Towards Publish/Subscribe Functionality on Graphs

Lefteris Zervakis[1] Christos Tryfonopoulos[1] Vinay Setty[2]
Stephan Seufert[2] Spiros Skiadopoulos[1]
[1] University of the Peloponnese, Tripolis, Greece
[2] Max Planck Institute, Saarbrücken, Germany
{zervakis,trifon,spiros}@uop.gr {vsetty,sseufert}@mpi-inf.mpg.de

ABSTRACT

In this work, we introduce the publish/subscribe paradigm to support continuous query processing over evolving graphs and motivate it for a number of applications and a variety of possible continuous queries. To the best of our knowledge, this is the first work in the literature that considers supporting publish/subscribe in graphs; we focus specifically on massive and dynamically evolving graphs due to the nature of the problem and the type of targeted applications. To this end, we design a proof-of-concept filtering algorithm for supporting structural matching of continuous graph queries against updates in the evolving graph and demonstrate the need for efficient filtering by experimentally comparing our algorithm against a baseline approach.

1. INTRODUCTION

In the modern digital era, *graphs* are ubiquitous and ever-present as they model a vast number of different problems, including social networks, knowledge bases, information and communication networks, distributed systems, biological interactions, and hyper-linked web-pages. Moreover, in many of these problems, graphs are not the reasonably-sized, static snapshots that data scientists are commonly assuming. Contrary, in typical applications graphs are *massive* in scale, *evolving* at varying rates (depending on the nature of the problem modelled), and often naturally *distributed* among different machines. Thus, the typical computational paradigm of posing a graph-related query (e.g., aiming at structural matching or certain graph properties) to the system and waiting to receive an answer seems insufficient at environments with such scale and dynamicity.

To address the aforementioned issues, researchers have shifted their attention towards less static modelling of graphs by adopting paradigms such as *evolving graphs* or *graph streams*. In these scenarios, a graph is considered as a stream of modifications (i.e., node and edge additions/deletions/updates) that trigger the (incremental) re-computation of some graph *properties* (e.g., density), cumulative *measures* (e.g., diameter), or subgraph *matching/mining*. Thus, the graph itself is considered to be dynamic, but the actual queries (i.e., computations) on the streaming input are typically *static* and cannot be modified. Moreover, the efficient evaluation of such computations is heavily dependent on the algorithm chosen for the task and evaluating more query types (let alone more than one queries) at once against the evolving graph is currently not supported. This happens because the adopted model focuses on mining the evolving graph and proves insufficient to address the challenges posed by applications that require the *filtering* of graph changes against *a set of (user) queries*.

In our work, we adopt the *publish/subscribe (pub/sub)* paradigm to evaluate *continuous queries* against an evolving graph. In this context, users (or services that act on users' behalf) pose continuous queries containing *sub-graph*, *path*, and *structural/attribute* constraints. In this model, continuous queries are appropriately *indexed* and efficiently evaluated against graph updates, avoiding the matching of the entire query database against every change in the graph stream. In this way, the matching queries are identified and the appropriate users are notified accordingly. This computational model allows us to offer functionality beyond anything supported by the current state-of-the-art and enables us to provide useful tools that allow users to subscribe to graph changes of interest. In the next section, we outline possible *applications* that may benefit from graph pub/sub and identify useful *query classes* for each application.

To this end, our contributions are summarised as follows. Initially, we advocate the application of the pub/sub paradigm in an evolving graph domain and highlight possible applications and useful queries; to the best of our knowledge this is the *first work* in the literature that deviates from the typical mining of evolving graphs to offer continuous query functionality to the end-user. Additionally, we highlight the problem of *indexing* the continuous query database to achieve efficient filtering of graph updates and demonstrate the necessity of *effective indexing structures* to support pub/sub in dynamically evolving graphs. Finally, we develop a proof-of-concept *filtering algorithm* that supports structural matching of continuous graph queries and experimentally demonstrate that it accelerates filtering by *four orders of magnitude* compared to a baseline approach.

2. APPLICATIONS AND OPEN ISSUES

To demonstrate the wide applicability of the proposed graph pub/sub paradigm we identify representative applications and useful query classes for these applications.

2.1 Applications for graph pub/sub

In this section, we discuss the application of graph pub/sub in different domains.

