Graph Databases for TLS Scans

Johannes Kunz, Markus Sosnowski*, Patrick Sattler†
*Chair of Network Architectures and Services, Department of Informatics
Technical University of Munich, Germany
Email: johannes.f.kunz@tum.de, sosnowski@net.in.tum.de, sattler@net.in.tum.de

***Abstract*—In the analysis of big-data, graph databases get a lot of attention due to their capability of dealing with large, unstructured, and rich data. The existing graph storage and analysis systems have different strengths and weaknesses depending on the use-case.**

In this paper, we will examine the three popular graph databases Neo4j, Apache Giraph, and MsGraphEngine in the context of storing and analyzing TLS scans. We will compare their storage architecture and graph processing capabilities to conclude that MsGraphEngine and Apache Giraph are better suited for our large-scale data analysis than Neo4j, where MsGraphEngine can be best adapted to our needs.

***Index Terms*—graph database, tls scan, big data, olap**

1. Introduction

In the field of graph processing, traditional database systems such as Relational Database Management Systems (RDBMS) and diverse NoSQL stores face problems due to the irregularity, richness, size, and the structure of the data [1]. A plethora of new databases called Graph Databases (graph DBs) have been developed, thus, specialized on storing and processing graphs. They can help reveal new information in the data by efficiently applying graph algorithms.

Graph DBs differ in data architecture and access, data distribution, query language and execution, and support for different transactions; see Section 2. Hence, the preferable graph DB is specific to the use-case. In this paper, we analyze and compare three popular graph DBs, namely *Neo4j*, *Apache Giraph*, and *MsGraphEngine* in the context of processing Transport Layer Security (TLS) scans. TLS is a widespread security protocol on the internet.

In Section 2 we explain the fundamentals of graph databases. We then define our problem and derive requirements on the Graph DB in Section 3. Finally, we analyze our candidates and compare them according to our requirements in Sections 5 and 6, respectively.

2. What are Graph Databases?

Graph Databases are specialized databases for handling graphs. They can cope with data-inherent properties that other DBs struggle with. In contrast to tabular data, graphs have an irregular structure, e.g. different incoming and outgoing numbers of edges per node [2]. Often, nodes and edges contain rich data such as labels and properties, which are inconsistent in size. Moreover, graphs can grow large in size, as we will see in Section 3.2, while modifications are made and information is ever-changing. On top, graph algorithms require irregular access to nodes and edges and, therefore, a low latency [1]. Ultimately, Graph DBs are designed to efficiently run graph algorithms.

Due to the requirements, a variety of different approaches have emerged, based on which graph databases can be distinguished. In the following, we will briefly explain these key concepts.

2.1. Graph Models

There exist three graph models, of which variants are implemented by graph DBs. The most common is the *Labeled Property Graph (LPG)* model [1]. The *Resource Description Framework (RDF)* and the *Hyper Graph* model are used by fewer databases and do not concern the DBs discussed here.

A Property Graph is defined as the tuple $(N, E, \rho, \lambda, \sigma)$ [3]. Let N and E denote finite sets of nodes (also called vertices) and edges such that $N \cap E = \emptyset$. Then, the function $\rho : E \to (N \times N)$ maps the edges to their corresponding start and end node. Additionally, the graph contains rich data associated to nodes and edges. Such data can be labels and properties, defined by the functions $\lambda : (N \cup E) \to \mathcal{P}^+(L)$ and $\sigma : (N \cup E) \times P \to \mathcal{P}^+(V)$, respectively. Here, L, P, and V are sets of labels, property names, and property values, respectively. The operator $\mathcal{P}^+(\cdot)$ denotes the power set excluding the empty set.

2.2. Storage Architectures

The storage architecture determines the efficiency and scalability of graph operations. Its index structures must provide an efficient way of querying the elements of the graph, while maintaining modifiability when it scales [1].

Some Graph DBs are based on more fundamental databases such as *key-value stores* or *wide-column stores*; see [1]. *Native* graph DBs use specially adapted storage architectures for graphs.

2.3. Types of Transactions

There are two types of transaction which a graph DB can be optimized for, namely Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP).

With OLTP, the user performs many smaller transactional queries that are processed interactively on the client

doi: 10.2313/NET-2020-11-1_04

for more in-depth analysis. This requires a low latency for transactions. The queries are local in nature, i.e., do not affect the graph on a global scale [1]. The graph database mainly acts as a storage system.

