Chapter 3

Related Work

In this Chapter, we introduce and discuss existing contributions in the fields that are related to the four research questions posed in Section 1.1. We present different approaches for processing data in Section 3.1 and describe existing frameworks for the analysis of dynamic graphs in Section 3.2. We introduce existing algorithms for the computation of different metrics in Section 3.3 and describe approaches for the efficient storage of graphs in Section 3.4.

3.1 Data Processing

In this Section, we present existing approaches for processing data. We introduce general concepts in Section 3.1.1, which are also applied when processing and analyzing graphs. We describe existing frameworks that implement these concepts in Section 3.1.2, many of which are used as a foundation for graph analysis frameworks.

3.1.1 Data Processing Concepts

A *stream* is a potentially infinite sequence of data elements that appear over time [257, 142, 19, 228, 4]. This concept is closely related to stream-based graph algorithms, which we introduced in Section 2.6.3. In general, all changes that occur in a dynamic system over time can be considered as such a continuous *data stream*. Examples are sensor measurements and stock price fluctuations. Their analysis is referred to as *computation on data streams*, *single-pass analysis*, or *sequential I/O*. It is commonly performed in real-time, i.e., new elements are processed as soon as they appear [295]. For the analysis in this scenario, so-called *streaming algorithms* are used. They are required to use space sublinear in the number of elements in the stream and therefore not allowed to store all data in memory. Various problems have been investigated, including the clustering of ata points [134, 5], the approximation of element frequencies [210], or the numveration of frequent patterns at arbitrary time granularities [123]. This *stream-based processing* approach contrasts the concept of *batch-based processing*. Here, elements are not processed as they appear but grouped into batches and processed altogether.

Bulk Synchronous Parallel (BSP) is a design paradigm for parallel algorithms [311, 121, 48]. It serves as the basis for many existing graph analysis frameworks. BSP enables the distributed processing of local computations. The computation is organized in so-called *super steps*. Within each step, the distributed processes concurrently compute their current task asynchronously. They communicate with each other to exchange data stored remotely. When a process finishes a task, it waits until all other processes reach the same so-called *barrier*. The results from all processes are then aggregated and the execution of the next super step is initiated. The BSP paradigm is well-suited for problems that can be split into sub-problems that require only parts of the globally available data.

MapReduce is a concept to facilitate data processing on large clusters [85, 86]. It is used by various data as well as graph processing frameworks for the distribution of work. A MapReduce program consists of the two procedures *Map()* and *Reduce()*. The *Map()* procedure filters and sorts the input data. The results are then summarized for further processing by the *Reduce()* procedure. Implementations of MapReduce programs commonly consists of three steps: *Map()*, *Shuffle()*, and *Reduce()*. In the first step, each worker processes its local data using the *Map()* operation and outputs keys that describe the task assignment. Next, the workers redistribute data among each other based on these keys. In the final step, workers process the assigned data in parallel.

3.1.2 Data Processing Frameworks

Hadoop [o36] is a Java framework for storing and processing large data sets in a distributed setting [55, 288, 326]. In general, it assigns computation tasks based on the locality of data to reduce communication overhead between the workers. Hadoop consists of three components: data storage, resource management, and processing. As its distributed storage solution, it uses the *Hadoop Distributed Filesystem* (HDFS), an implementation of the Google File System (GFS) [122]. *Yet another Resource Negotiator* (YARN) [o37] is used as resource manager, which assigns computation tasks to the workers [313]. As a batch processing framework, Hadoop uses *MapReduce*, an implementation of the MapReduce concept. Hadoop is used by many data and graph processing frameworks as foundation for distributing data as well as computation tasks.

Spark [o80] is a data processing tool for the parallel computation on clusters [339, 338]. It uses a restricted shared memory approach and is written in Java, Python, R, and Scala. Spark operates on *resilient distributed datasets* (RDDs), a read-only data structure distributed across all cluster nodes. This enables the iterative processing of datasets required for machine learning and processing SQL-like queries. A Spark cluster is managed either by a native implementation or existing approaches like YARN or Mesos [o54]. As Spark's distributed file system, many solutions can be used, including HDFS, Cassandra [o9], MapR-FS [o52], Amazon S3 [o3], and Kudu [o49]. As an input data stream, Spark provides interfaces to a large number of existing providers, such as the message broker Kafka [o46], the stream aggregator Flume [o21], the Twitter streaming API [o93], the distributed messaging library ZeroMQ [o95], the Amazon platform Kinesis [o2], and TCP/IP sockets. *Spark SQL* [o82] was developed on top of spark [332]. It adds the representation of data in so-called *DataFrames*, a data structure that supports the representation of structured data in Spark. In addition, Spark SQL provides interfaces for the manipulation of DataFrames as well as SQL support. *Spark Streaming* [o83] extends the core of Spark with support for pseudo stream-based analysis [340]. It splits a data stream into micro-batches that consists of all data elements that appeared within a short time frame. Each batch is then applied to modify the RDDs. Afterwards, they are processed in it entirety in the existing architecture. Therefore, Spark only provides a framework for the consecutive snapshot-based analysis of dynamic graphs as discussed in Section 2.6.2. In case the last batch would be supplied together with the updated RDDs, batch-based approaches could be supported as well (cf. Section 2.6.4). The use of Spark to implement the stream-based analysis of dynamic as described in Section 2.6.3 is not possible.

