Graph Patterns Indexes: their Storage and Retrieval

Jaroslav Pokorný
Faculty of Mathematics and Physics
Charles University, Prague
Czech Republic
pokorny@ksi.mff.cuni.cz

Michal Valenta
Faculty of Information Technologies
Czech Technical University, Prague
Czech Republic
valenta@fit.cvut.cz

Jaroslav Ramba
Faculty of Information Technologies
Czech Technical University, Prague
Czech Republic
rambajar@fit.cvut.cz

ABSTRACT

We propose a method for indexing graph patterns within a graph database. A graph database consists of a labelled property graph. The index is organized in a hash table and stored in the different database than the database graph. The method enables to create, use, and update indexes that are used to speed-up the process of matching graph patterns. The prototype implementing the method was analyzed for Neo4j graph database engine. Pattern indexes are stored in the embedded database MapDB. Three graph databases are used for experiments with pattern indexes. The paper provides a comparison between queries with and without using indexes.

CCS CONCEPTS

• **Information Systems → Data Management Systems**; *Query Languages*; • **Information Storage Systems → Record Storage Systems**; Index file organization

KEYWORDS

graph databases, graph querying, indexing graph patterns, Neo4j

ACM Reference format:

J. Pokorný, M. Valenta, and J. Ramba. 2018. Graph Patterns Indexes: their Storage and Retrieval. In *Proceedings of the 19th International Conference on Information Integration and Web-based Applications & Services, (iiWAS'2018). Yogykarta, Indonesia, November 2018*, 5 pages. https://doi.org/10.1145/3282373.3282374

1 INTRODUCTION

A need to efficiently store and process highly connected data has resulted in a renewed interest in the database category known as the graph database (GDB). A GDB is based on graph theory, uses nodes, properties, and edges. A GDB considers relationships between entities as a core aspect of its data model. The model is built on the idea that though there is a value in discrete information about entities, there is more value in relationships between them. A node is an entity such as a person, movie, object, or relevant piece of data and an edge represents the relationship between two nodes, e.g. friendship, who-buys-what, etc. Nodes and edges may have various properties (key/value pairs) attached to them.

Relaxing usual DBMS features, a *native GDB* can be any storage solution where connected elements are linked together without using an index (so-called *index-free adjacency*) [10]. A GDB can contain one (large) graph *G* or a collection of small to medium size graphs. The former includes, e.g., graphs of social networks, Semantic Web, geographical databases, the latter is especially used in scientific domains such as bioinformatics and chemistry or datasets like DBLP. The query processing over a graph collection involves, e.g., finding all graphs in the collection that are similar to or contain similar subgraphs to a query graph. We focus on the first category of GDBs in this paper**.**

Similarly, to traditional databases, we use the notion of a *graph database management system* (GDBMS). GDBMSs proved to be very effective and suitable for many data handling use cases. For example, specifying a graph pattern and a set of starting points, it is possible to reach an excellent performance for local reads by traversing the graph starting from several root nodes, and collecting and aggregating information from nodes and edges. On the other hand, GDBMSs have their limitations [8], e.g., they are usually not consistent, since have very restricted tools to ensure a consistency. Graph querying is a key issue in any graph-based application. Its functionalities include two natural operations: *graph pattern matching* and *graph navigation* [1]. Specifically, a *pattern match query* searches over a graph *G* to look for the existence of a pattern graph in *G*. Pattern matching is a fundamental processing requirement for graph data applications. It occurs in application scenarios like recommender systems, analyzing hyperlinks in WWW, complex object identification, software plagiarism detection, or traffic route planning. Graph navigation includes, e.g., shortest path queries and reachability queries. e.g., to find whether a concept subsumes another one in an ontological database.

