An Overview of Methods for Handling Evolving Graph Sequences

Andreas Kosmatopoulos[1](✉), Kalliopi Giannakopoulou[2], Apostolos N. Papadopoulos[1], and Kostas Tsichlas[1]

[1] Department of Informatics, Aristotle University of Thessaloniki, Thessaloniki, Greece
{akosmato,papadopo,tsichlas}@csd.auth.gr

[2] Department of Computer Engineering and Informatics, University of Patras, Patras, Greece
gianakok@ceid.upatras.gr

Abstract. Graph data structures constitute a prominent way to model real-world networks. Most of the graphs originating from these networks are dynamic and constantly evolving. The state (snapshot) of a graph at various time instances forms an evolving graph sequence. By incorporating temporal information in the traditional graph queries, valuable characteristics regarding the nature of a graph can be extracted such as the evolution of the shortest path distance between two vertices through time. Most modern graph processing systems are not suitable for this task since they operate on single very large graphs. In this work we review centralized and distributed methods and solutions proposed towards handling evolving graph sequences.

Keywords: Snapshots · Evolving graph sequences · Temporal graphs

1 Introduction

Modern times, have witnessed a rapidly expanding volume of data generated by significantly different types of sources. A substantial portion of the available data, such as data originating from social networks, citation networks, sensor networks and others [13], can be modeled into graph data structures. The vertices of these graphs represent the entities of each network while the edges express relationships between the different entities. As an example, in a graph corresponding to a social network, the vertices denote the users of the network and the edges signify the friend-relationships between them.

A common characteristic of most real-world networks is that they do not remain static and are constantly evolving. For instance, the state of Facebook on one day is different to its state on the following day since there have been new user accounts created and friendships formed or deleted. Other networks, such as citation networks, only grow larger as they move forward in time since, due to the network's nature, vertices and edges are only added and never deleted.

It follows that, there exists a range of queries that aim to provide further insight on the nature of each network by incorporating temporal aspects in the traditional graph processing methods. Some examples of these queries would be to determine the evolution of a graph's diameter, the shortest path distance of two vertices through time and the vertex degree distribution of a graph at different time instances.

By periodically collecting the state of a graph at various time instances we form an *evolving graph sequence*. Current centralized and distributed graph processing systems such as Pregel [14], Neo4j [15], Trinity [19], Giraph [7] and others focus on processing single and very large graphs without supporting temporal extensions to the typical graph processing queries. As a result, these systems are not inherently suitable for performing analysis on evolving graph sequences.

Most of the research conducted towards handling evolving graph sequences aims to exploit the commonalities that exist between a graph in different time instances in order to improve space or time efficiency. As an example, even though social networks are dynamic and change over time, the majority of the users and the friend relationships between them remain the same across multiple time instances. For that reason, a system that effectively handles evolving graph sequences should perform better compared to a single graph processing system that operates on the individual graphs of the sequence.

The work performed in the area is in an inceptive stage and thus we present solutions for both centralized and parallel or distributed approaches. Among the centralized methods is the FVF framework by Ren et al. [17] that groups the sequence graphs into clusters and operates on them. Another method was proposed by Koloniari et al. [10] and it is based on maintaining a log of operations (defined as deltas) that occur in the graph between various time instances and employing it to reconstruct the graph at a particular time instance. Caro et al. [2,4] proposed space-efficient methods that utilize compact and self-indexed data structures to reduce the total space cost. Finally, methods have been proposed [1,8,18,21] that index the sequence in a manner that permits the efficient evaluation of certain queries. In the parallel and distributed setting there have been two main methods proposed: The DeltaGraph system [9] is based on the principle of deltas and aims to efficiently store and retrieve the graph at specific time instances. Finally, the G* system [11,12,20] is a parallel graph database that focuses on taking advantage of the commonalities present between a graph in different time instances to store the sequence in an efficient manner.

