Insert here your thesis' task.

Czech Technical University in Prague
Faculty of Information Technology
Department of of software engineering

Master's thesis

Indexing of patterns in graph DB engine Neo4j I

Bc. Martin Troup

Supervisor: Ing. Michal Valenta, Ph.D.

29th June 2015

Acknowledgements

I would like to thank everybody that directly or indirectly contributed to the work introduced in the thesis. My special thanks go to MSc. Michal Bachman, the creator of GraphAware Framework, who was always willing to help me with answering questions about the topic. Next I would like to thank Ing. Michal Valenta, Ph.D., the supervisor of my Master's thesis, who was always there when I needed and shared his rich experience not just about the topic with me. I would like to thank Bc. Jaroslav Ramba, the creator of parallel Master's thesis involving similar topic, for sharing his knowledge of the topic. I would like to thank my parents for their support. I would never be able to finish the work without them. Finally, I would like to thank all professors at CTU FIT for providing me with scientific background necessary for finishing the work.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 29th June 2015 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Troup, Martin. *Indexing of patterns in graph DB engine Neo4j I.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2015.

Abstrakt

V této práci je představena nová metoda pro indexování grafových vzorů v grafové databázi. Metoda je navržena a implementována pro grafovou databázi Neo4j. Metoda umožnuje vytváření, používání a aktualizování indexů, které jsou použity k urychlení při vyhledávání vzorů v databázi. Práce dále nabízí srovnání dotazů bez použití a s použitím indexů. V práci je navíc uvedeno srovnání s konkurenční metodou, která je představena v souběžné magisterské práci.

Klíčová slova grafové databáze, Neo4j, GraphAware framework, indexování vzorů, teorie grafů, grafové vzory

Abstract

This thesis introduces a new method for indexing graph patterns within a graph database. The new method is analyzed, designed and implemented for Neo4j graph database engine. It enables to create, use and update indexes that are used to speed-up the process of matching graph patterns. The thesis provides a comparison between queries with and without using indexes. It also provides a comparison of the method with an alternate method that is presented in a concurrent master's thesis.

Keywords graph databases, Neo4j, GraphAware framework, indexing patterns, graph theory, graph patterns

Contents

List of Figures

Introduction

A database system is a collection of programs that run on a computer and help the user to get information, to update information, to protect information, in general to manage information [1]. The most fundamental aspect in which various database systems differ is the way data or information is represented. Since 1970 when E.F.Codd presented relational model of data [2], relational databases using the model have become the most important systems for storing or retrieving data. The model organizes data into tables (also called "relations") that consist of records (rows) and its properties (columns). Each record can be uniquely identified by its key. Records can be linked between each other using those keys.

The recent growth of user-driven content has fueled a rapid increase in the volume and type of data that is generated, manipulated, analyzed and archived. In parallel to the fast data growth, data is also becoming increasingly semi-structured and sparse [3]. The fact that relational databases are often not good enough for handling this new kind of data has led to the emergence of a class of newer types of databases. These new, so-called NoSQL databases provide a mechanism for storage and retrieval of data that is no more based on relational model. Each of these databases can use different data structure (e.g. key-value, graph or document).

Graph databases are NoSQL representatives that use graph structure to store and retrieve data. They embrace relationships as a core aspect of its data model. It is built on the idea that even though there is value in discrete information about things, there is even more value in relationships between them. Graph databases proved to be very effective and suitable for many data handling use cases.

Commercial graph databases started appearing on the market in 2003. Hence it is still very young technology. From technological point of view, they are still behind relational databases that are here for much longer time. They still miss some of features relational database users are used to. On the other hand fundamental functions of graph databases are being constantly

developed and optimized. Rapid development of graph databases leads to its increasing usage in commercial world. Data is being moved from relational to graph databases within many current applications with promise to reveal information (mostly from relations) that has been hidden so far. Also new applications with functionalities based on graph databases are being created. In other words, with graph databases new possibilities come.

Data is the most valuable asset for many companies. Effective way of processing it is one of the key aspects of running a successful business. As mentioned above graph databases are relationships focused. The real world—unlike the forms-based model behind the relational database—is rich and interrelated [4]. But besides of discrete information about things and relationships between them there is even more that can be retrieved from those databases. Patterns (also called "shapes") are more abstract pieces of information that can be found in the graph. They are composed of elementary information. Querying those patterns can play a major role in many use cases including recommendations engines or fraud detection systems. Despite of high performance of graph databases, process of finding those shapes in graph can be very time-consuming. This thesis focuses on designing a method that can speed up the process and thus help when retrieving such important data. Such method is introduced as a process of indexing graph patterns.

Chapter 1 of the thesis introduces background for the research including basic information from graph theory and information about graph databases. In this chapter, Neo4j, the most popular representative of graph databases at the time of writing the thesis, is introduced. Also information about GraphAware framework, a tool that helps with solving some of advanced Neo4j use cases, is provided. Chapter 2 describes the analysis and design of the new method of indexing patterns in graph databases. Chapter 3 presents implementation of designed method using Neo4j graph database engine. Chapter 4 introduces benchmark that compares the method to another method implemented in parallel thesis but also to current Neo4j querying possibilities. Chapter 5 comments on the evaluation and suggests topics for further research in this area.

Chapter 1

State-of-the-art

1.1 Graph theory

Many real-world situations can conveniently be described by means of a diagram consisting of a set of points together with lines joining certain pairs of these points. For example, the points could represent people, with lines joining pairs of friends; or the points might be cities and some of them might be connected by airlines. Notice that in such diagrams one is mainly interested in whether or not two given points are joined by a line [5]. A mathematical abstraction of situations of this type gives rise to the concept of a graph. Graph is the main interest of graph theory. Graph theory is the core content of Discrete Mathematics, and Discrete Mathematics is the theoretical basis of Computer Science and Network Information Science [6].

In this section some of the basic definitions from graph theory will be presented. These are essential when manipulating data within graph structures. All definitions in this section are taken or adopted from Diestel [7], unless indicated otherwise.

Graph A graph is a pair $G = (V, E)$ of sets satisfying $E \subseteq (V \times V)$. The elements of V are *vertices* (or *nodes*) of the graph G, the elements of E are its *edges* (or *relationships*) Each $e \in E(G)$ is represented as a pair $(v, u$, where $v, u \in V(G)$. Whereas in graph theory *vertices* and *edges* are used, in computer science, particularly in graph databases *nodes* and *relationships* are used. Note that these terms are handled interchangeably throughout the thesis.

Order The number of vertices of a graph G is its *order*, written as $\mid G \mid$; its number of edges is denoted by $\mid\mid G \mid\mid$. Graphs are $finite$ and $infinite$ according to their order. In this thesis we consider all graphs to be $finite$.

Incident vertex A vertex v is *incident* with and edge e if $v \in e$; then e is an edge at v. Pair of vertices (v, u) that represent edge are both incident with the edge and are called its ends.

Vertex degree The set of edges vertex v is incident with is denoted $E(v)$. The degree of a vertex v, denoted as $d(v)$, is the number $\mid E(v) \mid$ of edges incident with vertex v.

Subgraph A graph $G' = (V', E')$ is called a subgraph of a graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. That means that all the vertices of G' are the vertices of G and all the edges of G' are the edges of G. This definition was adopted from Bin [6].

Path A path is a non-empty graph $P = (V, E)$ of the form $V = \{v_0, v_1, \ldots, v_k\}$, $E = \{(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)\}$, where k is a finite constant. The number of edges in a path (respectively graph), $\mid E \mid$, is its *length*. A *cycle* is a special case of a path with additional edge (v_k, v_0) in set E.

Directed graph A directed graph $DG = (V, E, init, ter)$ is a graph $G = (V, E)$ that has additional maps *init* and *iter*. Map $init : E \to V$ assigns each edge *an initial vertex init(e)* also called *tail* or *start node* in domain of graph databases. Map $ter : E \to V$ assigns each edge *a terminal vertex ter(e)* also called *head* or *end node* in domain of graph databases. Note that these terms are handled interchangeably throughout the thesis. The edge e is set to be directed from $init(e)$ to $ter(e)$. There may be several edges between the same two vertices. If $init(e) = ter(r)$, the edge is called *loop*. This definition was adopted from Bachman [8]

Vertex degree revisited When considering directed graphs, vertex degree $d(v)$ can be formulated as $(indegree + outdegree) - loops$. Indegree, denoted $d_{in}(v)$, is defined as the number $\mid E_{in}(v) \mid$ of edges incident with vertex v, where $E_{in}(v) \subseteq E(v)$ such that $\forall e_{in} \subseteq E_{in}(v)$, $ter(e_{in}) = v$. Similarly, outdegree, denoted d_{out}, is defined as the number $\mid E_{out}(v) \mid$ of edges incident with vertex v, where $E_{out}(v) \subseteq E(v)$ such that $\forall e_{out} \subseteq E_{out}(v)$, $init(e_{out}) = v$. In case of *loop* one is added to both, *indegree* and *outdegree*. Then $d(v) = d_{in}(v) + d_{out}(v) - \mid E_{in}(v) \cap E_{out}(v) \mid$. This definition was adopted from Bachman [8].

Bijection A function $f : A \to B$ is *injective* if $\forall x, y \in A$, $x \neq y \Rightarrow f(x) \neq f(y)$. Thus a function $f : A \to B$ is *injective* if and only if each element of set B has at most one preimage under f. A function $f : A \to B$ is *surjective* if for all $b \in B$, there exists at least one $a \in A$ such that $f(a) = b$. Thus, a function $f : A \to B$ is *surjective* if and only if each element of set B has at least one preimage under f. Finally a function

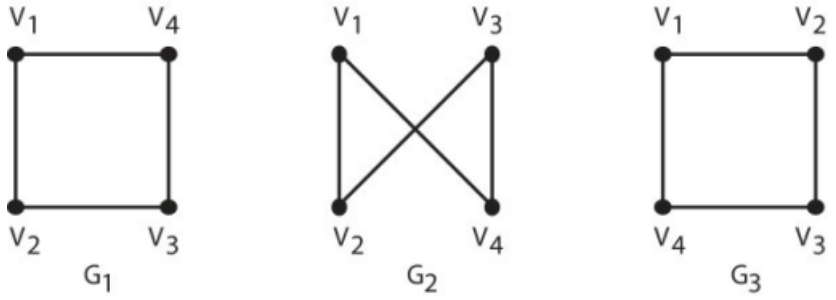


Figure 1.1: Isomorphic and automorphic graph examples

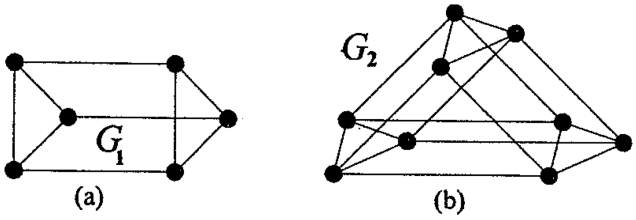


Figure 1.2: (a) G1 is vertex, but not edge symmetric (b) G2 is edge symmetric

$f : A \to B$ is a *bijection* if f is both *injective* and *surjective*. This definition was adopted from Gross and Yellen [9].

Isomorpishm, automorphism Let $G = (V, E)$ and $G' = (V', E')$ be two graphs. Graphs G and G' are called *isomorphic*, denoted $G \simeq G'$, if there exists a bijection $\varphi : V \to V'$ with $\forall u, v \in V, (u, v) \in E \Leftrightarrow \varphi(u)\varphi(v) \in E'$. Such a map φ is called an *isomorphism*; if $G = G'$, it is called an *automorhism*. Figure 1.1 shows three isomorphic graphs. The permutation producing the third graph from the first is an automorphism of the first graph [10].

Graph pattern In the context of this thesis, graph pattern is a graph $GP = (V, E)$, where $\forall v \in V(GP), d(v) > 0$. That means that each vertex $v \in V(GP)$ must be incident with at least a single edge $e \in E(GP)$.

Graph pattern matching Let $G = (V, E)$ be a graph and $GP = (V_{GP}, E_{GP})$ be a graph pattern (as it is defined above). In the context of this thesis, graph pattern matching is a process of finding all matches in a graph G for a given graph pattern GP. More specifically, it is a process of finding all subgraphs of G that are isomorphic to GP. Information about graph pattern matching is partially taken from Fan, Wan and Wu [11].

Graph symmetries Let $G = (V, E)$ be a graph. G is *vertex symmetric* if $\forall u_1, u_2 \in V(G)$, $\exists$ *automorphism* f such that $f(u_1) = u_2$. G is *edge symmetric* if $\forall e_1, e_2 \in E(G)$, $\exists$ *automorphism* f such that $f(e_1) = e_2$.

(a) DFS (b) BFS

Figure 1.3: Traversing order of DFS and BFS algorithms

Every edge symmetric graph is vertex symmetric. Cycles are always edge symmetric. Figure 1.2 shows vertex and edge symmetric graphs. This definition was adopted from Tvrdík [12].