Pub/sub on social graphs. *Advertising* in social networks is becoming a major source of revenue for large corporations such as Facebook and Twitter. Social networks attract advertisers as they can target users by leveraging on publicly available user profiles/demographics. Prompt identification of influential users, active monitoring of their evolving interests over time, and fast adaptation to social graph changes could increase the effectiveness of advertisements and provide an advantage over the competition. To realise the above requirements, there is a need for a pub/sub tool that will allow advertisers to subscribe to evolving social graphs for subgraphs with given properties and notify them in real time whenever a match occurs.

Moreover, identifying communities of users with similar interests in a network of user purchases (such as Amazon) may provide a valuable extension for *(social) recommendation* systems. In such a scenario, given the explicit or implicit preferences of a given user, the challenge lies in identifying a community with similar interests and using their purchases to provide useful recommendations to the user. Given that such communities are dynamic in nature, there is a need for a pub/sub tool that will process continuous queries targeted to real-time community identification and/or group formation detection incrementally as the graph evolves.

Pub/sub in protein interaction graphs. *Protein-protein interactions* (PPIs) are particularly useful in biology research and are typically modelled as graphs, with proteins as nodes and identified interactions between them as edges, stored in central repositories. In this setup, the resulting PPI graph for an organism is continuously evolving by (i) the addition of new nodes/edges through the identification of new proteins/interactions, (ii) the deletion of edges due to false positives in the interaction identifications methods, and (iii) the modification of edge weights though the verification (or invalidation) of already discovered interactions. Moreover, bias in the PPI identification method may lead to differences between the actual and observed PPI network, which means that the graph may continuously change.

In such a setting, scientists that want to stay informed about newly discovered interactions, their relation and interplay with existing ones, and the important properties that may be inferred from such discoveries have to repeatedly resort to querying tools that are unable to capture the evolution of the graph. Thus, there is a clear need for a pub/sub solution that provides continuous querying functionality over the PPI networks; such a service would notify subscribed users whenever a graph structure of interest or of certain properties is identified/registered in the repository.

Pub/sub in other graphs. Other interesting graph pub/sub applications include curation and pattern identification in *knowledge graphs*, monitoring of *traffic networks*, and intrusion detection in *communication networks*.

2.2 Useful continuous query classes

In this section, we briefly discuss useful query classes that could be supported in the context of graph pub/sub.

Structural and attribute matching. In our model, users will be able to subscribe to specific *subgraphs* or *motifs* (subgraphs with a fixed number of nodes found often in a graph) that match given attribute-value predicates and get notified when the evolving graphs matches their queries. This would allow user/product monitoring in social graphs, clique/k-motif identification for certain proteins in PPIs, and quality control for monitored entities in knowledge graphs.

Clustering coefficient. Continuous queries that specify a *clustering coefficient* (or any similar) measure will be useful to identify communities for targeted advertising, predict/validate PPI interactions, and track trending entities/items in knowledge graphs.

Shortest path. *Shortest path* continuous queries are especially useful on PPI graphs as they enable biologists to perform functional correlations and structural annotations between (closely located) proteins. In this scenario biologists want to get notified when the shortest path between two given proteins drops below a provided threshold; tracking shortest paths between nodes can be beneficial for several other continuous queries such as betweenness centrality and Steiner tree computation discussed below.

Clique and motif enumeration. Continuous queries that are used to subscribe for certain thresholds or top-K style statistics for given *cliques* and *motifs* are particularly useful in PPI graphs for detecting functionally related proteins and protein complexes. In this scenario, a user would like to be notified when a given clique or motif becomes frequent (i.e., its number of instances exceeds a predefined threshold or is within the top-k most frequent patterns) in the PPI graph.

Betweenness centrality. *Betweenness centrality* is defined as the fraction of shortest paths passing through a node. Intuitively, nodes with higher betweenness centrality in social graphs have higher visibility and injecting promoted content at those nodes would increase the advertising effect. Similarly, betweenness centrality is a key measure for identifying important proteins in a PPI network since proteins that demonstrate high betweenness centrality are more likely to be essential proteins with interesting functional and dynamic properties. Betweenness centrality may be used as an additional constraint in continuous queries once a subgraph of interest (e.g., a subgraph with specific attribute values or above a certain clustering coefficient) is identified.

Node degree. Even though *node degree* is a simple metric, together with betweenness centrality constitute the two key characteristics for identifying important proteins in PPI graphs, while continuous queries with node degree constraints could be used (in conjunction with other metrics) to notify knowledge graph curators of new/trending entities/items.