The rest of the work is organised as follows. In Sect. 2 we provide formal definitions regarding graphs, evolving graph sequences and a general problem definition. In Sect. 3 we present centralized methods and in Sect. 4 we focus on the parallel and distributed approaches. Finally, we conclude our work in Sect. 5.

2 Definitions

In this section we will provide some basic definitions about the general problem setting. First, we will formally define evolving graph sequences and then move on to discuss about the different query types that can be performed with regard to evolving graph sequences. Without loss of generality we will focus on undirected graphs since directed graphs mostly follow the same principles.

Definition 1 (Evolving Graph Sequence). *We define an evolving graph sequence $\mathcal{G}$ to be a collection of snapshots $\mathcal{G} = \langle G_1, G_2, G_3, \ldots \rangle$. A graph snapshot $G_i \in \mathcal{G}$ where $G_i = (V_i, E_i)$, corresponds to the graph G at time instance i and is characterized by a set of vertices V_i and a set of incoming and outgoing edges E_i.*

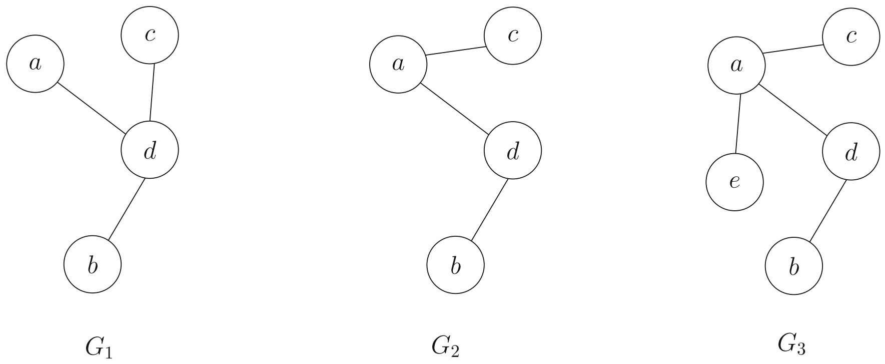

Fig. 1. An evolving graph sequence

The rate at which snapshots are obtained depends on the underlying network that the graph represents and is largely application-specific. Figure 1 depicts an evolving graph sequence which will serve as a running example for the remainder of this work. In this example, the evolving graph sequence $\mathcal{G}$ is composed of three snapshots G_1, G_2 and G_3 with each snapshot corresponding to the state of the graph G at time instances 1, 2 and 3 respectively. To obtain a particular snapshot from another snapshot in the sequence a set of operations has to be performed (e.g. by adding an edge between a and c in G_1 and removing the edge between c and d we obtain G_2). It is worth noting that a set of vertices or edges may not change at all in the entire sequence (e.g. b) and this fact can be exploited to reduce the total space or time cost when storing or querying the sequence respectively.

The aim of a system that handles evolving graph sequences is to efficiently store or index the sequence so as to answer historical analytic queries. We distinguish between two versions of the problem setting. In the *offline* version the entire sequence $\mathcal{G}$ is known beforehand and update operations are not supported in any snapshot (i.e. $\mathcal{G} = \langle G_1, G_2, G_3 \rangle$ only consists of G_1, G_2 and G_3 and no new snapshots are created). In the *online* version, $\mathcal{G}$ is constantly evolving and is not characterized by a "final" snapshot (i.e. $\mathcal{G} = \langle G_1, G_2, G_3, \ldots \rangle$ may eventually end up with more than three snapshots).

2.1 Query Types

The queries that can be performed upon evolving graph sequences can be characterized with respect to two main types [10]: their time domain and their graph scope. Regarding the time domain, queries are performed on either a particular time point or a time interval. In the case of a time point query we are interested in extracting a characteristic of the graph at a time instance t, while in a time interval query the objective is to study the evolution of a graph measure through an interval of time $[t, t']$. Queries are also distinguished by the scope of the graph that they operate on. More specifically, a query is focused on evaluating a graph measure concerning either a small set of vertices or the entire graph.