Flink [o17] is a distributed streaming data flow engine [109], written in Java and Scala. It enables data-parallel batch- and stream-based processing and provides native support for iterative algorithms. Flink stores data in distributed storage systems like HDFS or HBase [o39]. As input source, it provides interfaces to many system, such as Kafka, Flume, RabbitMQ [o71], the Twitter Streaming API, and Docker containers [o13]. Flink processes each element of a data stream separately but can also be used for batch-based processing. It is optimized for cyclic processes by using iterative transformations on collections. Overall, Flink can be used to implement the analysis of dynamic graph using snapshot-, batch-, and stream-based approaches.

Storm [o89] is a distributed processing framework for data streams, written in Clojure and Java. Its resources are managed by Storm-YARN [o90]. Storm acts as a data transformation pipeline with no specification of its termination. Storm applications are organized as *directed acyclic graphs* (DAGs). Its vertices, called *spouts* or *bolts*, model different components. They are connected by edges that represent directed streams to transport data from source to destination vertex. *Heron* [o40] is a framework for the distributed processing of streaming data [184]. It is written in Java and provides APIs for Java, Scala, and Python. Heron was developed by Twitter as a direct successor to Storm and is backwards-compatible. It is expected to be faster and more scalable to meet the real-time requirements of Twitter's applications. Both frameworks can be used as part of a data processing pipeline to pre-process a data stream but not as a single basis for building dynamic graph analysis on top.

Hama [o38] is a distributed data processing framework that implements the BSP paradigm. It is written in Java and consists of three components: *BSPMaster*, *GroomServers*, and *Zookeeper*. The BSP-Master is responsible for task management, job scheduling, and control of the super steps. GroomServers are the workers that execute the tasks provided by the master. The synchronization of the components at the end of each super step is orchestrated by the Zookeeper. Hence, Hama could be used to implement dynamic graph analysis using separate snapshot-based processing.

Samza [o72] is an asynchronous framework for stream processing. It was initially developed by LinkedIn [o51] and is written in Scala and Java. Samza uses Kafka's pub/sub message queue for communication and YARN as resource manager. Even though it is a powerful stream processor, it has not yet been used as basis for the analysis of dynamic graphs.

3.2 Graph Analysis

In this Section, we present existing approaches and frameworks for the analysis of dynamic graphs, both related to research questions *Q1*, *Q3*, and *Q4*. Frameworks can be used to benchmark and compare the performance of algorithms (*Q1*) as well as graph representations in case multiple are supported (*Q4*). Existing approaches for analyzing dynamic graphs are directly related to *Q3*, which is concerned with the parallelization of dynamic graph analysis. We describe the basic properties of each frameworks, such as the underlying data processing framework and which graph analysis concepts it provided. We discuss which graph metrics are provided by the framework and to which extent they provide the visualization of graphs and the results computed during an analysis.

We divide the frameworks into three classes: separate snapshot-based, consecutive snapshot-based, and stream-based frameworks (cf. Section 2.6). Separate snapshot-based frameworks are not designed for the analysis of dynamic graphs and do not support to change the graph itself. For each timestamp of interest, the corresponding snapshot must be read separately, which results in an I/O overhead. In contrast, consecutive snapshot-based frameworks allow for the adaptation of a graph over time. The analysis is also performed using snapshot-based algorithms. Stream-based frameworks provide the stream-based transformation of a graph and facilitate its analysis using either snapshot- or stream-based algorithms.

Please note that we restrict our discussion to those graph processing frameworks that are, to the best of our knowledge, most commonly used in academia. Other notable consecutive snapshot-based frameworks include GraphJet [284, o30], FlashGraph [344, o16], Blogel [334, o7], Giraph++ [303], Pregelix [58, o67], and Mizan [161, 174, o55] while Chronos [138] and KineoGraph [65] are recent stream-based graph frameworks.

We introduce concepts for the analysis of graph snapshots in Section 3.2.1. We describe frameworks that apply separate snapshot-based processing in Section 3.2.2, frameworks for consecutive snapshot-based analysis in Section 3.2.3, and stream-based frameworks for the analysis of dynamic graphs in Section 3.2.4.

3.2.1 Snapshot-based Graph Analysis Concepts

The most common way to analyze a graph snapshot is the use of *sequential* or *serial algorithms*. All operations are executed sequentially in a single thread as opposed to *parallel* or *concurrent* execution [127, 133, 108]. Often, these algorithms follow directly from the properties they should compute. Therefore, sequential algorithms are commonly used for the separate or consecutive snapshot-based processing of dynamic graphs. Their translation into parallel algorithms is not straight-forward in most cases [127, 133].

