query subgraph is posed against an enormous SN, we do not have the option of matching the subgraph in a naive way against the graph – without intelligent pre-processing, the query would simply take too long.

(Q2) Consider a financial network whose nodes are account holders and banks - edges capture banking transactions including bank transfers, lines of credit, etc. A financial crimes investigator who has determined that a particular bank (say *Bank1*) is suspicious might want to ask a query of the form: "Find all vertices $?v_1, ?v_2$ such that $?v_1$ wired money to *Bank1* and *Bank1* received a wire from $?v_2$ and both $?v_1$ and $?v_2$ have a common friend $?v_3$ who has been labeled suspicious". The financial crimes investigator might believe that such pairs of people $(?v_1, ?v_2)$ are suspicious and worth a further look.

Both queries above contain multiple vertices and different relationships between them, demonstrating the need to execute complex queries over social networks. In addition, answering SPARQL queries in the Semantic Web's RDF framework largely involves subgraph matching. We show how to answer such queries and more complex ones over large social networks efficiently.

Fig. 1. Example social network

While all graphs can be represented trivially in relational database format, these queries both involve very large joins – thus, executing these queries on a standard relational DBMS

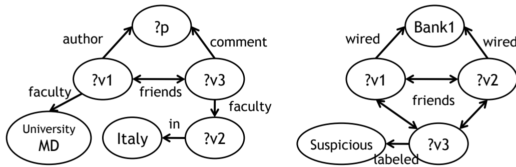

Fig. 2. Example queries Q1 (left) and Q2 (right)

incurs a very high cost. For instance, on a PostGres relational database using a graph of size 16M edges, queries of similar complexity took an average of 168 seconds to execute. In contrast, graph-based indexing systems such as DOGMA [1] take much less time (0.31 seconds on average) on the same hardware. The case for graph-based indexes has separately been argued in several works (see, e.g., [2]).

Section II defines a social network $\mathcal{S}$ as a labeled directed graph. We define queries over SNs as graphs Q – answers to queries are all subgraphs in $\mathcal{S}$ that "match" the query graph Q. Our cloud oriented approach to executing such queries has k slave nodes and one master node. Computation is completely distributed and asynchronous and our goal is to minimize communication between nodes. To achieve this, we start with a known probability distribution over queries (which can, for example, be derived from historical data of an SN) and we transform the SN $\mathcal{S}$ into a *weighted* graph. This transformation is achieved by identifying probabilities that a vertex in $\mathcal{S}$ will be retrieved from a slave node in response to a random query and by identifying probabilities that an arbitrary pair of vertices in $\mathcal{S}$ will be retrieved from a slave node. Section III defines these probabilities and shows how the minimal edge cuts of the resulting weighted graph correspond exactly to a partition of the graph that has minimal *expected* cost of execution when amortized across arbitrary queries. In Section IV, we present an algorithm to partition an SN across k slave nodes in accordance with the above strategy and includes algorithms for batch insertion of new edges (and vertices) into slave nodes. Within a slave node, any strategy to store graph data on a single disk (e.g., DOGMA [1]) can be used. Section V presents two cloud oriented query processing algorithms for subgraph matching queries — *other types of queries are not handled in this paper*. Section VI includes experimental results showing that our algorithms can efficiently answer complex queries over SNs containing 778M edges. In Sections VII and VIII we discuss related work and outline conclusions.

The approach to efficiently answering subgraph matching queries using a cloud of compute nodes presented in this work applies to general graph datasets and therefore makes this work applicable beyond the social networking realm. For instance, a significant fragment of the SPARQL query language for RDF, a popular data format used on the semantic Web, can be represented as subgraph matching queries.

II. Basic Notation

Throughout this paper, we assume the existence of an arbitrary, but fixed set $\mathcal{V}$ whose elements are called vertices. For example, $\mathcal{V}$ might consist of all strings that can form a valid userid and/or the set of all valid identifiers for comments in a social network like Facebook (for the sake of simplicity, we are ignoring files, etc.). We also assume the existence of a finite set $\mathcal{P}$ of *predicate symbols*.

A *social network* $\mathcal{S}$ is a triple (V, E, λ) where $V \subseteq \mathcal{V}$ is the set of vertices in the social network, $E \subseteq V \times V$ is a *multiset* of edges from vertices to vertices and $\lambda : E \rightarrow \mathcal{P}$ assigns a predicate symbol to each edge in E.

Throughout this paper, we assume an arbitrary, but fixed social network $\mathcal{S} = (V, E, \lambda)$ — we use the toy university SN example in Fig. 1 for illustrative purposes.

The *out neighborhood* of vertex v is the set $out(v) = \{u \mid (v, u) \in E\}$; the *in neighborhood* of node v is the set $in(v) = \{u \mid (u, v) \in E\}$. The neighborhood of v is the set $ngh(v) = out(v) \cup in(v)$. Each of these neighborhoods can be restricted to a particular predicate symbol p: for example, $out_p(v) = \{u \mid (v, u) \in E \wedge \lambda(v, u) = p\}$.

When formulating queries, we assume the existence of a set VAR of variable symbols ranging over V. Each variable symbol starts with a ?. A *query* Q is a triple (V_Q, E_Q, λ_Q) where $V_Q \subseteq V \cup VAR$, $E_Q \subseteq V_Q \times V_Q$ is a multiset of edges, and $\lambda_Q : E_Q \rightarrow \mathcal{P}$. We use VAR_Q to denote the set of variable vertices in query Q.