DFS and BFS[1]

Depth-first search (DFS) and breadth-first search (BFS) are two of the most useful graph traversal techniques. They allow one to search a graph in linear time and compile information about the graph. Both of them start from a root node. The way how such node is chosen from a graph depends on the purpose one of these algorithms is used for.

The algorithm of depth-first search starts from the root node and explores as far as possible along each branch before backtracking. It uses a stack (FIFO) to store visited nodes. Figure 1.3a shows the order in which a graph is searched.

The algorithm of breadth-first search starts from the root node and explores the neighbor nodes first, before moving to the next level neighbors. It uses a queue (LIFO) to store visited nodes. Figure 1.3b shows the order in which a graph is searched.

Random graphs [2]

A *random graph* is a graph in which some specific set of parameters take fixed values, but the graph is random in other respects. A graph $G = (V, E)$, as defined above, is a set of vertices and edges. Thus some values of these two sets can be taken fixed and some can be randomly generated when creating a random graph.

There are many ways how random graphs can be generated. For instance one can create a graph by fixing number of vertices n and the number of edges m. Then m pairs of vertices are chosen at random from all possible pairs and

[1] Information in this section was adopted from Kozen [13] and Wikipedia [14], [15].
[2] Information in this section was adopted from Newman [16].

connected with an edge. Even though this process works well, it is not robust enough when one wants to fix some specific characteristics of generated graph such as the average degree of vertex or the fact that the graph should have no multiedges or self-edges. For such cases *random graph models* were defined. To simplify, there are two basic models.

- $G = (n, m)$ is a random graph model that is defined as a probability distribution $P(G)$ over all simple graphs G with n vertices and m edges. The probability distribution is then chosen in respect with required graph characteristics.
- $G = (n, p)$ is a random graph model in which we have n vertices and we place an edge between each distinct pair with independent probability p. In this model, number of edges is not fixed.

There are many specific random graph models based on these basic ones. Here two of them are introduced.

- *Erdős–Rényi model* is a $G = (n, m)$ random model, where graph is chosen uniformly at random from the collection of all simple graphs of n vertices and m edges. Thus its probability distribution

  $$P(G) = \frac{1}{\Omega}$$

  where Ω is the total number of such simple graphs.
- *Barabsi–Albert model* is a random model where graph is being constructed iteratively. First a small number of vertices (m_0) with edges is created. Other vertices are being added one at a time. Each new vertex generates an edge to $m \leq m_0$ existing vertices. The probability that the new vertex will be connected to existing vertex i is

  $$p_i = \frac{k_i}{\sum_j k_j}$$

  where k_i is the degree of vertex i and the sum over k_j sums up degrees of all vertices in the graph. Notice that vertices with higher degree tend to quickly accumulate even more edges. This definition was adopted from Albert and Barabási [17].

1.2 Graph databases

Databases are tools for storing and retrieving data. Existing ones can be basically divided into two categories, SQL and NoSQL.

SQL databases were developed in 1970s when first data storage applications have appeared. Within these databases individual records are stored as

rows in tables (also called *relations*) , with each column storing a specific piece of data about that record. Data across tables can be linked via keys. Each row has such unique identifier. Structure and data types of these databases are fixed in advance. Thus to store a new data item the whole table must be changed. Vertical scaling is supported. In case of horizontal scaling it is possible to spread SQL databases over many servers, but significant additional engineering is generally required. SQL databases use SQL language to manipulate with data.

NoSQL databases are much younger databases with first implementations in 2000s. They are supposed to deal with limitations of SQL databases, particularly concerning scale, replication and unstructured data storage. For instance, horizontal scaling, as the way of adding new server instances, is very simple within these databases, since they automatically spread data across servers as necessary. Another advantage of these databases is that their schema is typically dynamic (also called *schema-free*). That means that records can add new information on the fly unlike SQL databases. Data within these databases are mostly accessed via simple APIs or SQL-like query languages. Data structure of NoSQL databases is no longer based on relational model but it varies on database type. We can organize existing NoSQL databases into several subcategories by storage model they use.

- *Key-value stores* have a single table with key and value columns only (also called *hash map*) as a data structure. All data is then stored within this table. It simplifies relational model as a whole complex of tables.

- *Column-oriented stores* are set between SQL databases and key-value stores. Although they use a single table like key-value stores do, it is not a simple hash map anymore. It consists of several predefined columns that are shared for all records. Within these columns each record can store different properties. Thus it expands key-value stores possibilities but keeps its dynamic schema.

- *Document stores* are optimized to work with document-oriented information also known as semi-structured data. They store whole documents and are very effective when retrieving partial information from them.

- *Graph databases* store data in graph. While using knowledge from graph theory they can provide users with data that could be way more difficult to retrieve when using SQL databases.

The number of NoSQL databases is rapidly increasing nowadays and new types are still being developed. Some of these NoSQL databases already have their own subcategories and some of them are being combined to retrieve even richer information and achieve the best performance possible when handling

with large volume data. Information in this section was adaped from Tiwari [3], and MongoDB [18].

As mentioned above, graph databases belong to the category of NoSQL databases. They are based on knowledge of graph theory since their data is stored within a graph structure. Once again graph consists of nodes and relationships between these nodes. Graph databases excel at dealing with highly interconnected data. These relationship-first oriented databases use the fact that links between things are at least as important as discrete information about these things when mining useful information from data. In such case graph structure is highly convenient since it enables effective traversing through nodes by following relationships [19].

There are many real world use cases when using graph structure to store data seems natural. The entire web for instance. It is defined as a set of hypertext documents connected via hyperlinks. In other words it is a huge graph, where documents are nodes and links are relationships between these nodes. Another example is a social graph. Social graphs demonstrate personal relations of internet users within social networks like Facebook or Twitter. There are many more real life examples of using graphs across all fields including science, government or business. In fact graphs can be found almost everywhere.

There are many existing graph database engines, some strictly graph oriented and some that combine graph structure with other database types. Neo4j, Titan and OrientDB are the most popular ones at the moment.

1.2.1 Neo4j

Neo4j is the highest ranked graph database engine according to DB-engines [20]. In fact it is an open-source NoSQL database implemented in Java and Scala running on the Java Virtual Machine. It has many features, but most importantly:

- Database is fully ACID. It means that all of its transactions guarantee: *atomicity* - a transaction will be either fully commited or fully reverted, *consistency* - a transaction will move database from one valid state to another, *isolation* - parallel transactions will run in isolation, *durability* - once a transaction was successfully commited all the changes it made to the system will be permanent.

- Database is provided in embedded or server mode. It means that Neo4j database can be run as a separated application or within an existing Java application.

- Database guarantees high performance queries. Neo4j has developed its own system to store data on disk which provides an effective data access. Due to efficient representation of nodes and relationships, database

guarantee constant time traversals for relationships in the graph both in depth and in breadth [21].

- Database is clustered. It means that several servers can connect and communicate to a single database. That offers two major advantages: fault tolerance - if a server fails, there are others to back up, load balancing - users are automatically allocated to the server with the least load.
- Database supports Cypher. Cypher is a very powerful graph query language for querying and updating data stored in a graph database. Similarly to SQL, it is a declarative language. Cypher uses ASCII-Art to make its queries human readable.

Labeled Property Graph

A graph defined by Diestel [7] is not enough powerful to be used to store all data possible. Hence, graph databases typically employ extensions of such data model. Neo4j defines and implements *Labeled Property Graph*. It is a graph data model that, again, consists of nodes and relationships but provides some additional characteristics. First of all, each node can have one or more labels that indicate in which categories specific nodes belong. If a node does not have any label assigned it is treated as anonymous. Each relationship must have exactly one label (also called *type*) assigned. Each node or relationship can have arbitrary number of arbitrary attributes. Each attribute consists of key, uniquely identifying the attribute, and a value [8]. All relationships must have direction assigned but can be treated as bidirectional when queried. Figure 1.4, adopted from Neo Technology [21], shows such an instance of Labeled Property Graph data model. Within this model data about books, their authors and about people who purchased them are stored.

Architecture [3]

Labeled Property Graph, defined in the section above, is a model that is consistent across graph database implementations but there are many ways how to encode and represent such graph in the database engines main memory.

Neo4j is a graph database with native graph storage. It, among other things, means that Neo4j exhibits so-called *index-free adjacency*. Such property corresponds to the way references are stored on a disk. In fact each node maintains direct references to its adjacent nodes. Thus each node acts as a micro-index of other nearby nodes. When using index-free adjacency, one traversing step can be $O(1)$ in algorithmic complexity. In other words, traversing in general is extremely cheap.

[3] Information about Neo4j architecture was adopted from Bachman [8] and Robinson, Webber and Eifrem [4]

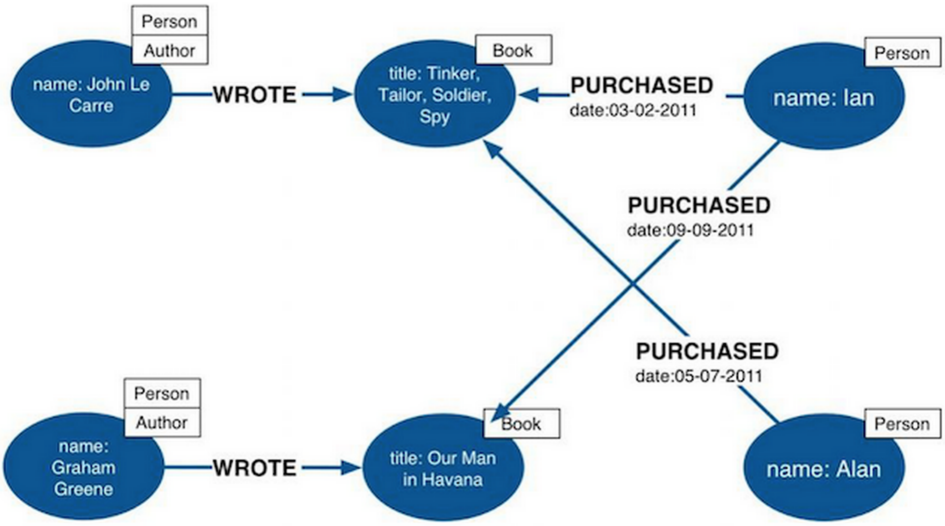


Figure 1.4: Labeled Property Graph

Figure 1.5 shows the high-level architecture of Neo4j. Database runs in JVM and exposes Core API to manipulate with data. Also Cypher, discussed at the beginning of the section describing Neo4j, can be used for such operations. On top of these basic components a number of other APIs for graph data storage and manipulation are provided, including a REST API, Traversal API and many others. Also custom APIs can be developed as server extensions [8]. Neo4j utilizes two types of caches:

- *File system cache*, also called *low level cache*, caches the Neo4j data in the same format as it is represented on the durable storage media. This cache layer is provided to speed up I/O operations. File system cache operates on Operating System layer.

- *Object cache*, also called *high level cache*, caches individual nodes, relationships and their properties in a form that is optimized for fast traversal of the graph. Nodes and relationships are stored as Java objects as soon as they are first accessed. Object cache operates on JVM layer.

Transaction management handles all database transactions. As mentioned before, Neo4j supports the ACID properties in order to fully maintain data integrity and ensure good transactional behavior. All transactions are always durably written to transaction log, which can be used to recover the store files in the event of a crash.

Data is physically stored in multiple separated record files. As shown in figure 1.5, each of these files contains the data for a specific part of the graph such as nodes, relationships, node labels, relationship types and properties for

both nodes and relationships. How exactly nodes and relationships are stored is shown in figure 1.6 and described below:

- *Node store file* stores node records. Each node is represented with exactly 15 bytes. Fixed-size records enable searching nodes by ID at cost $O(1)$. That can be achieved by calculating the offset, simply $ID * 15$, and jumping to that position in the node store file. The first byte of a record is the in-use flag that indicates if the record is currently being used to store a node. The next four bytes represent the ID of the first relationship connected to the node. The next four bytes represent the ID of the first property of the node. The next five bytes represent labels point to the label store for this node. The final byte is reserved for flags.

- *Relationship store file* stores relationship records. Again, to achieve optimal search algorithmic complexity, fixed-size records are used. Each relationship is represented with exactly 34 bytes. The first byte is the in-use flag, same as for node records. The next four bytes represent the start node and other four bytes represent the end node of the relationship. The next four bytes are reserved for a pointer to the relationship type stored in relationship type store. Then there are pointers for the next and previous relationship records for each of the start and end nodes, each by four bytes. The final byte is used for so-called *relationship chain*, that is the key component of Neo4j's traversal framework.