Pregel is a *vertex-centric approach* for the distributed analysis of large graphs [209]. It was developed by Google and has been adopted by many graph analysis frameworks [137, 219]. Pregel's "think like a vertex" approach is conceptually close to the BSP principle. The set of vertices is partitioned among the workers which store the respective induced subgraph of their partition. Edges between partitions are modeled as network connections between the workers. To execute a program in the BSP round-based manner, workers execute a program for each vertex that is contained in their respective partition. As a result, vertices can either vote to halt the program or send messages via their edges to neighbors. These messages are either sent locally to vertices of the same worker or over the network to vertices in other partitions. In the next super step, vertices receive messages and execute the program again. Pregel has proven to be a powerful approach that fits better to sequential graph problems than MapReduce [261, 246]. As an extension to Pregel, the *graph-centric approach* to "think like a graph" has been proposed [303]. It is designed to reduce the communication overhead that occurs when messages are sent inside partitions via the costly messaging interface. To achieve this, information about the partitions is provided to the developer. This allows for the optimization of algorithms and the decrease of messaging overhead.

Streaming graph analysis applies the general concept of stream-based processing to the analysis of graph snapshots [59, 162, 154]. Here, a graph is represented as a stream of its edges. This approach is useful in case a graph is too large to fit into randomly accessible memory. Algorithms that are only allowed to see this stream once are referred to as *single-pass algorithms*. Because of their restricted access to the graph, most graph properties can only be approximated. To enable more precise computations on graphs with lower space requirement, *semi-streaming* approaches have been developed [111]. Here, a linear number k or logarithmically many passes over the complete stream are allowed. This enables the exact computation of properties or their approximation with higher quality.

3.2.2 Separate Snapshot-based Graph Analysis Frameworks

Giraph is a framework for the distributed execution of sequential graph analysis [18, 66, 67, 213, o24], implemented in Java. It is inspired by Pregel and adopts its vertex-centric processing model. As a back-

bone for the communication and data exchange between workers, it uses Hadoop. *Giraph* itself provides implementations for the computation of connected components, page rank, single-source shortest paths, and random walks. Algorithms for the computation of other graph properties have been implemented by third-party projects. The most notable is the *Giraph*-based machine learning library Okapi [o59]. It adds algorithms for the computation of various similarity measures, sybil rank, and triangle counts.

Graph Processing System (GPS) is a Pregel-like graph analysis framework [272, o27], written in Java. Similar to *Giraph*, it adds to the basic constructs of Pregel via a master compute function to combine local and global computations. *GPS* also includes a re-partitioning scheme to decrease messaging overhead between workers. Furthermore, it provides an optimization called *LALP* to further reduce the network communication when processing graphs with a skewed degree distribution. So far, algorithms for the computation of various properties have been implemented, e.g., coloring, weighted shortest paths, page rank, as well as strong and weak connectivity.

Signal/Collect is a graph analysis framework [296, o78], written in Scala. It uses a vertex-centric programming model for the parallel computation on a multi-core system. During an analysis, information is sent along the graph's edges (*signal*). Then, these edges perform the actual computation (*collect*). It supports a synchronous execution mode similar to Pregel in specific and BSP in general. In addition, *Signal/Collect* provides an asynchronous as well as an interactive execution mode. Many algorithms for the computation of graph properties have been implemented, including clustering coefficient, page rank, single-source shortest paths, and vertex coloring.

Small-world Network Analysis and Partitioning (SNAP) is a parallel framework for the analysis and partitioning of large-scale graphs [22, o79], implemented in C. The parallelization is implemented using *Open Multi-Processing* (OpenMP) [81, o61], an API for shared-memory parallel programming. In addition to the partitioning of a graph, properties like centrality or modularity can be computed.

Medusa is a parallel graph processing framework [345, 346, o53]. It is implemented in CUDA, C, and C++. It enables users to leverage the parallelization capabilities of GPUs by writing sequential C or C++ code. So far, it only provides algorithms for the computation of single-source shortest paths and the execution of a breadth-first search.

GraphLab is a graph-based, distributed computation framework [200, 201, o31], written in C++. Similar to other frameworks, it was initially developed for the application in machine learning. Many toolkits have been implemented on top of it, which cover fields such as collaborative filtering, clustering, and computer vision. The graph library *PowerGraph* supports the basic computations such as triangle counting and page rank [129, o66].

Ligra is a parallel graph processing system [287, o50], written in C++. It is designed for the execution in a multi-core, shared memory environment. The API is minimal and implicitly provides parallel execution. Many graph properties can be computed, such as triangle count, page rank, k-core, and betweenness centrality.

