Graph Pattern Index for Neo4j Graph Databases

Jaroslav Pokorný[1](✉), Michal Valenta[2], and Martin Troup[2]

[1] Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic
pokorny@ksi.mff.cuni.cz

[2] Faculty of Information Technology, Czech Technical University, Prague, Czech Republic
{valenta, troupmar}@fit.cvut.cz

Abstract. Nowadays graphs have become very popular in domains like social media analytics, healthcare, natural sciences, BI, networking, etc. Graph databases (GDB) allow simple and rapid retrieval of complex graph structures that are difficult to model in traditional information systems based on a relational DBMS. GDB are designed to exploit relationships in data, which means they can uncover patterns difficult to detect using traditional methods. We introduce a new method for indexing graph patterns within a GDB modelled as a labelled property graph. The index is based on so called graph pattern trees of variations and stored in the same database where the database graph. The method is implemented for Neo4j GDB engine and analysed on three graph datasets. It enables to create, use and update indexes that are used to speed-up the process of matching graph patterns. The paper provides details of the implementation, experiments, and a comparison between queries with and without using indexes.

Keywords: Graph databases · Indexing patterns · Graph pattern · Graph database schema · Neo4j

1 Introduction

A *graph database* (GDB) is a database that uses the graph structure to store and retrieve data. A GDB embraces relationships as a core aspect of its data model. The model is built on the idea that even though there is value in discrete information about entities, there is even more value in the relationships between them. Relaxing usual DBMS features, a native GDB can be any storage solution where connected elements are linked together without using an index (a property called *index-free adjacency*).

Similarly to traditional databases, we will 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 one or several root nodes, and collecting and aggregating information from nodes and edges. On the other hand, GDBMSs have their limitations [5]. For example, they are usually not consistent, since have very restricted tools to ensure a consistency. This property is typical for NoSQL databases [13], where GDBMs are often included.

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. Thus, the goal of query processing is, e.g., to find all subgraphs of *G* that are the same or similar to the given query graph. We can consider shortest path queries, reachability queries, e.g., to find whether a concept subsumes another one in an ontological database, etc. The query processing over a graph collection involves, e.g., finding all graphs in the collection that are similar to or contain subgraphs similar to a query graph. We focus on the first category of GDBs in this paper.

Graph search occurs in application scenarios, like recommender systems, analyzing the hyperlinks in WWW, complex object identification, software plagiarism detection, or traffic route planning. Gartner[1] believes that over 70% of leading companies will be piloting a GDB by 2018.

One of the most fundamental problems in graph processing is pattern matching. Specifically, a *pattern match query* searches over a *G* to look for the existence of a pattern graph in *G*. This problem can be expressed in the different graph data models as Resource Description Framework, property graphs as well as in the relational model. A property subclass of property graphs can even be modelled using XML documents. We will focus on general property graphs in this paper. Both above mentioned GDB types, however, reduce exact query matching to the subgraph isomorphism problem, which is NP-complete [15], meaning that this querying is intractable for large graphs in the worst-case. In context of Big Data we talk about Big Graphs [6]. Their storage and processing require special technics.

An effective implementation of each DBMS highly depends on the existence and usage of indexes. Nowadays, some effective indexes for nodes and edges already exist in GDB implementations (see, e.g., the evaluation [3] mentioned in Sect. 2.1), 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 Sect. 2. Particularly, there already exist indexing methods for (various kinds of) graph pattern matching, see, e.g., works [1, 14, 16].

In the paper, we focus on Neo4j GDBMS[2] and its possibilities to express an index of graph patterns. Neo4j is an open-source native GDBMS for storing and managing of property graphs, that offers functionality similar to traditional RDBMSs such as a declarative query language Cypher[3], full transaction support, availability, and scalability through its distributed version [10]. Cypher commands use partially SQL syntax and are targeted at ad hoc queries over the graph data. They enable also to create graph

[1] https://www.gartner.com/doc/3100219/making-big-data-normal-graph, last accessed 2018/11/14.

[2] https://neo4j.com/, last accessed 2018/11/14.

[3] http://neo4j.com/developer/cypher-query-language/, last accessed 2018/11/14.

nodes and relationships. Our goal is to extend the Cypher with new functionality supporting more efficient processing graph pattern queries.

Our work is an extension of the paper [8] that introduces a new approach to graph pattern indexing in Neo4j graph database environment. In this paper we present details of implementation, new experiments, and their discussion. The rest of the paper is organized as follows. In Sect. 2 we summarize some related works divided into two categories of graph indexing methods: value-based indexing and structure-based indexing. Section 3 introduces a GDB model based on (labelled) property graphs. We continue with graph pattern indexing and the details of the new method based on so called graph pattern trees of variations. An implementation is described in Sect. 4 and related experiments in Sect. 5. Section 6 gives the conclusion.

2 Background and Related Works

In general, graph systems 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 who follow each other are regarded as a triangle.

One theme in graph querying is graph data mining finding frequent patterns. Frequent graph patterns are subgraphs that are found from a collection of graphs or a single massive graph with a frequency no less than a user-specified support threshold. Subgraph matching operations are heavily used in social network data mining operations.

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: *structure-aware* and *property-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.

2.1 Value-Based Indexing

Authors of [3] compare indexing used in two favourite GDBMSs – Neo4j and OrientDB[4]. The Cypher language of Neo4j enables to create indexes on one or more properties for all nodes that have a given label. OrientDB supports five classes of indexing algorithms: SB-Tree, HashIndex, Auto Sharding Index, and indexing based on the Lucene Engine (for fulltext and spatial data). SB-tree [4] is based on B-Tree with several optimizations related to data insertion and range queries. In Auto Sharding Index (key, value) pairs are stored in a distributed hash table.

[4] http://orientdb.com/orientdb, last accessed 2018/11/14.

Another native GDBMS 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. Node-centric indexes are local index structures built individually per node. In large graphs, nodes can have thousands of incident edges.

2.2 Structure-Based Indexing

The design principle behind a structural 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 have mainly focused on mining "good" substructure features for indexing. A good feature set improves the filtering power by reducing the number of candidate graphs, 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 more general subgraph features. Many methods take a path as the basic indexing unit. For example, the SPath algorithm [19] is centred on a local path-based indexing technique for graph nodes and transforms a query graph into a set of the shortest paths in order to process a query. The work [12] distinguishes three types of structure-based indexes: path-based index, subgraph-based index, and spectral methods.