Data manipulation [4]

In Neo4j, there are two basic approaches to query and update data within the database:

Imperative approach by providing the engine with specific instructions about how data should be processed. In that case, the engine only follows these instructions step by step. Data can be manipulated in such way using *Core API*. Core API handles graph data as Java objects and using its methods one can easily do elementary operations including creating and deleting both nodes and relationships or changing their properties and labels. Since Core API provides low level data access, it is not always easy enough to use it for advanced use cases involving data retrieval operations. For that reason, Traversal Framework API was developed on top of Core API. Figure 1.7 shows how traversing works when using Traversal Framework. Its main component, *Traversal Description*, defines traversing process by:

[4] Information about data manipulation within Neo4j database was adopted from Neo4j manual [22]

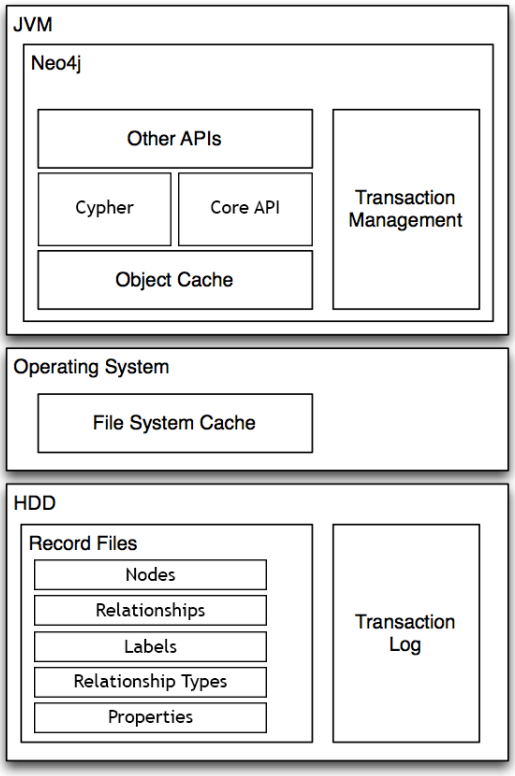


Figure 1.5: Neo4j high-level architecture

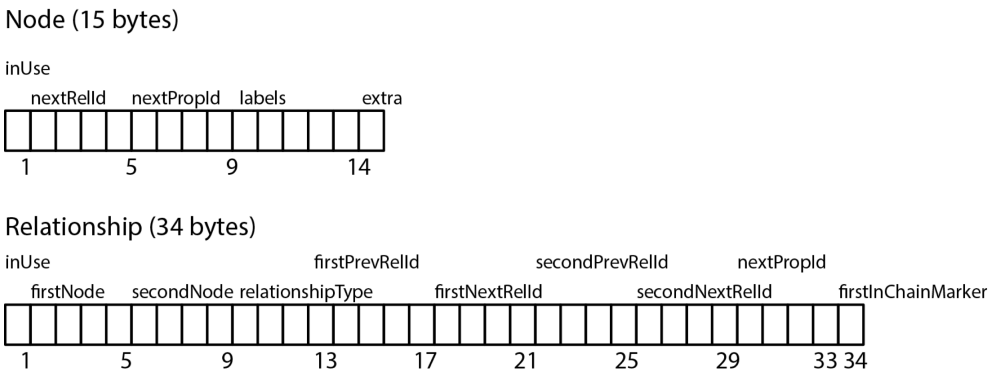


Figure 1.6: Neo4j data storage

- *Order* defines that way graph should be explored.
- *Evaluator* decides what should be done at a specific position in the graph when traversing.
- *Uniqueness* declares if nodes or relationships can be revisited when traversing.
- *PathExpander* names allowed ways to traverse.

The whole process of traversing is handled by *Traverser* component that holds result of traversing.

Declarative approach by declaring what should be done with data without providing the engine with detailed instructions. In that case the engine itself decides how to achieve these desired results. Data can be manipulated in such way using *Cypher*. Cypher, as mentioned above, is a SQL-inspired graph query language that allows for expressive and efficient querying and updating data within the graph database. Each Cypher query is turned into what is called *execution plan*. Such plan consists of small pieces called operators. Each operator is responsible for a small part of the overall query. There are two strategies how a query can be turned in such plan:

- *rule based planner* produces execution plans based on hard coded rules.
- *cost based planner* uses statistics about graph to assign cost to alternative plans and picks the cheapest one.

Even though basic Cypher queries are used in this thesis, describing its syntax is out of its scope. Cypher syntax is described in detail in Robinson, Webber and Eifrem [4].

Indexes [5]

A database index is a data structure that improves speed of retrieving data by providing the database with quick jump points on where to find the full references, much like book indexes. Although indexes speed up querying data, they also bring additional cost in terms of maintaining them and also require additional storage space. They are mostly valuable in situations when the number of database read operations is significantly higher than the number of database write operations.

Three options how to index discrete data (i.e. nodes and relationships) within the database have been introduced in Neo4j so far:

[5] Information from this section was adopted from Small [23]

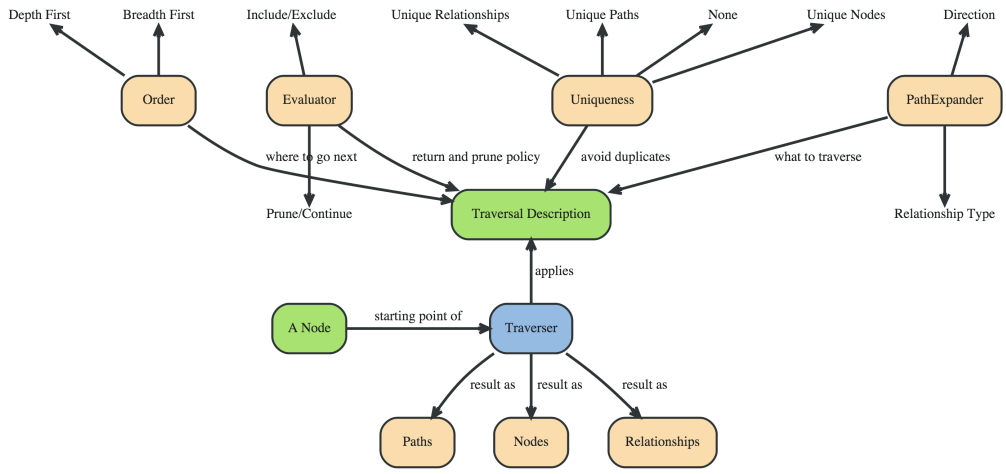


Figure 1.7: Traversal Framework API

- *Legacy indexes*, also called *manual indexes*, are powered by Lucene (open-source search software) outside the graph. These allow nodes and relationships to be indexed under a *key:value* pair. To use such index, one needs to handle all operations such as creating index, adding, deleting or updating data within the index manually. Legacy indexes are generally used as pointers to start nodes for a query but they provide no automatic ability to speed up queries.

- *Auto-indexes* are basically legacy indexes that are updated automatically. They need to be configured before a database is started. Also only one automatic index for nodes and one for relationships can be configured. That is done by listing the properties that are to be indexed.

- *Schema indexes*, introduced in Neo4j 2.0, are built around the concept of node labels. Such indexes can be created over given property for all nodes that have a given label. They are also automatically managed and kept up to date by the database whenever the graph is changed. Since schema indexes use labels, they are only available for nodes. Unlike legacy indexes, these indexes can actually speed up queries. If relationships indexing or full text indexing (provided by Lucene) is not needed, it is preferable to use schema indexes.

1.2.2 GraphAware Framework [6]

GraphAware is an open-source Neo4j framework. It was first introduced in Bachman's MSc Thesis called *GraphAware: Towards Online Analytical Pro-*

[6]Information from this section was adopted from GraphAware Git repository [24] and during personal consultations with Michal Bachman, author of GraphAware framework.

cessing in Graph Databases [8]. The framework aims at speeding up development with Neo4j by providing a platform for building useful generic as well as domain-specific functionality, analytical capabilities, graph algorithms, and more [25].

GraphAware framework was mainly built to help with solving some of Neo4j advanced use cases. Among other things, the framework is useful when dealing with following use cases:

- **Custom APIs**, also called *unmanaged extensions*, are pieces of code implemented and tested in Java. These are deployed with Neo4j in the same instance with the purpose to somehow extend its functionality. Using the framework, one can easily implement such API for many different purposes, such as providing functions that are missing in Cypher, providing custom input/output formats or restricting database access.

- **Transaction-Driven Behavior** is based on the fact that Neo4j enables to hook into the transaction handling process and inspect these transactions that are being processed. In other words, one is able to go through all database changes the transaction would make if it was successfully commited and perhaps modify the transaction in some way. Such functionality can be handful when solving many use cases such as notifications of modified data, additional modifications (e.g. assigning external IDs to newly created nodes) or schema enforcement (i.e. enforcing schema in the database, that is natively schema-free). GraphAware framework makes it easy to build, test, and deploy custom Transaction-Driven functionality by providing an appropriate interface.

- **Asynchronous Computation** is useful when one wants to run some background jobs in Neo4j. In that case, GraphAware framework makes it possible to build such functionally that is executed in the background during quite periods. Neo4j is primarily OLTP, meaning that requests drive transactions. That means quite periods are entered when the main, request driven, transaction process slows down, typically over night. Asynchronous computation is essential for many use cases including precomputing recommendations based on very time-consuming algorithms or continuously gathering any kind of similarities, centralities or statistics.

From the high-level point of view, GraphAware framework can be divided into two main components:

- **GraphAware Server** is a Neo4j server extension that allows building custom REST APIs [7] on top of Neo4j using Spring MVC [8]. Such custom

[7] REST APIs are such APIs whose methods can be accessed via HTTP methods.

[8] Spring MVC is a Java framework that, among other things, implements model-view-controller pattern.

APIs, as mentioned above, can be easily deployed and used as Neo4j extensions within the database.

- **GraphAware Runtime** is a runtime environment for both embedded and server Neo4j database deployments. This component allows to use pre-built GraphAware modules or to create custom modules that typically extend the core functionality of the database by:
  - transparently enriching/modifying/preventing ongoing transactions in real-time (Transaction-Driven Behaviour)
  - performing continuous computations on the graph in the background (Asynchronous Computation)

When using the framework one can easily implement a custom module that provides similar functionality to ones mentioned in this section and expose its methods via REST API. Such module can be then deployed within the database as a Neo4j extension. A number of such modules are already provided within the framework. Following ones are used in the thesis:

- **Algorithms** is a library of graph algorithms for Neo4j. It includes two key functionalities, *graph generators* and *path finding*. Graph generators consist of a number of randomly generated graph implementations. Erdos-Renyi model and Barabasi-Albert model, mentioned in section with graph theory, are also included. Randomly generated graphs are usually used for testing purposes. Path finding, on the other hand, allows to find a given number of shortest paths between two nodes. It extends the original shortest path finding functionality in Neo4j, so one can specify the desired number of results. Methods from the library are also exposed via REST API.
- **GraphAware Test** is a module that enables to test code that, in some way, interacts with Neo4j database. It enables to test code on three different levels:
  - **GraphUnit** enables simple graph unit-testing.
  - **Integration testing** enables integration testing of Neo4j-related code.
  - **Performance testing** enables to measure performance of Neo4j-related code or to run experiments with such code. Multiple parameters can be set when running these tests, including level of cache used.
- **Improved Transaction Event API** provides a convenient way to find out what data is about to be modified during the transaction that is about to be committed. To keep data safe, Neo4j requires every mutating operation on the graph to be run in transaction. Moreover Neo4j

provides an interface with the possibility to react to these mutations right before they are commited or right after they are commited. That is useful when one wants to prevent some mutations from happening, to log changes or to perform additional mutations to the graph. GraphAware framework extends such functionality so information about mutations within single transaction can be easily accessed. It is then very easy to access information about all created, deleted and changed both nodes and relationships.

Although there are many more modules already implemented and ready to use in GraphAware framework, they are out of the scope of this thesis. However, all of them accessible and documented in GraphAware Git repository [24].

Chapter 2

Analysis and design

Labeled Property Graph as a data model enables to store not only discrete information about things, but also relations between these things. When retrieving data from a graph database, one may want to query not only single nodes or relationships, but also more complex units consisting of these basic elements. Such units, called *graph patterns* [9], can contain valuable information for many use cases. The fact that 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 graph databases usually provide.

Even though Neo4j provides several APIs for data manipulation including Core API and Traversal API, Cypher is usually used for such operations. Since Cypher uses ASCII-Art to express queries, it is very easy to use and thus very popular. Process of graph pattern matching can be also done using Cypher. One can easily express a graph pattern as a Cypher query that is then turned into an execution plan that ensures to find all matches for given graph pattern in the database.

A wide variety of graph patterns can be found across different graph databases. 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 $GP = (V, E)$, where $V = \{v_1, v_2, v_3\}$ and $E = \{(v_1, v_2), (v_2, v_3), (v_3, v_1)\}$, is called a *triangle*. In Cypher, a triangle can be expressed in a few different ways, but preferably, in the context of this thesis, as

$$(n_1) - [r_1] - (n_2) - [r_2] - (n_3) - [r_3] - (n_1)$$

. A triangle is used to demonstrate individual steps during analyzing and designing a new method of indexing graph patterns.

Figure 2.1 shows a specific triangle that comes from a social graph. It visualizes people and friendships between them. There are three people, represented as nodes, with label *Person* and each with its *name* property in the

[9] Graph pattern definition can be found in graph theory section of chapter 1.

