A survey of computation techniques on time evolving graphs

Shalini Sharma

Institute of Information System and Applications,
National Tsing Hua University,
Hsinchu, Taiwan
Email: shalu24sharma@gmail.com

Jerry Chou*

Department of Computer Science,
National Tsing Hua University,
Hsinchu, Taiwan
Email: jchou@lsalab.cs.nthu.edu.tw
*Corresponding author

Abstract: Analysing time evolving graphs (TEG) in a timely fashion is critical in many application domains, such as social network. Many research efforts have been made with to address the challenges of volume and velocity from dealing with such datasets. However, it remains to be an active and challenged research topic. Therefore, in this survey, we summarise the state-of-art computation techniques for TEG. We collect these techniques from three different research communities: 1) the data mining for graph analysis; 2) the theory for graph algorithm; 3) the computation for graph computing framework. Based on our study, we also propose our own computing framework DASH for TEG. We have even performed some experiments by comparing DASH and graph processing system (GPS). We are optimistic that this paper will help many researchers to understand various dimensions of problems in TEG and continue developing the necessary techniques to resolve these problems more efficiently.

Keywords: big data; time evolving graphs; TEGs; computing framework; algorithm; data mining.

Reference to this paper should be made as follows: Sharma, S. and Chou, J. (2020) 'A survey of computation techniques on time evolving graphs', *Int. J. Big Data Intelligence*, Vol. 7, No. 1, pp.1–14.

Biographical notes: Shalini Sharma is working towards her PhD at Large-Scale System Architecture Lab (LSA Lab), National Tsing Hua University(NTHU), Taiwan. She is doing research on algorithms for time evolving graphs. Her research interest includes parallel and distributed computing, graph processing and incremental algorithms.

Jerry Chou is an Associate Professor at Computer Science Department, National Tsing Hua University(NTHU) since 2011. He received his PhD from Computer Science and Engineering department at UCSD in 2009. Before joining NTHU, he worked in the data management group in Lawrence Berkeley National Lab (LBNL). His research interests including high performance computing, cloud computing, data management, and distributed and parallel systems. His work has led to over 40 publications in international conferences and journals, and he has served as the reviewer and program committee member in several high impact journals, including TPDS, JDPC, etc.

1 Introduction

As defined in graph theory, graph can be described by a set of vertices and edges associated with attribute values on them. It is a commonly used and powerful data representation for describing data with relationships in various science domains, such as social network (Rapoport and Horvath, 1961), traffic network (Chen et al., 2009), web graph (Barlow, 2003), power grid (Watts and Strogatz, 1998) and biology (Jeong et al., 2001). Many research work have been done for graph data in the past but most of them were discussed with respect to a *static graph*, where the graph structure is given and fixed in the problem definition. Not until recently, researchers started to pay attention on the challenges of time evolving graph (*TEG*) where the graph structure can be changed dynamically with time by a set of update events, such as edge addition/deletion, vertex addition/deletion and attributes modification. Let us take the social network data from Facebook as an example. The data can be represented as a graph by mapping the users as vertices and their friendships as edges; it is a graph evolving over time. According to a study (Noyes, 2018), in every 60 seconds 510,000 comments are posted, 293,000 statuses are updated and 136,000 photos are uploaded. Hence many valuable information can only be obtained by analysing the graph data across time periods, such as tracking the trends of activities and changes in user behaviours.

A traditional approach to deal with TEG is to convert the data into a set of graph snapshots at each time interval and then apply the techniques for static graph on these individual graph snapshots. However as the *volume* of graph data and *velocity* of graph update events continue to grow at an unprecedented rate, to store and process these graph snapshots independently can result in significant amount of wastage in storage space and computation power. Therefore, many new challenges urgently needs attention from research community to develop a new set of algorithms, techniques and tools specifically for TEG. In this work, we give a thorough survey on some of these recent attempts with the aim to help researchers understand various dimensions of problems in TEG and resolve these problems more efficiently.

Processing TEG from user's perspective is about tracking the changes in the graph. These changes are tracked using fundamental algorithms and further these algorithms are implemented on computing frameworks to perform analytics on TEG. Hence the three different aspects of processing TEG are:

1 graph analytic

2 graph algorithms

3 graph frameworks.

Due to the different characteristics of these three aspects, there are different research communities focusing on them too. For example, the data mining community focuses on graph analytic, graph theory community focuses on graph algorithms and the distributed computing community focuses on frameworks. The approaches in each of these aspects are dependent on each other and form a processing stack from the lowest level of computations to the highest level of applications. We will further explain these three aspects by using an advertisement in social media as example.

1 *Graph analytic:* analysis of a graph is the key to business intelligence and provides current trends by analysing user's behaviour.To post an advertisement in social media, we need to find the most influential users in a graph. Such user's can be found by computing centrality of a network.

2 *Graph algorithm:* algorithm is a procedure to solve a problem. For example the centrality of a graph can be computed by calculating all pair shortest path. The user which is part of maximum number of shortest path would be the most influential.

3 *Graph frameworks:* framework is a platform that stores and updates the graph. For example, in order to compute all pair shortest path, we need to implement it on a graph framework through an API which handles the communication and graph data.

In this survey, we collect the recent research work on TEG from the above mentioned aspects. In graph analytics and graph algorithms, we will discuss several use cases and in graph frameworks, we will discuss various design issues for building frameworks. In graph analytic, we will discuss finding centralities, detecting communities, detecting rare categories and link prediction. In graph algorithms, we will discuss about shortest path, pattern matching, minimum spanning tree (MST) and path connectivity (PC). In graph frameworks, we will discuss the frameworks for TEG. Following our survey, we also propose our own computing framework architecture for TEG, called DASH and we briefly describe the designs and challenges to implement this computing framework. Moreover we have even shown initial results by comparing DASH with graph processing system (GPS). In the end, we have listed down four major challenges of TEG, i.e., *tracking change*, *computing change*, *locating change* and *storing change*. We also summarised all the approaches and researches discussed in this paper on the basis of these challenges.

The rest of the paper is organised as follows. First, we summarise the work of graph analytic, graph algorithms and graph processing frameworks in Sections 2, 3, and 4 respectively. Then we present our proposed system architecture for TEG in Section 5 followed by the survey discussions it is presented in Section 6. Finally, the paper is concluded in Section 7.

2 Graph analytic

Graph analytic is all about tracking the change in graphs. It studies the change and provides the knowledge of the current trends and characteristics of data. It plays major role in business intelligence to build applications, such as recommendation systems. In this paper, we focus on the data mining techniques for analysing graphs and discuss how these techniques are extended for the graphs that evolve over time. In particularly, we summarise the techniques for three graph analytic problems, including centrality, community detection and link prediction with respect to change. They are all fundamental problems for mining graph data, like social networks.

2.1 Centrality

Centrality is a problem to identify the most important or central elements of a network graph. Depending on the problem definitions, there are various kinds of centrality like degree centrality, closeness centrality, page-rank centrality, betweenness centrality, etc. While centrality is a well-known problem, only closeness centrality and betweenness centrality have been studied on TEG so far. Hence we discuss these approaches below.