It is remarkable, that different graph index structures have been used for different kinds of substructure features, but no index structure is enabled to support all kinds of substructure features. Authors of [18] propose a Lindex, a graph index, which indexes subgraphs contained in database graphs. Nodes in Lindex represent key-value pairs where the key is a subgraph in a GDB and the value is a list of database graphs containing the key. Frequent subgraphs are used for indexing in gIndex [16]. An introduction to graph substructure search, approximate substructure search and their related graph indexing techniques, particularly feature-based graph indexing can be found in [17]. In [20], the authors introduce a structure-aware and attribute-aware index to process approximate graph matching in a property graph.

A detailed discussion of different types of graph queries and a different mechanism for indexing and querying GDBs can be found in [11].

3 Modelling of Graph Databases

Although GDBMS can be based on various graph types, we will use a (*labelled*) *property graph model* whose basic constructs include:

- entities (nodes),
- properties (attributes),
- labels (types),

5 http://www.sparsity-technologies.com/, last accessed 2018/11/14.

6 http://titan.thinkaurelius.com, last accessed 2018/11/14.

- relationships (edges) having a direction, start node, and end node,
- identifiers.

Entities and relationships can have any number of properties, nodes and edges can be tagged with labels. Both nodes and edges are defined by a unique identifier (Id). Properties are of form key:domain, i.e. only single-valued attributes are considered. In graph-theoretic notions we also talk about *labelled and directed attributed multigraphs* in this case. It means the edges of different types can exist between two nodes. These graphs are used both for a GDB and its database schema (if any). In practice, this definition is not strictly enforced. There are GDBMSs supporting more complex property values, e.g. the already mentioned Neo4j.

When retrieving data from a GDB, one may want to query not only single nodes or relationships, but also more complex units consisting of these basic elements. Such units, *graph patterns*, can contain valuable information for many use cases. The fact that the graph can easily express such information is one of the main benefits of using such data model. Thus graph pattern matching is one of the key functionalities GDBs usually provide. In Sect. 3.1 we discuss shortly graph patters definable in the Cypher language and two basic methods for their indexing. Section 3.2 focuses on so called graph pattern trees of variations appropriate for organizing variations of a single graph pattern. Updating the index after performing DML operations is described in Sect. 3.3.

3.1 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.

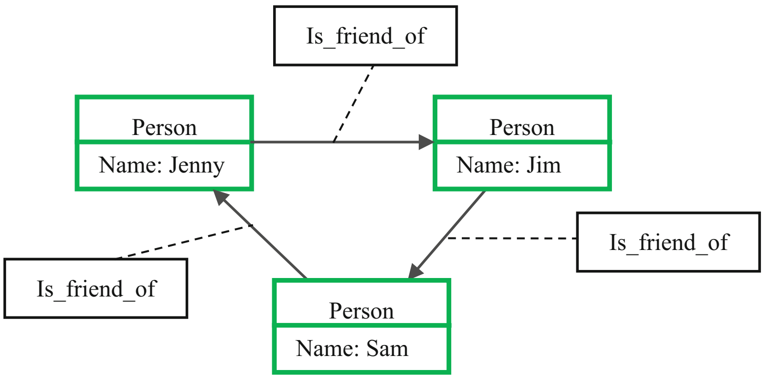


Fig. 1. Example of triangle [8].

One of widely used graph patterns, defined as GP = (V_p; E_p), where $V = \{v_1; v_2; v_3\}$ and $E = \{(v_1; v_2); (v_2; v_3); (v_3; v)\}$, is called a *triangle*. In Cypher, a triangle can be expressed in a few different ways, but preferably, e.g., as

$$(\mathtt{n}_1) - [\mathtt{r}_1] - (\mathtt{n}_2) - [\mathtt{r}_2] - (\mathtt{n}_3) - [\mathtt{r}_3] - (\mathtt{n}_1)$$

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. Figure 1 shows a triangle coming from a social graph. To retrieve such pattern using Cypher is easy for Neo4j. The problem arises when we focus only on structural features of the graph and want, e.g., all such triangles of people with their friendship. Then a structure-based index can be helpful.

A graph pattern index is basically a data structure that stores pointers that reference graph pattern units within the database. Indexes can be either stored in the same database as the actual data or in any external data store. We use here the former variant. The latter was used, e.g., in [9], where the embedded database MapDB[7] was used for this purpose.

3.2 Graph Pattern Tree of Variations

An important feature of our approach is that a new index can be created for each different graph pattern.

Due to labelling nodes and edges of GDB, patterns of the same structure can occur in different variations (see Fig. 2). All variations of a single graph pattern can be organized into a tree-like structure, called *graph pattern tree of variations*. A part of such tree for a triangle is shown in Fig. 2. Nodes represent individual graph pattern variations. A root node of the tree is reserved for the basic graph pattern variation with no additional information about nodes and relationships. Children of each node represent variations that provide some additional information compared to its parent nodes (i.e. when traversing deeper in the tree, more information about graph pattern is specified).

When querying a particular graph pattern in the database, one can use either a specific graph pattern variation, or arbitrary ancestor graph pattern from the pattern variation tree.

The more the database engine knows about the queried graph pattern, the more it filters graph space based on such specific information. Especially, node and edge labels which are usually already indexed by value-based indexing methods can be used.

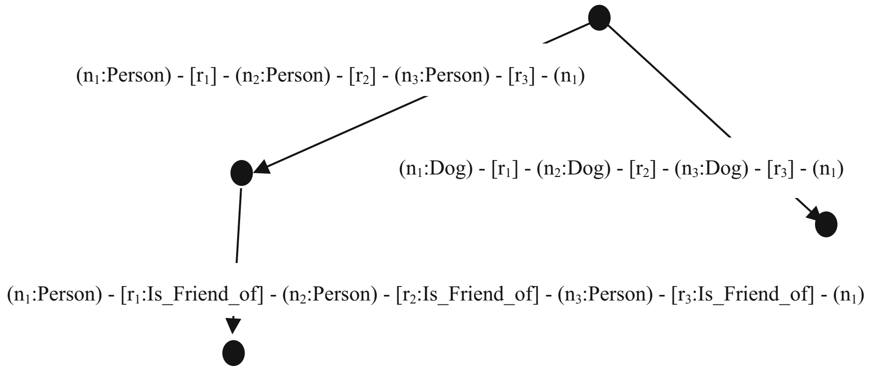
That means if one wants to query the basic graph pattern variation that represents the root node of its graph pattern tree of variations, it will be faster to query it using the structure-based index proposed in this paper. On the other hand, querying variations that already provide a lot of information about nodes and relationships (such variations are situated on the lowest levels of the tree, including its leaves) it may be faster to use a value-based indexing method.