An effective implementation of each DBMS highly depends on existence and usage of indexes. Nowadays, some effective indexes for nodes, edges, and properties (*value indexes*) already exist in GDBMS implementations, while *structure-based indexes*, which may be very useful for subgraph queries and for relationship-based integrity constraints checking, are yet rather the subject of research as it is described in Section 2. Particularly, there already exist indexing methods for various kinds of graph pattern matching (see, e.g., work [5]). However, most subgraph isomorphism algorithms are based on a backtracking method, which computes the solutions, by incrementally enumerating and verifying candidates for all nodes in a query graph.

The main aim of the paper is to analyze the possibilities of structure-based graph indexing in GDBs and subsequent design and implementation of a graph index for the selected GDBMS. We focus on Neo4j[1] and its possibilities to express graph patterns and their implementation through the web service REST[2]. Performance testing will be done on several typical patterns on an appropriate data sample. Neo4j is an open-source native GDBMS that offers functionality similar to traditional RDBMSs such as full transactional support, a declarative query language Cypher[3], availability and scalability through its distributed version. Our goal is to extend the Cypher with new functionality supporting more to index various graph patterns.

The method for indexing graph patterns, including operations to create an index, query using an index and update an index, is implemented for Neo4j. The major benefit of Neo4j is its intuitive way of modelling and querying graph-shaped data. Internally, Neo4j stores edges as double linked lists. Properties are stored separately, referencing the nodes with corresponding properties.

It is also important to mention that all existing indexes must be updated when each of mentioned DML operations is applied to a database. It is done so within the same transaction that executed a DML operation. If a transaction is successfully committed, indexes will be updated. If a transaction is roll-backed, indexes will remain in the same state as they were the transaction was initialized.

The rest of the paper is organized as follows. In Section 2, we introduce basic notions of GDBs and summarize some related works concerning graph pattern matching and indexing. In Section 3, we deal with indexing graph patterns and their storing in a database way. Related experiments are described and discussed in Section 4. A comparison with another implementation of pattern indexes is presented. Section 5 gives the conclusions.

2 BACKGROUND AND RELATED WORKS

Supposing a set of nodes V and a multiset of edges E, where $E \subseteq V \times V$, a *multigraph* G is an ordered pair (V, E). In the world of GDBs we will use a (*labelled*) *property multigraph mode*l whose basic constructs include: entities (V), relationships (E) having a direction, start node, and end node, and identifiers, properties (attributes), and labels (types). Entities and relationships can have any number of properties, nodes and edges can be tagged with labels. A unique identifier (`Id`) defines both nodes and edges. Properties are of key-value pairs, i.e. only single-valued properties are considered. In graph-theoretic notions, we also talk about *labelled and directed attributed multigraphs* in this case.

Graph pattern matching reduces i.e., we retrieve in G all subgraphs, which are isomorphic to the query graph. The problem is NP-complete [15], meaning that this querying is intractable for large graphs in the worst-case. Subgraph isomorphism is the most basic problem of graph pattern matching. Despite NP-completeness, many algorithms have been proposed in order to solve it in practice, e.g., [15] or more recent VF3 [2] and STW [14].

In general, GDBMs use various graph analytics algorithms supporting with finding graph patterns, e.g., connected components, single-source-shortest-paths, community detection, triangle counting, etc. Triangle counting is used heavily in social network analysis. It provides a measure of clustering in the graph data, which is useful for finding communities, and measuring the cohesiveness of local communities in social network websites like LinkedIn or Facebook. In Twitter, three accounts, which follow each other, are regarded as a triangle.

2.1 Indexing in GDBs

Indexing is used in GDBs in many different contexts. Due to the existence of properties values in a GDB, graph indexes are of two kinds, in principle: *property-aware* and *structure-aware*. They occur in GDBMS in various forms from a fulltext querying support over indexing nodes, edges, and property types/values to indexes based on indexing non-trivial subgraphs. Some GDBMSs use search engines like Apache Lucene[4] as index backend.