- *Betweenness centrality:* betweenness centrality measures the importance of a node in a graph. It calculates shortest path between every pair of node and then determine betweenness centrality by calculating the number of shortest paths passing through every node. One of the most popular algorithm for calculating betweenness centrality is Brandes (2001) algorithm and it runs in $O(nm)$ time, where n and m are the number of edges and vertices. Betweenness centrality computes shortest path for every pair of node which itself is an expensive operation and hence there is a need of an improvised algorithm. Kourtellis et al. (2015) proposed a framework for computing vertex and edge betweenness centrality incrementally in large evolving graphs. It maintains the latest update of both vertex and edge betweenness centrality by assuring that there is no impact on the computational cost.
- *Closeness centrality:* closeness is a metric assigned to each node depending on the distance from other nodes. A node which is closer to many other nodes is called an influential node. In social media, it will be faster to broadcast a message from these influential nodes. Kas et al. (2013) demonstrate the design of an incremental closeness algorithm that supports efficient computations of all-pair shortest paths in a graph where edges are added and removed continuously. They even provide an insight to the need of incremental algorithms for efficient computation. The main idea is to respond to the changes over time in the analysed network by performing early pruning and propagating the updates only to the affected parts of the network. However there is a tradeoff between storing pre-computed and redundant information. It uses closeness centrality as an incremental metric. This metric is responsible for the speed with which closeness centrality can be calculated for all nodes in a network.

2.2 *Community detection*

Community is a structural property of real networks. It is a fully connected subgraph. Communities in social media could be a set of influencers, group leaders, mediators, followers, etc. The solution designed for this problem should be able to discover changes and quickly react by modifying the underlying community structure. We are going to discuss two problems related to communities, i.e., evolutionary clustering (EC) and incremental clustering (IC). EC algorithms try to determine new clustering results with minimum changes. IC is just an incremental algorithm to do clustering and focus only on the current results.

When we talk about community behaviour, it comprises of detecting communities and studying the change in communities' structure. Amelio and Pizzuti (2015) detects the change using population and sensor-based methods. They propose a framework which does three main tasks:

1. finding community structure in the underlying network
2. automatically discovering changes when they occur
3. quickly reacting to changes by applying specific strategies that properly handle diversity in the genetic population in order to achieve a new optimum.

The two strategies that are evaluated to react to the changes in order to obtain an optimum are hypermutation (HM) and random population initialisation (RI). In HM, the value of each gene of a chromosome is modified by randomly assigning to one of the neighbours and in RI, the overall population is reinitialised by using the new network.

After the extraction of communities from the social graph, the next step is to detect the change and analyse the evolution of the community. This evolution can be tracked by defining a 'change' and detecting it periodically. FICET (Liu et al., 2015) is a framework to detect and track community evolution incrementally, by using both current and historical data. The core communities are extracted in a core subgraph and expanded to the whole graph gradually. It does not need to give weights to the snapshots or temporal costs or the number of communities to find. Moreover it can even keep high quality of clustering simultaneously. They introduced four validation measures such as the normalised mutual information (NMI), the modularity value (MQ), the number of communities and the running time to compare FICET with other existing approaches.

Most of the traditional clustering algorithms focus on topological structure of the graph, which means that every partition will have similar topological structure. Inc-cluster (Zhou et al., 2010) ignores the topological structure and performs partitioning in such a way that each partition is similar in terms of a vertex's attributes. Inc-cluster is an improvement to SA-cluster. SA-cluster performs matrix multiplication to compute random walk distances between graph vertices. Every time edges are updated, random walk distances are updated by repeatedly doing matrix multiplication. Inc-cluster proposes an algorithm which incrementally updates random walk distances. They performed experiments and compared the clustering quality and speedup with other clustering algorithms and achieved significant speedup.

2.3 *Link prediction*

Link prediction is used to predict future links in advance or to predict missing links in an incomplete or corrupted data. In social media, link prediction could be used to recommend new friends that are connected to your existing friends. It aims to estimate the likelihood of existence of edges by using the current structure of the graph. Existing link prediction methods fail to generalise the graph stream settings because the graph snapshots where link prediction is performed, are no longer readily available in memory or even on disks for effective graph computation and analysis. In social networks, the interaction between participants including friend requests, document/photo sharing and message exchanges are fast, transient, and are of very large scale. It is desirable to support online link prediction in social graph streams for user/commodity recommendation, market analysis and business intelligence. Zhao et al. (2016) formalised the streaming link prediction problem in graph streams and identified a series of fundamental and neighbourhood-based link prediction measures. It includes Jaccard coefficient, common neighbour and Adamic-Adar as target measures for streaming link prediction. They even designed effective graph sketches for accurate estimation in graph streams. The broader idea described here is the identification of structure in different target measures and encode such salient, structure-aware correlations into different space-efficient graph sketches (constant space complexity per vertex) for streaming link prediction. Cost effective graph sketches are also designed based on Min-Hash and vertex biased sampling. Specifically, they design Min-Hash-based graph sketches to estimate the Jaccard coefficient measure and vertex-biased reservoir sampling-based graph sketches to estimate the common neighbour and Adamic-Adar measures.

2.4 Rare category detection

Rare category detection is used to solve many problems like synthetic ID detection, insider threat detection, fraud detection, etc. In suck kind of problems, only a small percentage is of interest and it is known as minority class. The traditional approach to identify such classes is random sampling. However in case of TEG, those minority classes change their behaviour intentionally due to fear of getting caught. Zhou et al. (2015) addresses this problem by proposing two incremental algorithms to detect initial examples of minority classes. Their main idea is to update the detection model by local changes instead of reconstructing at every update. They also propose another algorithm with softer upper bounds.

3 Graph algorithms

Algorithms are simply a set of rules for computation. Before computing, we should know where to compute and how to compute. *Where* refers to the portion of the graph that needs re-computation due to graph update events. *How* refers to the techniques of algorithms that are required to reduce the amount of computation. Graph algorithms plays a major role in processing of TEG by *locating* the change and *computing* the change.

3.1 Pattern matching

Graph pattern matching is finding a subgraph in a large graph that is similar to a given graph. It is widely used in social network analysis. Pattern matching focuses on the techniques to compute, i.e., how to compute. Pattern matching can be based on various models, i.e., graph simulation, bounded simulation and subgraph isomorphism. The two research works that we are going to mention are the unboundedness of incremental pattern matching on all the above models (Fan et al., 2011) and incremental pattern matching based on graph simulation in a distributed environment (Kao and Chou, 2016).

Fan et al. (2011) proposed incremental algorithms for various models of pattern matching. It investigated bounded simulation, subgraph isomorphism, and graph simulation. This is the first work to study the boundedness of pattern matching in terms of $O|CHANGED|$, where CHANGED is addition/deletion of nodes/edges. It shows that all the algorithms are unbounded. However, it is bounded for special cases like unit deletion and insertion.