Conversely, OLAP is mostly executed on the server, i.e. the server performs exhaustive graph algorithms, while the client awaits the result. The requests are fewer, but often span the whole graph. The main goal is a high throughput, which is often achieved by a high degree of parallelization on the server [1].

2.4. ACID

A database complies with *ACID* if it ensures the following properties. Transactions are always performed completely or not at all (Atomicity). Data always remains consistent, i.e., a completed transaction always produces valid output (Consistency). Concurrent transactions cannot see intermediate results (Isolation). The result of completed transactions persists (Durability) [4]. This also applies to graph DBs.

3. Problem Statement

This section describes the dataset that needs to be processed and the resulting requirements on the database.

3.1. Input Data

We use the TLS scanner *goscanner*, see [5], to scan the internet for servers using the TLS protocol. The output are *.csv* tables containing information such as used certificates, ports, and protocols about all servers. The servers' IP is a unique identifier. An exemplary scan is shown in Table 1.

TABLE 1: Exemplary TLS scan

Host	Port	Server Name	Protocol
2a00:1450:4001:81f::200e	443	google.com	TLSv1.2
172.217.22.78	443	google.com	TLSv1.3
192.30.253.113	443	github.com	TLSv1.3

We want to store this table together with additional information as a directed graph connecting servers, server properties, domain names, certificates, and certificate authorities. We use labeled edges to express relations. An exemplary graph is shown in Figure 1.

In the future, scanners may gather even more information, such as more server properties or direct relations between servers. This input data structure is therefore not fixed, but rather the current state.

3.2. Requirements on the Graph Database

In the following, we derive a list of requirements on the graph DB based on the properties of our dataset.

3.2.1. Scalability. TLS scans easily reach sizes up to 200 GB with over 400M entries. Thus, the graph DB must be able to efficiently deal with large numbers of nodes and edges, while the data quantity per node is comparatively low (limited to mostly the node name and a few properties).

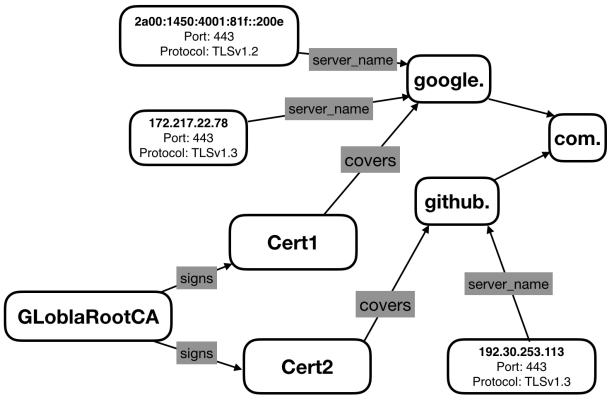

Figure 1: Exemplary graph structure of a TLS scan corresponding to Table 1

3.2.2. Fast Loading. TLS scans are performed on a frequent basis. Once a new scan is ready, the old one is obsolete. Thus, the most recent scan shall be inserted into the graph DB as quickly as possible to have more time for analysis. Due to the huge data size, the insertion speed is a relevant parameter.

3.2.3. Adaptability. As explained above, our input data structure is not fixed. If we want to focus on different aspects in our analysis, we may need to change the graph structure or add additional information. Thus, the graph DB must to adapt to our needs.

3.2.4. OLTP and OLAP. As we focus on graph analysis, OLAP will be more suited for us than OLTP. Due to the scale of the dataset, we need a system designed for high throughput.

However, the support of OLTP could be useful as the client requests the server for information via an API. Thus, the graph algorithm running on the client can be written in the programming language we prefer. Also, it requires less computational power of the server and is good for locally inspecting the data, e.g., getting information about one specific TLS server.

4. Related work

There exists a variety of surveys on graph DBs, each specialized on different aspects. In 2019, Besta et al. [1] published an exhaustive survey covering general design, data models and organization, data distribution, and transactions and queries of graph DBs. Over 40 databases were compared and classified accordingly. Focusing on scalability and performance, Barpis and Kolovos [6] investigated the performance of graph DBs compared with RDBM Systems in the context of model-driven engineering. In 2015, Kaliyar [7] published a brief overview over popular graph DBs, such as Neo4j, DEX, HyperGraphDB, and Trinity, concentrating on data modeling. A benchmark of graph algorithms performed on different graph DBs was published in 2013 by McColl et al. [2]. In 2018, Patil et al. [8] reviewed both implemented and theoretical computational techniques for graph DBs.