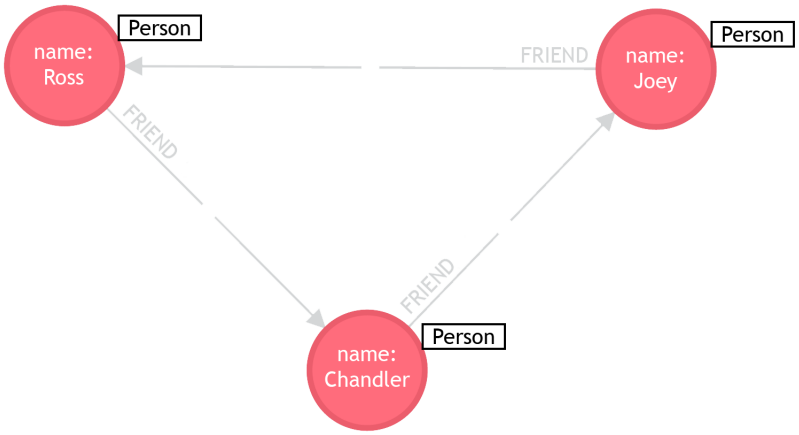


Figure 2.1: A specific triangle from social graph

triangle. Relationships of type *FRIEND* are directed, as it is restricted in Labeled Property Graph, but can be handled as bidirectional. It is very easy to retrieve such specific triangle using Cypher, since there is already some information known about it. The problem arises when one wants to retrieve all such triangles of people with their friendships. If the social graph consists only of *Person* labeled nodes and relationships of type *FRIEND* then Cypher needs to scan the whole graph to retrieve such data. Scanning the whole graph for the purpose of finding all matches for given graph pattern is very time-consuming. Thus a new method of indexing graph patterns is introduced. The main goal of the method is to speed up the process of graph pattern matching for general graph patterns.

2.1 Requirements

2.1.1 Functional requirements

- the method will provide following index operations:
  - create index
  - delete index
  - query using index
  - update index after DML operations
- the method will provide index operations via following REST API:

- GET `http://.../pattern-index/{indexName}/{cypherQuery}` to execute query using *indexName* index. The output (i.e. result of the query) will be in JSON format.
- POST `http://.../patern-index/{indexName}/{graphPattern}` to create a new index, further referred as $indexName$, with given graph pattern.
- DELETE `http://.../pattern-index/{indexName}` to delete *indexName* index.
- GET `http://.../pattern-index/count/{indexName}` * to get the number of indexed graph pattern units in *indexName* index.

2.1.2 Non-functional requirements

- The prototype of the method will be implemented in Java.
- The prototype of the method will be implemented as a plugin for Neo4j graph database engine.

2.1.3 Limitations

- When creating index or querying using index, all nodes and relationships will be named using identifiers within the *MATCH* clause of Cypher used. Such restriction prevents from vague graph pattern definitions. For example:

  $$(n_1) - [r_1] - (n_2) - [r_2] - (n_3) - [r_3] - ()$$

  can be understood as

  $$(n_1) - [r_1] - (n_2) - [r_2] - (n_3) - [r_3] - (n_1)$$

  but also as

  $$(n_1) - [r_1] - (n_2) - [r_2] - (n_3) - [r_3] - (n_4)$$

  . First one represents a triangle and second one does not (i.e. they are different graph patterns). One can easily prevent such confusion by naming all nodes and relationships as suggested.

- When querying using index, following Cypher query format will be expected:

  MATCH <pattern> WHERE <condition> RETURN <expression>

- When querying using index, there will be no restrictions involving IDs of nodes in *WHERE* clause of the query. If there is any such restriction applied, it is suggested to query without using index. If Cypher is aware of any IDs of nodes, it can locate them by these given IDs in $O(1)$ and look for graph pattern matches only in their neighborhood instead of scanning the whole graph. Such process is very effective and thus a new method would not bring any improvement (i.e. the method of graph pattern matching is to be used for general graph patterns).

2.2 Analysis

2.2.1 Graph pattern index storage

Indexes, as mentioned in the first chapter, are used to speed up querying data in some specific use cases. Such performance improvement is done by providing the database with quick jump points on where queried data is. Then the whole database does not have to be scanned in the purpose of finding them. In the context of graph pattern matching, matches for given graph pattern within the database (also called *graph pattern units*), represent such data. Thus all these graph pattern units must be found in the database at first and then index that maps positions of these units within the database must be created. Since it is created, one can execute queries using such index.

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. Also the index data structure itself can vary. In this thesis two of many different approaches of storing graph pattern indexes are introduced. One of them is then used when implementing a new method of indexing graph patterns and the other is used in concurrent master's thesis.

- Indexes are stored inside the graph database where the actual data is. Thus they use a graph to store its data. The advantage of this approach is that graph pattern units can be referred directly from the index. Another advantage is that indexes can be queried using Cypher as well as other data in the database. The main disadvantage of this approach is that index data (i.e. meta data) can be mixed with the actual data within the database. That can happen in case the database does not support meta data. Neo4j, for example, does not support them at the time of writing the thesis. Although they are not physically separated, they can, at least, be separated on the logical level by assigning self-made meta labels to them. This approach is used when implementing a new method of indexing graph patterns in this thesis. Thus detailed information concerning advantages and disadvantages of this approach is provided in following section of this chapter.

- Indexes are stored outside of the database where the actual data is. They are stored in a key-value store that uses a map (i.e. a collection of key-value pairs) as a data structure. The advantage is that key-value stores provide high performance read and write operations. Another advantage is that index data is physically separated from the actual data. The disadvantage is that indexes cannot directly refer their graph pattern units. That means they do not keep actual pointers to the database but they keep identifiers instead. Based on these identifiers, graph pattern units must be found in the graph database. Another disadvantage is that transactions must be handled across two different databases. A two-phase commit protocol must be used to preserve ACID transaction properties.

2.2.2 Graph pattern variations

It is important to mention that a new index must be created for each different graph pattern (i.e. index that was created based on a specific graph pattern cannot be used when querying different graph pattern). Property Labeled Graph, as already mentioned, extends a graph model by providing nodes with labels, relationships with types and enables to use properties for both nodes and relationships. Thanks to these additional graph characteristics, there can be multiple variations of a single graph pattern expressed. These variations share the same structure but differ in some of these mentioned characteristics and thus represent different data. For example,

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

and

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

are two different variations of a triangle expressed in Cypher. Both obviously share the same graph pattern structure. However, the first one will match all triangles whose nodes will be *Person* labeled, whereas the second one will match those, whose nodes will be *Dog* labeled. Thus they will match exclusive sets of triangle units.

All variations of a single graph pattern can be organized into a tree-like structure, called *graph pattern tree of variations* [10]. Part of such tree for a triangle is shown in figure 2.2. 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

[10] Graph patten tree of variations is a term defined for the purpose of this thesis.

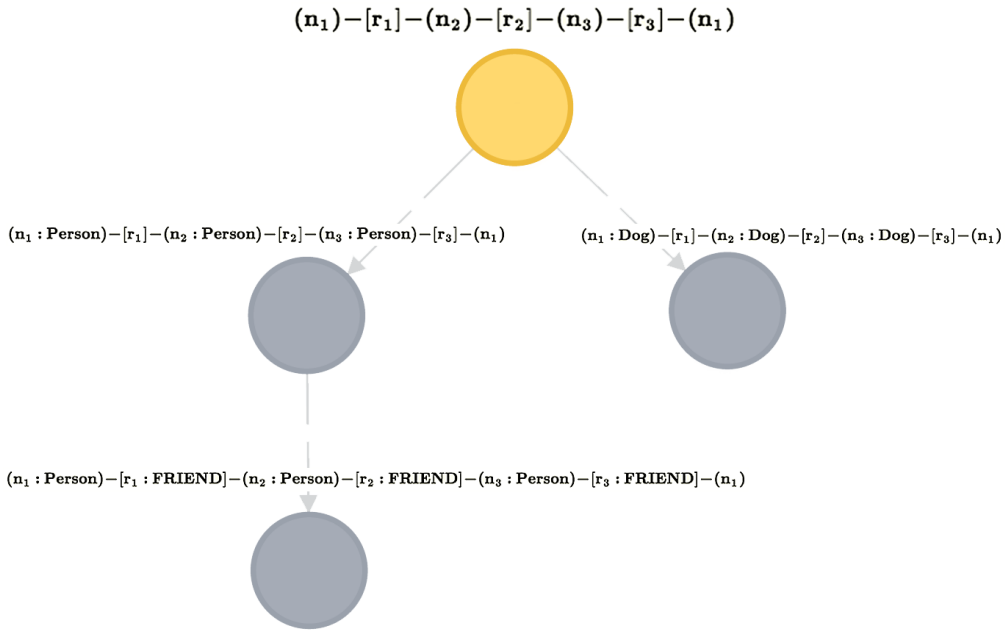


Figure 2.2: Tree-like structure with graph pattern variations

information about graph pattern is specified). Leaves of such tree represent specific graph pattern units (with IDs of nodes and relationships declared) within the database.

Graph pattern can be represented by a set of its variations. When querying a graph pattern, one actually queries a specific variation of such graph pattern. Thus it must be said explicitly which one is queried. If it is not said so, the basic variation with no additional information about nodes and relationships is used.

Each graph pattern's variations can be organized in its graph pattern tree of variations. Such structure is useful when querying any graph pattern's variation using index.

Lemma 1. *Index, created for a specific variation of a graph pattern, can be used to query the variation and also all of its descendants within its graph pattern tree of variations.*

Proof. Let $Q = (V, E)$ be a specific variation of a graph pattern that the index was created for and $D_Q = \{D_1, D_2, \ldots, D_n\}$ a set of variations that are in the position of descendant nodes of Q within its graph pattern tree of variations. Let $M(GP)$ be a function that for a variation of a graph pattern, given as a parameter, returns the set of graph pattern units within the database. Each

variation in the position of a descendant node within the graph pattern tree of variations share the structure with its parent node and adds some additional information about nodes and relationships. It implies $\forall D \in D_Q, M(D) \subseteq M(Q)$. The index maps all graph pattern units of a variation it was once created for. Therefore it can be also used for descendants of the variation, since a set of graph pattern units for each of them is a subset of the variation's set of graph pattern units. Then, by filtering units from the variation set based on additional information of descendant variation, the subset is achieved. □

The more Cypher knows about the queried graph pattern, the more it filters graph space based on such information and thus the more effective it is. That means if one wants to query the basic graph pattern variation that represent the root node of its graph pattern tree of variations, it will be faster to query it using the index instead of using a simple Cypher query. 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) will be faster when using a simple Cypher query. In other words, there is a level of the tree, where querying variations on such level stops being more effective when using the index compared to simple Cypher queries. The height and the depth of the tree depends on the graph pattern the index was created for and data within the database.

2.2.3 DML operations affecting index

As mentioned above, a graph patten 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, they would not provide reliable information when queried.

First of all, it is necessary to define all elementary DML operations that can affect an index. Transactions consist of one or more of these elementary operations. An index is updated after each one of these operations is applied on a database. The order in which they are applied is not important since they are not interdependent.

Creating a node

No existing graph pattern units can be destroyed and no new ones can emerge by creating a node. Since no existing data is affected by creating a new node, no existing graph pattern units can be destroyed. The index is up to date when creating a new node which means that the only graph pattern units

that could emerge would have to involve such node. Since the node has no relationships when created, it, by definition, cannot be a part of any graph pattern unit. Thus no new graph pattern units can emerge when creating a node.

Creating a relationship

No existing graph pattern units can be destroyed but new ones can emerge when creating a relationship. For the same reason as when creating a node, no existing units can be destroyed. The fact that new graph pattern units can emerge when creating a relationship is proven by an example. Let's say a database that contains, among other data, a triangle shown in figure 2.1 is used. Also an index for a basic variation of a triangle, expressed in Cypher as:

$$(n_1)-[r_1]-(n_2)-[r_2]-(n_3)-[r_3]-(n_1)$$

is created. Let's say a new relationship of *BROTHER_IN_LAW* type is created between nodes representing people with names Ross and Chandler. Then there are two different triangles that share the same nodes in the database. One of them is mapped by the index because it existed within the database before creating a new relationship. The second one is not. Thus a new graph pattern unit has emerged and the index must be updated.

Deleting a node

No new graph pattern units can emerge and no existing ones can be destroyed when deleting a node. The index is up to date and no new data is added to the database during deleting a node. Then no new graph pattern units can emerge. The fact that no existing graph pattern units can be destroyed while deleting a node is proven by an example. There are three possible ways how a database can handle the operation of deleting a node:

1. Only a node with no relationships on it can be deleted. Otherwise it is not allowed to do so (i.e. deleting a node with some relationships on it causes an integrity constraint violation). When a database uses such approach, the only time a node can actually be deleted is when it does not have any relationships on it. In that case such node, by definition, cannot be a part of any graph pattern unit. Thus by deleting it, no graph pattern unit can be destroyed. When deleting a node that has one or more relationships on it, first its relationships must be deleted. Such process can be done by using a complex transaction composed of more elementary DML operations. In other words, the operation of deleting a node by itself cannot cause a destruction of any graph pattern units. This approach is used in Neo4j database engine and thus it is the approach that is handled when designing a new method of indexing graph patterns.