Graph-tool is a Python module for the efficient analysis and visualization of graphs [252, o28]. It is implemented in C++ and based on the *Boost Graph Library* (BGL) [o8]. Algorithms are implements using OpenMP. *Graph-tool* supports directed as well as undirected graphs, the assignment of arbitrary weights via property maps, and the representation of graphs as adjacency lists or adjacency matrices. It does not support dynamics to the graph but provides a large number of algorithms for the computation of graph properties, including all-pairs shortest paths, degree distribution, similarity measures, assortativity, clustering coefficient, community structure, and various centrality measures. In addition, algorithms for determining graph isomorphism, minimum spanning trees, connected components, and maximum flow are included. The framework also provides graph generators for random and power-law graphs. *Graph-tool* provides an own implementation for the visualization of graphs as well as interfaces to *Graphviz*, a standalone graph visualization tool [103, o33].

Pajek is a Windows-only framework for the analysis and visualization of large graphs [35, 36, 83, o63], written in Delphi. It supports directed and undirected graphs including their weighted versions. *Pajek* provides a vast amount of algorithms for the computation of graph properties, including connectivity, shortest paths, centrality measures, clustering, and community structure. *PajekXXL* is a memory-efficient version of *Pajek*, which can process graphs with up to 999,999,999,997 vertices. For the visualization of graphs, various graph drawing algorithms have been implemented.

Cassovary is another graph analysis framework with a focus on space efficiency [o92] It was developed by Twitter in Scala and provides APIs for Java and other JVM languages.

X-Stream is an edge-centric graph processing framework [269, o94], implemented in C and C++. It provides the multi-threaded analysis of large graphs, represented as a stream of their edges, on a single machine. Thereby, it enables the analysis of graphs that are too large to fit into memory using semi-streaming approaches. As input, *X-Stream* requires a graph to be stored in a binary edge format and can handle up to a size of 3TB, i.e., 64,000,000,000 edges. So far, algorithms for computing single-source shortest paths, betweenness centrality, strong connectivity, and triangle count have been implemented. In

X-Stream, the analysis is restricted to the execution on a single machine. Therefore, the *Chaos* framework was developed as an extension [268]. It supports the distribution of an analysis to multiple machines and facilitates the processing of graphs containing more that 1,000,000,000,000 edges.

3.2.3 Consecutive Snapshot-based Graph Analysis Frameworks

Java Universal Network/Graph Framework (JUNG) is a Java library for modeling, analyzing, and visualizing graphs [245, o45]. It supports directed, undirected, parallel, and hyper graphs with arbitrary weights assigned to vertices and edges. Some snapshot-based algorithms for the computation of properties like page rank and centrality measures are provided. *JUNG* contains an extensive visualization component which includes the interactive modification of the represented graph.

GraphStream is another framework for the representation and visualization of dynamic graphs [100, o32], written in Java. It implements a stream-based maintenance of dynamic graphs and contains a powerful visualization component. *GraphStream* provides a selection of graph generators and the possibility to assign attributes to vertices and edges. While its focus is on the visualization of graphs and their properties, it also contains rudimentary support for the snapshot-based computation of graph properties.

Gephi is a framework for the interactive visualization of static and dynamic graphs [34, o23]. Multiple graph drawing algorithms have been implemented to enable the visualization of graphs in various ways. It is limited to graphs with less than 1,000,000 vertices and edges. Dynamic graphs can be described as a stream of updates and analyzed using snapshot-based algorithms. Many have been implemented, including betweenness centrality, closeness centrality, shortest paths, diameter, clustering coefficient, page rank, and community detection.

NetworkX is another graph framework with an emphasis on analysis and graph generation [279, o58]. It is implemented in Python and provides representations for directed, undirected, and parallel graphs. A huge amount of snapshot-based algorithms has been implemented for the analysis of dynamic graphs. These graphs can either be read from many formats or created using internal generators. The initial state of a dynamic graph is generated using a static models. The changes to the dynamic graph over time are modeled using so-called *transformations*. The visualization component allows for the 2- or 3-dimensional representation of graph snapshots.

Stanford Network Analysis Platform (SNAP) is a graph analysis system for large graphs [189, o85]. It is part of the *Stanford Network Analysis Project* [o87] alongside the *Stanford large Network Dataset Collection* [o84]. *SNAP* is implemented in C++ and also available for Python (Snap.py) [o86]. Graphs and their changes over time can be generated via models in the framework or read from files. For the analysis of graphs, the framework only provides means to define snapshot-based algorithms.

GraphChi is a vertex-centric graph analysis framework [186, o29]. It is an extension of *GraphLab* and adds capabilities for modeling and representing dynamic graphs. The main version of GraphChi is implemented in C++. A second version, implemented in Java, exists alongside a graph database implemented in Scala. With its Pregel-like analysis approach, GraphChi only includes support for snapshot-based algorithms. Basic examples like community detection, triangle counting, as well as the computation of page rank and connected components have been implemented.