In other words, there is a level of the tree, where querying variations on such level stops being more effective when using the structure-based index compared to value-based index. The height and the depth of the tree depends on the graph pattern the index was created for and data within the database.

[7] http://www.mapdb.org/, last accessed 2018/11/14.

3.3 Updating Graph Pattern Index

A graph pattern index maps all graph pattern units that are matched by a graph pattern the index was created for. Such graph pattern units exist within the database and so can be manipulated via DML operations. Thus, they can be updated in such way they no longer match the graph pattern. Also, when adding new data to the database, new graph pattern units can emerge. For that reason, each graph pattern index must always map its graph pattern units that currently exist within the database. That means each index must be updated each time a DML operation is applied on the database. Otherwise, indexes would not provide reliable information when queried.



(n_1:Person) - [r_1] - (n_2:Person) - [r_2] - (n_3:Person) - [r_3] - (n_1)
(n_1:Dog) - [r_1] - (n_2:Dog) - [r_2] - (n_3:Dog) - [r_3] - (n_1)
(n_1:Person) - [r_1:Is_Friend_of] - (n_2:Person) - [r_2:Is_Friend_of] - (n_3:Person) - [r_3:Is_Friend_of] - (n_1)

Fig. 2. Tree-like structure with graph pattern variations [8].

The intended DML includes operations like creating a node, creating a relationship, deleting a node, deleting a relationship, updating a node, and updating a relationship. Except the first one, all these operations affect the index, i.e. the index must be updated. 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 rollbacked, indexes will remain in the same state as before the transaction was initialized.

4 Implementation

The method for indexing graph patterns, including operations to create an index, query using an index and update an index, is implemented for the Neo4j GDB engine.

The major benefit of Neo4j is its intuitive way of modelling and querying graph-shaped data. Internally, it stores edges as double linked lists. Properties are stored separately, referencing the nodes with corresponding properties.

It is not easy to describe our implementation on a limited space, because it is tightly bind to database engine itself (Neo4j version 2.2. was used). It uses and extends Neo4j internal classes in order to achieve better integration. On the other hand, the implementation is done in a conceptually clean way and its shorted description presented in this paper may also explain a lot about extension of a database engine by additional services (like indexing method) in general. Implementation details can be found in [14].

This section is organized as follows: in Sect. 4.1, the idea of structure-based index implementation inside the GDB is explained, it provides necessary terminology used in the rest of the section. Section 4.2 describes an architecture of Neo4j and GraphAware framework classes involved in our implementation and explains the concept of our implementation. Sections 4.3, 4.4 and 4.5 claim to explain implementation of individual usage of structure-based indexes, i.e., their creating, using for querying and keeping them actual with DML operations over the database. Section 4.6 provides information how our implementation can be installed and used.

4.1 Overview

Our index implementation is done in the same GDB as basic graph data. We introduced an additional graph representing all indexes in the database. This graph has a root providing approach to all indexes.

Implementation of an index consists of a two-level tree. The first level has one node representing the index and containing appropriate metadata. This top-level index node is related to common root mentioned above. The second level of index representation consists of a set of graph pattern units. Each unit represents one pattern (triangle in our case). There are direct relationships to appropriate nodes in the database from each pattern unit.

Figure 3 describes an implementation of triangle shape index. The first and the second layer in the figure represents pure index data, while the third layer represents original data in GDB.

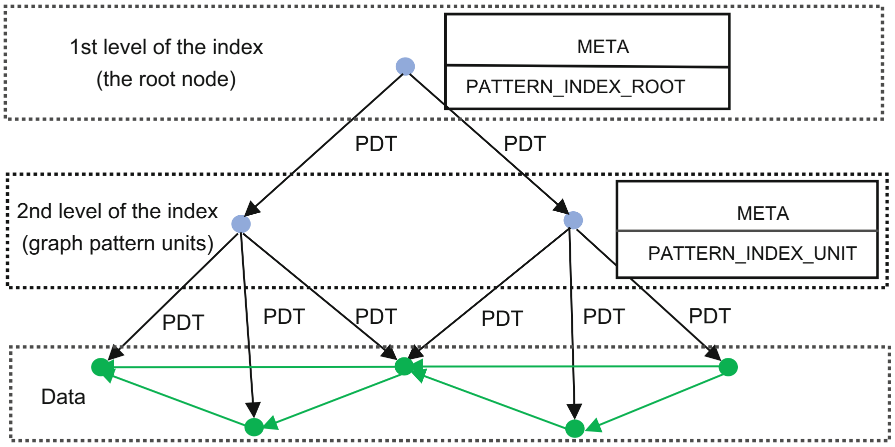


Fig. 3. Implementation of triangle shape index [8].

Index data is separate from original data on the logical level. This is done by assigning a special `META` label to nodes of the index and a special

PATTERN_INDEX_RELATION type (label) to relationships, they are denoted PDT in the figure. A special additional labels are used to differentiate the index root (PATTERN_INDEX_ROOT) and second level nodes representing particular indexes (PATTERN_INDEX_UNIT).

For the following discussion it is important to understand notions of graph pattern units and index units.

Graph Pattern Units. They are formed from actual data in the database. They are uniquely identified by relationships. In our example with triangle shape we may care about permutations in the result set or there may be more relevant relationships between two nodes, i.e. multigraph.

Index Units. Index units belong to index structure. They are uniquely identified by (ordered) list of its nodes. One index unit may point to several graph pattern units.

4.2 Architecture of Implementation

Implementation Background. Neo4j GDB engine is written in Java. It provides Core API which is used to communicate with a GDB. It is also used in our implementation.

GraphDatabaseService interface, the key part of Core API, is used as the main access point to a running Neo4j instance. It provides many methods for querying and updating data, including operations to create nodes, get nodes by id, traverse a graph and many more. Each of such operations must be executed within a transaction to ensure ACID properties. More than one operation can be applied to a database within a single transaction. For this purpose the interface also provides a method to create a new transaction. Neo4j then enables to use Transaction interface to build transactions in a very easy way. When using this interface, all operations within a transaction are enclosed in try-catch block.