2.1.1 Value-based Indexing. We present indexing in three native GDBMSs. The Cypher language of Neo4j enables to create indexes on one or more properties for all nodes that have a given label. Sparksee[5] uses B+-trees and compressed bitmap indexes to store nodes and edges with their properties. Titan[6] supports two different kinds of indexing to speed-up query processing: graph indexes and node-centric indexes. Graph indexes allow efficient retrieval of nodes or edges by their properties for sufficiently selective conditions. They can be composite or mixed. The former are very fast and efficient but limited to equality lookups for a particular, previously defined combination of property keys. Mixed indexes can be used for lookups on any combination of indexed keys and support multiple conditions predicated in addition to equality depending on the backing index store. Node-centric indexes are local index structures built individually per node. In large graphs, nodes can have thousands of incident edges.

Some GDBMs are multi-model. For example, OrientDB[7] supports key-value, document, and graph data model. It uses even five classes of indexing algorithms: SB-Tree [7], HashIndex, Auto Sharding Index, and indexing based on the Lucene Engine (for fulltext and spatial data). We remind, that the SB-tree is a variant of the B-tree optimized for sequential multi-page disk access during long range retrievals. Mpinda et al. [6] compare indexing for Neo4j and OrientDB. Neo4j's queries retrieval has less performance than OrientDB especially on several number of nodes. Therefore, having a large number of nodes, whereby indexing techniques are necessary, it is recommended to use OrientDB.

2.1.2 Structure-based Indexing. The design principle behind a structure-based index is to extract and index structural properties of database graphs, typically at insertion time, and use them to filter the search space rapidly in response to a query. Previous works

[1] https://neo4j.com/ (retrieved on 8. 10. 2018)
[2] https://restfulapi.net/ (retrieved on 8. 10. 2018)
[3] http://neo4j.com/developer/cypher-query-language/ (retrieved on 8. 10. 2018)
[4] https://lucene.apache.org/core/ (retrieved on 8. 10. 2018)
[5] http://www.sparsity-technologies.com/ (retrieved on 8. 10. 2018)
[6] http://titan.thinkaurelius.com/ (retrieved on 8. 10. 2018)
[7] http:// www.orientdb.com/ (retrieved on 8. 10. 2018)

have mainly focused on mining "good" substructure features for indexing. A good feature set improves the filtering power by reducing the number of candidate subgraphs, which leads to a reduction in the number of subgraph isomorphism tests in the verification step. Subtree features are also mined for indexing, and they are less time-consuming to be mined in comparison with mode general subgraph features. Many methods take a path as the basic indexing unit. For example, the SPath algorithm [18] is centered on a local path-based indexing technique for nodes in *G* and transforms a query graph into a set of shortest paths in order to process a query. The key problem of SPath lies in the inefficiency of offline processing. Furthermore, it is quite expensive to perform updates over these indexes. Srinivasa distinguishes in [13] three types of structure-based indexes: path-based index, subgraph-based index, and spectral methods. Spectral methods for indexing GDB employ concepts from spectral graph theory, where features of database graph(s) are represented as vectors in a hypothetical space.

Unfortunately, no index structure is enable to support all kinds of substructure features. Authors of [17] propose a Lindex, a graph index, which indexes subgraphs contained in database graphs. An introduction to graph substructure search, approximate substructure search and their related graph indexing techniques, particularly feature-based graph indexing can be found in [16]. In [18], the authors introduce a structure-aware and attribute-aware index to process approximate graph matching in a property graph.