2. If a node gets deleted, all of its relationships get, by default, deleted as well. This approach is similar to the one above. The only difference is that by deleting a node, one can also delete its relationships without knowing it.

3. The relationship remains in the database even though one or both of its end nodes were deleted. This approach is ignored here, because it is in a conflict with the fact that a relationship, in the context of this thesis, is defined by its end nodes.

Deleting a relationship

No new graph pattern units can emerge but some existing ones can be destroyed when deleting a relationship. For the same reason as when deleting a node, no new units can emerge. The fact that existing graph pattern units can be destroyed when deleting a relationship is proven by an example. Let's say a database with a triangle from figure 2.1 is used again. Also an index, created for a specific variation of a triangle, express in Cypher as:

$$(n_1)-[r_1 : Friend]-(n_2)-[r_2 : Friend]-(n_3)-[r_3 : Friend]-(n_1)$$

is used. The index maps the position of the triangle from figure 2.1 because it matches the graph pattern the index was created for. If Ross and Chandler are not friends anymore, the relationship between them gets deleted. Then the friendship triangle between them and Joey is broken. In such case, the index must be updated.

Updating a node

New graph pattern units can emerge and some existing ones can be destroyed when updating a node [11]. Such facts will be proven by an example. Let's say a database with a triangle from figure 2.1 is used again. Also two indexes are created for this database, both for different variations of a triangle. First for one expressed in Cypher as:

$$(n_1\,\{name : 'Chandler'\})-[r_1]-(n_2)-[r_2]-(n_3)-[r_3]-(n_1)$$

and second for one expressed as:

$$(n_1\,\{name : 'Rachel'\})-[r_1]-(n_2)-[r_2]-(n_3)-[r_3]-(n_1)$$

Let's assume, for the purpose of this example, that there is only one node with name Chandler and no nodes with name Rachel in the database. In this scenario, the first index maps only a single graph pattern unit, the one shown

[11] Updating a node covers operations including changing, creating or deleting a label of a node or changing, creating or deleting a property of a node.

in figure 2.1. The second one does not map any units. If a value of name property of node representing Chandler gets changed to Rachel, both indexes must be updated. The first one will map no units and the second one will map a triangle involving a node with name Rachel.

Updating a relationship

New graph pattern units can emerge and some existing ones can be destroyed when updating a node [12]. The operation of updating a relationship affects the index the same as the operation of updating a node. Note that changing type of a relationship is not supported in Neo4j. Such operation can be simulated by deleting the relationship and creating a new one.

2.3 Design

Designing of a new method of indexing graph patterns involves a few steps. First of all an index data structure is introduced. Such structure should be optimized so querying using an index is as much efficient as possible. Then a process of building a new index is introduced. Finally a process of updating indexes after DML operations is introduced. It is important to keep in mind that there can be multiple indexes created for a database (i.e. they all must handle updates after each DML operation to be consistent).

2.3.1 Graph pattern index structure

Indexes are stored within the graph database together with the actual data. They use a graph data structure to store its data since it is a native data structure of graph databases. More specifically they use a restrictive form of a graph, a tree. Such tree consists of exactly two levels. Indexes also use extended graph characteristics introduced in Labeled Property Graph.

Figure 2.3 shows a simple index for a triangle. Red colored nodes represent the actual data, whereas blue colored ones represent the index. Index data is separated from the actual 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 to relationships of the index. The root node on the first level of the tree provides a high level view of the index. Apart from the *META* label, it is also labeled with a special *PATTERN_INDEX_ROOT* label. The root node is identified by its properties including a name of the index and a graph pattern the index is built for. On the second level of the tree, there is a set of nodes that represent so-called *index units*. These are labeled with a *META* label again, but also with a special *PATTERN_INDEX_UNIT*

[12] Updating a relationship covers operations including changing, creating or deleting a type of a relationship and changing, creating and deleting a property of a relationship.

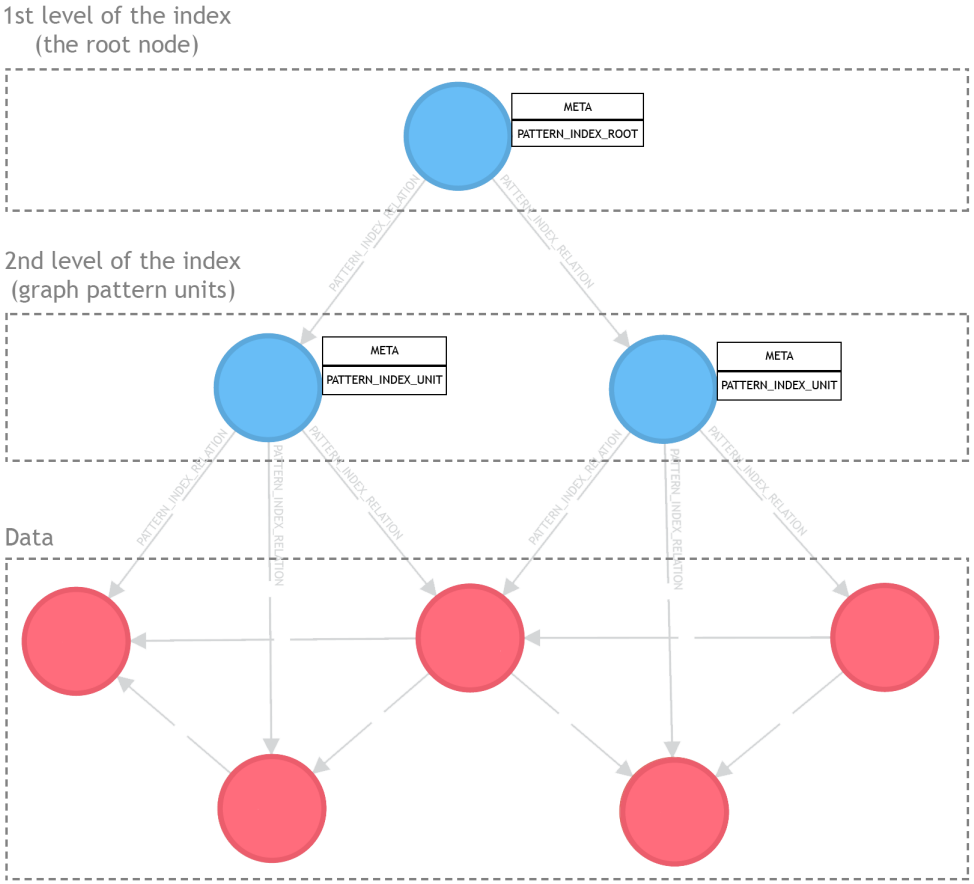


Figure 2.3: An index for a triangle graph pattern

label. Index units point to actual matches for the graph pattern within the database, called graph pattern units.

Note that there is a difference between graph pattern units and index units. Whereas graph pattern units are formed by actual data, index units belong to index data and point to actual graph pattern units. Also graph pattern units are uniquely identified by its relationships, whereas index units are identified by its nodes. That means a single index unit can point to multiple graph pattern units since there can be multiple relationships between two nodes. Thus each index unit has a *specificUnits* property that holds a list of all graph pattern units to which it points.

The fact that indexes use a tree data structure and store its data within the same database where actual data is, brings many advantages but some disadvantages as well. A brief summary of these is provided.

\+ Indexes must be updated each time a DML operation is performed

against a database. Thanks to the fact that index data and actual data is stored within the same database, a single transaction can be used to perform DML operations and to update indexes at the same time. In other words, no distributed transactions are needed and thus a two phase protocol does not have to be used.

+ Index data can be queried in the same way actual data is. For example, if one wants to know about all graph pattern indexes that are created for a database, it is only necessary to find all nodes with *PATTERN_INDEX_ROOT* label. Also, one can easily find out how many index units a single index consists of by counting outgoing relationships of type *PATTERN_INDEX_RELATION* of appropriate root node. Querying index data can be done by using a graph query language or by using APIs that are provided by a specific graph database engine.

+ Indexes point directly to graph pattern units from index units via relationships of *PATTERN_INDEX_RELATION* type. Thus it is easy to detect if a data node is part of a specific graph pattern unit by checking incoming index relationships on such node. This is very useful when handling a process of updating indexes. For example if a data node gets modified, one can easily detect which graph pattern units involve such node and thus which index may need to be updated.

− If a database does not support metadata, index data and actual data is mixed on the physical level. In this case, data is only separated on the logical level by assigning a special label to index nodes and a special type to index relationships. For example if one wants to get a total number of actual data nodes within the database, it is necessary to explicitly exclude nodes of index data. Also additional index data may affect database statistics that are used to evaluate execution plans for queries. Cypher, for instance, uses two types of planners for evaluating such plans. One of them, cost planner, uses these statistics. Thus it is sensitive to data in the database when choosing appropriate execution plan for a query. Moreover additional graph pattern units can be formed by combining index nodes and actual data nodes. Such phenomenon can be seen in figure 2.3. There are two additional triangles that are formed by nodes from 2nd level of the index together with actual data nodes. Since index data is not to be seen by users of the database, these triangles cannot be valid graph pattern units. Thus they must be ignored when creating a new index.

− A tree graph pattern index structure within a graph database require more storage space compared to, for example, a simple external hash map that can be used for such purpose as well. Each index is stored within a two level tree. Exactly $u + 1$ nodes are used for each index,

where u is the number of index units within such index. Each of these nodes have some properties. Also $u + (u * n)$ relationships are used, where n is the number of nodes the indexed graph pattern consists of.

2.3.2 Creating an index

The main goal when creating an index for a specific graph pattern is to build an index tree structure as it is described above. First of all, a database must be scanned in order to find all graph pattern units that match given graph pattern. Then for each group of graph pattern units that share the same nodes, a single node representing an index unit is created. Also all nodes within a group of graph pattern units are linked to appropriate index unit using a *PATTERN_INDEX_RELATION* relationship. For each graph pattern unit an identifier is created. Such identifier is created by concatenating IDs of relationships that form a graph pattern unit. Before the process of concatenating, IDs are first sorted in ascending order. The identifier is then stored within a *specificUnits* property of appropriate index unit. Finally, the root node of the index is created. An index name and a graph pattern that the index is built for are stored as properties within such node. To complete the tree structure of the index, each index unit is linked to the root node using a *PATTERN_INDEX_RELATION* relationship.

2.3.3 Querying using an index

Since an index is created, it is very easy to use it when querying. As mentioned above, each graph pattern index is represented by the root node of its index tree structure. Thus by retrieving all nodes with *PATTERN_INDEX_ROOT* label, one can gain basic knowledge about all graph pattern indexes that exist within the database. It is necessary to provide a name of the index that should be used when querying. Based on the name, appropriate root node is chosen from the set. Index units are reached by traversing outgoing relationships of type *PATTERN_INDEX_RELATION* from chosen root node. These nodes together hold a complete list of graph pattern units within the database. Also they have a direct access to nodes that form these units. Based on the relationship between a queried graph pattern variation and the one the index was built for, there two ways how matched graph pattern units should be retrieved.

- The queried graph pattern variation is the same as the one the index is built for. In this scenario, graph pattern units can be retrieved directly.

- The queried graph pattern variation is descendant of the one the index is built for. In this scenario, graph pattern units must be further filtered

based on additional information provided by queried graph pattern variation [13].

- If the relationship between these two is any other, the index cannot be used when querying given graph pattern variation.

2.3.4 Updating indexes

There are only a few operations that, if executed individually within single transactions, can affect graph pattern indexes. These operations, including creating and deleting a relationship and updating either a node of a relationship, are introduced in the analytical part of this chapter. This section further focuses on the way indexes are updated after one of these operations is applied to a database.

The process of updating indexes is explained for each of mentioned DML operations. It is demonstrated using a triangle index, but the process is identical for any other graph pattern. 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 commited, indexes will be updated. If a transaction is rollbacked, indexes will remain in the same state as they were the transaction was initialized.

Deleting a relationship

Figure 2.4 shows the process of updating a triangle index after a relationship is deleted. The process is described in a few steps.

1. The relationship with ID 5 is set to be deleted.
2. Before deleting the relationship, its end nodes are checked.
    - At least one of these end nodes does not have any relationships of type *PATTERN_INDEX_RELATION* on it. Then the relationship set to be deleted cannot be a part of any graph pattern unit. To be a part of a graph pattern unit, both of its end nodes would have to be involved in such unit as well. Thus both of them would have to have at least a single relationship of type *PATTERN_INDEX_RELATION* on them. In this case, the relationship can be deleted without the chance of affecting any of graph pattern units.
    - Both of its end nodes have some relationships of type *PATTERN_INDEX_RELATION* on them, but no pair of these relationships share

[13] Detailed information about variations of a graph pattern is provided in Graph pattern variations section of this chapter.