Suppose $\mathcal{S}$ is an SN and Q is a query. A *substitution* for query Q is a mapping $VAR_Q \cap \mathsf{VAR} \rightarrow V$. If θ is a substitution for query Q, then $Q\theta$ denotes the replacement of all variables $?v$ in V_Q by $\theta(?v)$. Hence, the graph structure of $Q\theta$ is exactly like that of Q except that nodes labeled with variables are replaced by vertices in the SN $\mathcal{S}$. A substitution θ is an *answer* for query Q w.r.t. SN $\mathcal{S}$ iff $Q\theta$ is a subgraph of $\mathcal{S}$. The *answer set* for query Q w.r.t. an SN $\mathcal{S}$ is the set $\{\theta \mid Q\theta$ is a subgraph of $\mathcal{S}\}$. For example, the answer to the query (Q1) w.r.t. the SN in Fig. 1 is $\{(?v1 = \text{Prof Dooley}, ?v2 = \text{Universita Calabria}, ?v3 = \text{Prof Calero}, ?p = \text{Paper ABC})\}$.

III. Partitioning Social Networks Across a Compute Cloud

This section describes how an SN may be "split" across a compute cloud so that we can efficiently process subgraph matching queries. We assume that a compute cloud consists of k "slave" nodes and one "master" node. Slave nodes communicate *directly* without going through the master, thus preventing the master from becoming a communication bottleneck. The master takes an initial query Q and directs it or parts of it to one or more slave nodes that then complete the computation of the answer with *no further interaction with the master* till the complete answer is assembled. At this stage, the complete answer is shipped to the master which sends the result to the user. The master is primarily an interface to the user.

We transform the SN first into a weighted graph. Intuitively, the weight of an edge (v_1, v_2) refers to the sum of the probability that v_2 will be retrieved immediately after v_1 and

vice versa when an arbitrary query is processed. Intuitively if this probability is (relatively) high, then the two vertices should be stored on the same slave node. We then use these to partition the SN across k slave nodes so that expected communication costs are minimized.

Throughout this section, we assume there is a probability distribution $\mathbb{P}$ over the space of all queries. Intuitively, $\mathbb{P}(q)$ is the probability that a random query posed to a SN is q. For any real world SN like FaceBook or Orkut, $\mathbb{P}$ can be easily learned from frequency analysis of past query logs.

A. Probability of Vertex (Co-)Retrievals

A *query plan* $qp(Q)$ for a query Q is a sequence of two types of *operations*: the first type retrieves the neighborhood of vertex v (from whichever slave node it is on), and the second type performs some computation (e.g. check a selection condition or perform a join) on the results of previous operations. This definition is compatible with most existing definitions of query plans in the database literature.

Definition 3.1 (Query trace): Suppose $x = qp(Q)$ is a query plan for a query Q on an SN $\mathcal{S}$. The *query trace* of executing x on $\mathcal{S}$, denoted $qt(x, \mathcal{S})$, consists of (i) all the vertices v in $\mathcal{S}$ whose neighborhood is retrieved during execution of query plan x on $\mathcal{S}$, and (ii) all *pairs* (u, v) of vertices where immediately after retrieving u's neighborhood, the query plan retrieves v's neighborhood (in the next operation of x). □

Traces contain consecutive retrievals of vertex neighborhoods. This allows us to store neighborhoods of both u and v on the same slave node, avoiding unnecessary communication.

When processing a query, we make the reasonable assumption that index retrievals are cached so that repeated vertex neighborhood retrievals are read from memory and hence the query trace $qt(x, \mathcal{S})$ can be defined as a set rather than as a multiset. The probability distribution $\mathbb{P}$ on queries can be used to infer a probability distribution $\tilde{\mathbb{P}}$ over the space of feasible query plans. $\tilde{\mathbb{P}}(x) = \sum_{Q \in \mathcal{Q}: qp(Q)=x} \mathbb{P}(Q)$. This says that the probability of a query plan is the sum of the probabilities of all queries which use that query plan.[1] We can now define the probabilities of retrieval and co-retrieval as follows.

Probability of retrieving vertex v. The probability, $\mathbb{P}(v)$, of retrieving v when executing a random query plan is $\sum_{x \in qp(\mathcal{Q}): v \in qt(x,\mathcal{S})} \mathbb{P}(x)$. Thus, the probability of retrieving v is the sum of the probabilities of all query plans that retrieve v.

Probability of retrieving v_2 immediately after v_1. The probability $\mathbb{P}(v_1, v_2)$ of retrieving v_2 immediately after v_1 is $\sum_{x \in qp(\mathcal{Q}): (v_1,v_2) \in qt(x,\mathcal{S})} \mathbb{P}(x)$. This says that the probability of retrieving v_2 immediately after v_1 is sum of the probabilities of all query plans that retrieve v_2 immediately after v_1.

B. Partitioning an SN Across a Cloud

We can associate a weighted graph $\mathsf{WG}(\mathcal{S})$ with the SN $\mathcal{S} = (V, E, \lambda)$. The weighted graph is the complete graph $(V, V \times V, w)$ where $w(v_1, v_2) = \mathbb{P}(v_1, v_2) + \mathbb{P}(v_2, v_1)$.

[1]In the rest of the paper, we will abuse notation and denote both PDFs by $\mathbb{P}$.

The following important theorem shows that the minimal edge cuts[2] of $\mathsf{WG}(\mathcal{S})$ correspond to the partition of $\mathcal{S}$ across k slave nodes that minimizes expected cost of executing a query.