GraphX is a graph processing framework on top of Spark [331, 130, 330, o81], written in Java. Like its predecessor *Bagel*, it represents dynamics of a graph by modifying the RDDs that represent it. At its core, *GraphX* uses Hadoop for the distribution of jobs and the management of resources. It provides a Pregel- as well as a MapReduce-like API. In both cases, the graph analysis is only performed using snapshot-based approaches that must be executed for each timestamp of interest. As examples, algorithms for counting triangles and computing connected components as well as page rank have been implemented.

Spargel is an API for graph processing on top of Flink [o19]. It provides a Pregel-like approach for the analysis of graph snapshots, which can be changed over time. *Spargel* supports directed and weighted graphs as well as the possibility to model parallel edges. So far, algorithms for the computation of connected components, page rank, and shortest paths have been implemented.

3.2.4 Stream-based Graph Analysis Frameworks

Gelly is an extension to the Flink-based *Spargel* framework for graph analysis [o18]. Recently, an API for stream-based maintenance of dynamic graphs was added. It allows for the modification of graphs via compound vertex and atomic edge updates. While *Gelly* does not support graph generators, it provides operations for modifying a graph over time, including filtering, joining, reverting, and union operations. Like *Spargel*, the framework uses a vertex-centric approach for the snapshot-based computation of properties. So far, properties like page rank, connected components, single-source shortest paths, label propagation, and community detection can be computed.

GraphTau is a stream-based graph analysis framework built on top of Spark [149]. It uses *GraphX* for the graph representation and Spark Streaming to model streaming input. Each snapshot of a graph is modeled as a pair consisting of a *vertex RDD* and an *edge RDD*. Over time, new snapshots are created by applying a *DeltaRDD* to them for each transition. *GraphTau* supports two computational models: *Pause-Shift-Resume* (PSR) and *Online Rectification* (OR). In PSR, the computation on the current snapshot is paused when a new one arrives. All result, computed to far, are shifted to the new snapshot and the computation is resumed. Obviously, this leads to imprecisions and only allows for the computation of approximate results. In OR, a computation is set back to a state where the latest changes have not been taken into consideration yet and then continued on the new snapshot. Using *GraphTau*, properties like connected components and page rank can be computed. So far, the framework is not publicly available.

Spatio-Temporal Interaction Networks and Graphs Extensible Representation (STINGER) is a stream-based graph analysis framework [20, 102, 263, o88]. It is written in C++ and an advancement of the *Small-world Network Analysis and Partitioning* framework with bindings for Java and Python. At its core, it provides a data structure for storing sparse dynamic graphs with semantic information attached to them. Adjacency lists are maintained as linked lists of arrays and meta data is stored directly in vertices. Its graph maintenance components provides a stream- and a batch-based mode. Vertices and edges can be inserted or removed and their respective attributes updated. So far, stream-based algorithms for the computation of connected components, community structure, and betweenness centrality have been implemented as well as a parallelized batch-based computation of the clustering coefficient. In addition, snapshot-based algorithms for executing a breadth-first search, extracting the k-core, and performing agglomerative clustering are available.

3.3 Algorithms for Dynamic Graph Analysis

In this Section, we present algorithms for the computation of graph properties using different approaches. They directly relate to the second research question (*Q2*) which considers the development of new algorithms for the efficient analysis of dynamic graphs. We present sequential algorithms in Section 3.3.1. We briefly discuss parallel algorithms in Section 3.3.2 and introduce vertex-centric approaches in Section 3.3.3. In Section 3.3.4, we illustrate streaming algorithms and give examples of dynamic algorithms in Section 3.3.5.

3.3.1 Sequential Algorithms

We consider an algorithm to be *sequential* or *serial* if it executes all operations one after the other as opposed to parallel or concurrent processing. These algorithms have been researched for many decades and are commonly applied for the snapshot-based analysis of dynamic graphs. In the remainder of this Section, we give examples of commonly used properties and the respective sequential algorithms for their computation.

The set of (weakly/strongly) *connected components* of a graph is a partition of V such that the subgraph induced by each subset is (weakly/strongly) connected while any superset is not. The size distribution of a graph's components is of interest in many areas, including neural networks [254, 217], botnet detection [259], and network resilience [230]. Weakly connected components can be determined by executing a breadth-first search (BFS) or a depth-first search (DFS), a straight-forward computation as noted by Hopcraft and Tarjan [145]. The strongly connected components can be computed by executing two BFSs [283, 76] or by performing a single DFS while maintaining a stack of visited vertices [301]. Maintaining a second stack can speedup this computation [93]

As a *community*, we consider a subgraph whose vertices are densely interconnected but have only a relatively small number of connections to vertices in other communities. Determining the community structure of a graph helps to understand the components and their connections of biological, social, and computer networks [124, 248, 235]. For their detection, a large number of sequential algorithms has been developed [116, 329]. Many algorithms like *DeltaQ* [234] and *Fast Unfolding* [51] attempt to minimize the modularity of communities, i.e., the ratio of intra- to inter-community connections. Other algorithms apply the concept of expanding spheres [24] or the round-based propagation of labels [258, 193] to determine community structures.