So far, no survey compares the above-mentioned graph databases Neo4j, Apache Giraph, and MsGraphEngine

doi: 10.2313/NET-2020-11-1_04

based on the requirements stated in Section 3.2, i.e, the special case of large-scale data sets with a possibly changing data structure, the need for fast loading, and OLAP.

5. Analysis of the Graph Databases

In the following, we introduce the three graph DBs Neo4j, Apache Giraph, and MsGraphEngine.

5.1. Neo4j

The database Neo4j is one of, if not the most popular Graph DB, hence, why we have chosen this DB for comparison in our paper. According to its creators, Neo4j benefits from their first-mover advantage and a thriving community. It claims to scale well, to support highly parallel graph processing and to have high loading speeds [9]. It is implemented in Java.

5.1.1. Storage Architecture. Neo4j stores edges, nodes, and properties in *records* of fixed size. These are addressable, contiguous blocks of memory. Records storing a node or an edge are called *node records* and *edge records*, respectively. Properties are stored in *property records* and can hold up to four properties. For large property values, there is an additional dynamic store [1].

These three types of records are used to store a property graph (see Section 2.1) as depicted in Figure 2. Let n1 and n2 denote two nodes, which are connected by the edge e2. Both nodes have further connections to nodes that are not explicitly depicted for simplicity.

Node records contain a reference to the edge record of the first edge that is attached to the node. In our example the record of n1 points to the record e2. It also stores labels and a pointer to a property record [1]. For administration purposes, the node record keeps flags, which are omitted in the figure.

Edge records contain pointer to the node records of the start and end node. It stores one label and a pointer to a property record. Each edge record belongs to the adjacency lists (AL) of the start and end node. These adjacency lists are implemented as doubly linked lists [1]. Hence, an edge record also stores forward and back pointers for both ALs. Due to these linked lists, Neo4j supports *index-free* adjacency., i.e., no special index structure is required for querying the adjacencies of a node.

Due to the index-free adjacency, Neo4j deals well with large graph sizes. No index structure needs to be kept up-to-date.

The storage is *disk-based*, meaning that not the entire graph is kept in the RAM [10].

5.1.2. Data Distribution and Types of Transactions. Neo4j does not support distributed data across servers. The data may be replicated but cannot be segmented [1].

It fully supports ACID in all transactions [10] and can be used for both OLAP and OLTP [1].

5.2. Apache Giraph

Apache Giraph is an open-source implementation of Google's Graph Analysis System Pregel. It was famously

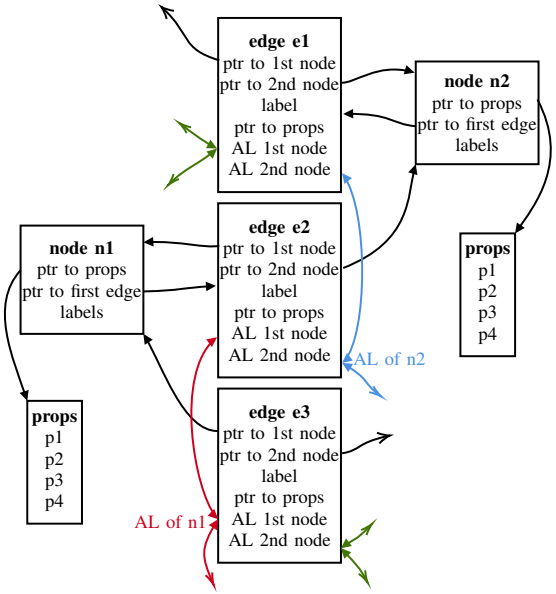

Figure 2: Storage architecture of Neo4j

used by Facebook to process a large-scale Graph with a Billion nodes and more than a Trillion edges within minutes [11], making it an interesting candidate for us.

Giraph is much more than just a Graph storage system. It was designed to perform exhaustive graph analysis on distributed servers and to interface with a variety of services in Apache's framework *Hadoop*; see [12].

5.2.1. Distributed Computing. The system Hadoop by Apache is the basis for Giraph's capability to deal with big data. It is an open-source implementation of the *MapReduce* programming model developed by Google. Its core idea is to split the computation in a *Map* and a *Reduce* function that both accept and generate key/value pairs. In the first step, Map generates a set of intermediate values with intermediate keys. The intermediate values are grouped by the intermediate keys. In the second step, the Reduce function reduces each group to a more simple value. Both steps can be parallelized on multiple machines [13]. Giraph uses a variant of MapReduce, also called *map-only* [12], meaning that there are no Reduce steps between Maps.