The authors in Kao and Chou (2016) further proposed a distributed incremental algorithm for graph simulation pattern matching model. The basic design of distributed algorithm is similar to the sequential algorithm. The re-computation operations are only triggered from the vertices where graph update events occur. If a vertex is added or removed from the matching set, the status of the neighbours needs to be re-evaluated as well. Different from the sequential algorithm that uses a stack to traverse the vertices requiring re-evaluation, the distributed algorithm needs to propagate messages to the vertices that need re-evaluation. Only the distributed algorithm for graph simulation model was proposed because it is the most relaxed matching model which allows vertices to determine their matching status based on local information. Other stricter matching models may require global information too, and thus cannot be scaled effectively in distributed computing environment. Overall, all these incremental pattern matching algorithms showed significant improvements on computing time over traditional batch computations, especially when the percentage of changes to the graph is small.

3.2 Page rank

Page rank depends on the computations of previous results or messages from the previous iteration. It only considers location of the computation therefore it focuses on ‘where to compute’. The main task of the algorithm is where to detect the change in the graph when the graph is not notified about the changes. We are going to discuss two algorithms in this section and the main idea is to avoid complete crawling of web graph and find a way in which page rank computed at any point of time is nearly optimal. The major task is finding the changed portion of a graph as it saves a lot of computational cost. Desikan et al. (2005) proposed an algorithm which can find the changed portion of the graph by partitioning the graph in such a way that the changed and unchanged graphs remain in the separate partition. The state of the vertex is only affected by the incoming edge, so it can determine the changed part. In the end only the changed part needs to be recomputed. Bahmani et al. (2012) used probing approach to find the changed portion of the graph. The algorithm decides which part of the graph should be probed to compute page rank. It is based on the idea of crawling a smaller portion of graph and approximating the page rank computation. They proposed two algorithms: randomised and deterministic. The randomised algorithm works on the basis of intuition that nodes with higher page rank needs more probing. The deterministic algorithm uses priority probing.

3.3 PC and MST

PC and MST are well known graph computation problems. It also focuses on 'where to compute'. Web search engine can be a good example for such class of algorithms where search engine crawls the web periodically and updates the graph of web links without getting notified about the changes. The challenge of this algorithm is that it can only probe a small portion of graph at one time. The approach used in both MST and PC is based on selection-sorting method. The vertex and edges need to be sorted in some order (ascending/descending) depending upon the problem. After sorting, the required edges or nodes are selected as a resultant graph.

Unlike common algorithms that gets notifications from the graph whenever it changes, the algorithms proposed by Anagnostopoulos et al. (2012) does not notify. The objective of this algorithm is to obtain the resultant path in case of PC, and spanning tree in case of MST. There is a need of a probing algorithm which selects node or edges, re-computes them and validates the result. If the result produced is not appropriate, then continue probing. The change in the graph is limited to edge-swapping.

The purpose of PC is to find a Eular circuit in a graph where every node is visited exactly once and there is a distinct path between each pair of node. They proposed two algorithms, i.e., one path (maintains one path throughout the execution) and two path (maintains two paths simultaneously). A MST is one which costs the least among all spanning trees. In MST, general sorting is performed to keep the approximation of ordering of all the edges. At each time step, Kruskal's greedy strategy is used to maintain MST. The results of MST are even applied to Matroids to find maximal independent set. The authors in this paper focused on the 'how' aspect. They developed algorithms that are directly related to computation.

3.4 Summary

Table 1 summarises all the algorithms discussed previously on the basis of four parameters:

1 boundedness
2 changes allowed
3 method used
4 result types.

Fan et al. (2017) emphasised the importance of boundedness for the feasibility of TEG algorithms. An algorithm is locally bounded if the computation is based on a set of neighbouring nodes in the changed graph instead of the original graph. An algorithm is relatively bounded if the computation is based on the changed graph over a set of time intervals in order to avoid re-computation. Unit changes are allowed in a TEG and they are insertions, deletions and edge swapping. As discussed earlier, the methods used by these algorithms are incremental, partitioning, probing and selection sorting. The results of these algorithms are optimal or approximate in nature.

Table 1 Graph algorithms summary

Problem	*Algorithm*	*Boundedness*	*Changes allowed*	*Method*	*Result type*
Pattern matching	Fan et al. (2011)	Unbounded (GS, BS, ISO)	Insertions, deletions	Incremental	Optimal
Pattern matching	Kao and Chou (2016)	Unbounded (GS)	Insertions, deletions	Incremental, distributed	Optimal
Pagerank	Desikan et al. (2005)	Locally-bounded	Insertions, deletions	Partitioning, incremental	Optimal
Pagerank	Bahmani et al. (2012)	Locally-bounded	Insertions, deletions	Priority probing, intuition probing	Approx.
Path connectivity	Anagnostopoulos et al. (2012)	Relatively-bounded	Edge-swapping	Selection sorting, probing	Approx.
MST	Anagnostopoulos et al. (2012)	Relatively-bounded	Edge-swapping	Selection sorting, probing	Approx.

Table 2 Graph computing framework summary

Framework	*Platform*	*Programming model*	*Communication model*	*Data mgmt. strategy*
Kyrola et al. (2012)	Single node	Vertex-centric	Asynchronous	Parallel sliding window
Chen et al. (2012)	Heterogeneous	Vertex-centric	Pull	Partitioning (bandwidth)
Malewicz et al. (2010)	Distributed	Vertex-centric	Synchronous	Partitioning (vertex cut)
Roy et al. (2013)	Single node	Edge-centric	Synchronous	Partitioning (edge cut)
Yuan et al. (2014)	Single node	Path-centric	Synchronous	Partitioning (paths)
Simmhan et al. (2013)	Distributed	Subgraph-centric	Synchronous	Partitioning (GoFS)
Zhang et al. (2014)	Distributed	Vertex-centric	Asynchronous pull	Partitioning
Gonzalez et al. (2012)	Distributed	Vertex-centric	Asynchronous synchronous	Partitioning (edge cut)
Filippidou and Kotidis (2015)	TEG	Vertex-centric	Asynchronous	Partitioning (edge cut)
Abdolrashidi and Ramaswamy (2016)	TEG	Vertex-centric	Asynchronous	Partitioning (incremental)
Tsourakakis et al. (2014)	Distributed	Vertex-centric	Asynchronous	Partitioning (edge cut)
Ediger et al. (2012)	Shared memory	Vertex-centric	Asynchronous	Snapshots (asynchronous)
Cheng et al. (2012)	Distributed	Vertex-centric	Asynchronous pull	Snapshots (asynchronous)
Han et al. (2014)	Distributed	Vertex-centric	Asynchronous	Snapshots (periodic)
Shi et al. (2018a)	GPU	Vertex-centric	Asynchronous	Partitioning (colouring-based)

4 Graph frameworks

Computing framework is a platform in which we can efficiently perform various operations in graphs. Building a framework is a very difficult task for TEG because it needs to store and update the changes in the graph. To facilitate computations on TEG, there are various frameworks which are proposed, and some are implemented too. Table 2 summarises all the frameworks discussed in this section on the basis of the design issues.