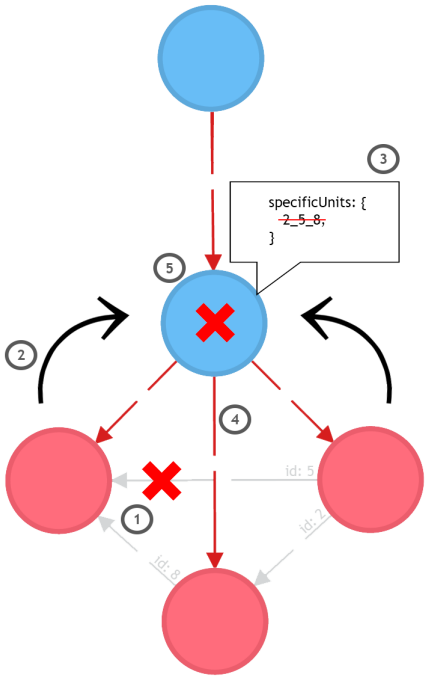


Figure 2.4: The process of updating an index after a relationship gets deleted

the same start node. It means that both of end nodes of the relationship that is set to be deleted are part of mutually exclusive sets of graph pattern units. In this case, the relationship is not part of any graph pattern units. Thus it can be deleted without the chance of affecting any of graph pattern units.

- Both of its end nodes have some relationships of type *PATTERN_INDEX_RELATION* on them and some pairs of these relationships share the same start nodes. In this case, both end nodes together are part of some graph pattern units. That means the relationship set to be deleted may or may not be a part of some of these units. For this reason common start nodes, representing index units, must be further inspected.

3. Index units that have relationships to both of end nodes of the relationship that is set to be deleted must be inspected. Note that these units do not necessarily belong to the same index. Each of these units has a *specificUnits* property that holds a list of graph pattern units that belong to it. Graph pattern units are represented within such lists by their relationships in a form of string identifiers [14]. All identifiers that

[14]Graph pattern identifiers are introduced in the analytical part of this thesis.

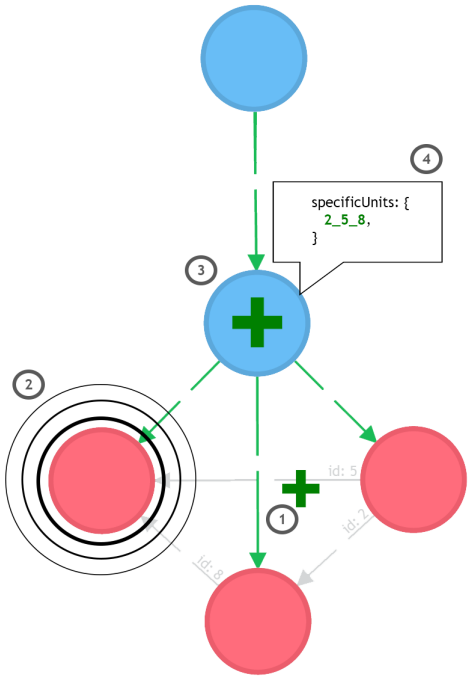


Figure 2.5: The process of updating an index after a relationship is created (appropriate index unit does not exist yet)

involve the relationship that is set to be deleted must be removed from appropriate lists. All index units that remain with empty lists within their *specificUnits* property after such operation must be deleted. They no longer map any graph pattern units.

4. Index units that no longer map any graph pattern units after step 3 must be deleted. However, they still have some relationships on them. Thus these must be deleted first. Each of index units has one or more outgoing relationships and exactly one incoming relationship on it. Outgoing relationships link an index unit to nodes that form its graph pattern units. Whereas the only incoming relationship links an index unit to the tree node of the index tree structure. All such relationships are deleted for each of these index units.

5. Index units whose relationships were deleted in step 4 are deleted themselves. After that, all indexes are successfully updated and the relationship that is set to be deleted can actually be deleted.

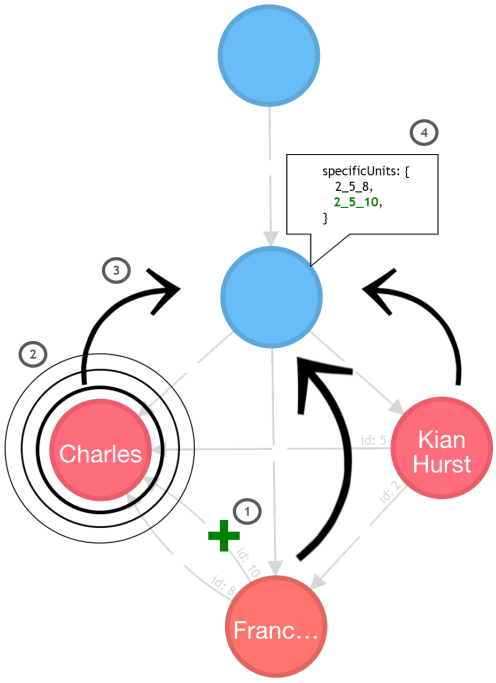


Figure 2.6: The process of updating an index after a relationship is created (appropriate index unit already exists)

Creating a relationship

Figures 2.5 and 2.6 show the process of updating a triangle index after a relationship is created. The process is described in a few steps.

1. The relationship with ID 5 is created (let's assume it did not exist within the database before).

2. Since this step, the whole process is done separately for each index within the database.

   It is necessary to find out if any new graph pattern units for an index emerged by creating the relationship. One of end nodes of newly created relationship is chosen randomly [15] in order to find all graph pattern units

[15] The intention of doing such operation is to find newly emerged graph pattern units. These must unconditionally involve the relationship whose creation started this process, otherwise they would already exist within the database before. Obviously they must also involve both of its end nodes and thus it does not matter which one is chosen in order to find newly emerged graph pattern units.

that involve such node for an index within a database. A set of found graph pattern units may be divided into three groups.

(a) A group of graph pattern units that already existed within the database before the operation of creating a relationship was applied. An index already map these, thus they further do not need to be handled in order to update it.

(b) A group of newly emerged graph pattern units whose nodes not yet form a different graph pattern unit within an index. These graph pattern units are handled separately, as it is shown in figure 2.5.

(c) A group of newly emerged graph pattern units whose nodes already form one or more different graph pattern units within an index. These graph pattern units are handled separately, as it is shown in figure 2.6.

3. This step vary for groups defined at step 2.

(b) Graph pattern units, as described at step 2(b), are grouped by their nodes (i.e. if two graph pattern units share the same nodes, they belong to the same group). For each of these groups a new index unit is created. Such index unit is then linked to the root node of an index via a relationship of *PATTERN_INDEX_RELATION* type. The index unit is also linked, using the same type of relationship, to each of nodes that form appropriate group of graph pattern units.

(c) Appropriate index units already exist for graph pattern units described at step 2(c). These must be found in order to be updated.

4. An identifier must be created for each of graph pattern units defined at step 2(b) and 2(c). Such identifier is then stored within a *specificUnits* property of appropriate index unit. At this moment an index is updated.

Updating nodes and relationships

It is important to remind that by updating nodes or relationships, new graph pattern units can emerge and some existing ones can be destroyed. It is not important if it is a node or a relationship that gets updated. The process of updating indexes is the same for both cases. It can be described in a few steps.

1. A node or a relationship gets updated. Such operation involves changing, creating or deleting a label of a node, a type of a relationship or a specific property of a node or a relationship.

2. Since this step, the whole process is done separately for each index within the database.

First of all the node of interest is set. It is either the node that gets updated or one of randomly chosen end nodes of the relationship that gets updated during the DML operation that started this process. Next all graph pattern units that involve the node of interest are found for an index within a database. It is the same operation as described in step 2 of the process of updating indexes after a relationship is created.

3. This step is identical to step 3 of the process of updating indexes after a relationship is created.

4. This step is identical to step 4 of the process of updating indexes after a relationship is created. After this step, an index is updated to newly emerged graph pattern units.

5. The node of interest can be part of one or more graph pattern units. Updating such node or its incident relationship can cause a destruction of some of these graph pattern units. Thus all graph pattern units that involve the node of interest and existed within the database before the update operation was applied must be checked. Those that get destroyed by the update operation must be removed from an index. That is done by deleting their identifiers from *specificUnits* property of appropriate index units.

6. Those index units that remain, after step 5, with empty list of graph pattern units within their *specificUnits* property must be deleted together with their relationships. Such operation is similar to steps 4 and 5 of the process of updating indexes after a relationship is deleted. After this step, an index is updated to graph pattern units that are destroyed by the update operation.

Chapter 3

Realisation

Implementation of a new method of indexing graph patterns includes a few operations, as it described in the analytical part of this thesis. First of all, the method provides the ability to create a new index for any given graph pattern. Such operation includes building a tree index structure within the same graph database, where actual data is. Next it enables to use any of created indexes in order to speed up queries involving appropriate graph patterns. Finally the method ensures that all existing indexes within the database are updated each time a DML operation is applied to the database.

3.1 Implementation background

The method of indexing graph patterns is implemented for Neo4j graph database engine. It is written in Java, since Neo4j itself is mostly written in Java. Neo4j provides Core API. While working with indexes, such API is used to communicate with a graph database.

`GraphDatabaseService` interface, the key part of Core API, is used as the main access point to a running Neo4j instace. 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. An example of a transaction encapsulation in Java:

```
GraphDatabaseService graphDb;
...
try ( Transaction tx = graphDb.beginTx() )
{
    // operations on the graph
    tx.success();
}
```

`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 access 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.

3.2 Implementation of the method of indexing graph patterns

The implementation of the method of indexing graph patterns involves several classes. `PatternIndexModel` class is the core class of the method. Figure 3.1 shows the class and its relationships with other classes of the method. 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.

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

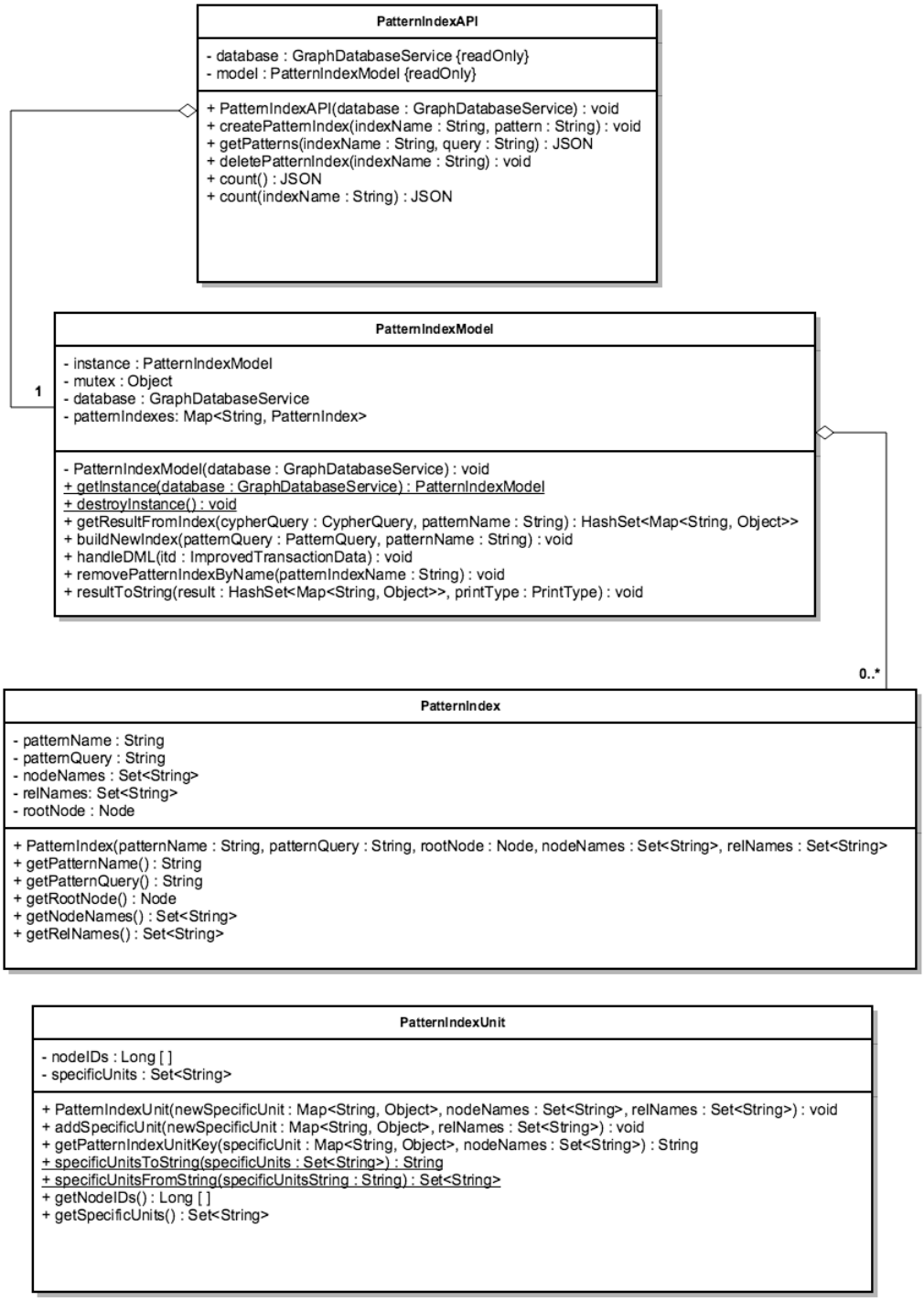


Figure 3.1: UML class diagram - main classes

`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 in the analytical part of this thesis, each index tree consists of a single root node and index units that map actual graph pattern units within a database. Instances of `PatternIndexUnit` 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. As mentioned above, Neo4j supports only primitive data types for properties. For this case, `PatternIndexUnit` class provides methods that enable to convert a list of identifiers representing graph pattern units to a single string and vice versa.