3 INDEXING GRAPH PATTERNS

A wide variety of graph patterns can be found across different GDBs. Graph patterns have different information value that is based on type of data stored within a database and use cases that involve these graph patterns. One of widely used graph patterns, defined as GP = (V_P, E_P), where V_P = {a; b; c}, E_P = {(a; b); (b; c); (c; a)}, $V_P \subseteq V$, and $E_P \subseteq E$, is called a *triangle*. In Cypher, a triangle can be expressed in a few different ways, but preferably as

(a)-[r1]-(b)-[r2]-(c)-[r3]-(a)

i.e., triangle patterns look for three nodes adjacent to each other regardless of edge orientation. That is, a direction can be ignored at query time in Cypher, i.e. the database graph behind can be handled as bidirectional. To retrieve all triangles from a GDB, we can evaluate the Cypher query:

MATCH (a)--(b)--(c)--(a) RETURN a,b,c

The complexity of a triangle query is O(|V| * (|V_P| + |E_P|)). The associated algorithm finds the pattern from each node in the depth-first search way [4]. The problem arises when we focus only on structural features of the graph and want, e.g., all triangles of people with their friendships. Then a structure-based index can be helpful.

A *graph pattern index* is a data structure that stores pointers that reference graph pattern units within the GDB. Indexes can be either stored in the same database as the actual data or in any external data store. We use here the latter variant based on the work [10], where the embedded database MapDB[8] was used.

In the graph pattern index, we store not all the data, but only node and edge `Ids` that give us a unique description of the pattern.

[8] http://www.mapdb.org/ (retrieved on 8. 10. 2018)

We are using both node and edge `Ids` in order to support multigraphs structure indexing. To store the index, we have chosen a hash table. A single table that will contain the structure shown in Table 3.1 will represent each indexed pattern. The key (line) of the hash table will represent the indexed pattern (list of nodes and list of edges). The first part of the key is the node `Id` set in ascending order and separated by one underscore. Then it follows the separation of two underscores between the node list and edge list. The second part of the key is a list of edge `Ids` that are as well as the nodes ranked ascending. The values (pointers to the GDB) are omitted in the table.

Table 3.1: Example of a triangle pattern

triangle	
key	*value*
_nodeId11_nodeId27_nodeId34__relId1009_relId4555_relId4565_	
_nodeId18_nodeId24_nodeId65__relId1203_relId2743_relId3001_	
_nodeId48_nodeId52_nodeId87__relId1078_relId2225_relId4563_	

4 IMPLEMENTATION AND EXPERIMENTS

Finding the triangles is the most typical example that we can illustrate to show the effectiveness of our index implementation. In real life, we will meet this query in social networks where we are recommended to be friends with people who are friends of our friends. In this way, we have a certain probability that we are also familiar with them or in the future, we will familiarize with them and close the triangle of friendship [4]. Such network analysis is a building block for any recommending algorithms that are applied over the given network. Complexity of network, clustering coefficient, the total number of triangles or the average probability for new patterns (the emergence of a new friendship edge) can be used, e.g., to identify the largest network communities [12].

All bellow presented results are achieved by measuring within a test environment provided by GraphAware framework[9] on Neo4j 2.2. The following configuration is used when performing measurements: 2.5GHz dual-core Intel Core i5, 8GB 1600MHz memory DDR3, Intel Iris 1024 MB, 256 GB SSD, OS X 10.9.4.

To achieve the most accurate results, measurements are always performed multiple times and their results are averaged. Measuring is done for all cache types provided by Neo4j, i.e., No-cache (Neo4j instance with no caching), Low-level cache and High-level cache.

4.1 Querying index

We first tested query performance. For this purpose, we generated an Erdos-Renyi random graph (see, e.g., [3]) *G* as a 100,000-node community network with 500,000 edges. The results of selective queries are always the same set containing all the triangles in *G*. All the queries were evaluated without and with index. Graphical presentation of the measurement is in Fig. 1.

We can see the query time with index is more than 100times faster on each cache option. There were 157 triangle shapes in the

[9] https://github.com/graphaware/neo4j-framework (retrieved on 8. 10. 2018)

database. The index creation times were 78s for High-level cache, 3min for Low-level cache, and nearly 4 hours with No-cache option. The index size was 77 KB. Only the largest triangle database measurement is presented here, measurements on scaling the database can be found in [10]. Other experiments were done on Cineasts and Transaction databases. A simplified schema of Cineasts is in Fig. 2. Its GBD size was 72,000 nodes and 106,651 edges. Measurements are to see in Fig. 3.