Most of the queries can be mapped to a combination of these two categories. As an example, consider the query "How has the shortest distance between a and c evolved over time instances t_s and t_e" which can be defined as a time interval query that focuses on a set of vertices. Similarly, the query "What is the diameter of G at time instance t_i" is a time point query that is concerned with the entire graph.

3 Centralized Methods

Having provided the basic definitions with respect to the problem of handling evolving graph sequences, we move on to methods and solutions proposed for centralized environments. We begin with the FVF framework by Ren et al. [17] followed by the works of Koloniari et al. [10] and Caro et al. [4]. We conclude the section by discussing indexing methods for evolving graph sequences that tackle certain historical queries.

3.1 The FVF Framework

The first centralized method we review is the FVF (FIND - VERIFY - FIX) framework proposed by Ren et al. [17]. The authors describe a method that consists of two phases, a preprocessing phase and a query-processing phase, and additionally propose storage models for the evolving graph sequences that support the aforementioned framework.

In the preprocessing phase the initial snapshots of the sequence are grouped into smaller clusters of similar snapshots. This is performed by defining a graph similarity measure and by incrementally adding snapshots in a cluster (starting from the first snapshot in the sequence) until a graph similarity threshold has been surpassed. At that point, a new empty cluster is created and the above procedure is repeated until all the snapshots have been examined. For each cluster, two representative graphs $G_\cap$ and $G_\cup$ are extracted which are the largest common subgraph and the smallest common supergraph of all snapshots in the cluster respectively. For example, if we assume that G_1 and G_2 from Fig. 1 are grouped in the same cluster their respective $G_\cap$ and $G_\cup$ graphs correspond to the graphs in Fig. 2.

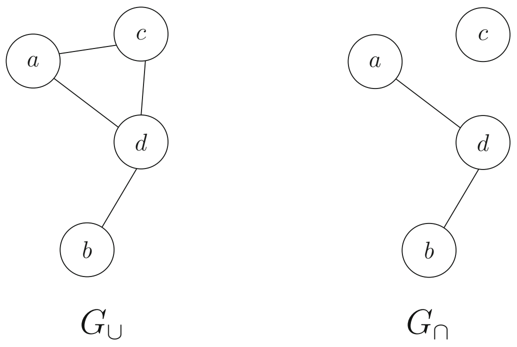

Fig. 2. $G_\cup$ and $G_\cap$ for the evolving graph sequence in Fig. 1

In the query-processing phase the authors use the clusters and their representative graphs to answer shortest path and closeness centrality queries. At first they evaluate the solution to a query for the representative graphs of the cluster ("FIND" step) on the basis that the solution will readily apply to a number of the snapshots in the cluster. In the "VERIFY" step, the evaluated solution is tested with each snapshot in the cluster in conjunction with a set of intuitive lemmas. For each snapshot that the evaluated solution does not apply, the framework attempts to "FIX" the solution so that it also applies to the aforementioned snapshot.

The authors also propose three storage models that can be used along with the FVF framework. The models make use of the similarities exhibited between successive snapshots and between representative graphs of successive clusters to reduce the total space cost of the evolving graph sequence. Finally, they assess their work through extensive experiments on both real and synthetic datasets.

3.2 Using Graph Deltas for Historical Queries

The authors in [10] advocate the use of graph deltas to support historical queries on evolving graph sequences. They begin by stating the operations that are supported on each snapshot, namely, $addNode(u_i)$, $remNode(u_i)$, $addEdge(u_i, u_j)$, $remEdge(u_i, u_j)$ which correspond to the addition or removal of a vertex u_i and the addition or removal of an edge between two vertices u_i and u_j respectively. Graph deltas are defined to be sets of such operations that when applied on a particular snapshot they yield another snapshot of the sequence. For example in Fig. 1, G_3 can be obtained by applying $\{addNode(e), addEdge(a, e)\}$ to G_2.