4.1 General graph computing techniques

Graph computation is a vast field for discussion. In this section, we have categorised the frameworks based on their design principles, i.e.,

1 platform

2 programming model

3 communication model

4 data management strategy.

4.1.1 Platform

Large graph cannot be stored on disk, so we have four different platforms: single node system, distributed system, heterogeneous system, and graphics processing unit (GPU).

- *Single node:* single node systems take advantage of shared memory that allows efficient inter-process communication, as multiple tasks have access to the same memory. It reduces the communication cost by this process. However the disadvantage is there is limited memory in single node systems. Graphchi (Kyrola et al., 2012) was the first framework to use disk-based system for computing large graphs. Graphchi (Kyrola et al., 2012) tells us how to manage a large graph on a disk and deal with graph mutation.
- *Distributed system:* when the graph is too large and cannot be stored on a single machine, we use distributed systems. A distributed system consists of multiple processing units where each unit has private memory. The major challenges in distributed environment are communication cost and barrier synchronisation. Malewicz et al. (2010) started exploiting the distributed architectures of commodity clusters to enable efficient processing of large volumes of data.

- *Heterogeneous systems:* hardware and network topology does not have to be uniform in a cluster. Some machines may have a newer generation of hardware or some machines may be better connected than others. A heterogeneous environment refers to an environment where not every processing unit is equally powerful. Surfer (Chen et al., 2012) recognises heterogeneity challenges for cloud computing and tries to partition the graph based on available bandwidth between machines.
- *GPU:* because GPU provides massive degree of parallelism and high processor bandwidth, they can be used to process large graphs with billions of edges and nodes. Shi et al. (2018b) did a survey on key issues of graph processing on GPUs. Frog by Shi et al. (2018a) proposes a hybrid colouring model to partition the graph. They overlap the transferring time of data with execution of kernel function in order to improve the system performance.

4.1.2 Programming model

Most of the programming models are vertex-centric. The other programming models are proposed to optimise the performance and to minimise the communication overhead. Pregel (Malewicz et al., 2010) is a vertex-centric model where kernel runs on each vertex in parallel. X-stream (Roy et al., 2013) is an edge centric graph processing model on a single machine and addresses the disk access problem by serialising it. Path centric approach (Yuan et al., 2014) aims to improve convergence speed by traversing and partitioning the graph on the basis of paths. GoFFish (Simmhan et al., 2013) tries to minimise the communication between vertices by introducing subgraph level programming model. Vertices in a subgraph can share information. The information can be shared directly between the subgraph.

4.1.3 Communication model

- *Synchronous/asynchronous:* synchronous execution (Simmhan et al., 2013) of a graph algorithm can be depicted as a sequence of iterations, delimited by a global barrier. Each iteration performs updates based on values from the last iteration (in parallel). Updated values are only exchanged between iterations. Asynchronous execution (Zhang et al., 2014) lets updates be performed on the most recent data. Synchronisation is performed as soon as possible, rather than through a global barrier, resulting in an irregular communication interval. Frameworks that make a distinction between local and remote vertices can benefit from local asynchronous computation while synchronisation of remote values is still performed in synchronous iterations.
- *Push/pull:* in a push style flow, the information flows in a forward direction. As soon as an update occurs, message is sent to all neighbours. In a pull style flow, information flows in the reverse direction, and the active vertex updates their shortest path length by proactively reading the lengths of their neighbours' paths. Message-based communication naturally map to a push-based communication flow, while Shared Memory maps to a pull-based flow.

4.1.4 Data management strategy

Graph partitioning is a very old problem which has been studied for decades. The usual applications of graph partitioning include divide and conquer algorithms, parallel computing, de-clustering algorithms and many more. When the size of graph is large, then graph partitioning can be used for web searches, locating hot spots, trace targets, etc. All the existing graph partitioning algorithms are appropriate for large graphs; however they cannot be helpful in case of TEG.

- *Edge cut/vertex cut:* the partitioning can be performed either on edge cut or vertex cut. An edge-cut evenly assigns vertices to partitions with a minimal number of crossing edges. Computations need to be expressed on edge-level to allow for efficient parallel computation. A vertex-cut evenly assigns edges to partitions with a minimal number of crossing vertices. For many real-world graphs, where degree distribution follows a power law, a vertex-cut leads to a more balanced partitioning. Edge cut (Gonzalez et al., 2012) is more efficient and maintains data consistency where as in vertex cut the state of a vertex keeps on changing.

4.2 TEG computation technique

Wickramaarachchi et al. (2014) discussed various challenges and issues involved in processing TEG. The purpose of such framework is to help the users in implementing graph algorithms efficiently. Gao et al. (2015) discussed five major issues that should be considered for designing frameworks:

1 graph distribution

2 on-disk graph organisation

3 programming model

4 message model

5 synchronisation policies.

There are many frameworks for parallel processing of large scale graphs and they are widely discussed in Doekemeijer and Varbanescu (2014) but very few have been proposed for TEG so far. Hence, we are discussing these challenges related to TEG in the following sections.

Since the graph is changing, the two major challenges are:

1 graph repartitioning
2 asynchronous snapshots.

Every time the graph changes, it needs to be repartitioned therefore we need efficient graph repartitioning algorithms. Another challenge is to track the changes in the graph, hence taking snapshots is very important. The cost of taking snapshots is considerable high.

4.2.1 Graph repartitioning

The purpose of partitioning is to split the data across partitions in such a way that most of the computations can be performed locally. An efficient partition has minimum number of edges across partitions. In case of TEG, the nodes and edges are streamed in an arbitrary order.

Filippidou and Kotidis (2015) proposed a full TEG partition scenario where changes can be processed simultaneously by the system. There is also an online node placement heuristic technique that takes decision of placing incoming nodes on the fly. All the statistics required can be evaluated from the summary created by compact summary structure from compressed spanning trees (CST). CST helps in performing partitioning on demand, whenever required. The way of partitioning and placement of graph data on computing nodes has a significant impact on the performance of the cluster.

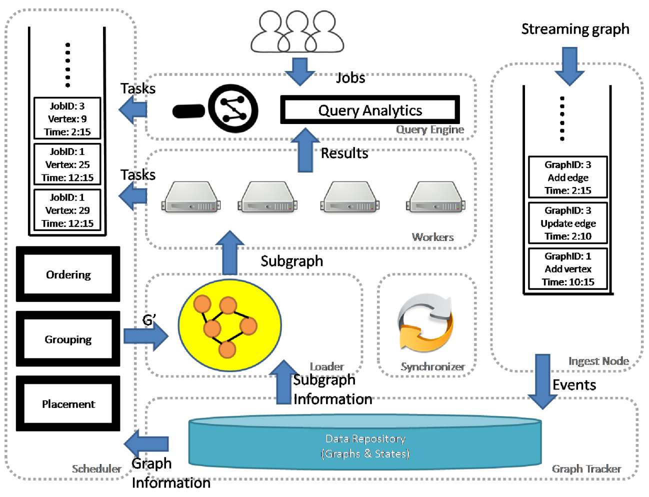
Abdolrashidi and Ramaswamy (2016) proposed a cost sensitive approach for partitioning the dynamic graphs. It incorporates multiple performance factors such as communication cost, number of intra-node edges and load distribution among computing nodes. The incremental algorithms are also proposed which uses the cost heuristics to handle graph modification events.