Node and Relationship are other two important interfaces provided by Core API. Node interface provides methods that cover all possible operations with a single node, including manipulation with its properties, labels, and relationships. Relationship interface, on the other hand, provides methods to mostly accessed information about such relationship, including its end nodes or type. Note that it is not possible to change a type of a relationship in Neo4j. Such operation must be simulated by deleting and re-creating a relationship. It is also important to mention that Neo4j supports only primitive data types when storing properties of nodes and relationships. Thus more complex data types must be converted to string values before storing within their properties.

Since Neo4j 2.2, GraphDatabaseService interface enables to execute queries using Cypher. For this purpose a method execute is provided. Cypher query is passed as a string parameter to this method. Result of such query is organized in a table and returned as an instance of Result class. It is not necessary to enclose such operation in transaction try-catch block since it is, by default, executed within a transaction.

Graph Index Implementation. The implementation of the method of indexing graph patterns involves several classes. `PatternIndexModel` class is the core class of the method.

Figure 4 shows classes and their relationships. A single instance of `PatternIndexModel` class is created for a Neo4j database. It handles all operations that involve manipulation with graph pattern indexes. `PatternIndexModel` uses a singleton pattern, thus its instance can be obtained by calling `getInstance` method of the class. Instance of Neo4j database is passed as a parameter to this method. The class further provides methods to create a new index, query using an existing index, delete an existing index, and to handle updating of all existing indexes at once.

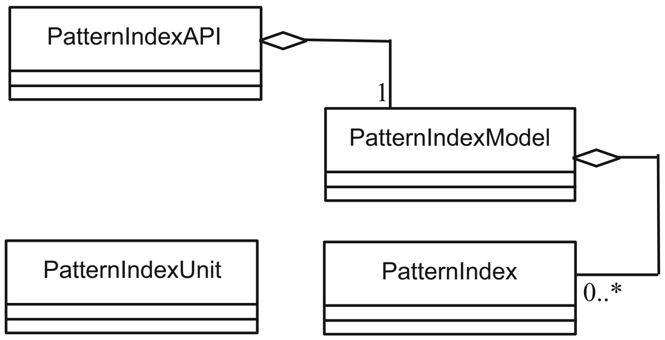


Fig. 4. Main classes.

Instance of `PatternIndexModel` manages all indexes that are created for a database. Each of these indexes is represented as an instance of `PatternIndex` class. Such instance holds basic information about an index including its name, graph pattern that the index was created for and other information that is necessary for its functionality. It also holds a root node of its index tree structure. All existing indexes are accessible within the instance of `PatternIndexModel` class via a single map. Such map consists of key-value pairs, such that a key represents a name of an index and a value represents appropriate instance of `PatternIndex`.

As mentioned above, each index tree consists of a single root node and index units that map actual graph pattern units within a database. Instances of `Pattern IndexUnit` class are used to represent specific index units. An index unit is identified by a group of nodes that form graph pattern units mapped by such index unit. Each index unit can map one or more graph pattern units. Thus each instance of `PatternIndexUnit` holds a set of nodes that identify it and a set of string identifiers that represent specific graph pattern units that belong to it. Identifiers of graph pattern units are persisted within a `specificUnits` property of appropriate index units in a database.

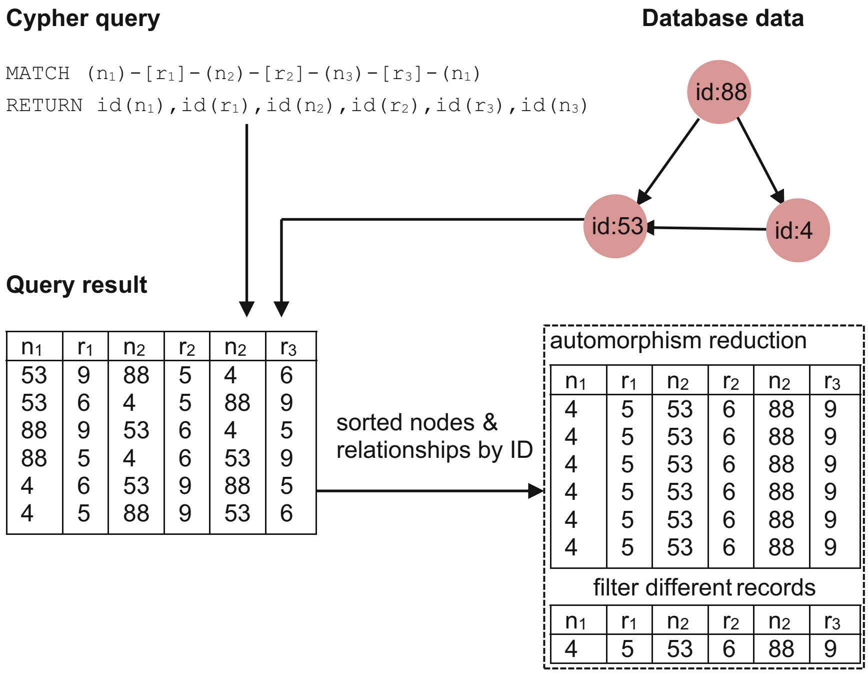
In order to query the database using index, the abstract class `QueryParse` is created. Two classes named `PatternQuery` and `CypherQuery` implement this class and are used for querying itself.

Additionally the class `DatabaseHandler` which provides manipulating methods for creating and processing individual parts of the index is provided.

4.3 Index Creation

To create a new index, `buildNewIndex` method is provided within `PatternIndexModel` class. When calling such method, a user must provide a name of the index and also a graph pattern that the index should be created for. Cypher, more specifically its `MATCH` clause, is used to express such graph pattern. First of all, the process of graph pattern validation is applied to given graph pattern.