Furthermore, they define complete deltas to be sets of operations that when applied on the first snapshot of the sequence they are able to yield any of the sequence's snapshots.[1] Additionally, inverted deltas are defined to be sets of

[1] Certain snapshots require applying only a subset of the operations in a complete delta.

operations that when applied on a snapshot G_t they yield a snapshot $G_{t'}$ where $t' < t$, that is, $G_{t'}$ occurs "earlier" in the sequence than G_t.

Having defined the different types of deltas, the authors discuss snapshot materialization techniques and policies. More specifically, while any of the sequence's snapshots may be reconstructed if a complete and invertible delta and another one of the sequence's snapshots are maintained, it may be to the method's benefit to also maintain interposed snapshots to speed up snapshot materialization.

The next body of the work proposes three plans for efficient query processing. Perhaps the most universal of the proposed plans is a two-phase query plan that first materializes a particular snapshot according to the techniques discussed and then executes the query on the materialized snapshot. Finally, the authors discuss potential optimizations, delta indexing approaches and present some preliminary results of their solutions.

3.3 Compact Sequence Representations

Until this point the previous work we discussed was focused on reducing the total time cost of queries on evolving graph sequences. In the following works by Caro et al. [4] the authors address the problem of reducing the space cost when handling evolving graph sequences. Their proposed methods are heavily based on compact and self-indexed data structures that coupled with certain compression techniques (such as ETDC [3] and the PForDelta technique [22,23]) achieve overall high space efficiency with a good trade-off on the total time cost of the queries.

The authors use the concept of *contacts* as described by Nicosia et al. [16] to define temporal graphs.[2] A contact is defined to be a 4-tuple (u, v, t_s, t_e) that signifies the existence of an edge between vertices u and v during the time period $[t_s, t_e]$. The collection of all contacts is equivalent to the temporal graph itself, while, a particular snapshot G_t corresponds to the set of contacts (u, v, t_s, t_e) such that $t \in [t_s, t_e]$.

Next, operations that can be performed upon temporal graphs are presented. Those include:

- neighbor queries (i.e. report all neighbors of a vertex u),
- reverse neighbor queries (i.e. report all vertices that have a vertex u as neighbor),
- active edge queries (i.e. does there exist an edge between two vertices u and v at time instance t?),
- retrieving a snapshot of the graph at time instance t,
- edge state change queries (i.e. report all edges that have had their state changed at time instance t, that is all contacts that $t_s = t$ or $t_e = t$)

[2] Throughout the remainder of this work we will use the terms "evolving graph sequences" and "temporal graphs" interchangeably.

After a brief overview of the compression techniques and compact data structures they use in their work, the authors focus on the four temporal graph representations they propose along with their implementations that take advantage of the compression techniques. The first representation, called EdgeLog is an index that maintains for every vertex v in the temporal graph a list with the neighbors of v. Each neighbor of v is also equipped with a list containing all the time intervals that the particular edge exists in the sequence. The EdgeLog structure for a sequence composed by the graphs G_1 and G_2 of the example in Fig. 1, is depicted in Fig. 3.