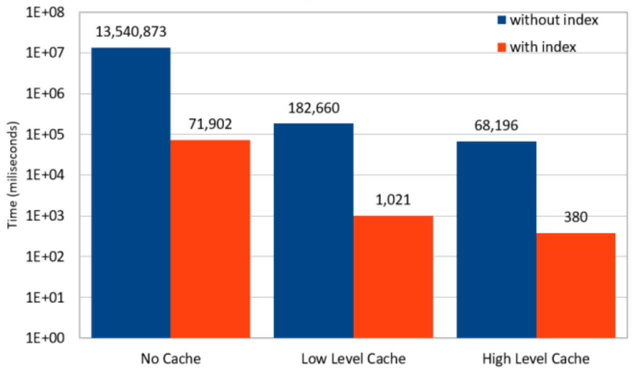

Figure 1: Social graph, triangle index - query time

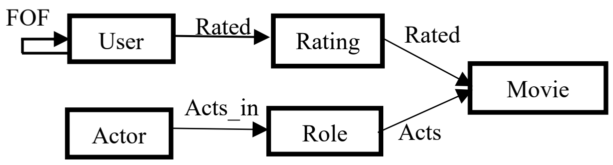

Figure 2: Cineasts database schema

The measured query specified triangles with a forth edge from the initial node (*funnel shape index*).

MATCH (a)--(b)--(c)--(a)--(d) RETURN a,b,c,d

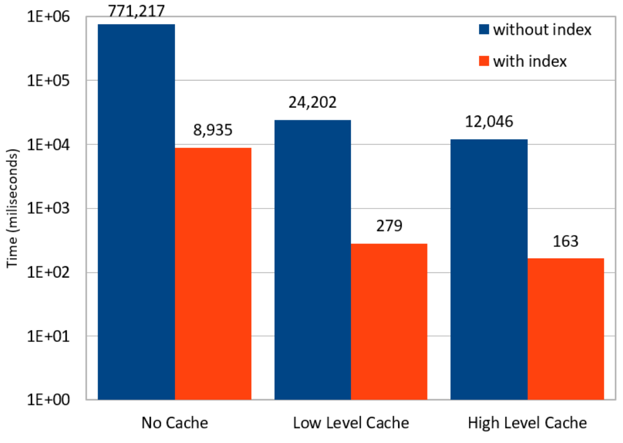

Figure 3: Cineasts database, funnel shape index - query time

We can see that funnel shape index on Cineasts database is yet more effective for query operation than in the case of triangle shape index on the Social graph. Construction times for different cache options were 17 s, 26 s, and 14 min, respectively. There were 86 different shapes in the database and the index size was 48 KB.

Transaction database. Its nodes represent credit cards and edges their mutual transactions.

Each edge is randomly created with a predetermined probability. The GBD size was 10,000 nodes and 99,970 edges. The measured query specified diamonds with a diagonal edge (*rhombus shape index*). Measurements are depicted in Fig. 4.

MATCH (a)--[e]--(b)--[f]--(c)--[g]--(a)--[h]--(d)--[i]--(b)
RETURN a,b,c,d

Again, rhombus-shape index provides more than 100times speed-up for querying. Index creation times for individual cache options were 73 s, 2 min, and 1,5 hour, respectively. There were 70 different shapes in the database and index size was 74 KB. The complete analysis and design decisions can be found in [10].