Dense subgraph. Trending story/topic detection is an important area where dense subgraphs are known to be of benefit [2]. Contrary to [2], where the focus is on identifying the top-k densest subgraphs in a social media stream, our focus is on providing *dense subgraph* as a thresholded constraint in continuous queries (in conjunction with attribute/structural matching) to allow monitoring of developing stories/users in social graphs or trending items/entities in knowledge graphs.

Steiner tree. In knowledge graphs, continuous queries could be used to notify graph curators when a new *Steiner tree* is formed between given entities consisting of specifically labeled edges, in the spirit of [7] but modified for pub/sub in evolving graphs. Subscription to Steiner tree formation between nodes (or to Steiner points) could be used to monitor knowledge graph quality or emergence of interesting links.

3. ALGORITHMS AND EVALUATION

In this section, we outline two proof-of-concept filtering algorithms that match continuous queries against graph updates and present a concise experimental evaluation of their performance that highlights the need for efficient filtering.

3.1 Overview of filtering algorithms

In the context of our proof-of-concept implementation, we developed two algorithms that are able to filter continuous queries aiming at structural properties of the evolving graph. In our setup, continuous queries were arbitrary graphs that express user interests and are appropriately indexed depending on the filtering algorithm at hand. The evolving graph (against which the continuous queries are constantly, i.e., in every update, evaluated) is also indexed to allow for faster matching against the query database.

The first algorithm we developed has no query indexing strategy and scans the query database sequentially in a brute force manner (hence the name BF) on every graph update to determine matching queries. The BF algorithm stores the full query database in a linked list using an appropriate representation that allows it to match (at publication time) each continuous query against the indexed (evolving) graph. BF was implemented to serve as a simple baseline that would demonstrate the necessity of appropriate query indexing structures that will enhance the filtering process.

The second algorithm we designed decomposes the continuous query to the vertex pairs that form the query graph and uses these pairs as keys to store the query identifier at an inverted index (hence the name INV). In this way, a continuous query graph with k edges is decomposed into k vertex pairs and its identifier is stored at k different hash table buckets, one for each pair. An auxiliary table T is used to store the number of vertex pairs that are contained in each continuous query and the number of already matched pairs. When a new graph update is published, the vertexes that are involved in the update are used to locate all affected continuous queries in the inverted index and appropriately update T. Subsequently, T is scanned to determine whether any newly matching queries have arisen as a result of the new update. If so, the appropriate notifications are created and sent to the subscribed users.

As we show in the next section, employing even a simple indexing solution such as INV leads in significant improvements in filtering time over an exhaustive method.

3.2 Experimental evaluation

In this section, we present a series of experiments that compare the performance of algorithms BF and INV.

Evolving graph. We utilised the *Wikipedia Pagelinks* graph obtained from the *DBpedia* website as our evolving graph; the initial *Wikipedia Pagelinks* graph contains more than $19M$ unique pages (graph nodes) and over $158M$ links between these pages. To simulate the graph evolution, we obtained a snapshot of the original graph that contained $1M$ triples and simulated the graph evolution by adding those triples to an (initially empty) graph. The publication events (graph updates) resulted to a directed graph with more than $1.2M$ pages (vertices) connected by $1M$ links (edges).

Continuous query database. Since no database of continuous (graph) queries was available to us, we used the final graph to artificially generate realistic query databases of varying sizes and characteristics. The generated continuous queries belonged to three different query classes that capture different information needs, i.e., chains, stars, and arbitrary graphs, and each query class was chosen equiprobably. For our evaluation we generated query databases of varying (i) sizes –namely $qDB = \{10K, 30K, 50K\}$ queries, (ii) average query length –namely $qL = \{4, 5, 6\}$ triples/query, and (iii) matching percentage –namely $qPer = \{5\%, 10\%, 15\%\}$ of all queries matched the final graph.

Technical configuration. All algorithms were implemented in Java. For the indexing of the graph the JGraphT graph library was used. A PC with a Core *Xeon* 2.0GHz and $10GB$ RAM running Ubuntu Linux 14.04 was used. The time shown in the graphs is wall-clock time and the results of each experiment are averaged over 10 executions to eliminate any fluctuations in time measurements.