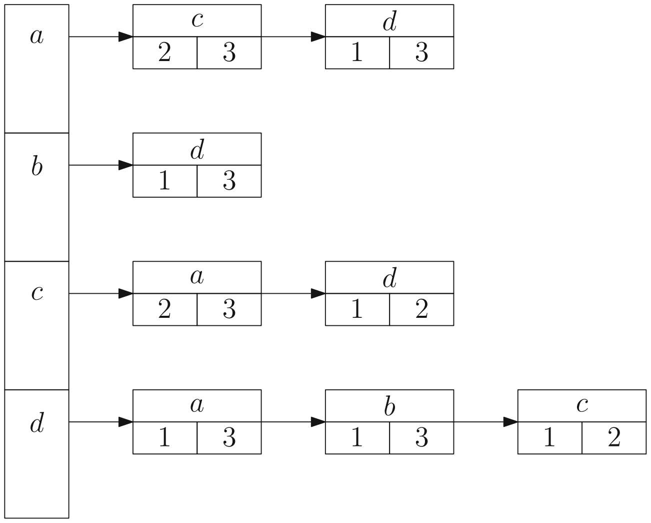

Fig. 3. EdgeLog and EveLog for the evolving graph sequence in Fig. 1

The second representation, called EveLog, follows a similar approach to the first. More specifically, EveLog is composed of a list with all the vertices that appear in the temporal graph. For each vertex v, there exists a list with all the "events" related to v (i.e. edge state change along with the vertex at the other end of the edge). The third representation is titled Compact Adjacency Sequence (CAS) and is based on the use of the Wavelet tree, while the fourth representation (CET) is based on the Interleaved Wavelet tree which is a data structure proposed in the same work as an additional asset to handling temporal graphs.

The work is concluded with extensive experimental evaluation over synthetic and real datasets through which the authors reach an interesting conclusion that there isn't a single best data structure for all the queries performed on temporal graphs. As a last remark, we should note that the above work focuses on the offline version of the problem, yet it also mentions alterations and modifications that need to be done in order for the solutions to apply to the online version.

3.4 Constructing Indices for Specific Queries

The work presented so far mostly focuses on efficiently storing, maintaining and retrieving the snapshots of an evolving graph sequences. There have been methods proposed in literature that instead aim to index the evolving graph sequence in a manner that permits the effective evaluation of specific queries. We present some notable examples in the section that follows.

Akiba et al. [1] describe dynamic indexing schemes that permit them to answer distance queries on either the last snapshot (current) or in any "older" snapshot in the sequence. Furthermore, they support the historical distance change-point query that reports all the time instances in the sequence where the distance between two vertices u and v changes. It is worth noting that in their work, they handle graphs that only support vertex additions and edge additions.

An other method that concentrates on answering shortest path queries was proposed by Huo et al. [8]. The authors make use of a Temporally Evolving Graph structure to store all the updates that occur in the sequence and proceed to use variations of Dijkstra's algorithm [5] to compute shortest paths. Furthermore, they speed up their solutions by making use of preprocessing indexes, namely, Contraction Hierarchies [6].

Yang et al. [21] propose an algorithm that discovers most frequently changing components in an evolving graph sequence. They begin by defining measures of change between vertices and the general problem of extracting the most frequently changing component and proceed to present their solutions.

Finally, Semertzidis et al. [18] tackle the problem of answering historical reachability queries. Their proposed index structure is called TimeReach and it is built in a manner that takes advantage of the strongly connected components that are present in a graph.

4 Parallel and Distributed Methods

In this section we turn our attention to methods and solutions that were proposed for parallel and distributed environments. The two systems that we will be analyzing are the DeltaGraph system by Khurana et al. [9] and the G* parallel graph database by Labouseur et al. [11,12,20].

4.1 The DeltaGraph System

Khurana et al. [9] designed and implemented a distributed system called Delta Graph that aims to efficiently store and retrieve snapshots from an evolving graph sequence. DeltaGraph supports time point (singlepoint) queries, time interval snapshot queries and multiple time point (multipoint) queries. Furthermore, along with the graph structure a query is also able to return the attributes of vertices and edges (e.g. name, weight etc.) The system is composed of two main components: the DeltaGraph index structure and the GraphPool in-memory data structure.

The DeltaGraph index is described as a rooted hierarchical graph structure that resembles a tree with adjacent leaves connected to each other in a bidirectional manner. The leaves of the structure correspond to snapshots of the sequence while the inner nodes correspond to graphs that can be obtained by applying a differential function (e.g. Intersection) to its children. The edges between the nodes store sets of deltas that are used to obtain a child node from its parent and they are horizontally partitioned between workers. It should be noted at this point that the only data stored are the sets of deltas and not the graphs themselves although the authors advocate the materialization of specific snapshots in DeltaGraph so as to speed up query time.