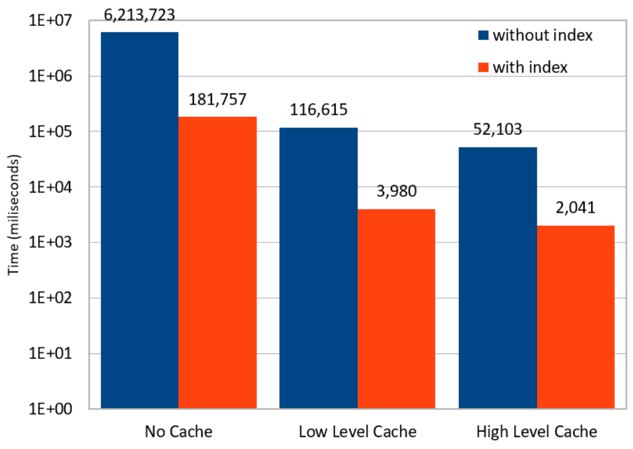

Figure 4: Transaction database, rhombus index - query time

4.2 DML operations and comparison with another implementation

Index must be updated together with DML operations on base data. In our implementation, update of index is done in the same transaction as a DML statement. We did measurements on all three databases for the following DML operations: creating index, creating a relationship, deleting a relationship, and deleting a node. All these DML operations may affect existing indexes. Measurements are done using all cache variants (see above).

The last mode is the most suitable for our purpose and, not surprisingly, it provides the best performance. Therefore, we present only High-level cache measurements bellow. Moreover, we compared our prototype implementation with another structure-based indexing method described in [10], where three GDBs were used: social graph, music database, and transactions database.

The music database stores data about artists, information about tracks they recorded, and labels that released these records. Measurements are presented in Tables 4.1-4.3. Table captions contain information about particular graph sizes used for experiments. The method described in this paper is referred as

Hash/map index, the method described in [10] is referred as Tree index, the column named Simple query represents no index usage.

Table 4.1: Social graph, triangle index, 10,000 nodes, and 50,000 relationships

Operation	Simple query [ms]	Tree index [ms]	Hash/map index [ms]
create index	-	5 242	4 985
query index	6 881	403	363
create rel.	25	29	35
delete rel.	146	157	163
del. n. with rel.	228	277	924

Table 4.2: Music database, funnel index, 12,000 nodes, and 50,000 relationships

Operation	Simple query [ms]	Tree index [ms]	Hash/map index [ms]
create index	-	36 238	47 620
query index	41 794	1 243	2 028
create rel.	29	64	66
delete rel.	257	283	309
del. n. with rel.	375	459	3 655

Table 4.3: Transaction database, rhombus index, 10,000 nodes, and 100,000 relationships

Operation	Simple query [ms]	Tree index [ms]	Hash/map index [ms]
create index	-	5 242	4 985
query index	6 881	403	363
create rel.	25	29	35
delete rel.	146	157	163
del. n. with rel.	228	277	924

Our prototype is tightly related to Neo4j. Querying results are (not surprisingly) very affected by cache method (see Fig. 1, 3 and 5). Long times for queries without indexes show that evaluation of such kind of queries in Neo4j is not optimized at all. Our indexes improved the response time appr. 100times, but queries using indexes in our implementation are not realized only on them. Such kind of optimizer decision, as it is usual in relational databases, would need a lot of work in Neo4j Cypher evaluation engine, and of course, it would bring much better response. On the other hand, index size is small compared to data itself, and their maintenance along with DML operation on data (see tables 4.1, 4.2 and 4.3) is pretty cheap compared with the query speedup even in our simple prototype implementation. These are reasons why we think our approach is promising and suitable for further research especially with conjunction of optimizer indexes usage improvement.

5 CONCLUSIONS

We designed a prototype for indexing patterns for graphs stored in a Neo4j GDBMS. We tested the Neo4j query to trace the indexed patterns and then implemented the proposed prototype. In the selected databases, we showed examples of queries to which the index would be suitable. The results confirmed that for the cases we have designed, the index very quickly speeds up the querying and not too much influences the INSERT/DELETE operations of the node and edges in the GDB. We also compared the prototype of the index to the prototype referred in [9], where the index is built as a tree structure stored in Neo4j. The index stored directly in the GDB influences less the node DELETE operation than the index proposed here. In the index search, the prototypes do not differ significantly. However, if there were any deviations in querying the index, then the number of patterns after the reduction that the index searched was always decisive. The effectiveness of reducing the prototype always depended on the specific content of the GDB, and therefore we cannot say exactly, what is better and what is worse.