Fennel (Tsourakakis et al., 2014) tries to formulate a graph partitioning objective function which consists of two elements: cost of edge cut and cost of sizes of individual clusters. There could be three types of streaming orders: random, BFS and DFS. This paper proposes a k-partitioning algorithm which is responsible for the decision regarding placement of new vertices, however the location of the vertex cannot be changed again. One-pass streaming algorithm is also proposed by using a greedy strategy. It assigns each arriving vertex to a partition such that the objective function of the k graph partitioning problem is maximised.

4.2.2 Asynchronous snapshots

We need a special data structure that enables to manage such huge graphs and analyse the changes in the graph. It should support fast insertions, fast deletions and fast updates on graph. Stinger (Ediger et al., 2012) prevents locking mechanism but it does not support distributed systems. It maintains snapshots in a shared memory system. It generates snapshots after fast update in the memory.

Figure 1 Architecture (see online version for colours)


Streaming graph
Jobs
Tasks
Query Analytics
Query Engine
Results
Tasks
Workers
JobID: 3 Vertex: 9 Time: 2:15
JobID: 1 Vertex: 25 Time: 12:15
JobID: 1 Vertex: 29 Time: 12:15
GraphID: 3 Add edge Time: 2:15
GraphID: 3 Update edge Time: 2:10
GraphID: 1 Add vertex Time: 10:15
Subgraph
Ordering
Grouping
Placement
G'
Loader
Synchronizer
Ingest Node
Subgraph Information
Events
Data Repository (Graphs & States)
Scheduler
Graph Information
Graph Tracker

A kineograph (Cheng et al., 2012) is a distributed system that can process continuously changing graphs. It separates graph processing and graph updates. All the updates are transformed into sequence of transactions. Kineograph handles parallelism, consistency by guaranteeing atomicity, fault tolerance by handling ingest node and graph node failure, and periodically returns updated results. It supports static graph algorithms by operating over a snapshot, distributed across nodes. It consists of an efficient graph engine which supports incremental computations.

Chronos (Han et al., 2014) is specifically optimised to run in-memory iterative computations. It maintains snapshots periodically and construct it on the fly. This paper focuses on two things: locality and scheduling. Locality can be divided into two types: time-locality which can be created as time progresses linearly and structure-locality which can only be an approximate, as it is challenging to project a graph structure into a linear space. Chronos favors time-locality when there are multiple snapshots in memory. Chronos schedules graph computations to leverage the time locality. Temporal graph engines calculate the results across snapshots rather than around each vertex. Chronos is responsible for all the decisions regarding batch operations associated with each vertex across multiple snapshots.

5 DASH: proposed framework

To address the problem of processing time-evolving graph, we propose a system DASH with new system architecture which is distributed, asynchronous and dynamic. The goal of DASH are as follows:

1 manage graph data dynamically instead of pre-partitioning, in order to avoid long graph pre-partition time
2 schedule the task at runtime with workload aware property to achieve better load balance

3 use asynchronous and dynamic computing model with incremental algorithm to speed up convergence of computation

4 load only the necessary data to compute node in order to reduce memory usage.

In this section, we have explained the architecture of DASH followed by experimental evaluation.

5.1 Architecture

After our survey on various frameworks in the earlier sections, we propose a framework that is required for TEG. This framework consists of a dynamic execution model in which jobs are defined as an incremental algorithm. In this section we have explained the framework in detail. Before explaining each component in detail, we will define: execution model, jobs, tasks and graph state. The architecture is shown in Figure 1.

1 An *execution model* refers to the execution of components of a framework, i.e., jobs and tasks.

2 A *job* is an incremental algorithm which runs on the query engine and consists of several tasks. The job will be executed dynamically by breaking it into tasks. This framework uses vertex driven approach where components interact with each other via message passing.

3 A *task* is represented as a message; it is a single computation of the job. A task can be defined as task $\{jobID, vertexID, timestamp, execinf\}$; where jobID is an unique identifier for a job, vertexID is an unique identifier for every vertex in the graph, timestamp is the time at which the job starts and execinf contains all the parameters required for the execution of job. Tasks are independent and can be scheduled and run on any node at run time. Thus it removes the synchronisation barrier.

4 An *event* can be addition or removal of an edge or a vertex, i.e., event $\{timestamp, graphID, action\}$.

5 *Graph state* represents the information with respect to job execution, i.e., how does the graph look like at the time of computation. In our framework, graph state is maintained by graph tracker and it provides the state information to the loader as well as the scheduler.

To avoid an overhead of maintaining and storing snapshots, we propose a dynamic way for tracking the graph. We reconstruct the graph dynamically during the computation. This might seem to be a delayed operation but we believe it can overlap this time with the computation time. DASH is basically an event driven framework which follows publish/subscribe model. Whenever a user tries to query the graph, those queries should be converted into algorithms which can be executed incrementally. The whole framework is designed in such a way that it supports incremental algorithms. We explained above that the graph state is stored temporarily at the time of computation, which is even a way to implement incremental approach. All the components with their functionality are explained below.

5.1.1 Ingest nodes

When input data arrives in the system, it consists of several events, i.e., add edge E or remove edge E. The purpose of having ingested nodes is to connect the source of data to our framework. The action occurs either on vertices or edges, such as:

1 an edge consists of a source and destination vertex, *edge (srcID, desID)*

2 a vertex consists of the name of the vertex, *vertex (name)*.

The two main functions of ingest nodes are:

1 accepting the incoming data and converting it into graph events, i.e., graph-update operations

2 sending graph-update events to the graph tracker in a distributed manner.

5.1.2 Graph tracker

The graph tracker has three roles:

1 it is responsible for storing the events sent by ingest nodes

2 it also provides graph information to the scheduler

3 it sends the subgraph which is a set of vertices to the loader.

An important challenge for graph tracker is to store and keep record of events. There could be many possible ways to do so, for example we can store it in the memory structure. In this case, we need to solve the synchronisation issue as addressed by Stinger (Ediger et al., 2012) or we can store it as an update event file. We need to make sure that it loads efficiently from the disk as stated by xstream (Roy et al., 2013). The last option could be to store it in the graph database which generally does not scale well.

5.1.3 Query engine

Before explaining the role of query engine, the term Task should be made clear. There are two types of tasks in our framework, it can be either a new task or a currently running task. Query engine is an interface between user and framework. It receives queries from user and converts them into set of jobs. One job consists of many tasks. To convert user queries into jobs, there is a requirement of a different query language. Our query engine follows publish subscribe model as described in Cheng et al. (2012). This is a pull-based mechanism in which user can subscribe to receive the results. The basic two functions of query engine are:

1. it converts the jobs submitted by user to a set of computation tasks and sends them to the job scheduler
2. after computation has successfully finished, it collects all the results and publish them to the subscribed users.