Theorem 3.2: Assuming uniform costs (across all slave nodes) for retrieving a vertex neighborhood and sending a message to a node, the partition of $\mathcal{S}$ which minimizes the total query execution time of a random query coincides with the partition that minimizes the cost of a minimal edge cut of $\mathsf{WG}(\mathcal{S})$. □

To see why Theorem 3.2 is true, suppose we choose a query Q uniformly at random. Q's cost depends on the time to retrieve vertex neighborhoods of all vertices $v \in qt(qp(Q), \mathcal{S})$. The total time needed to retrieve vertex neighborhoods is independent of the partition (as we assume all slave nodes are equally fast at retrieving vertex neighborhoods). Thus we need a constant amount of time to retrieve $ngh(v)$ for all $v \in qt(qp(Q), \mathcal{S})$ irrespective of the partition block containing v.

Vertex co-retrievals costs, however, are dependent on the partition $\mathcal{P} = \{P_1, \ldots, P_n\}$ of $\mathcal{S}$. Suppose $(v_1, v_2) \in qt(qp(Q), \mathcal{S})$ with $v_1 \in P_i$, $v_2 \in P_j$, and $i \neq j$. In this case, the neighborhood retrieval for vertex v_1 is followed by that of v_2. As v_2 is in a different partition block residing on another slave node, we need to send a message to this node at a communication cost of c which adds to the total query execution time and depends on the partition $\mathcal{P}$. We define the indicator random variable $X_{(v_1,v_2)} = c$ if $(v_1, v_2) \in qt(qp(Q), \mathcal{S})$, and $X_{(v_1,v_2)} = 0$ otherwise. Using standard expected values, the total expected cost of communication for Q is $\mathbb{E}(\sum_{(v_1,v_2) \in E_d} X_{(v_1,v_2)}) = \sum_{(v_1,v_2) \in E_d} \mathbb{E}(X_{(v_1,v_2)}) = \sum_{(v_1,v_2) \in E_d} \mathbb{P}(E_{(v_1,v_2)}) \times c$, where $E_d = \{(v_1, v_2) | v_1, v_2 \in V_{\mathcal{S}}, v_1 \in P_i, v_2 \in P_j, i \neq j\}$.

As the size of an edge cut is the sum of weights of all edges that connect vertices in different partition blocks, the partition which minimizes the edge cut of the graph constructed in the statement of Theorem 3.2 is also the partition which minimizes the total expected cost of communication and therefore the variable part of the total expected query execution time.

IV. Partitioning in Practice

Even though Theorem 3.2 says we can partition the graph using edge cuts, this method is not usable in practice because computing minimal edge cuts is NP-complete [3]. Hence, we need fast algorithms to partition the graph. In this paper, we provide algorithms for batch insertion of edges (including potentially new vertices) into a social network $\mathcal{S}$. We do this via the important notion of a vertex force vector.

Definition 4.1 (Vertex force vector): Let $\mathcal{P} = \{P_1, \ldots, P_k\}$ be a partition of $\mathcal{S}$ and consider any

[2]An edge cut C of a weighted graph is a partition of the vertices into "blocks". An edge (u, v) in the graph is said to *cross the edge cut* C if u is in one block of the partition and v is in another. The size of an edge cut is the sum of the weights of the edges that cross the cut. C is said to be a *minimum cut* iff there is no other cut C' such that the the size of C' is less than the size of C.

block P_i. The *vertex force vector*, denoted $|\vec{v}|$, of any vertex $v \in \mathcal{S}$ is a k-dimensional vector where $|\vec{v}|[i] = f_{\mathcal{P}}(\sum_{x \in ngh(v) \cap P_i} w((v,x)))$ and $f_{\mathcal{P}} : \mathbb{R}^+ \rightarrow \mathbb{R}$ is a function called the *affinity measure*. □

A vertex force vector intuitively specifies the "affinity" between a vertex and each partition block as measured by the affinity measure $f_{\mathcal{P}}$. An affinity measure takes the connectedness between a vertex v and the respective partition block as an argument. The vertex force vector captures the strength with which each partition block "pulls" on the vertex and is used as the basis for a vertex assignment decision. If an inserted edge introduces a new vertex v, we first compute the vertex force vector $|\vec{v}|$ and then assign v to the partition block P_j where $j = argmax_{1 \leq i \leq k} |\vec{v}|[i]$. COSI uses an affinity measure that is a linear combination of three factors. We discuss the choice of coefficients in the experimental section.

Connectedness. Obviously, evaluating the connectedness of a vertex v to a partition block P_i is crucial for edge cut minimization – we measure this as the number of edges that connect v to the vertices in P_i.

Imbalance. Balanced partitions lead to even workload distribution, thus enhancing parallelism. Let $|P_i|_E = \sum_{x \in P_i} deg(x)$ be the number of edges in P_i; let T be an estimate (even a bad one) of the total number of edges that a given graph is expected to be. Then a reasonable measure of imbalance is the standard deviation of $\frac{\{|P_i|_E\}_{1 \leq i \leq k}}{T}$.

Excessive size. In addition to imbalance, we regulate the size of partition blocks by comparing the actual size of a block to its expected one. If a block grows beyond its expected size, we want to punish such growth more aggressively than imbalance does alone by reducing the affinity further according to the metric $\min(-\frac{|P_i|_E - \frac{T}{k}}{T}, 0)$.

A. Batch Edge Insertion

We now show how to insert a new set of edges into an SN, given that a partition $\mathcal{P} = P_1, \ldots, P_k$ of the SN already exists.[3] A naive GreedyInsert insertion algorithm iterates over all new vertices v: for each vertex v it computes the vertex force vector and assigns v to the block P_i such that $|\mathbf{v}|[i]$ is maximal — fortunately we can do better. This greedy algorithm is used as a baseline in our experiments.