ACKNOWLEDGMENTS

This work was partially supported by the Charles Uni. project Q48.

REFERENCES

[1] R. Angles, M. Arenas, P. Barcelo, A. Hogan, A. Reutter, and D. Vrgoc. 2017. Foundations of Modern Query Languages for Graph Databases. arXiv:1610.06264

[2] V. Carletti, P. Foggia, A. Saggese, and M. Vento. 2017. Introducing vf3: A new algorithm for subgraph isomorphism. In: *Proceedings of Int. Workshop on Graph-Based Representations in Pattern Recognition*. Springer (2017) 128-139

[3] R. Diestel. 2016. Graph Theory, Springer GTM 173, 5th ed.

[4] D. Easley, J. Kleinberg. 2010. Networks, Crowds, and Markets: Reasoning about a Highly Connected World. 1st edition, Cambridge Univ. Press, U.K.

[5] J. Lee, W.S. Han, R. Kasperovics, and J.H. Lee. 2012. An in-depth comparison of subgraph isomorphism algorithms in graph databases. In *Proceedings of VLDB*, 133-144

[6] S.A.T. Mpinda, L.C. Ferreira, M.X. Ribeiro, and M.T.P. Santos. 2015. Evaluation of Graph Databases Performance through Indexing Techniques. *Int. Journal of Artificial Intelligence & Applications* (IJAIA) Vol. 6, No. 5, 87-98.

[7] P.E. O'Neil. 1992. The SB-tree: An Index-Sequential Structure for High-Performance Sequential Access. *Informatica*, 29 (1992) 241-265

[8] J. Pokorný. 2015. Graph Databases: Their Power and Limitations. In *Proc. of 14th Int. Conf. on Computer Information Systems and Industrial Management Applications* (CISIM 2015), K. Saeed and W. Homenda (Eds.), LNCS 9339, Springer, (2015) 58-69

[9] J. Pokorný, M. Valenta, and M. Troup. 2018. Indexing Patterns in Graph Databases. In *Proceeding of the* 7th Int. Conference on Data Science, Technology, and Applications (DATA 2018), Scitepress, (2018), 313-321.

[10] J. Ramba. 2015. Indexing graph structures in graph database machine Neo4j II. Master's thesis, Czech TU in Prague, Faculty of Information technology.

[11] I. Robinson, J. Webber, and E. Eifrém. Graph Databases. O'Reilly Media (2013).

[12] B. Serroura, A. Arenasb, S. and Gómez. 2011. Detecting communities of triangles in complex networks using spectral optimization. Journal Computer Communications, Vol. 34, Issue 5 (2011) 629-634.

[13] S. Srinivasa. 2012. Data, Storage and Index Models for Graph Databases. In: Graph Data Management: Techniques and Applications, Sh. Sakr and E. Pardede (Eds), IGI Global, 47-70.

[14] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. 2012. Efficient subgraph matching on billion node graphs. In *Proceedings of PVLDB* 5 (2012) 788-799.

[15] J.R. Ullmann. 1976. An algorithm for subgraph isomorphism. *J. ACM*, 23(1), 1976.

[17] X. Yan, J. Han, J. 2010. Graph Indexing. Chapter 5 in C.C. Aggarwal and H. Wang (eds.), Managing and Mining Graph Data, Advances in Database Systems 40, Springer, 161-180.

[18] D. Yuan, P. Mitra. 2013. Lindex: a lattice-based index for graph databases. *The VLDB Journal* 22 (2013) 229–252.

[19] L. Zhu, W.K. Ng, and J. Cheng. 2011. Structure and attribute index for approximate graph matching in large graphs. *Inf. Syst.* 36(6) (2011) 958-972.