5.1.4 Scheduler

Scheduler in our system plays an important role, and we employ a scheduling algorithm that optimises the performance of our system. First, our scheduling algorithm is data locality aware. It maintains and updates a mapping table between data and worker. In the process of decision making about which of the worker to assign the task, it will first check if the data has been loaded into any of the worker. The worker who already has the graph data gets higher priority of being assigned to the task because of the reduction of network traffic from the scheduler to the graph tracker, and the graph tracker to the worker.

Second, our scheduling algorithm ensures that there is load balancing among workers by dynamic task scheduling. The performance of load balancing is obvious in parallel program and distributed system. The performance of the system may have huge degradation when load is unbalanced since every compute node, process or thread have to wait until all the tasks finishes their computation. As a result, unbalanced load leads to long waiting time and low performance.

Algorithm 1 Scheduling algorithm

```
Input: Vertex $v$ of task, list with all the workers $workerList$
Output: Worker ID $w$ for vertex $v$
 1: $workerDataLocalityList = []$
 2: for each $w \in workerList$ do
 3:     /* Check which worker has data of vertex $v$ */
 4:     if $hasVertexData(w, v)$ is true then
 5:         $workerDataLocalityList.append(w)$
 6:     end if
 7: end for
 8: if $workerDataLocalityList.length()$ is 0 then
 9:     /* Assign to worker who has least load */
10:     return $getLeastLoad(workerList)$
11: else
12:     /* Assign to worker who has data locality and least load */
13:     return $getLeastLoad(workerDataLocalityList)$
14: end if
```

While it is hard to maintain load balancing when computing on TEG in systems with pre-partition, DASH can perform load balancing no matter which graph partition the task needs due to our dynamic scheduling algorithm. If there is no data locality for the task or there are multiple workers that have data, scheduler will assign the task to the worker with least load by observing the queue length of each worker. By assigning tasks in run time, DASH can have good load balance.

Algorithm 1 is the algorithm of our scheduler. Given the vertex ID of a task, the scheduler returns the ID of the worker where the task will be executed on. The scheduler first tries to select the worker with the least load from the workers with data. If there is no worker that has data, the scheduler selects any worker with the least load. This is implemented by constructing a $workerDataLocalityList$ in line 2–7, and selecting the least loaded worker from line 8–13. This scheduling algorithm is simple yet effective, it not only avoids complicated computation that might slower down the scheduling process but also achieves the goal of good data locality and load balance.

5.1.5 Loader

The task of loader is to load the requested graph into the memory of compute node on the basis of the subgraph, i.e., set of vertices that are provided by graph tracker. It introduces more vertices to connect the disconnected components. It receives subgraph from graph tracker which is actually a data structure containing the set of vertices. It also receives graph information required for scheduling, i.e., set G' which consists of the grouping components of scheduler.

5.1.6 Worker

This is the execution unit of our framework. The responsibilities of workers are as follows:

1. the workers consist of a local execution queue, where the data waits until they are ready for execution and CPUs to be available
2. to maximise the CPU execution, the ordering of tasks depends on the ready data
3. ut either generates new task or send them to scheduler or sends results to query engine
4. the task migration among nodes which is known as work stealing.

5.1.7 Synchroniser

As the name suggests, synchroniser is responsible to ensure the integrity of data in a framework. In DASH we need synchroniser because we allow jobs to overlap. It does the following tasks:

1. it handles vertex duplication among subgraph during loading and maintaining consistency of data
2. it stores the graph states and flushes out all the results.

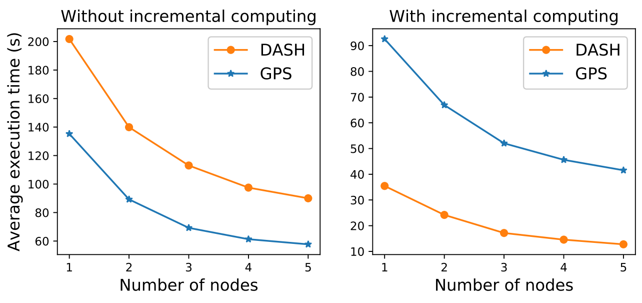
5.2 Experimental evaluation

We evaluate DASH on the cluster with six nodes, each node containing one AMD Opteron 6282 SE CPU (16 cores and 2.6 GHz). We use three representative graph algorithms as benchmark on three graphs. In order to evaluate performance on power-law graph, we generate RMAT graph with parameter a = 0.45, b = 0.22, c = 0.22. The number of vertices are 640,000 and edges are 6400,000. We also compared our system with state-of-the-art distributed system, GPS by Salihoglu and Widom (2013). For each experiment, we ran three times and report the average in our plots. We choose single source shortest path to evaluate our system. Single-source shortest path (SSSP) is an algorithm that calculates the shortest distance from a given source vertex to all other vertices. For SSSP, we use the update events of edge weight. First, we randomly select a ratio of edges in the graph. Second, for each edge weight w, we assign a new weight random from 1 to w. Third, we construct the update events file containing the task of destination vertex of selected edges and updated weight.

5.2.1 *Scalability*

First we compare the scalability of DASH with GPS, which is critical to the performance of distributed GPS. As plotted in Figure 2, it shows that for running algorithms without incremental computing, our system does not perform as well as GPS due to the overhead of dynamic graph loading. However, for running algorithms with incremental computing, our system can take advantage of incremental computing and minimise overhead of dynamic graph loading, hence has better performance than GPS.

Figure 2 Scalability of DASH and GPS with and without incremental computing (see online version for colours)

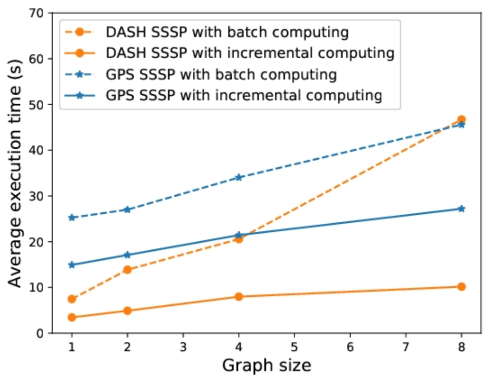


5.2.2 *Overall performance*

Next we compare execution time of DASH and GPS with different problem size running SSSP. Graph size 1 in Figures 3 and 4 is the same size as RMAT graph , other graphs which are 2×, 4× and 8× of this graph are also generated by RMAT algorithm.

As we can see in Figure 3, DASH with incremental computing can have short execution time and increase stably as graph size increases. The good performance is due to our dynamic and asynchronous system model that has balanced load among workers and helps computation converge faster. The reason why our system without incremental computing has poor performance on large graph is due to the overhead of large amount of graph data loaded at runtime, that allows the communication time dominating the execution time.