Computing the *single-source shortest path lengths* (SSSP) means to determine the length of the shortest path from one vertex, the source, to all others. Computing this property for each vertex as source solves the *all-pairs shortest path length* (APSP) problem. Both find applications in the analysis of biological, social, computer, and chemical networks [231, 32, 52]. Solutions to these problems have been researched for a long time with the most prominent solutions being the algorithms by Dijkstra [92], Bellman-Ford [39], and Floyd-Warshall [115]. More recently, algorithms have been proposed to solve

these problems on large graphs [227, 119, 172, 328]. Others have been developed for the application in graphs with small integer weights [120], arbitrary integer weights [286], or real weights [63]. In addition, heuristics have been proposed that only approximate the distribution of shortest paths in a graph [351].

The *betweenness centrality* of a vertex is the number of all shortest paths between pairs of vertices that it is part of. Thereby, it expresses the importance of each vertex in many biological, social, and computer networks [157, 33]. Furthermore, it has applications in measuring the tolerance of a network to targeted attacks [80]. Its computation is time-consuming and only few efficient algorithms have been developed [56]. To approximate the betweenness centrality in reasonable time, sampling-based approaches are often applied instead of exact computations [21, 264].

Counting the number of closed *triangles* in a graph reveals insights into the interconnection of its components. When put into the relation of potential triangles, i.e., pairs of neighbors of each vertex, measures like the *local clustering coefficient*, *average clustering coefficient*, or *global clustering coefficient* can be derived. Initially, local and average clustering coefficient were introduced by Watts and Strogatz to understand and model the small-world effect in social networks [320]. The global clustering coefficient, or *transitivity*, was later introduced by Newman as a measure to quantify the quality of social network generators [237]. These properties help to understand the relation between neighbors of a vertex in scientific networks [30], network growth models [314], brain networks [271], and social networks [226]. Many algorithms have been developed to compute these metrics exactly [275]. In addition, various approximations, especially for the transitivity, have been developed [276, 308, 309].

We consider classes of isomorphic subgraphs of size k as *k-vertex motifs*. The relative occurrences of subgraphs that belong to a motif class are assumed to indicate the nature and functionality of many networks [225]. They are used in the analysis of graphs from a wide range of applications, including Internet Point-of-Presence maps [112, 113], natural language processing [46, 45], and Peer-to-Peer networking [183, 136]. Furthermore, they are of great importance in the analysis of biological networks [255, 205] and are used to understand protein interaction networks [215, 7, 270, 72], cellular networks [176], and the structure of genes [160, 285]. In addition, the concept of motifs has also been extended by the inclusion of temporal aspects [180, 159] and the definition of degree-based motifs [224]. Because of this huge field of application to many different areas, a multitude of sequential algorithms has been developed to count the occurrences of motifs in a graph. The first approaches like ProMotif [148], mfinder [170], MAVisto [278], and NeMoFinder [64] provided tools to count motifs of small sizes. On larger graph, they perform rather poorly, mainly because of their inefficient subgraph enumeration. This changed with the development of Fanmod [324], an efficient implementation of the RAND-ESU algorithm [322, 323] that all recent approaches have been compared to. Recent algorithms like Kavosh [169] or MODA [242] improve the efficiency of enumerating all subgraphs. G-Tries [262] is based on the idea of creating dedicated representations of subgraphs. ACC [221, 222] uses combinatorial techniques to speed-up the computation. To speedup the computation, sampling-based approaches have also been developed [171, 323, 128].

The computation of various graph properties is straight-forward. Therefore, their computation has not been studied widely. This includes properties like the degree distribution [99, 57, 140], the PageRank [247], the rich-club coefficient [348], and the assortativity [232, 233]. Most of them can easily be computed by iterating over the set of all vertices or edges and executing simple calculations.

3.3.2 Parallel Algorithms

For many sequential algorithms, parallel versions have been developed to speedup the analysis of a single snapshot in a multi-threaded environment [87, 256]. Examples are parallel algorithms for computating the betweenness centrality [208], counting triangles [300], finding motifs [277, 343], and computing connected components [202]. While this conversion of sequential algorithms to parallel versions is straight-forward for some properties and algorithms, it is inherently difficult for others. The main reasons for such difficulties are the data-driven and unstructured character of graph problems, their poor locality, and the high ratio of data access to actual computation [204].