Our COSI_Insert algorithm leverages graph *modularity* [4] to identify a strongly connected subgraph that is loosely connected to the remaining graph. However, modularity cannot be used blindly as our balance requirement must also be met.

Definition 4.2 (Modularity): The *modularity* of a partition $\mathcal{P}$ of an undirected graph $G = (V, E)$ with weight function $w : E \rightarrow \mathbb{R}$ is defined as

$$mod(\mathcal{P}) = \sum_{P \in \mathcal{P}} \left(\frac{W(P,P)}{2\,|E|} - \frac{deg_W(P)^2}{(2\,|E|)^2} \right)$$

where $deg_w(v) = \sum_{x \in V} w((v,x))$ is the weighted degree of vertex v, $W(X,Y) = \sum_{x \in X, y \in Y} w((x,y))$ is the sum of edge weights connecting two sets of vertices $X, Y \subset V$, and $deg_W(X) = \sum_{x \in X} deg_w(x)$ is the weighted degree of a set of vertices $X \subset V$. □

Intuitively, blocks with high modularity are densely connected subgraphs which are isolated from the rest of the graph. Our algorithm iteratively builds high modularity blocks and then assigns all vertices in a block to one slave node based on the vertex force vector. Let $B \subset V$ be a set of vertices. We generalize the notion of a vertex force vector to sets of vertices B by defining $\left|\vec{B}\right|[i] = f_{\mathcal{P}}\left(\sum_{v \in B} \sum_{x \in ngh(v) \cap P_i} w((v,x))\right)$. The intuition behind our partitioning algorithm is that assigning vertices at the aggregate level of isolated and densely connected blocks yields good partitions because (i) we respect the topology of the graph, (ii) most edges are within blocks and therefore cannot be cut, and (iii) force vectors of sets of vertices combine the connectedness information of many vertices leading to better assignment decisions.

The COSI_Partition algorithm (Fig. 3) initially assigns each vertex to be its own block via the assignment function c. The assignment function from initial blocks to the partition $\mathcal{P}$ is initialized to $\emptyset$ and subsequently updated to account for prior vertex assignments according to the parameter function α. The algorithm then repeatedly iterates over all new vertices u and determines whether moving u into any neighboring block increases modularity. If so, u is moved into the block which yields the largest increase and the assignments are updated. We continue iterating over all new vertices until the number of vertex moves l in the previous iteration falls below some threshold δ. This first part of the algorithm determines modular blocks by iteratively moving individual vertices to maximize modularity improvement ($\Delta M(u,t)$ denotes the change in modularity resulting from moving u into the block of t).

	Algorithm COSI_Partition **Input:** Undirected graph $G = (V, E)$, partial assignment function $\alpha : V \rightarrow \mathcal{P}$, weight function $w : E \rightarrow \mathbb{R}$ **Output:** Total assignment function $\alpha : V \rightarrow \mathcal{P}$
1	$(c : V \rightarrow \mathcal{C}) \leftarrow \emptyset$; $c(v) = \{v\} \forall v \in V$; $(b : \mathcal{C} \rightarrow \mathcal{P}) \leftarrow \emptyset$
2	**for all** $v \in$ domain(α)
3	$b(c(v)) = \alpha(v)$
4	**repeat**
5	$l \leftarrow 0$
6	**for all** $u \in V -$ domain(α)
7	$x \leftarrow$ argmax$_{t \in ngh(u)}[\Delta M(u,t) = \frac{2W(\{u\},b(t)) - 2W(\{u\},b(u)-u)}{2\lvert E\rvert} - \frac{2deg_w(u)\left[deg_W(b(t)) - deg_W(b(u)-u)\right]}{(2\lvert E\rvert)^2}]$
8	**if** $\Delta M(u,x) > 0$
9	$c(x) \leftarrow c(x) \cup \{u\}$; $c(u) \leftarrow c(u) - \{u\}$; $c(u) \leftarrow c(x)$; $l \leftarrow l + 1$
10	**until** $l < \delta$
11	$G_{\mathcal{C}} \leftarrow (\mathcal{C}, E_{\mathcal{C}})$ where $E = \{(x,y) \mid \exists u \in c^{-1}(x), v \in c^{-1}(y) : (u,v) \in E\}$
12	$w_{\mathcal{C}} \leftarrow E_{\mathcal{C}} \rightarrow \mathbb{R}$ where $w_{\mathcal{C}}((x,y)) = \sum_{u \in c^{-1}(x)} \sum_{v \in c^{-1}(y)} w((u,v))$
13	**if** $\lvert\mathcal{C}\rvert < \theta_1$ **or** $\frac{\lvert\mathcal{C}\rvert}{\lvert V\rvert} > \theta_2$
14	**for all** $i = 1, \ldots, n$
15	$e_i \leftarrow 0$
16	**for all** C randomly chosen from $\mathcal{C} -$ domain(b)
17	$b_i(C) \leftarrow P_m$ where $m = argmax_{1 \leq j \leq k} \left\lvert\vec{C}\right\rvert[j]$
18	$b \leftarrow b_j$ where $j =$ argmin$_{1 \leq i \leq n} \sum_{C_s, C_t \in \mathcal{C}, b_i(C_s) \neq b_i(C_t)} w((C_s, C_t))$
19	**else**
20	$b \leftarrow$ COSI_Partition$(G_{\mathcal{C}}, b, w_{\mathcal{C}})$
21	**for all** $v \in V$
22	$\alpha(v) \leftarrow b(c(v))$
23	**return** α