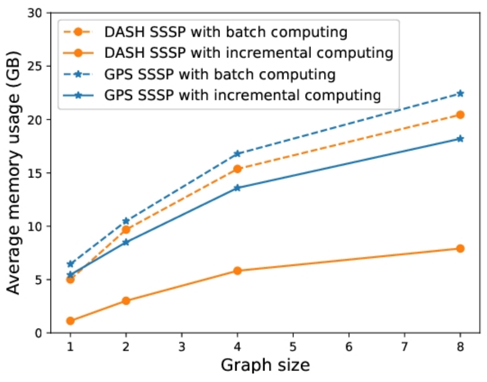
Figure 3 Average execution time with different problem size (see online version for colours)



5.2.3 *Memory usage*

In-memory GPS that uses less memory is capable of processing larger graph. Here we evaluate memory usage for each system with and without incremental computing. In Figure 4, DASH with incremental computing only uses 35% of memory compared to GPS. This is because graph data changed in update events is relatively smaller in the whole graph, causing a memory waste if the system loads the whole partition of graph into local memory. With incremental computing and dynamic graph loading, DASH only loads necessary data into worker nodes and has lower memory usage.

Figure 4 Average memory usage with different problem size (see online version for colours)



5.2.4 *Heterogeneous environment*

To evaluate system running in heterogeneous environment, we fixed total number of CPU cores of all the workers in the system and make number of CPU cores on single worker different from each other. Figure 5 shows DASH can utilise all the heterogeneous resources well and outperforms GPS for more than 8x speedup. This is because the dynamic and asynchronous design of our system, which can assign more tasks to worker nodes with more computing power. On the other hands, static and synchronised system such as GPS cannot schedule the tasks flexibly and must wait until all the worker nodes finish their tasks to move on to the next *superstep*.

Figure 5 Comparison of system running on homogeneous and heterogeneous environment (see online version for colours)

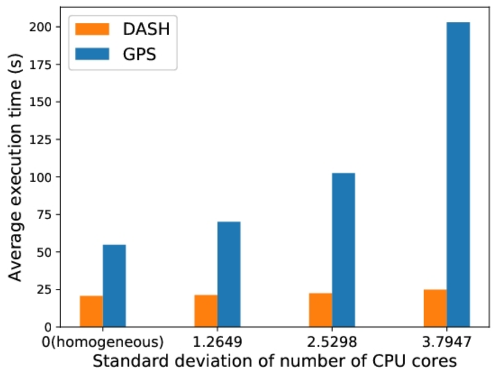
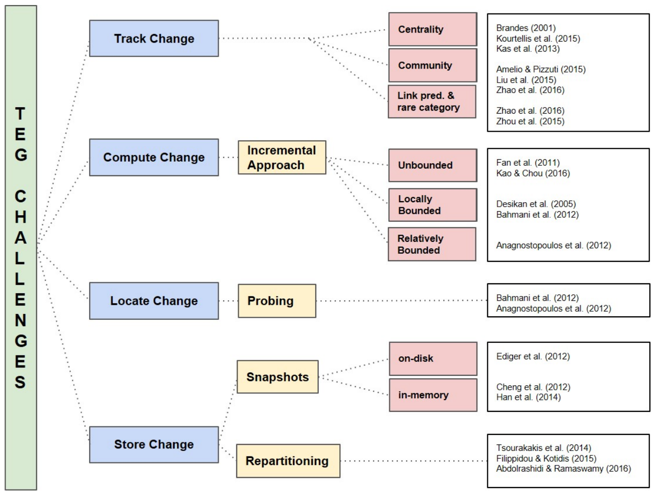


Figure 6 TEG challenges (see online version for colours)



6 Survey discussion

In order to cover the challenges of TEG, we surveyed many papers. The problems in processing TEG could be interpreted differently for different research communities and can be categorised as below and have been summarised in Figure 6.

1 *track change:* to know if the graph has changed or not and to analyse the change over a period of time; for example how often the centrality of a graph changes

2 *compute change:* since the graph has changed, the algorithms should be computed again with minimum re-computation

3 *locating change:* to find the location of the change from where the computation can start

4 *storing change:* since the graph is changing very fast, these changes should be stored as soon as they occur.

7 Conclusions

In this paper, we conducted a survey on various dimensions of TEG like graph analytic, graph frameworks and graph algorithms. This survey provides an insight at various challenges and techniques involved in analysis of TEG. We concluded that we need faster algorithms to find out the changes occurring in TEG. We even found out that we need an incremental approach to improvise computation. We have even studied some of the existing computational frameworks and we found out that there is a need for a faster analytic and processing model. This requirement motivated us to propose an efficient framework for TEG, DASH. This paper states the challenges that are faced by the existing frameworks and how these challenges can be overcome by DASH. Moreover our experimental results show how DASH can perform better than GPS using an incremental approach. We propose various solutions on how to tackle individual challenges. We have an asynchronous execution model which maintains global status table and prevents synchronisation barrier too. We also have a dynamic scheduler which consists of three components responsible for different tasks. All the components of our framework communicate by message passing and most of the decisions are taken by the scheduler. We are optimistic that this survey will help many researchers to understand various dimensions of problems in TEG and continue developing the necessary techniques to resolve these problems more efficiently. Our proposed computing framework is an example of that, and we will test it further with more algorithms and optimise it to improve the performance.

References

Abdolrashidi, A. and Ramaswamy, L. (2016) 'Continual and cost-effective partitioning of dynamic graphs for optimizing big graph processing systems', in *2016 IEEE International Congress on Big Data (BigData Congress)*, pp.18–25.

Amelio, A. and Pizzuti, C. (2015) 'An evolutionary dynamic optimization framework for structure change detection of streaming networks', in *2015 6th International Conference on Information, Intelligence, Systems and Applications (IISA)*, pp.1–6.

Anagnostopoulos, A., Kumar, R., Mahdian, M., Upfal, E. and Vandin, F. (2012) 'Algorithms on evolving graphs', in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference, ITCS'12*, ACM, New York, NY, USA, pp.149–160.

Bahmani, B., Kumar, R., Mahdian, M. and Upfal, E. (2012) 'Pagerank on an evolving graph', in *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD'12*, ACM, New York, NY, USA, pp.24–32.

Barlow, J. (2003) 'The laws of the web. Patterns in the ecology of information [Review]', *Interface: The Journal of Education, Community and Values*, Vol. 3, No. 3 [online] http://bcis.pacificu.edu/journal/2003/03/huberman.php.

Brandes, U. (2001) 'A faster algorithm for betweenness centrality', *Journal of Mathematical Sociology*, Vol. 25, No. 2, pp.163–177.

Chen, R., Yang, M., Weng, X., Choi, B., He, B. and Li, X. (2012) 'Improving large graph processing on partitioned graphs in the cloud', in *Proceedings of the Third ACM Symposium on Cloud Computing, SoCC'12*, ACM, New York, NY, USA, pp.3:1–3:13.

Chen, Z., Shen, H.T., Zhou, X. and Yu, J.X. (2009) 'Monitoring path nearest neighbor in road networks', in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, SIGMOD'09*, ACM, New York, NY, USA, pp.591–602.

Cheng, R., Hong, J., Kyrola, A., Miao, Y., Weng, X., Wu, M., Yang, F., Zhou, L., Zhao, F. and Chen, E. (2012) 'Kineograph: taking the pulse of a fast-changing and connected world', in *Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys'12*, ACM, New York, NY, USA, pp.85–98.