If the graph pattern is valid, all graph pattern units that match given graph pattern are found within a database. This is done by executing a simple Cypher query. The process of matching graph pattern units using a simple Cypher query is very time consuming. From each node within a database depth-first search or breath-first search (it is based on Neo4j settings) is applied in order to find matches for a graph pattern starting from such node. By applying such process a group of matched graph pattern units is retrieved. Some of them might be mutually automorphic. This is caused because of the fact that searching is done from each node individually.



```
MATCH (n1)-[r1]-(n2)-[r2]-(n3)-[r3]-(n1)
RETURN id(n1),id(r1),id(n2),id(r2),id(r3),id(n3)
```

n_1	r_1	n_2	r_2	n_2	r_3
53	9	88	5	4	6
53	6	4	5	88	9
88	9	53	6	4	5
88	5	4	6	53	9
4	6	53	9	88	5
4	5	88	9	53	6

n_1	r_1	n_2	r_2	n_2	r_3
4	5	53	6	88	9
4	5	53	6	88	9
4	5	53	6	88	9
4	5	53	6	88	9
4	5	53	6	88	9
4	5	53	6	88	9

n_1	r_1	n_2	r_2	n_2	r_3
4	5	53	6	88	9

Fig. 5. Creating index - automorphism reduction.

Then such group can be also referred to as a group of automorphism groups of graph pattern units. Figure 5 shows a result for a query. In this case, the database consists only of a single triangle. There should be a single triangle matched but instead there are six records (i.e. graph pattern units represented in rows) in the result. Note that all of them belong to a single automorphism group. It is necessary to reduce automorphism in a group of matched graph pattern units such that each automorphism group consists of only a single graph pattern unit. For this purpose records are sorted by IDs of nodes and IDs of relationships separately and then only different records (i.e. graph patterns) are filtered.

After a group of graph pattern units is found within a database and automorphism is reduced, an index tree structure is built. Its index units will map these graph pattern units. Identifiers are created for specific graph pattern units and then stored within `specificUnits` property of appropriate index units. Also basic information about an index, including its name and graph pattern with parsed identifiers, is stored within properties of its root node. After that, a new index is stored under its name within `patternIndexes` attribute of `PatternIndexModel` instance.

4.4 Index Querying

To query using an index, `getResultFromIndex` method is provided within `PatternIndexModel` class. A user must provide an index to be used and also a query to be executed within an index when calling this method. The first step of the whole process of querying using an index is to find the root node of appropriate index. For this purpose an instance of `PatternIndexModel` is loaded into memory each time an instance of Neo4j is started up.

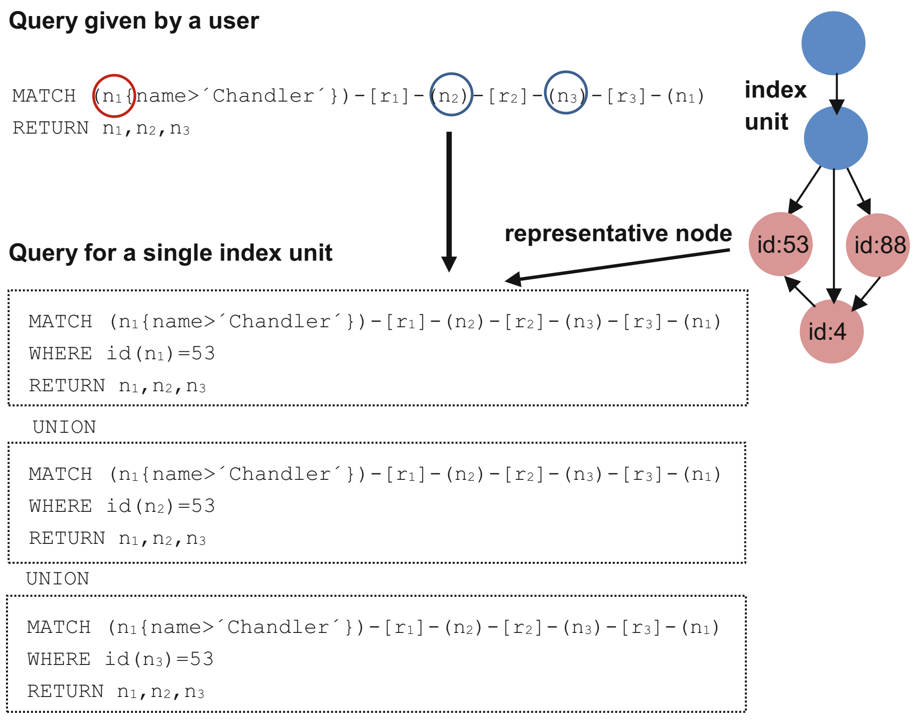
After the root node is retrieved directly from memory, all index units that belong to appropriate index must be collected. This is done by traversing outgoing relationships of the root node that head to these units.

The process of querying using an index can be split into executing given query on top of each graph pattern unit that is mapped by appropriate index. Results of these queries are then merged to present the final result for given query. That is indeed a lot of queries to be executed. Thus it is better to execute the query not on top of each single graph pattern unit but perhaps on each group of nodes that together with their relationships form one or possibly more graph pattern units. Such groups of nodes are also referred to as index units. Each of such index unit is defined by a different set of nodes that form one or more graph pattern units. Index units are subgraphs within a GDB. Unfortunately, querying on top of subgraphs is not supported in Neo4j at the moment.

The following process is applied for each of index units of appropriate index. One of nodes that together form an index unit is chosen to represent such index unit. The node will be further referred to as a *representative node*. Then all matches for given query that involve such node are found within a database. This is done by executing multiple similar queries, where a representative node is put on every node position within a graph pattern expressed in a `MATCH` clause of the query. Results of these queries are then merged.

In Cypher, this can be solved by using a single composed query. Figure 6 shows such composed query for a single index unit, which is represented by a node with ID 53.

In general, a single composed query consists of *s* subqueries, where *s* is the number of nodes within a graph pattern expressed in a `MATCH` clause of given query. Note that a graph pattern for which appropriate index is created consists of the same amount of nodes. As said at the beginning, this whole process is done for each index unit of appropriate index. Thus it is necessary to execute *n* such composed queries, where *n* is the number of index units of appropriate index. Finally, results of these queries are merged to present the final result for given query. This approach is used when implementing the method of indexing graph patterns.



```
MATCH (n1{name>´Chandler´})-[r1]-(n2)-[r2]-(n3)-[r3]-(n1)
RETURN n1,n2,n3
```

```
MATCH (n1{name>´Chandler´})-[r1]-(n2)-[r2]-(n3)-[r3]-(n1)
WHERE id(n1)=53
RETURN n1,n2,n3
 UNION
MATCH (n1{name>´Chandler´})-[r1]-(n2)-[r2]-(n3)-[r3]-(n1)
WHERE id(n2)=53
RETURN n1,n2,n3
 UNION
MATCH (n1{name>´Chandler´})-[r1]-(n2)-[r2]-(n3)-[r3]-(n1)
WHERE id(n3)=53
RETURN n1,n2,n3
```

Fig. 6. Querying using index.