Fig. 3. COSI_Partition algorithm

The algorithm constructs a new graph $G_{\mathcal{C}}$ from the original graph G by collapsing all vertices assigned to the same block

[3]This can be used to create a partition of an SN for the first time by assuming $\mathcal{S} = \emptyset$.

during the modularity finding phase. If the graph $G_{\mathcal{C}}$ has less than some threshold number of vertices θ_1 or if its size is not significantly smaller than the original graph G, we stop and assign vertices to partition blocks. Otherwise, we call COSI_Partition recursively on the collapsed graph $G_{\mathcal{C}}$ to find modular blocks comprised of modular blocks, thereby building multiple levels of modular graphs. If the collapsed graph is small enough, we sequentially assign each vertex in $G_{\mathcal{C}}$ to the partition block which maximizes the force vector component as we did before. We repeat this process n times using different random permutation of the vertices and choose the assignment that minimizes the edge cut. Finally, we map the vertex assignment onto the original graph G, thereby projecting it down one level. COSI_Partition is guaranteed to respect prior vertex assignments (as specified by parameter α). Moreover, the size of the collapsed graph $G_{\mathcal{C}}$ is a constant factor smaller than the original graph and hence the size of the input graph decreases exponentially as the function calls itself recursively which contributes to the speed of the algorithm.

V. Query Answering

The COSI_basic parallel algorithm, shown in Fig. 4, operates asynchronously and in parallel across all slave nodes. A user issues query Q to the master node which "prepares" the query. In particular, it selects one constant vertex c from Q and determines the slave node S that hosts c using the function call location(c). The prepared query is then forwarded to S.

```
    Algorithm COSI_basic
    On master node
    Input: Graph query Q
    Output: Answer set A, i.e. set of substitutions θ s.t. Qθ is a subgraph of S
 1  for all z ∈ V_Q ∩ VAR
 2     R_z ← null /* no candidate substitutions for any vars in the query initially */
 3  for all z ∈ V_Q ∩ (S ∪ V)
 4     R_z ← {z}/* constant vertices only have themselves
                 as possible substitutions */
 5  qid ← next query ID /* uniquely identifies query */
 6  c ← argmin_{v∈V_Q} h_opt(v)/* pick optimal vertex to process next */
 7  send (Q, qid, c, {(c → c)}, {R_z}) message to location(c) slave node
    On slave node
    Input: Graph query Q, query ID qid, designated vertex c,
           partial substitution θ, candidate sets {R_z}, local index D
 8  for all edges e = (c, v) incident on c and some v ∈ V_Q ∩ VAR
 9     if R_v = null
10        R_v ← retrieveNeighbors(D, c, λ_Q(e)) /* use index to retrieve
                 all nbrs of c with same label as e * /
11     else
12        R_v ← R_v∩ retrieveNeighbors(D, c, λ_Q(e)) /* restrict space
                 of possible subst. for z */
13  if ∄v, u ∈ V_Q ∩ VAR : (u, v) ∈ E_Q/* have we found an answer? */
14     {v_1, . . . , v_l} ← {v | v ∈ V_Q ∩ VAR ∧ v ∉ θ}
15     for (s_1, . . . , s_l) ∈ R_{v_1} × R_{v_2} × . . . × R_{v_l}
16        θ' ← θ ∪ (v_1 → s_1) ∪ . . . ∪ (v_l → s_l)
17        Send (qid, θ') to master
18  else if ∃w ∈ V_Q : R_w = ∅/* reached dead end? */
19     return "NO"
20  else
21     w ← argmin_{v∈V_Q ∧|R_v|>0} h_opt(v)/* pick optimal vertex to process next */
22     for all m ∈ R_w
23        θ' ← θ ∪ {w → m}
24        send (Q, qid, m, θ', {R_z}) message to location(m) slave node
```

Fig. 4. COSI_basic algorithm

The algorithm proceeds depth first, substituting vertices for variables in Q one at a time. We maintain a set of result candidates for each variable in Q. This set is either uninitialized or is a superset of all result substitutions for that particular variable. The slave node query algorithm assumes there is an index retrieval function retrieveNeighbors(D, v, l) that retrieves $ngh_l(v)$ in sorted order from the local index D (which could be implemented many ways) on the slave. For now, h_{opt} arbitrarily chooses the next vertex to be substituted. A better definition of h_{opt} will be provided in the COSI_heur algorithm.

Incoming queries come with a selected variable to be instantiated with a vertex ID. The algorithm updates the candidate result sets by retrieving the neighborhood of the newly substituted vertex from the index. Since the result sets are sorted, this operation takes linear time. It then checks if any results have been found or whether the current substitution cannot yield a valid result. All query results are sent to the master which returns them to the user. If neither condition holds, the algorithm selects the next variable v' to be substituted and forwards the query to those slave nodes that host potential substitution candidates for v'.