3.3.3 Vertex-centric Algorithms

Soon after the development of Pregel [209], the vertex-centric algorithms gained a lot of attention. With its clear and intuitive programming model, the development of many algorithms became easy and allowed for the straight-forward distribution of large computation tasks. In this context, vertex-centric algorithms for the computation of PageRank [209, 202] and triangle counting [101] have been developed and implemented in most vertex-centric analysis frameworks. To determine the connected components of a graph, vertex-centric algorithms commonly solve the *HashMin* problem. Here, each vertex is assigned the smallest identifier of a vertex in its own component [260, 101, 335]. Many of these algorithms are based on

Shiloach-Vishkins parallel Algorithm for the computation of shortest paths [202]. Furthermore, vertex-centric algorithm have been developed to compute single-source shortest paths [202, 219] as well as approximations for the diameter, i.e., the longest shortest path between any pair of vertices [202].

3.3.4 Streaming Algorithms

In the scenario of streaming graph analysis, only a small number of single-pass algorithms has been developed. This includes graph partitioning [294, 269, 240, 307] as well as approximations of the triangle count [27, 154]. For other problems, such as *Max*, *MaxNeighbor*, *MaxTotal*, and *MaxPath*, it has been shown that they require too much memory to be solved efficiently using only a single pass over the edge stream [142].

Therefore, most graph algorithms in the streaming scenario apply the semi-streaming approach of processing the complete stream multiple times. Here, k- and $log(|V|)$-pass algorithms with lower space requirements have been developed that expect the edges ordered as incidence list or can handle them in arbitrary order [156, 37, 59]. Furthermore, k-pass approximations of the page rank [274] and the weighted, unweighted, or bipartite matching problem [111, 220] have been developed as well as $\frac{log|V|}{loglog|V|}$-pass approximations of diameter [111] and the distance between vertex-pairs in weighted graphs [111].

3.3.5 Dynamic Algorithms

Dynamic graph algorithms, also referred to as *online algorithms*, compute the properties of a dynamic graph over time. As input, they take an initial graph and a list of updates to it. In contrast to the streaming algorithms discussed in Section 3.3.4, they maintain a complete copy of the graph in memory. Each update is applied to the graph's in-memory representation and used as input for an algorithm to update the corresponding results. In regular intervals or uppon request, the properties of interest are queried and computed from the (intermediate) results [177]. Note that the concept of dynamic algorithms corresponds to our notion of stream-based algorithms, as introduced in Section 2.6.3.

Many dynamic algorithms for the update or maintenance of graph properties have been developed. To compute the connected components in dynamic graphs, spanning trees are commonly established and maintained for each update in the stream of changes [89, 265, 337].

Dynamic algorithms for detecting and maintaining communities have also been developed. They are based on distributed algorithms for the detection in delay tolerant networks [147] or define transitions between community assignments for single updates [238]. As the quality of detected communities deteriorates after a large number of updates, some approaches propose to regularly re-compute the community assignment using sequential algorithms [42, 297, 132].

For maintaining the all-pairs shortest path lengths [199], rooted spanning trees for each vertex are maintained where the distance of each vertex in the tree equals the shortest path length in the graph. While this approach is similar to dynamic algorithms for computing connected components, the maintenance of these trees is more complex and can require their complete re-computation in certain scenarios. Many algorithms have been developed and differ in the graph type that they are applicable to. Some approaches only consider weight changes [267], can handle edge insertions and weight decreases [117], or are specialized for edge deletions in undirected, unweighted graphs [44]. Other algorithms process all types of updates [118] or are restricted to positive integer or real weights [177, 90].

Exact algorithms for the computation of betweenness centrality are restricted to edge additions [131, 188] while heuristics are able to handle weighted as well an unweighted dynamic graphs [41].

To maintain the correct count of local and global triangles in a dynamic graph, a dynamic algorithms was developed [244]. Here, the main idea is that each edge addition creates new open triangles and closes existing ones. Hence, the respective counter can be increases by investigating the adjacencies of both connected neighbors.

Even though the detection of motifs has a wide range of application in many different areas, there is no stream-based algorithm yet to maintain the corresponding counts in a dynamic graph.

3.4 Graph Representation

In this Section, we give an overview of existing approaches for the efficient storage and access of graphs in memory. All of them are directly related to the fifth research question ($Q5$) which is concerned with determining efficient representations for the analysis of dynamic graphs. We introduce approaches for the efficient representation of graphs in Section 3.4.1. In Section 3.4.2, we describe the basic concepts

of graph databases and present examples of existing solutions. We introduce approaches for the profile-guided selection of efficient data structures in Section 3.4.3 and describe concepts for adaptive data structure selection in Section 3.4.4.

3.4.1 Efficient Graph Representation

Many frameworks have been developed for the separate snapshot-based graph analysis as introduced in Section 3.2.2. Here, the graph snapshot is written once during initialization and only read during analysis. Other frameworks implement consecutive snapshot-based graph analysis as introduced in Section 3.2.3. The graph representation is changed over time by applying batches of updates to it. The current state is analyzed afterwards, requiring only read access to the graph. Similarly, stream-based frameworks, introduced in Section 3.2.4, change the graph representation for each update separately followed by read operations of the analysis using stream-based algorithms.