To answer a singlepoint query for a time instance t, the system locates through a binary search among the leaves the two adjacent leaves that "encompass" the query point t. Afterwards, it finds the minimum-weight path from the root to either of the two leaves, where the weight of an edge is set to be equal to the size of its respective delta. For multipoint queries, the system follows the same procedure with the difference being that instead of finding a path with minimum weight the system has to find the lowest-weight Steiner tree between the root and the multiple time instances.

The other component of the system is the GraphPool data structure which maintains in-memory a combination of materialized snapshots. More specifically, GraphPool maintains the current graph, historical snapshots and materialized graphs in a single combined graph. To determine which graphs contain a certain component or attribute the system makes uses of a mapping table. Finally, GraphPool is responsible for keeping the current graph index updated and cleaning up historical snapshots that are no longer needed.

4.2 The G* Graph Database

The last system we will be reviewing is the G* graph database by Labouseur et al. [11,20] that focuses on taking advantage of the commonalities that exist between snapshots in a sequence so that they are stored in an efficient manner.

In the G* system, each server is assigned a set of vertices along with all the outgoing edges of each vertex in the set. This achieves significant data locality since obtaining all of a vertex's edges can be accomplished without the need to contact any of the other servers. Furthermore, since the snapshots in a sequence exhibit similarities between them, G* avoids storing redundant information by only storing each version of a vertex once and, in that way, data that isn't modified between different snapshots isn't needlessly stored again.

Additionally, each server maintains an index named Compact Graph Index (CGI) that stores a single $(vertexID, disk\_location)$ pair for each vertex version that exists in a combination of the sequence's snapshots. For example, the CGI of a server maintaining vertex c of Fig. 1 would contain two pairs related to c: A pair for version c_1 in $\{G_1\}$ and another pair for version c_2 in $\{G_2, G_3\}$. It should be noted that the CGI has a low space overhead and can be mostly or fully kept in memory. As a last remark, the authors have proposed splitting the

Table 1. Summary of the works reviewed

Citation	Setting/Environment	Purpose/Approach
[17]	Centralized	Snapshot Storage & Retrieval, Shortest Paths, Closeness Centrality Queries
[10]	Centralized	Snapshot Storage & Retrieval, Two-Phase Query Plan
[4]	Centralized	Snapshot Storage & Retrieval, Compact and Self-Indexed Data Structures
[1]	Centralized	Historical Distance Queries
[8]	Centralized	Shortest Path Queries
[21]	Centralized	Discovery of Most Frequently Changing Components
[18]	Centralized	Historical Reachability Queries
[9]	Distributed	Snapshot Storage & Retrieval
[11]	Distributed	Snapshot Storage & Retrieval

CGI in a specific manner when a large number of graph combinations has been formed in its contents.

In a similar spirit to the storage module of G*, the CGI can also be used with regard to query processing to ensure that each version of vertex or edge is only processed once per query evaluation. Furthermore, the G* system supplies three types of primitives that can be used to construct graph query operators: summaries, combiners and bulk synchronous parallel (BSP) operators. Finally, in [12] the authors discuss snapshot replication and distribution techniques.

5 Conclusions

A significant fraction of contemporary networks can be modeled into graph data structures that are dynamic and constantly evolving. By integrating temporal information with typical graph queries we can obtain an improved understanding of a graph's overall nature. In this work we reviewed methods and systems proposed that aim to efficiently handle evolving graph sequences. A concise summary of the works presented can be seen on Table 1.

Acknowledgments. This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales. Investing in knowledge society through the European Social Fund."