The COSI_basic algorithm uses two optimizations to reduce messaging costs: (i) if the source and destination of the message in Lines 22-23 coincide, then the algorithm recursively calls itself with updated data structures rather then sending the message; (ii) the query processor groups all messages with the same query ID targeted at the same slave node and sends them in one message, thus reducing communication cost. COSI_basic does not rely on central orchestration – it uses depth first search so the branches of the search tree are traversed in parallel while ensuring that no branch gets explored multiple times. After forwarding the prepared query to a slave node, the master waits for incoming results of that query and forwards those to the user. As we explore branches in parallel, the master cannot be notified when the search for query results has completed. Keeping track of the current number of parallel executions for each query would introduce significant synchronization cost. Instead, the master keeps track of the time t_{last} at which the last result of a running query has come in. If the difference between the current time and t_{last} exceeds a threshold, the master asks all slave nodes for a list of query IDs of all currently running queries. The master merges these lists and closes all queries whose IDs are not contained. To avoid the case where a query is being forwarded to another slave node at the very moment that the master asks for all query IDs, each slave node keeps query IDs in their local list up to a certain grace period.

A. The COSI_heur algorithm

The choice of the next variable to be instantiated has profound implications on the running time of COSI_basic, as some substitutions yield larger branching factors in the search than others. COSI_heur handles this by choosing the variable vertex v' which has the lowest cost according to function h_{opt} . First, to reduce branching factor, we could choose the variable vertex v' with the smallest number of result candidates. This heuristic only considers the branching factor of the immediate next iteration, but is nevertheless an important metric to consider in the cost heuristic. Second, whenever we instantiate a vertex on a remote partition block,

we have to send a message to the appropriate slave which is expensive. Therefore, we consider the fraction of result candidates which are not stored locally as a cost metric. When we have to send a query to remote slaves for further processing, we would like to distribute the workload evenly across all slaves. Hence, we also analyze the distribution of result candidates by slave via the cost metric

$$ds(v) = \sqrt{\sum_{1 \leq i \leq k} \left(|R_v^i| - \frac{|R_v|}{k} \right)^2}$$

where R_v^i is the set of result candidates for vertex v restricted to those which reside on slave node i. Finally, we define

$$h_{opt}(v) = |R_v| \times (1 - \frac{|R_v^l|}{\alpha \times |R_v|}) \times (1 + \beta \times \frac{ds(v)}{|R_v|})$$

where l is the ID of the local slave node and α and β are constants that determine how much the model favors locality over parallelism. Our experiments study how α, β impact query run-times.

VI. Implementation Experimental Results

COSI is implemented in Java. We developed a communication infrastructure for the compute nodes based on the Java NIO libraries which is used to send the graph data during the loading and the queries during the query answering stages. The communication infrastructure handles contention at individual nodes and variations in network latency. It is optimized to ensure that the dispatcher's requests for outstanding queries are answered quickly. COSI uses a modified version of the DOGMA graph database [1] for local storage of the graph data. DOGMA itself interfaces against BerkeleyDB 4.8 [5] for on-disk storage. However, we emphasize that COSI is implemented independently from the underlying graph storage backend and can utilize alternative databases. The dispatcher node handles all user requests and maintains a vertex label lookup table on disk using BerkeleyDB. To reduce storage size as well as message size, all vertex labels are mapped onto unique IDs at the dispatcher node. When a query is issued to the dispatcher, it first retrieves the vertex ids for the labels before forwarding the query to the storage node. Conversely, it looks up the vertex labels for all ids contained in a query answer before forwarding it to the issuing client. Moreover, to facilitate efficient retrieval, vertex IDs include a namespace which uniquely identifies the storage node; thus, vertices can be directly located without a routing table.

In our experiments, we used a cluster of 16 compute nodes out of which one served as a dispatcher and the remaining 15 nodes served as storage nodes. All storage nodes had an identical hardware configuration with two Intel Xeon Quad Core 2.3 GHz Processors, 8 GB of RAM, and 73 GB SAS 10k RPM hard drive. The dispatcher's hardware differed slightly with 16 GB of RAM and two 146 GB 10k RPM SAS disks in RAID1 mirror configuration.

We fixed the coefficients for the affinity measure by hand. Both, the imbalance and excessive size metric, were given an equal weight of 1. The connectedness measure was set relative to the number of edges we considered per batch. We experimented with different batch sizes and found best performance for half a million edges.

By the definition of COSI_basic query answering algorithm in Figure 4, any two co-retrieved vertices must be connected. Hence, we set the probability of vertex co-retrieval to 0 for all unconnected pairs of vertices. The co-retrieval probability for connected vertices was set to 1.[4] Ideally, these probabilities would be derived from an analysis of query execution logs, however, such data was not available to us.

We used the social network data set studied in [6] for experiments. This data set contains 778M edges and describes personal relationships and group memberships crawled from Facebook, Orkut, Flickr, and LiveJournal.

To evaluate the performance of our proposed partitioning and query answering strategies, we designed 11 queries of varying size. In designing the queries, we first fixed the query graph topology and then randomly chose edge and vertex labels from the social network dataset while ensuring that the resulting query has a non-empty result set. The size of a query graph is measured by the number of edges and vertices it contains. We list some of the queries used in our experiments in the appendix.

A. Performance of COSI_Partition

Fig. 5 compares COSI_Partition's performance with that of the GreedyInsert algorithm. To validate our experiments, we used a random partitioning scheme, which assigns vertices to slave nodes uniformly at random, as the naive baseline in our experiments and report all results in comparison to this baseline. COSI_Partition achieves a substantial 36% improvement in edge cut over the naive baseline at a total running time of 10.5 hours for all 778M edges. GreedyInsert only achieves a marginal improvement in edge cut. COSI_Partition significantly outperforms greedy batch insertion by 33% with only slightly higher imbalance as measured in the standard deviation in partition block size relative to average size of a block. We observe that COSI_Partition substantially outperforms both baselines on the important edge cut quality metric.