However, graph algorithms are often hard to parallelize. Therefore, the algorithms are executed in *super-steps*. Each super-step is parallelized on the machines. Once the machines have finished the super-step, the next one is started. This is also known as the *Bulk Synchronous Parallel* programming model. The programming is *vertex based*, meaning that nodes are the fundamental entities of processing. In between two super-steps, nodes can send messages to other nodes [10].

Hadoop manages the exchange of messages, code, and other information between the many workers and the master that controls the computation.

5.2.2. Storage Architecture. Nodes, edges, and messages are stored as Java objects and are serialized to byte arrays. To avoid memory management issues and to minimize Java's expensive garbage collection, Giraph performs parts of the memory management on its own [12].

Nodes are stored in objects instantiated by the class *Vertex*. Each vertex object has a unique ID, a value (a generic), and a set of outgoing edges; see [14].

The outgoing edges of a node are stored in an object that can be an instance of different classes implementing the *OutEdges* interface. According to the use-case, the user can choose between different implementations. In case one wants to efficiently iterate over all outgoing edges, a byte array or a linked list may be most efficient. For random access, on the other hand, a hash map for indexing the edges may be more suitable. The user is also allowed to create own implementations of the OutEdges interface [12].

Objects of the class *Edge* store a pointer to the target node and a value that is a generic.

Without different specification, Giraph stores the entire graph in the RAM of the distributed machines and is, thus, *memory-based* [15].

5.2.3. Types of Transactions. Due to its nature, Giraph supports OLAP but is not suited for OLTP. It is optimized for graph analysis rather than for storing graphs. Hence, there is no point in investigating the compliancy with ACID.

5.3. MsGraphEngine (Trinity)

Microsoft's Graph Engine Trinity is a distributed graph processing framework based on a globally addressable key-value store. It can be customized using the Trinity Specification Language (TSL) [16].

5.3.1. Storage Architecture. The core element of Trinity is a key-value store, where values are addressable system-wide across several machines. The values (called *cells*) are binary blobs of variable size that can hold arbitrary data. Their usage is specified via the Trinity Specification Language. Thus, the user can customize the graph schema and adapt to the needs. Cells may hold node or edge data or associated rich data [17]. A communication framework passes messages between machines, which can also be adjusted with TSL.

The key/value store is partitioned in trunks of fixed size with individual hash tables for addressing. Each machine keeps multiple trunks. The entire store is loaded in the RAM [17].

5.3.2. Distributed Query Execution. In Trinity, there exist three types of workers: slaves, proxies, and a client. Slaves store trunks and process and answer incoming messages. Proxies only deal with messages and do not store graph data. They can be used for gathering and relaying results from slaves to the client. The client is a worker that interfaces with the user [17].

The programming model is vertex centric. As with Giraph, consecutive super-steps are performed. At each step, a node receives messages from a fixed set of nodes, which have been sent in the previous super-step [17].

5.3.3. Types of Transactions. There is no inherent mechanism ensuring ACID. However, Trinity provides measures such as spin locks for each storage cell to guarantee consistency [17].

It can be used for both OLAP and OLTP.

6. Comparison of the Graph Databases

In this section, we compare our candidates based upon our requirements.

Scalability. All three graph DB discussed above are scalable in size in terms of their graph model implementation. There are no index structures that would become much more inefficient with growing size. However, with bigger the graph size, more storage space and computational power is needed. This is where Giraph and Trinity outperform Neo4j. They are specifically designed to run distributed graph analysis taking advantage of the RAM memory and the computing power of several machines. Neo4j can run distributed, but with copies of the data set, improving availability rather than storage capacity. It uses disc memory, which is cheaper and persistent but not as fast as the in-memory storage of Giraph and Trinity. The performance of the latter is throttled by the network speed of the cluster [17].

Adaptability. Neo4j is designed for storing labeled property graphs, i.e., its graph model is fixed. This can lead to both memory overhead and design constraints if the data does not suit the model. In Giraph the user has a certain freedom by choosing generic types, selecting different implementations (see OutEdges) and writing own classes. Trinity has even more degrees of freedom. Its language TSL lets the user define the entire graph schema and communication protocols. Thus, there is less storage overhead than in Giraph [17]. However, the user has to manually set up the database in TSL. Additionally, Giraph stores Java runtime-objects including their meta-data, causing storage overhead [17]. Again, Trinity or Giraph may be the better choice.