References

1. Akiba, T., Iwata, Y., Yoshida, Y.: Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In: 23rd International World Wide Web Conference, WWW 2014, Seoul, Republic of Korea, 7–11 April 2014, pp. 237–248 (2014)
2. Brisaboa, N.R., Caro, D., Fariña, A., Rodríguez, M.A.: A compressed suffix-array strategy for temporal-graph indexing. In: Moura, E., Crochemore, M. (eds.) SPIRE 2014. LNCS, vol. 8799, pp. 77–88. Springer, Heidelberg (2014)
3. Brisaboa, N.R., Fariña, A., Navarro, G., Paramá, J.R.: Lightweight natural language text compression. Inf. Retr. **10**(1), 1–33 (2007)
4. Caro, D., Rodríguez, M.A., Brisaboa, N.R.: Data structures for temporal graphs based on compact sequence representations. Inf. Syst. **51**, 1–26 (2015)
5. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische mathematik **1**(1), 269–271 (1959)
6. Geisberger, R., Sanders, P., Schultes, D., Delling, D.: Contraction hierarchies: faster and simpler hierarchical routing in road networks. In: McGeoch, C.C. (ed.) WEA 2008. LNCS, vol. 5038, pp. 319–333. Springer, Heidelberg (2008)
7. Apache Giraph: http://giraph.apache.org/
8. Huo, W., Tsotras, V.J.: Efficient temporal shortest path queries on evolving social graphs. In: Conference on Scientific and Statistical Database Management, SSDBM 2014, Aalborg, Denmark, June 30–July 02, 2014, pp. 38:1–38:4 (2014)
9. Khurana, U., Deshpande, A.: Efficient snapshot retrieval over historical graph data. In: 29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, 8–12 April 2013, pp. 997–1008 (2013)
10. Koloniari, G., Souravlias, D., Pitoura, E.: On graph deltas for historical queries. In: WOSS (2012)
11. Labouseur, A.G., Birnbaum, J., Olsen, P.W., Spillane, S.R., Vijayan, J., Hwang, J., Han, W.: The G* graph database: efficiently managing large distributed dynamic graphs. Distrib. Parallel Databases **33**(4), 479–514 (2015)
12. Labouseur, A.G., Olsen, P.W., Hwang, J.: Scalable and robust management of dynamic graph data. In: Proceedings of the First International Workshop on Big Dynamic Distributed Data, Riva del Garda, Italy, 30 August 2013, pp. 43–48 (2013)
13. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection, June 2004. http://snap.stanford.edu/data
14. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, 6–10 June 2010, pp. 135–146 (2010)
15. Neo4j: http://neo4j.org/
16. Nicosia, V., Tang, J., Mascolo, C., Musolesi, M., Russo, G., Latora, V.: Graph metrics for temporal networks. In: Holme, P., Saramäki, J. (eds.) Temporal Networks, pp. 15–40. Springer, Heidelberg (2013)
17. Ren, C., Lo, E., Kao, B., Zhu, X., Cheng, R.: On querying historical evolving graph sequences. PVLDB **4**(11), 726–737 (2011)
18. Semertzidis, K., Pitoura, E., Lillis, K.: Timereach: historical reachability queries on evolving graphs. In: Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, 23–27 March 2015, pp. 121–132 (2015)

19. Shao, B., Wang, H., Li, Y.: Trinity: a distributed graph engine on a memory cloud. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, 22–27 June 2013, pp. 505–516 (2013)
20. Spillane, S.R., Birnbaum, J., Bokser, D., Kemp, D., Labouseur, A.G., Olsen, P.W., Vijayan, J., Hwang, J., Yoon, J.: A demonstration of the g_* graph database system. In: 29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, 8–12 April 2013, pp. 1356–1359 (2013)
21. Yang, Y., Yu, J.X., Gao, H., Pei, J., Li, J.: Mining most frequently changing component in evolving graphs. World Wide Web **17**(3), 351–376 (2014)
22. Zhang, J., Long, X., Suel, T.: Performance of compressed inverted list caching in search engines. In: Proceedings of the 17th International Conference on World Wide Web, WWW 2008, Beijing, China, 21–25 April 2008, pp. 387–396 (2008)
23. Zukowski, M., Héman, S., Nes, N., Boncz, P.A.: Super-scalar RAM-CPU cache compression. In: Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, Atlanta, GA, USA, 3–8 April 2006, p. 59 (2006)