All these frameworks are built for the efficient analysis of graphs. The underlying representation of a graph and its components is fixed and selected by the developers. Which representation is actually the most efficient one highly depends on many factors: the graph size and topology, the size of batches, the type of updates contained therein, the analysis-frequency, the metrics of interest, and the algorithms used for their computation. All of them influence the access patterns of read and write operations to the components of the graph, e.g., the list of all vertices V and the adjacency list of each vertex $adj(v)$. Obviously, there is not a single representation that performs best for all scenarios.

A lot of work has been done to develop compact representations of graphs. These approaches do not focus on runtime efficiency but on obtaining a small memory footprint [50]. They often are not even applicable to arbitrary graphs as they are developed for separable or sparse graphs only [49, 298].

Special graph representations for dynamic graphs have also been developed. Their underlying data structures are tuned for memory [207] or runtime efficiency [20, 102, 206]. Like the representations of graphs in frameworks, they are fixed and cannot be adapted to different scenarios.

In all cases, specific data structures are used to represent the components of a graph. While this selection of data structures achieves great performance in some cases, it performs poorly in others.

3.4.2 Graph Databases

Graph databases have been developed to represent and query elements and their relationships [16]. These relationships are modeled as edges between the elements with weight-like properties attached to them. This enables the simple description and fast execution of hierarchical queries on the represented data. In clear contrast, relational databases represent the relations between elements as their own properties and store them in tables [71]. When modeling many relationships between elements, this approach can lead to many *Join* operations, which can lead to a poor performance [316].

Dynamic graphs can be stored in graph databases by modeling the dynamics as relations or attaching lifetimes and timestamps to vertices and edges. Thereby, they allow for complex queries of the graph over time. But, they are neither suited for a large number of updates nor the efficient computation of topological graph properties for specific states [68].

So far, many different graph databases have been developed. While many have free community version, only few of them are open-source and used in academia. In the following, we give a brief overview of some of them. *Neo4J* [o57] is a transactional database for the storage and processing of graph data [321]. It is written in Java and considered to be the most used graph database [o12]. It enables access to the data via *Cypher*, a declarative graph query language with an open specification in the openCypher project [o60]. *OrientDB* [o62] is a multi-model database and the second most used graph database [o12]. It is written in Java and combines the features of a graph database and a document NoSQL database. OrientDB supports queries via SQL as well as *Gremlin* [o34], a graph traversal language. Gremlin was developed as part of *Tinkerpop* [o91], an Apache graph computing framework. It implements the property graph model *Tinkerpop Blueprints* and provides *TinkerGraph* as a reference implementation of a graph database.

The performance of different graph databases has been studies and compared extensively [16, 316, 15, 155]. For this purpose, many models have been developed to generate benchmarking workloads for graph databases [97, 23]. While these benchmarks also include workloads for the computation of graph properties using graph traversal, it became clear that they are not well-suited for such use cases [98].

3.4.3 Profile-guided Selection of Data Structures

Many approaches have been developed for profiling programs to facilitate their subsequent optimization. Frameworks like *Pin* [203] or *JFluid* [96] allow the instrumentation of existing programs to collect statistics about CPU usage, memory consumption, or call frequencies of code fragments.

In addition to this instrumentation, *Brainy* [158] enables the optimization of the data structures used by a program. Based on benchmarks of available data structures, it applies machine learning to generate rules like, e.g., *if operation o is called more than k times use data structure d*. After the analysis of a complete execution of the program, data structures are exchanged based on these general rules.

Such approaches are not applicable to the problem of dynamic graph analysis. The generated rules are generalized for all data types. Thereby, they do not take into account the specific runtime properties of handling vertices or edges in the respective lists. Furthermore, this profile-guided approach assumes a stable workload and size of the considered data structures. Hence, it does not consider changes that occur to size or access patterns during execution.

3.4.4 Adaptive Selection of Data Structures

Other approaches attempt to optimize the used data structures during run-time. Here, the data structures are selected during run-time and exchanged if necessary.

Just-in-Time data structures (JitDS) [84] is an extension of the Java language enabling the combination of multiple representations for a single data structure. For each instance, swap rules can be defined by an expert programmer to declare when and how to switch between representations. While this approach is powerful, it relies on the programmer's intuition and foresight to define such rules.

Chameleon [282] provides a framework for run-time profiling without the need to adapt the program. In case the program uses data structure wrappers provided by the framework, data structures can be replaced during run-time which comes at the high cost of performing a separate monitoring of each instance of such data structures.

Based on fixed rules for exchanging data structures as well, *CoCo* [333] requires the programmer to use wrappers provided by the framework in order to optimize the selected data structures during run-time.

With their use of pre-defined rules that do not adapt to the current properties of the graph and read accesses of the analysis, these approaches are not suited for optimizing the selection of data structures used to represent a dynamic graph during analysis.