4.5 Index Update

Indexes are updated automatically each time a DML operation is applied to a database. For this purpose `handleDML` method is provided within `PatternIndexModel` class. This method is called inside `beforeCommit` method provided within GraphAware framework. Such method is called each time a transaction is about to be executed. It also provides its users with an instance of `ImprovedTransactionData` that holds all nodes and relationships that are being created, deleted, and updated by currently running transaction. Such instance is passed to the `handleDML` method as a parameter.

In the context of `beforeCommit` method it is possible to traverse a graph in the state as it was before a transaction was executed. Let's say a user deletes a relationship within a transaction. Then `beforeCommit` method is called. Instance of `ImprovedTransactionData` holds the relationship within a list of deleted relationships. At this moment, one can, e.g., explore end nodes of such deleted relationship. When executing Cypher queries using execute method in this context, a database is always in the state as it would be if the transaction was successfully committed. These are key functionalities used when implementing methods for updating indexes.

`handleDML` method performs some additional operations to update indexes before currently running transaction is actually committed. If the transaction fails and it is to be rollbacked, all these additional operations are reverted as well. `handleDML` method calls other methods based on data that is being modified by the transaction. All of them update indexes based on specific DML operations. The process of updating indexes is described in the analytical part of this thesis.

4.6 Usage of Implementation

Neo4j can be either embedded in a custom application or run as a standalone server. When using embedded mode, Neo4j can be used within an application by simply including appropriate Java libraries. In this case, the method of indexing graph patterns can be used in the same way. Thus it can be included as an extra library. When using a standalone mode, Neo4j is accessed either directly through a REST interface or through a language-specific driver. For this case, a custom API, provided by `PatternIndexAPI` class, is exposed to access all operations of the method of indexing graph patterns. In this case, the method must be built first. After it is built, it is necessary to drop built jar file into the plugins directory of appropriate Neo4j installation. In other words, the method is used as a Neo4j plugin. Note that the method uses GraphAware framework libraries, so these must be included as well when installing the method.

5 Experiments

Experiments were done on three different graph datasets: Social graph with a triangle index, Music database with a funnel index, and Transaction database with a rhombus index. These datasets are shortly described in Sect. 5.1. In Sect. 5.2, we present experiments done with growing the database size and the structure index effectivity. Measurements are done on social graph with triangle index. Section 5.3 presents measurements on all three datasets. Here only one size of each dataset is used, but we present also time necessary to keep index actual along with DML operations on the database. Some hypothesis about the size of index are provided in Sect. 5.4. Section 5.5 brings a discussion of the measurements.

All results are achieved by measuring within a test environment provided by GraphAware framework. The following configuration is used when performing measurements: 2.5 GHz dual-core Intel Core i5, 8 GB 1600 MHz 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.

5.1 Graph Datasets

Social graph is a database that contains information about people and friendships between them. People, represented by nodes, have names and are distinguished to males and females by appropriate labels. Friendships between them are represented by relationships of Is_friend-of type. Such database of changeable size is generated by Erdős– Rényi model for generating random graphs (see, e.g., [2]). The generator is a part of used GraphAware framework[8]. Triangle index is built for a triangle graph pattern expressed in Cypher in Sect. 3.1.

Music database stores data about artists, detailed information about the tracks they recorded and labels that released these records. The database has a fixed size of 12 000 nodes and 50 000 relationships. It is one of the example datasets that Neo4j provides on its website[9]. The database contains 86 funnel patterns. Funnel index pattern (see, Fig. 7a) we used for this database has the following Cypher expression:

$$\texttt{(n1)} - \texttt{[r1]} - \texttt{(n2)} - \texttt{[r2]} - \texttt{(n3)} - \texttt{[r3]} - \texttt{(n1)} - \texttt{[r4]} - \texttt{(n4)}$$

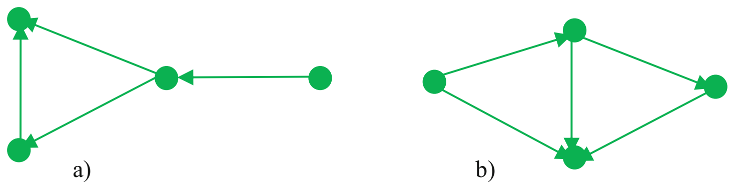


Fig. 7. Graph patterns used for Music DB (a) and Transaction DB (b) [8].

Transaction database stores data about transactions between bank accounts in a simplified way. Bank accounts, represented by nodes, are identified by account numbers. Transactions between bank accounts are represented by relationships. They have no properties on them since it is not crucial for the measurements. If used in a real database, they would probably hold some specific characteristics about them, e.g., a date of transaction execution or the amount of transferred money within a transaction.

Such database of changeable size was generated by a Cypher script that was created especially for this purpose. Such simple script creates bank accounts at first and then

[8] https://github.com/graphaware/neo4j-framework, last accessed 2018/11/14.

[9] http://neo4j.com/developer/example-data/, last accessed 2018/11/14.

generates a transaction relationship for each pair of these accounts with a given probability. The database we generated has 10 000 of nodes and 100 000 of relationships, it contains 70 rhombus patterns which were indexed. Rhombus index (see, Fig. 7b) is used for this database. It is formulated by the following Cypher expression

$$(\mathtt{n}_1) - [\mathtt{r}_1] - (\mathtt{n}_2) - [\mathtt{r}_2] - (\mathtt{n}_3) - [\mathtt{r}_3] - (\mathtt{n}_1) - [\mathtt{r}_4] - (\mathtt{n}_4) - [\mathtt{r}_5] - (\mathtt{n}_2)$$

5.2 Measurement on Social Graph Database with Triangle Index

The size of the database scales from 50 nodes and 100 relationships to 100 000 nodes and 500 000 relationships. A matching triangle graph pattern using a simple query (i.e. without a graph structure index) is nearly impossible for larger databases of this type.

There are two metrics used: time and the number of database hits (DBHits), i.e., total number of single operations within Neo4j storage engine that do some work such as retrieving or updating data.