Evaluation results. Figure 1 presents the results from the evaluation of the algorithms INV and BF. Specifically, Figure 1(a) shows the time in nanoseconds required to insert $50K$ continuous queries with $qPer = 5\%$ when varying the query length qL and the query database size qDB is increasing. We observe that the insertion time of all algorithms remains the same as the qDB size increases, while insertion time increases with the average query length. Algorithm BF is 9 times faster to index a continuous query with $qL = 6$ compared to Algorithm INV. This happens due to the nature of each algorithm; BF simply inserts the continuous query at the end of a linked list, whereas INV needs to decompose the query into triples, access the involved inverted index buckets, and update table T. Figure 1(b) shows the time in nanoseconds required to match a graph update against a database of stored queries when varying the query database size qDB, while $qL = 5$ and $qPer = 5\%$. We observe that the filtering time of all algorithms increases with the query database size; algorithm INV is more than four orders of magnitude, i.e., 45000 times, faster in filtering an update against a database of $50K$ continuous queries compared to BF. In Figure 1(c) the filtering performance of the two algorithms when varying the query length qL, while $qDB = 50K$ and $qPer = 5\%$, is presented. The same performance is demonstrated from both algorithms; INV is again more than four orders of magnitude faster than BF. Finally, Figure 1(d) shows the filtering performance of the two algorithms when varying the matching percentage $qPer$, while $qDB = 50K$ and $qL = 5$; similarly INV outperforms BF by more than four orders of magnitude.

Summing up. In summary, for a query database of 50K queries, algorithm INV is able to support a throughput of more than 2M updates/sec in contrast to BF that supports around 500 updates/sec. This significant difference in the performance of the two algorithms demonstrates the need for efficient query indexing structures that will enable us to support real-life dynamically evolving graphs.

4. RELATED WORK

Structural graph pattern search using graph isomorphism has been studied in the literature before [11, 6]. However, all existing techniques are designed for static graphs and are not suitable for processing continuous graph queries on evolving graphs. The problem of continuous sub-graph matching has been considered in [13] but the authors (i) assume a static set of sub-graphs to be matched against update events, (ii) use approximate methods that generate false positives, and (iii) apply the solutions on small (evolving) graphs.

Pub/sub is a widely used communication paradigm to process continuous queries on dynamic data. Several classes of pub/sub systems and subscription languages have been proposed in the literature [5]. Pub/sub solutions on ontology graphs are proposed in [10, 14], but they are limited to the RDF graphs and RDF specific subscriptions. Distributed pub/sub middleware for graphs has recently been proposed in [4], but the authors do not consider graph structure (they limit subscriptions to node attributes and node distance constraints). Finally, in [3] the problem of evaluating graph