Next a few classes for parsing Cypher queries are introduced. Original Cypher parsers are written in Scala within Neo4j to maximize their performance. In the context of the method of indexing graph patterns, some additional parsing functionalities are needed. Thus three classes are introduced for such purpose, including the abstract `QueryParser` class and two other classes that inherit from such class, `PatternQuery` and `CypherQuery`. These are shown in figure 3.2. `PattenQuery` parses a *MATCH* clause of a Cypher query that represents a graph pattern. A user provides such graph pattern in order to create a new index. `CypherQuery`, on the other hand, parses a whole Cypher query. A user provides such query when querying using appropriate index. These two classes are implemented to provide mostly two functionalities that are essential for the method of indexing graph patterns.

- Validating a Cypher query, either a whole query or just its *MATCH* clause that is used to express graph patterns. The process of validation for both alternatives is shown in figure 3.3. First of all, original Neo4j parsers are used to validate the structure of Cypher. Then it is checked that all nodes and relationships within a *MATCH* clause are named using identifiers. Such validation is required due to limitations of the method. These are set in chapter 2. Finally, in case of a whole Cypher query, one more additional validation is needed. Such validation makes sure that there are no node ID restrictions in a *WHERE* clause of a Cypher query. Such limitation is also described further in chapter 2.

- Parsing out identifiers of nodes and relationships from a *MATCH* clause of a Cypher query. When one wants to create a new index for a specific graph pattern, the graph pattern must be provided in a form of a *MATCH* clause of a Cypher query. During this process, identifiers of

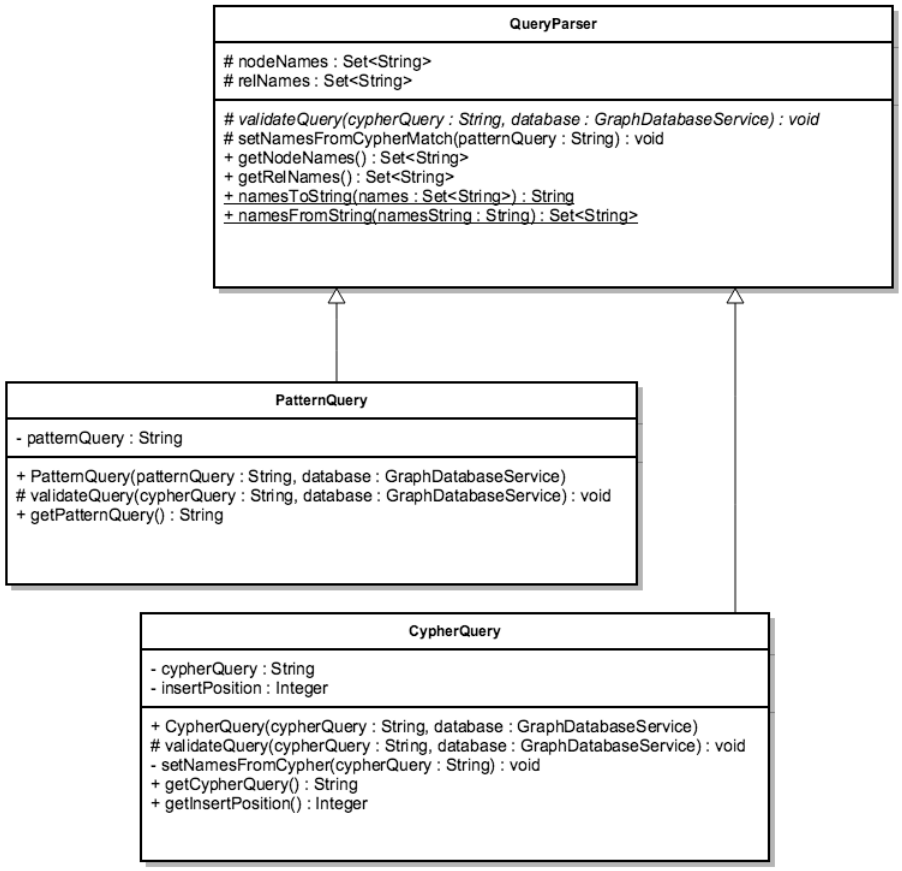


Figure 3.2: UML class diagram - Cypher parsers

nodes and relationships named by a user when providing a graph pattern are parsed out. Then they are set as `nodeNames` and `relNames` attributes of appropriate `PatternIndex` instance and further stored as properties of its root node within a database. Such data is needed when querying using an index and also when updating indexes after a DML operation is applied to a database.

`DatabaseHandler` is a class that handles most of database operations that involve querying and manipulating data. It consists only of static methods as it is shown in figure 3.4. These are used within methods of the main `PatternIndexModel` class that handle index operations. For example when building a new index tree, it is necessary to create a root node and some index units within a database. For this purpose, `createNewPatternIndexRoot` and `createNewPatternIndexUnit` methods are provided by the class.

All supportive classes are introduced now so it is time to finally describe main index operations. These are provided through main methods of the core

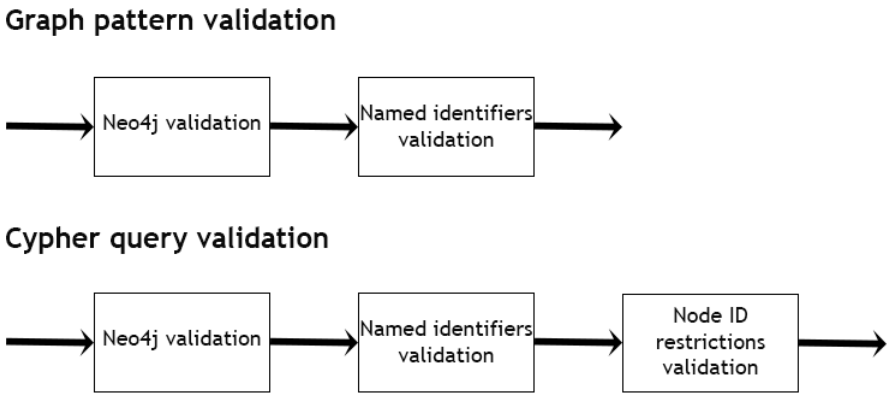


Figure 3.3: Cypher validators

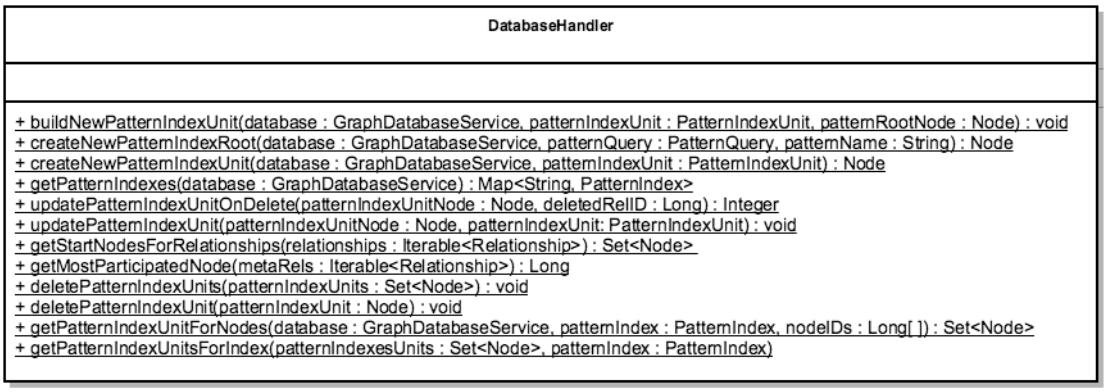


Figure 3.4: UML class diagram - database operations

`PatternIndexModel` class. Methods to create a new index, query using an existing one, delete an existing one, or to get the size of an existing index are also exposed via REST API. The API is provided within `PatternIndexAPI` class. Thus these operations can be either used directly by calling appropriate methods of `PatternIndexModel` or by HTTP requests when using the API. Other methods for updating indexes are called automatically each time a DML operation is applied to a database.

Before main methods of `PatternIndex` class for manipulation with indexes are introduced, it is important to remind the problem of mixing index data with the actual data. The problem is introduced in the analytical part of this thesis. Neo4j does not support meta data. Thus index data are assigned with special *_META_* label for nodes and special *PATTERN_INDEX_RELATION*

for relationships to separate them from the actual data on the logical level. Index data together with the actual data can actually form additional graph patterns within a database. These are not valid since index data is not to be seen by a user. To avoid them when operating with graph pattern indexes, it is necessary to explicitly exclude them based on their meta tags (i.e. label for nodes and type for relationships) from all Cypher queries. This is done within a *WHERE* clause of a Cypher query.

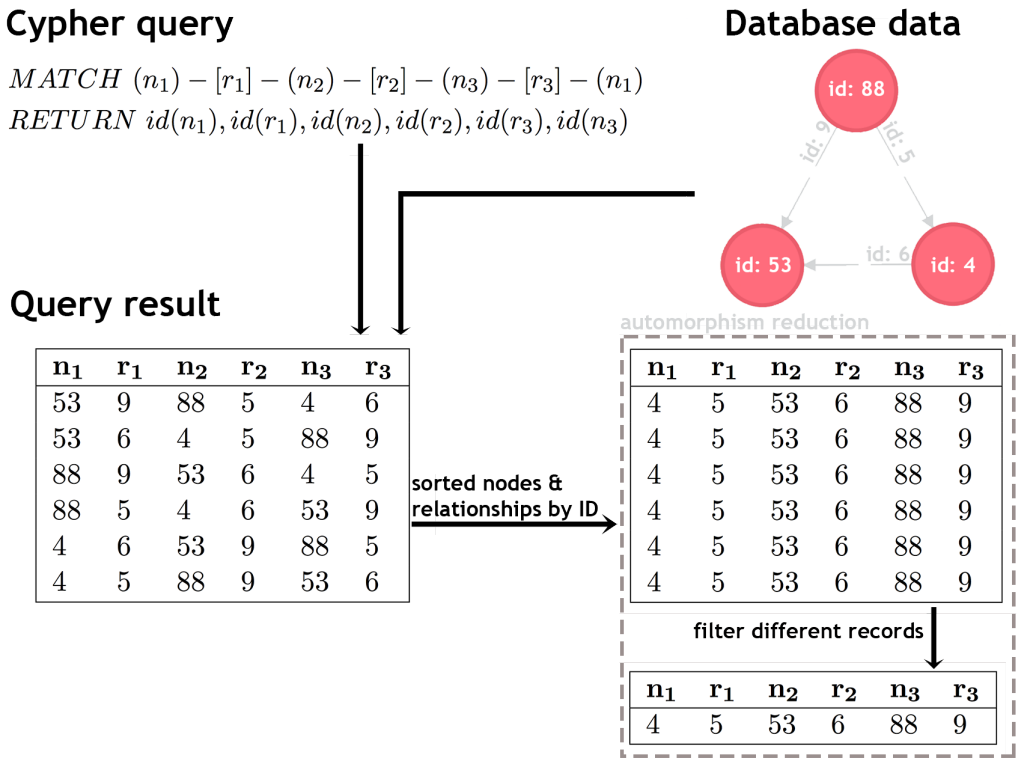
Now it is time to describe main four operations of the method of indexing graph patterns in detail.

Creating an index

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. Then such group can be also referred to as a group of automorphism groups of graph pattern units. Figure 3.5 shows a result for a query. In this case a 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.



n_1	r_1	n_2	r_2	n_3	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_3	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_3	r_3
4	5	53	6	88	9

Figure 3.5: Automorphism reduction

Deleting an index

To delete an index, `removePatternIndexByName` method is provided within `PatternIndexModel`. It is necessary to provide the name of an index to be deleted. An instance of `PatternIndexModel` provides a complete set of existing indexes within its `graphIndexes` attribute. It is a map that holds indexes as `PatternIndex` instances under their names. Thus appropriate `PatternIndex` instance representing the index to be deleted can be found using its name within the map. Such instance then provides a direct access to the root node of appropriate index tree. All index units within the tree are collected by traversing outgoing relationships of the root node. Then, all of them are deleted together with their relationships. The root node is the last one to be deleted. Finally the instance of `PatternIndex` is removed from the `graphIndexes` map.