Desikan, P., Pathak, N., Srivastava, J. and Kumar, V. (2005) 'Incremental page rank computation on evolving graphs', in *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web, WWW'05*, ACM, New York, NY, USA, pp.1094–1095.

Doekemeijer, N. and Varbanescu, A.L. (2014) *A Survey of Parallel Graph Processing Frameworks*, Report Number PDS-2014-003, ISSN 1387-2109.

Ediger, D., McColl, R., Riedy, J. and Bader, D.A. (2012) 'Stinger: high performance data structure for streaming graphs', in *2012 IEEE Conference on High Performance Extreme Computing*, pp.1–5.

Fan, W., Li, J., Luo, J., Tan, Z., Wang, X. and Wu, Y. (2011) 'Incremental graph pattern matching', in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD'11*, ACM, New York, NY, USA, pp.925–936.

Fan, W., Hu, C. and Tian, C. (2017) 'Incremental graph computations: doable and undoable', *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD'17*, ACM, New York, NY, USA, pp.155–169.

Filippidou, I. and Kotidis, Y. (2015) 'Online and on-demand partitioning of streaming graphs', *2015 IEEE International Conference on Big Data (Big Data)*, pp.4–13.

Gao, Y., Zhou, W., Han, J., Meng, D., Zhang, Z. and Xu, Z. (2015) 'An evaluation and analysis of graph processing frameworks on five key issues', in *Proceedings of the 12th ACM International Conference on Computing Frontiers, CF'15*, ACM, New York, NY, USA, pp.11:1–11:8.

Gonzalez, J.E., Low, Y., Gu, H., Bickson, D. and Guestrin, C. (2012) 'Powergraph: distributed graph-parallel computation on natural graphs', in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12*, USENIX Association, Berkeley, CA, USA, pp.17–30.

Han, W., Miao, Y., Li, K., Wu, M., Yang, F., Zhou, L., Prabhakaran, V., Chen, W. and Chen, E. (2014) 'Chronos: a graph engine for temporal graph analysis', in *Proceedings of the Ninth European Conference on Computer Systems, EuroSys'14*, ACM, New York, NY, USA, pp.1:1–1:14.

Jeong, H., Mason, S., Barabasi, A-L. and Oltvai, Z. (2001) 'Lethality and centrality in protein networks', *Nature*, Vol. 411, No. 6833, pp.41–42.

Kao, J-S. and Chou, J. (2016) 'Distributed incremental pattern matching on streaming graphs', in *Proceedings of the ACM Workshop on High Performance Graph Processing, HPGP'16*, ACM, New York, NY, USA, pp.43–50.

Kas, M., Carley, K.M. and Carley, L.R. (2013) 'Incremental closeness centrality for dynamically changing social networks', in *2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2013)*, pp.1250–1258.

Kourtellis, N., Morales, G.D.F. and Bonchi, F. (2015) 'Scalable online betweenness centrality in evolving graphs', *IEEE Transactions on Knowledge and Data Engineering*, Vol. 27, No. 9, pp.2494–2506.

Kyrola, A., Blelloch, G. and Guestrin, C. (2012) 'Graphchi: large-scale graph computation on just a PC', in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12*, USENIX Association, Berkeley, CA, USA, pp.31–46.

Liu, Y., Gao, H., Kang, X., Liu, Q., Wang, R. and Qin, Z. (2015) 'Fast community discovery and its evolution tracking in time-evolving social networks', in *2015 IEEE International Conference on Data Mining Workshop (ICDMW)*, pp.13–20.

Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N. and Czajkowski, G. (2010) 'Pregel: a system for large-scale graph processing', in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD'10*, ACM, New York, NY, USA, pp.135–146.

Noyes, D. (2018) *The Top 20 Valuable Facebook* [online] https://zephoria.com/top-15-valuable-facebook-statistics/ (accessed 7 February 2018).

Rapoport, A. and Horvath, W.J. (1961) 'A study of a large sociogram', *Behavioral Science*, Vol. 6, No. 4, pp.279–291.

Roy, A., Mihailovic, I. and Zwaenepoel, W. (2013) 'X-stream: edge-centric graph processing using streaming partitions', in *Proceedings of the 24th ACM Symposium on Operating Systems Principles, SOSP'13*, ACM, New York, NY, USA, pp.472–488.

Salihoglu, S. and Widom, J. (2013) 'GPS: a graph processing system', in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management, SSDBM*, ACM, New York, NY, USA, pp.22:1–22:12.

Shi, X., Luo, X., Liang, J., Zhao, P., Di, S., He, B.J. and Jin, H. (2018a) 'Frog: asynchronous graph processing on gpu with hybrid coloring model', *IEEE Transactions on Knowledge and Data Engineering*, Vol. 30, No. 1, pp.29–42.

Shi, X., Zheng, Z., Zhou, Y., Jin, H., He, L., Liu, B. and Hua, Q-S. (2018b) 'Graph processing on GPUS: a survey', *ACM Comput. Surv.*, Vol. 50, No. 6, pp.81:1–81:35.

Simmhan, Y., Kumbhare, A.G., Wickramaarachchi, C., Nagarkar, S., Ravi, S., Raghavendra, C.S. and Prasanna, V.K. (2013) 'Goffish: a sub-graph centric framework for large-scale graph analytics', *Proceedings of the Euro-Par 2014 Parallel Processing Conference*, Springer, Cham, pp.451–462.

Tsourakakis, C., Gkantsidis, C., Radunovic, B. and Vojnovic, M. (2014) 'Fennel: streaming graph partitioning for massive scale graphs', in *Proceedings of the 7th ACM International Conference on Web Search and Data Mining, WSDM'14*, ACM, New York, NY, USA, pp.333–342.

Watts, D.J. and Strogatz, S.H. (1998) 'Collective dynamics of 'small-world' networks', *Nature*, Vol. 393, No. 6684, pp.440–442.

Wickramaarachchi, C., Frincu, M. and Prasanna, V. (2014) 'Enabling real-time pro-active analytics on streaming graphs', *Algorithms*, Vol. 15, p.18.

Yuan, P., Zhang, W., Xie, C., Jin, H., Liu, L. and Lee, K. (2014) 'Fast iterative graph computation: a path centric approach', in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC'14*, IEEE Press, Piscataway, NJ, USA, pp.401–412.

Zhang, Y., Gao, Q., Gao, L. and Wang, C. (2014) 'Maiter: an asynchronous graph processing framework for delta-based accumulative iterative computation', *IEEE Transactions on Parallel & Distributed Systems*, Vol. 25, No. 8, pp.2091–2100.

Zhao, P., Aggarwal, C. and He, G. (2016) 'Link prediction in graph streams', in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pp.553–564.

Zhou, D., Wang, K., Cao, N. and He, J. (2015) 'Rare category detection on time-evolving graphs', in *2015 IEEE International Conference on Data Mining*, pp.1135–1140.

Zhou, Y., Cheng, H. and Yu, J.X. (2010) 'Clustering large attributed graphs: an efficient incremental approach', in *2010 IEEE International Conference on Data Mining*, pp.689–698.