DBHits metrics for varying size of databases are shown on Fig. 8. We can see that from the database of size 10 000 nodes/50 000 relationships index pattern is much more effective than "simple query" (i.e. the query without index usage). The amount of DBHits for index grows linearly with growing the database while it grows exponentially for simple query.

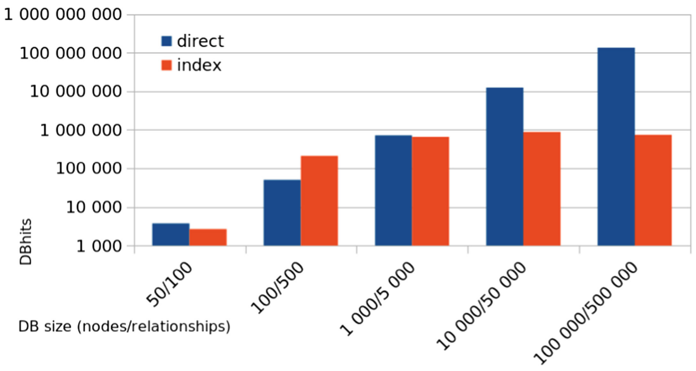


Fig. 8. Social graph, triangle index – DBhits metrics [8].

Time metrics are shown on Fig. 9. Again, we can see an exponential growth of time for a simple query and a linear growth for index. For the largest database of 100 000 nodes and 500 000 relationships, a query using an index is approximately 170 times faster and performs approximately 180 times less database operations than a simple query.

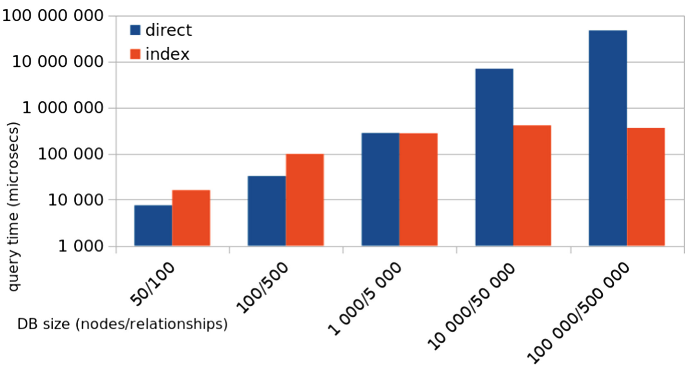


Fig. 9. Social graph, triangle index – time metrics [8].

5.3 DML Operations and Queries Measurement

The index must be updated together with DML operations on the base data. In our implementation update of the index is done in the same transaction as DML statement. We did measurements on all three databases mentioned in Sect. 4.1 for the following DML operations:

- creating an index,
- creating a relationship,
- deleting a relationship,
- deleting a node, and
- deleting a label of a node.

All these DML operations may affect existing indexes.

All measurements were done again for all cache types provided by Neo4j, i.e.:

- without caching,
- low level cache (i.e. file buffer cache), and
- high level cache (object cache).

The last mode is the most suitable for our purpose and, not surprisingly, it provides the best performance. For measurement results using another cache modes see [14].

In the Tables 1, 2 and 3 we present also a time of given operation without index usage to show additional costs for index maintenance. In Table 1, we present measured values done on a social graph with triangle index on the database having 10000 nodes and 50 000 relationships. Let us note, that there were 183 graph patterns on 179 nodes.

Table 1. Social graph, triangle index.

Operation	Simple query [μs]	Index [μs]
Create index	–	5 242 762
Query index	6 881 440	403 375
Create relationship	25 973	29 987
Delete relationship	146 820	157 726
Delete node with its relationships	228 399	277 835
Delete node label	17 475	19 380

Table 2 presents measurements for funnel pattern index on the Music database. The database consists of 12 000 nodes, 50 000 relationships and contains 86 funnel patterns.

Table 2. Music database, funnel index [8].

Operation	Simple query [μs]	Index [μs]
Create index	–	11 810 818
Query index	6 077 701	162 175
Create relationship	148 194	162 175
Delete relationship	289 220	283 263
Delete node with its relationships	484 171	665 391
Delete node label	64 420	66 610

Table 3 presents measurements for rhombus pattern index on the Transaction database. Database consists of 10 000 of nodes, 100 000 of relationships and contains 70 rhombus patterns.

Table 3. Transaction database, rhombus index [8].

Operation	Simple query [μs]	Index [μs]
Create index	–	36 238 883
Query index	41 794 378	1 243 503
Create relationship	29 659	64 432
Delete relationship	257 094	283 067
Delete node with its relationships	375 308	459 808
Delete node label	17 485	22 420

5.4 Index Size

Index size linearly grows with the number of pattern units found in the database and it also linearly grows with the number of nodes that the indexed pattern consists of. The index size can by asymptotically expressed as

$$\Theta(n_u * n_n)$$

where n_u represents the number of pattern units found in the database and n_n represents the number of nodes that the indexed pattern consists of. The exact number of nodes needed for the index is

$$n_\mathrm{nodes} = 1 + n_\mathrm{u}$$

where a single node represents the root of the index and n_u nodes represent individual pattern units found in the database. The exact number of relationships needed for the index is

$$n_\mathrm{rels} = n_\mathrm{u} + n_\mathrm{u} * n_\mathrm{n}$$

where n_u relationships connect the root node with all n_u pattern unit nodes. n_n relationships then connect individual data nodes belonging to a single pattern unit to its representative pattern unit node.

Table 4 presents index size using 3 different patterns and databases. There are 183 pattern units indexed in the first database which is more than double what is indexed in other two databases. Triangle pattern consists of 3 nodes whereas funnel and rhombus patterns consist of 4 nodes. This results in approximately the same size (in Mb) of index for all of 3 measured databases and patterns.

Table 4. Database and index sizes.

Database index	DB size (Mb)	Index size (Mb)	Pattern units
Social graph, triangle index	19,6	0,15	183
Music database, funnel index	89,6	0,2	86
Transaction database, rhombus index	17,2	0,1	70

5.5 Discussion

Let us note several interesting observations coming from our measurements:

- index creation time is not much higher than query time without an index for triangle and rhombus patterns, it is nearly two times higher for funnel index,
- query using an index is faster than the query without index for all three patterns, it is 17 times faster for triangle index, 112 times for funnel index, and 33 times for rhombus index,

- time to update the index in case of insert/delete a node or a relationship is on average 17% of time needed for DML operation itself.