Note that Trinity has a more restrictive programming model than Giraph. Between super-steps, nodes can send messages to arbitrary nodes in Giraph, whereas the receivers are fixed in Trinity. However, this increases the efficiency of Trinity's inter-machine communication [17].

ACID, OLAP, and OLTP. Giraph and Trinity do not have built-in support for ACID, whereas Neo4j does. We can use Neo4j for both OLTP and OLAP. However, the benefit of OLAP may be limited due to the lack of processing power of a single machine. Trinity is designed to handle both use-cases well. It is a hybrid of Neo4j and Giraph, so to speak, as Giraph is only useful with OLAP.

Input Loading. Solely based on the storage architecture, it is hard to estimate, which of the three contestants will have the fastest loading time for our data set.

7. Conclusion and future work

We have compared the graph databases Neo4j, Apache Giraph, and MsGraphEngine (Trinity) in the context of storing and analyzing TLS scans. In summary, Neo4j is most suited if we want to store and locally query a smaller graph without much analysis. Conversely, Apache Giraph is not suited for storing graphs but specialized for distributed and parallel analysis of big data. Trinity serves both and can be adjusted as needed. Future work could include implementing and benchmarking graph algorithms on the three DBs to be sure about the best choice.

doi: 10.2313/NET-2020-11-1_04

References

[1] M. Besta, E. K. Peter, R. Gerstenberger, M. Fischer, M. Podstawski, C. Barthels, G. Alonso, and T. Hoefler, "Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries," *ArXiv*, vol. abs/1910.09017, 2019.

[2] R. McColl, D. Ediger, J. Poovey, D. Campbell, and D. A. Bader, "A Brief Study of Open Source Graph Databases," *ArXiv*, vol. abs/1309.2675, 2013.

[3] R. Angles, "The Property Graph Database Model," in *AMW*, 2018.

[4] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Comput. Surv.*, vol. 15, no. 4, p. 287–317, Dec. 1983.

[5] O. Gasser and P. Sattler, "goscanner," https://github.com/tumi8/goscanner, 2020, [Online; accessed 03-May-2020].

[6] K. Barmpis and D. S. Kolovos, "Evaluation of Contemporary Graph Databases for Efficient Persistence of Large-Scale Models," *Journal of Object Technology*, vol. 13, pp. 3: 1–26, 2014.

[7] R. k. Kaliyar, "Graph databases: A survey," *International Conference on Computing, Communication & Automation*, 2015.

[8] N. Patil, P. Kiran, N. Kavya, and K. Patel, "A Survey on Graph Database Management Techniques for Huge Unstructured Data," *International Journal of Electrical and Computer Engineering*, vol. 81, pp. 1140–1149, 04 2018.

[9] N. Inc., "Top Ten Reasons for Choosing Neo4j," https://neo4j.com/top-ten-reasons/, 2020, [Online; accessed 20-March-2020].

[10] S. Sakr, F. Orakzai, I. Abdelaziz, and Z. Khayyat, *Large-Scale Graph Processing Using Apache Giraph*. Springer International Publishing, 2017.

[11] Facebook, "Scaling Apache Giraph to a trillion edges," https://www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-a-trillion-edges/10151617006153920/, 2013, [Online; accessed 04-May-2020].

[12] C. Martella, *Practical graph analytics with Apache Giraph*. Apress, 2015.

[13] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, 2004, pp. 137–150.

[14] T. A. S. Foundation, "Apache Giraph Parent 1.3.0-SNAPSHOT API," http://giraph.apache.org/apidocs, 2019, [Online; accessed 21-March-2020].

[15] S. Heidari, Y. Simmhan, R. Calheiros, and R. Buyya, "Scalable Graph Processing Frameworks: A Taxonomy and Open Challenges," *ACM Computing Surveys*, vol. 51, pp. 1–53, 06 2018.

[16] Microsoft, "Graph Engine," https://www.graphengine.io/, 2017, [Online; accessed 21-March-2020].

[17] B. Shao, H. Wang, and Y. Li, "Trinity: A Distributed Graph Engine on a Memory Cloud," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 505–516.

doi: 10.2313/NET-2020-11-1_04