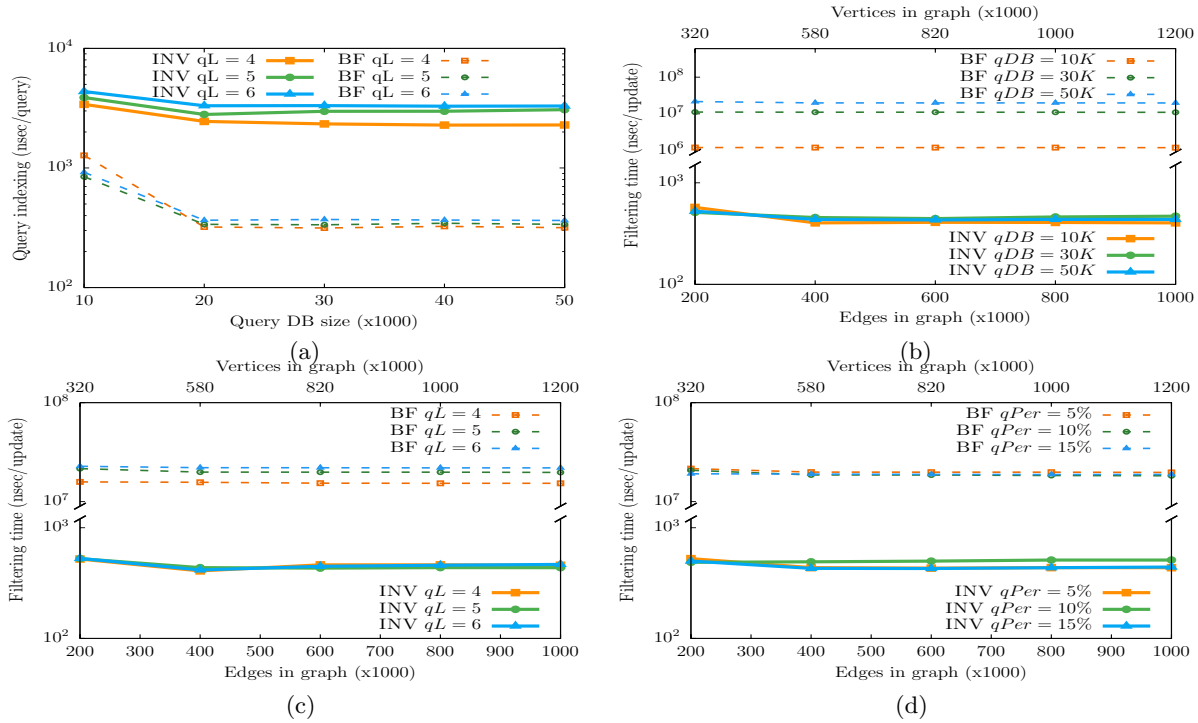

Figure 1: Experimental evaluation of algorithms BF and INV under different scenarios (semi-log plots).

constraints between publishers and subscribers is presented and applied to a distributed Web advertising scenario.

Another relevant area of research is graph streams; in [8, 9], algorithms to identify correlated graphs from a graph stream, by using a sliding window that covers a number of consecutive batches of stream data records, are proposed. This is different from our setting, as they aim at identifying subgraphs from a streaming graph that have Pearson correlation coefficients higher than a given threshold without considering the existing graph.

Sub-graph properties such as clustering coefficient and density have been considered with respect to evolving graphs before (e.g., top-k densest sub-graph maintenance [2] and dynamic community detection based on clustering coefficient [12]), but not in a pub/sub setup. Contrary to these graph mining approaches that focus on a specific property, we envision a pub/sub service that will support a rich set of continuous queries containing sub-graph, path, and structural/attribute constraints specified over evolving graphs. Finally, graph pub/sub relates to evolutionary network analysis [1], but in pub/sub the focus is not on maintenance/ analysis of the (evolving) graph.

5. OUTLOOK

We plan to investigate more sophisticated query indexing solutions that utilise statistical information on graph updates and commonalities between queries to achieve faster filtering. We also plan to extend our solutions to additional query classes (as identified in Section 2). Finally, we plan to develop a distributed solution for graph pub/sub that will fit naturally distributed evolving graphs and will be able to scale to massive amounts of data.

6. REFERENCES

[1] C. Aggarwal and K. Subbian. Evolutionary Network Analysis: A Survey. *ACM CSUR '14.*

[2] A. Angel, N. Sarkas, N. Koudas, and D. Srivastava. Dense subgraph maintenance under streaming edge weight updates for real-time story identification. *VLDB Endowment '12.*

[3] A. Broder, S. Das, M. Fontoura, B. Ghosh, V. Josifovski, J. Shanmugasundaram, and S. Vassilvitskii. Efficiently Evaluating Graph Constraints in Content-Based Publish/Subscribe. *WWW '11.*

[4] C. Canas, E. Pacheco, B. Kemme, J. Kienzle, and H.-A. Jacobsen. GraPS: A Graph Publish/Subscribe Middleware. *Middleware '15.*

[5] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM CSUR '03.*

[6] H. He and A. Singh. Closure-tree: An index structure for graph queries. *ICDE, '06.*

[7] G. Kasneci, M. Ramanath, M. Sozio, F. M. Suchanek, and G. Weikum. STAR: Steiner-Tree Approximation in Relationship Graphs. *ICDE '09.*

[8] S. Pan and X. Zhu. CGStream: continuous correlated graph query for data streams. *CIKM '12.*

[9] S. Pan and X. Zhu. Continuous top-k Query for Graph Streams. *CIKM '12.*

[10] M. Petrovic, H. Liu, and H.-A. Jacobsen. G-ToPSS - fast filtering of graph-based metadata. *WWW '05.*

[11] D. Shasha, J. Wang, and R. Giugno. Algorithmics and Applications of Tree and Graph Searching. *PODS '02.*

[12] C. Tantipathananandh, T. Berger-Wolf, and D. Kempe. A framework for community identification in dynamic social networks. *ACM KDD '07.*

[13] C. Wang and L. Chen. Continuous Subgraph Pattern Search over Graph Streams. *ICDE '09.*

[14] J. Wang, B. Jin, and J. Li. An Ontology-Based Publish/Subscribe System. *Middleware '04.*