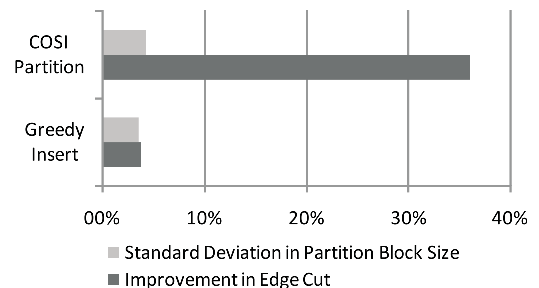

Fig. 5. Comparison of partitioning methods

[4]Note, that multiplying the probabilities of co-retrieval by a constant factor does not affect the edge cut minimization problem.

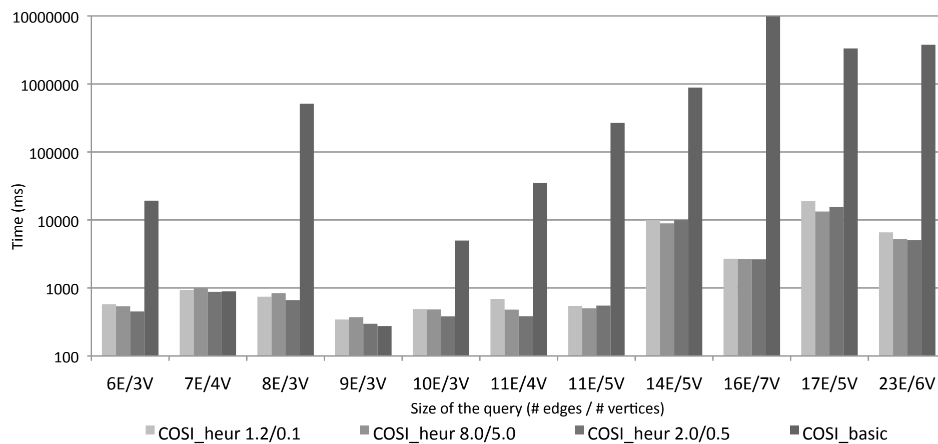

Fig. 6. Query times by query answering algorithm on the 778M edge Facebook/LiveJournal/Orkut data set

B. Query Answering

Fig. 6 compares COSI_basic against COSI_heur for three different parameter settings of the heuristic h_{opt}: $(\alpha = 1.2, \beta = 0.1)$ which strongly favors locality over parallelism, $(\alpha = 8.0, \beta = 5.0)$ which strongly favors parallelism over locality, and $(\alpha = 2.0, \beta = 0.5)$ which balances locality and parallelism. The queries have increasing complexity as measured by the number of edges (E) and variables (V) in the query graph. All query times were averaged across 6 independent runs with complete system restarts after each run to empty caches. Note, that the graph is plotted in logarithmic scale to accommodate the huge differences in query times.

COSI_heur drastically outperforms COSI_basic by up to 4 orders of magnitude on all but two queries, and the performance gap seems to grow exponentially with the query complexity. A close look at the difference in performance between the variants of COSI_heur reveals that the third configuration outperforms the first one on 9 queries, with a tie on the remaining 2, and outperforms the second configuration on 8 queries, being slower only on 3. These results suggest, that a balanced choice of parameters leads to a better h_{opt}.

C. Partition Impact

We derived theoretically in Theorem 3.2 that smaller edge cut leads to shorter expected query times. To verify the theorem experimentally, we compare the average query answering times over the partition generated by COSI_Partition against that produced by GreedyInsert. We have shown above that COSI_Partition produces partitions with lower edge cut which should reflect in shorter query times. We used the COSI_heur query answering with the third parameter configuration to compute the times for our set of queries and report the results in Fig. 7. The results support our hypothesis by showing that the partition produced by COSI yields significantly better query times for complex queries.

VII. Related Work

A wide variety of methods for SN analysis have been proposed and include tool kits such as UCINet [7] or Pajek [8]. However, most SN algorithms operate solely in memory, loading the entire graph from disk and then executing the analysis (see [9] for a survey of SN analysis software). For social networks of the size of Facebook, Flickr, or Orkut, such an approach becomes infeasible. To handle social networks of such magnitude one needs to store and query network data efficiently on disk. More importantly, complex queries involving even a few joins (as shown in the Introduction) can quickly cause such approaches to run into trouble. Ronen and Shmueli [10] introduce a social network specific query language and show how such queries can be answered on moderately sized datasets. However, their query language is geared toward users of a social network in helping them communicate with friends. Earlier work on database technologies for general graph data such as Lore [2] considered much smaller graphs than the social networks we study here. Graph structured RDF [11] data has been studied in the semantic Web community. Initial approaches to RDF storage [12], [13] stored triples in relational tables and then used a relational query engine to answer queries. [14] showed that storing RDF in a vertical database leads to significant query time improvements. [15] uses several B^+tree indexes for different permutations of subject, predicate, and object are maintained. All these approaches only work for single machines. In response to the increasing need of scalability when facing extremely large RDF datasets, two approaches have essentially been proposed so far: *scale up* and *scale out*. In scaling up, existing RDF databases, such as RDF-3X [15], Sesame [12], or YARS [16], are simply run on more powerful machines. As such it requires no technological innovation, but is extremely costly and limited by current hardware. In scaling out, multiple machines are utilized to store the data but all operations on the data are centrally executed. Parallel storage regimes, such as YARS2 [17], are cheaper but still limited in their scalability due to central execution. Our approach demonstrated efficient query answering across multiple machines without central orchestration.