Let us generalize the measurement and state several hypotheses about the efficiency of our implementation of pattern indexes:

- It was shown that starting from databases of size 10 000 of nodes and 50 000 of relationships queries using pattern indexes are more efficient than queries without them.
- Efficiency of pattern index increases with growing the database. Time and amount of database operations grow linearly for (triangle) pattern (Figs. 8 and 9).
- Complexity and size of the pattern used for index influence characteristics and efficiency of an index. We tested triangle, funnel, and rhombus patterns – all tested indexes are more than 17 times faster for querying, this ratio will growth with the size of the database.
- Time for keeping indexes actual seems to be under 20% of time necessary for DML operation.
- Experiments were done on the most general graph pattern indexes (the root of a graph pattern tree of variations, see Sect. 3.2 and index shapes in Figs. 1 and 7 for details).
- For practice usage, it is necessary to investigate the optimizer module for GDB engine. The module is responsible for creating execution plans including the decision when and which index is used for particular query evaluation.
- Method proposed in this paper stores data of structure-based indexes along the data which are indexed in the same database. It mixes together data and metadata and may cause undesirable effect on evaluation of queries which are not using structure based index.
- Unfortunately, Neo4j (and also others GDBMS) do not provide a separation space for data and metadata as it usual in relational database management systems.
- We already investigated another approach: store data about structural-based indexes in a separate storage, see [7] for details. Results we achieved are very similar to that presented in this paper.

6 Conclusions

In the paper a new method for indexing graph patterns was analyzed, designed and implemented for Neo4j GDBMS in order to speed up the process of matching graph patterns. The method enables to create, use and update multiple indexes, each created for a different graph pattern. Index data are organized in a tree structure and they are stored within the same database as the base data. This solution provides really fast approach from the index structure to data. On the other hand, it mixes index data and base data together in one common storage. It may negatively affect the evaluation of queries that do not use index patterns. We plan to address this issue in following research. It is the part of a more general topic how to store metadata and separate them from base data in GDBMS.

It is proved that using indexes which are created by the method introduced in this paper is beneficial for the process of matching graph patterns. In some cases queries using such indexes are extremely faster than simple Cypher queries. The paper aims to introduce the topic of indexing graph patterns and provides one of possible ways how to speed up the process of matching graph patterns within a GDB.

Acknowledgments. This work was supported by the Charles University project Q48.

References

1. Aggarwal, C.C., Wang, H.: Managing and Mining Graph Data. Springer, Boston (2010). https://doi.org/10.1007/978-1-4419-6045-0
2. Goldenberg, A., Zheng, A.X., Fienberg, S.E., Airoldi, E.M.: A survey of statistical network models. Found. Trends Mach. Learn. **2**(2), 129–233 (2009)
3. Mpinda, S.A.T., Ferreira, L.C., Ribeiro, M.X., Santos, M.T.P.: Evaluation of graph databases performance through indexing techniques. Int. J. Artif. Intell. Appl. (IJAIA) **6**(5), 87–98 (2015)
4. O'Neil, P.E.: The SB-tree: an index-sequential structure for high-performance sequential access. Informatica **29**, 241–265 (1992)
5. Pokorný, J.: Graph databases: their power and limitations. In: Saeed, K., Homenda, W. (eds.) CISIM 2015. LNCS, vol. 9339, pp. 58–69. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24369-6_5
6. Pokorny, J., Snášel, V.: Big graph storage, processing and visualization. In: Pitas, I. (ed.) Graph-Based Social Media Analysis, Chap. 12, pp. 391–416. Chapman and Hall/CRC, Boca Raton (2016)
7. Pokorný, J., Valenta, M., Ramba, J.: Graph patterns indexes: their storage and retrieval. In: Proceedings of the 19th International Conference on Information Integration and Web-Based Applications and Services (iiWAS 2018), Yogykarta, Indonesia, November 2018, 5 pages (2018)
8. Pokorný, J., Valenta, M., Troup, M.: Indexing patterns in graph databases. In: Proceedings of the DATA 2018, pp. 313–321 (2018)
9. Ramba, J.: Indexing graph structures in graph database machine Neo4j II. Master's thesis, Faculty of Information Technology, Czech Technical University in Prague (2015). (in Czech)
10. Robinson, I., Webber, J., Eifrém, E.: Graph Databases. O'Reilly Media, Menlo Park (2013)
11. Sakr, S., Al-Naymat, G.: Graph indexing and querying: a review. Int. J. Web Inf. Syst. **6**(2), 101–120 (2010)
12. Srinivasa, S.: Data, storage and index models for graph databases. In: Sakr, S., Pardede, E. (eds.) Graph Data Management: Techniques and Applications, Chap. 3, pp. 47–70. IGI Global, Hershey (2012)
13. Tivari, S.: Professional NoSQL. Wiley/Wrox, Hoboken (2015)
14. Troup, M.: Indexing of patterns in graph DB engine Neo4j I. Master's thesis, Faculty of Information Technology, Czech Technical University in Prague (2015). https://dspace.cvut.cz/bitstream/handle/10467/65061/F8-DP-2015-Troup-Martin-thesis.pdf?sequence=1&isAllowed=y
15. Ullmann, J.R.: An algorithm for subgraph isomorphism. J. ACM **23**(1), 31–42 (1976)
16. Yan, X., Yu, P.S., Han, J.: Graph indexing: a frequent structure-based approach. In: Proceedings of SIGMOD Conference, pp. 335–346. ACM (2004)

17. Yan, X., Han, J.: Graph indexing. In: Aggarwal, C.C., Wang, H. (eds.) Managing and Mining Graph Data. Advances in Database Systems, vol. 40. Springer, Boston (2010). https://doi.org/10.1007/978-1-4419-6045-0_5
18. Yuan, D., Mitra, P.: Lindex: a lattice-based index for graph databases. VLDB J. **22**, 229–252 (2013)
19. Zhao, P., Han, J.: On graph query optimization in large networks. VLDB Endow. **3**(1–2), 340–351 (2010)
20. Zhu, L., Ng, W.K., Cheng, J.: Structure and attribute index for approximate graph matching in large graphs. Inf. Syst. **36**(6), 958–972 (2011)