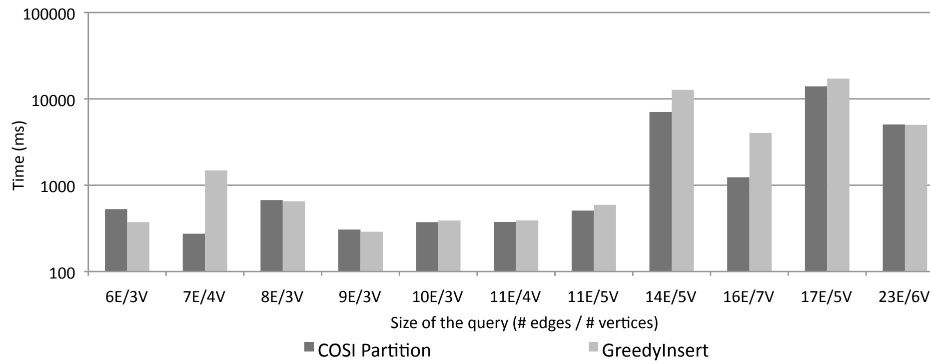

Fig. 7. Query times by partitioning method on the 778M edge Facebook/LiveJournal/Orkut data set

VIII. Conclusions

In this paper, we study subgraph matching on very large graph data using a cloud architecture. We develop a probabilistic model for retrieval of individual vertices and successive retrieval of vertices and use this model to show how splitting an SN $\mathcal{S}$ across k compute nodes can be reduced to a kind of minimal edge cut problem. As finding such edge cuts is NP-complete, we propose the COSI_Partition algorithm for creating such an index based on minimizing disconnectedness, imbalance and excessive size. Our COSI_basic algorithm and COSI_heur algorithms rely on a distributed communication mechanism between slave nodes to efficiently process queries on very large datasets. We show our framework works efficiently, answering many complex queries over a 778M edge real-world SN dataset derived from Flickr, LiveJournal, and Orkut in under one second.

References

[1] M. Bröcheler, A. Pugliese, and V. S. Subrahmanian, "DOGMA: A disk-oriented graph matching algorithm for RDF databases," in *ISWC*, 2009, pp. 97–113.

[2] R. Goldman, J. McHugh, and J. Widom, "From semistructured data to XML: migrating the Lore data model and query language," in *WebDB*, 1999, pp. 25–30.

[3] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, pp. 359–392, 1999.

[4] V. Blondel, J. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, p. P10008, 2008.

[5] BerkeleyDB, "*http://www.oracle.com/technology/products/berkeley-db*."

[6] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *IMC*, 2007, pp. 29–42.

[7] S. P. Borgatti, M. G. Everett, and L. C. Freeman, "Ucinet for windows: Software for social network analysis," *Harvard: Analytic Technologies*, 2002.

[8] W. Nooy, A. Mrvar, and V. Batagelj, "Exploratory social network analysis with pajek," *Structural analysis in the social sciences*, vol. 27, 2005.

[9] M. Huisman and M. A. V. Duijn, "Software for social network analysis," *Models and methods in social network analysis*, p. 270316, 2005.

[10] R. Ronen and O. Shmueli, "Evaluating very large datalog queries on social networks," in *EDBT*, 2009, pp. 577–587.

[11] O. Lassila and R. Swick, *Resource Description Framework (RDF) model and syntax specification*. W3C, 1998. [Online]. Available: http://www.w3.org/TR/1999/RECrdf -syntax-19990222.

[12] J. Broekstra, A. Kampman, and F. van Harmelen, "Sesame: An architecture for storing and querying RDF data and schema information," in *Spinning the Semantic Web*, 2003, pp. 197–222.

[13] K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds, "Efficient RDF storage and retrieval in Jena2," in *SWDB*, 2003, pp. 131–150.

[14] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach, "Scalable semantic Web data management using vertical partitioning," in *VLDB*, 2007, pp. 411–422.

[15] T. Neumann and G. Weikum, "RDF-3X: a RISC-style engine for RDF," *PVLDB*, vol. 1, no. 1, pp. 647–659, 2008.

[16] A. Harth and S. Decker, "Optimized index structures for querying RDF from the Web," in *LA-Web*, 2005, pp. 71–80.

[17] A. Harth, J. Umbrich, A. Hogan, and S. Decker, "YARS2: A federated repository for querying graph structured data from the web," in *ISWC*, 2007, pp. 211–224.

Appendix: Subset of Evaluation Queries

In the following, we list 3 of the 11 subgraph matching queries used in the experiments. Question marks denote variables and numbers denote anonymized vertex labels. Queries are represented as lists of directed edges:

```
start-vertex -- edge-type --> end-vertex

Query 1
?A -- member --> group675
?A -- knows --> ?B
?B -- member --> group224
?C -- knows --> ?B
?C -- member --> group333
?B -- knows --> ?A

Query 2
?A -- member --> group3085
?B -- knows --> ?A
?B -- knows --> ?C
?C -- knows --> ?A
?A -- member --> group4087
?C -- knows --> ?B
?B -- knows --> ?D

Query 9
?A -- member --> group682
?A -- knows --> ?B
?B -- knows --> ?A
?D -- knows --> ?A
?B -- knows --> user121
?D -- knows --> ?B
?D -- member --> group511
?A -- knows --> ?D
?C -- knows --> ?F
?C -- knows --> user1208
?F -- knows --> ?D
?F -- knows --> ?A
?B -- knows --> ?F
?C -- member --> group1209
?A -- knows --> ?F
?D -- knows --> ?F
?F -- member --> group311
```