Querying using an index

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. It means that all existing graph pattern indexes, accessible via their root nodes, are loaded just once, right after a Neo4j instance is started up. Thus an instance of `PatternIndexModel` lives and dies with Neo4j. It speeds up querying using an index, since a database does not have to be searched first in order to find appropriate root node. Instead, it is directly retrieved from memory.

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. All indexed graph pattern units are then accessible within *specificUnits* property of nodes that represent index units. Thus they could be directly retrieved. Although the problem arises when one wants to additionally filter the set of graph pattern units to be retrieved. There are two options how to filter them[16].

- **Projection** as a process of filtering is used when one does not want to retrieve complete information about graph pattern units consisting of nodes and relationships, but some particular information instead. Let's say a database that consists of people and their friendships is used and an index for a triangle is created. Figure 2.1 shows one of graph pattern units that could be mapped by such index. Now when querying using this index, one might be interested, for example, only in names of people that form such friendship triangles. In other words, only values of name property of nodes that form triangles should be retrieved instead of complete list of nodes and relationships. Particular information to be retrieved from indexed graph pattern units can be expressed within a *RETURN* clause of Cypher query that is given by a user.

- **Selection** as a process of filtering is used when one wants to retrieve only a subset of the complete set of indexed graph pattern units within an index. As it is mentioned in the analytical part of this thesis, an index is not necessarily used just for a graph pattern variation that it was created for. It can also be used when querying its descendant variations. Let's say the same database as above is used and the same graph pattern is created as well. Then one may want to retrieve, for example, a subset of indexed triangles from a database, where at least one of people has name Chandler. Such filtering can be done by querying using an index, such that queried graph pattern variation in a *MATCH* clause of Cypher query is a descendant of the one the index was created for.

Filtering can be done by one of mentioned options or by their combination as well. For this purpose, a user provides not only an index to be used, but also a Cypher query to declare what should be retrieved from indexed

[16]These options are defined for the purpose of this thesis.

graph pattern units when querying using an index. From a user point of view, the only difference compared to simple Cypher query is that an index that should be used is provided together with a query. Note that a graph pattern variation, expressed in a *MATCH* clause of such query, must be either the same or descendant of the variation for which the index, used for the query, was created. Otherwise the index cannot be used for such query.

In fact, there is a significant difference between the way a simple Cypher query and a query using an index are executed. A query without using an index has to be executed on top of a whole database. A query that uses an index can be executed just on top of graph pattern units that are mapped by such index. All matches for given query will be found within a database even if only these units are searched. This is guaranteed thanks to the relationship between a graph pattern variation expressed in a *MATCH* clause of given query and the variation for which the index was created.

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 graph database. Unfortunately querying on top of subgraphs is not supported in Neo4j at the moment of writing the thesis. Thus several approaches in order to execute queries on top of index units are introduced in this thesis. That is, as mentioned above, necessary to get final result for given query.

- **Approach no. 1**
  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 3.6 shows such composed query for a single index unit, that 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.

There are some strategies how to choose a representative node for an index unit. Since only one node is chosen for this purpose, there can be multiple index units that are represented by the same node. For example, figure 2.3 shows two index units that share one of nodes that form them. Thus they can be represented by different nodes but also by the same node. In case they are represented by the same node, only one composed query, as described above, is executed instead of two. Otherwise two same queries would be executed. In other words, the number of representative nodes does not have to correspond to the number of index units of appropriate index since a single representative node can be shared between multiple index units. Also the smaller set of nodes that represent all index units, the smaller number of composed queries to be executed. These are some possible strategies of choosing a representative node for an index unit.

- A representative node is chosen randomly for each index unit. This is the easiest strategy but a random size of total set of representative nodes is produced.
- A representative node for each index unit is the one with the smallest internal Neo4j ID. The smallest set of representative nodes is not guaranteed, but it is in average smaller than when chosen randomly.
- A representative node for each index unit is the one with the biggest number of incoming relationships of *PATTERN_INDEX_RELATION* type. This strategy does not necessarily produces the smallest set of representative nodes, but in average produces smaller set than strategies above. The strategy is used when implementing the method of indexing graph patterns.
- A representative node for each index unit is the one with biggest number of incoming relationships of *PATTERN_INDEX_RELATION* type that are linked strictly to index units that belong to appropriate index. Although this strategy produces the smallest set of representative nodes, it is very time consuming.

- **Approach no. 2**
  This approach is similar to the first one. A composed query must be, again, executed for each of index units. Although the structure of a single composed query varies from the first approach. Instead of a single node that represents an index unit, all nodes that form such unit are

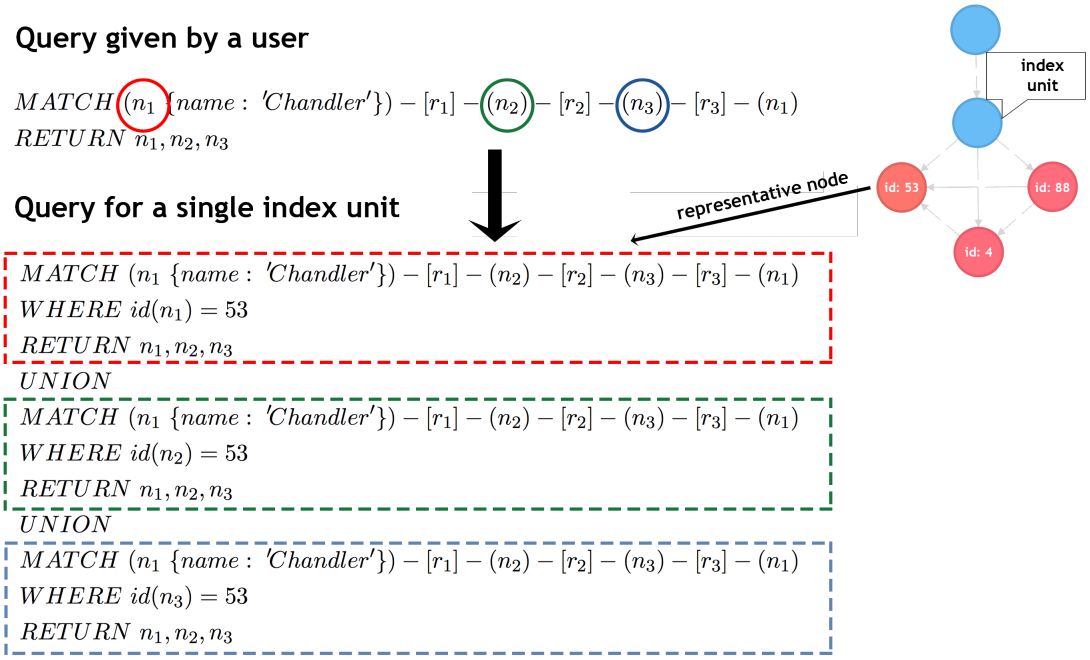


Figure 3.6: Approach no. 1 - a query for a single index unit

used for this purpose. Then all permutations of different mappings from nodes that form an index unit to node positions within a graph pattern expressed in a *MATCH* clause of the query must be applied when querying. Thus a composed query consists of $s!$ subqueries, where s is the number of nodes within a queried graph pattern. Figure 3.7 shows how such composed query with all permutations for a single index unit is created. Results of all subqueries are merged to present a result for such index unit. This approach is not applicable since the number of subqueries within a query for a single index unit grows exponentially with the number of nodes within a queried graph pattern.

- **Approach no. 3**
  When using this approach an extra Neo4j instance is needed. It is started up first with an empty database. Following process is done for each index unit of appropriate index. Nodes with relationships that form graph pattern units mapped by an index unit are copied together with their properties to the extra database. The intention is to separate data to be queried form the rest. Then a Cypher query given by a user is executed on top of such database. After this is done, all data within the extra database are deleted and the whole process starts again with next index unit. The problem is that copied nodes and relationships have different internal Neo4j IDs in both databases. Also it requires

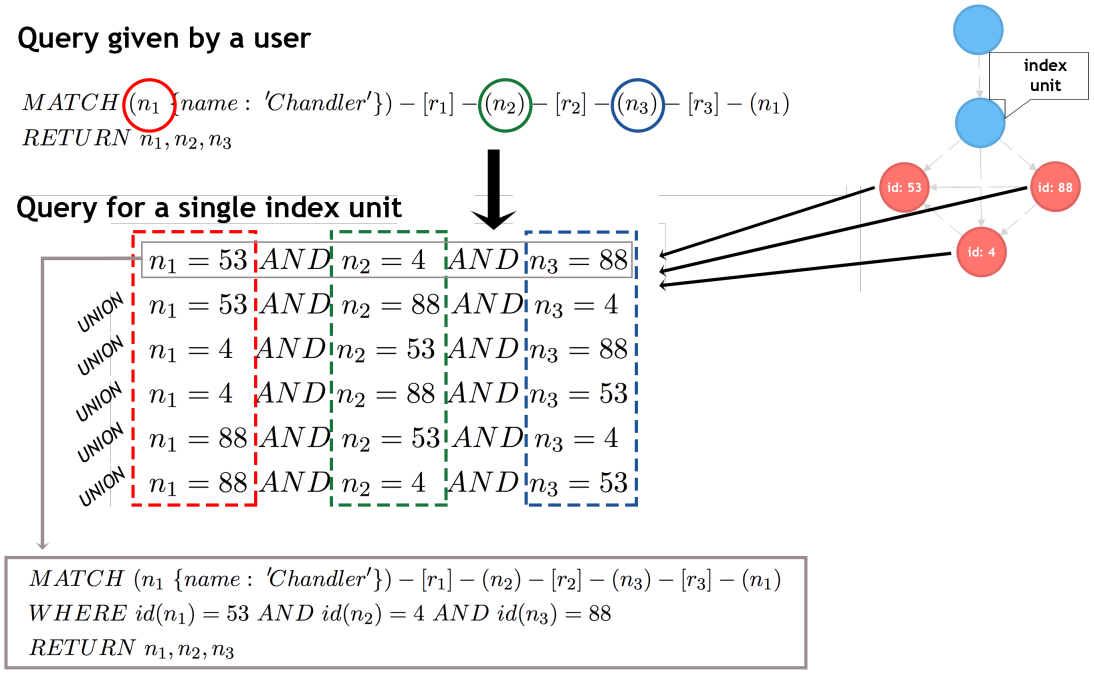


Figure 3.7: Approach no. 2 - a query for a single index unit

additional space and additional costs to run an extra Neo4j instance. Moreover the process of copying nodes and relationships, together with their labels, types and properties between two databases is expensive. Especially when it has to be done as many times, as the number of index units within appropriate index. These are many reasons not to use this approach for the purpose.

- **Approach no. 4**
  This approach is similar to the third approach. It also uses an extra Neo4j instance. The different is that index units are not copied to the extra database one by one. Instead there are all copied there at once. Then only a single Cypher query, given by a user, is executed on top of this data. This approach is slightly better than third approach, since there is only a single Cypher query executed against the extra database. Although, again, the extra Neo4j instance must be run and the same amount of data must be copied between databases. Thus this approach is not recommended to use for the purpose as well.

Updating indexes [17]

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 `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, for example, 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 commited. 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 commited. 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.

- `handleRelationshipDelete` method to update indexes after some relationships are deleted within the transaction.
- `handleNodeDelete` method to optimize the process of updating indexes after delete operations. As already mentioned, deleting a node by itself does not affect indexes. Although additional operations can be performed with deleted nodes to optimize the process of updating indexes. It is mainly used during transactions with more than one DML operations.
- `handleCreate` method to update indexes after some nodes or relationships (or both) are created within the transaction.
- `handleUpdate` method to update indexes after some nodes or relationships (or both) are updated within the transaction.

[17] Detailed description of these methods is out of scope of this thesis. For this purpose documentation is included within the code.

3.3 Instalation of the method of indexing graph patterns

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 it 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.

Chapter 4

Mearusements

This part of the thesis is focused on proving theoretical characteristics of graph pattern indexes. Matching general graph patterns is a very time consuming process. Thus graph pattern indexes are introduced to speed it up. Obviously an index must be created before it can be used when querying. Such operation involves building an index data structure with redundant information that provides quick jump points on where to find the full references of matched graph pattern units. It brings additional cost and requires additional space since an index structure must be built and stored for each graph pattern. It also brings additional cost since these indexes must be updated each time a DML operation is applied to a database.

The method for indexing graph patterns, including operations to create an index, query using an index and update an index, introduced in this thesis is implemented for Neo4j graph database engine. In Neo4j the process of matching graph patterns can be done by using provided APIs including Core API or Traversal API, but mostly Cypher is used for this purpose. Cypher is also an important part of the method for matching pattern indexes. From the user point of view, a simple query and a query using an index does not vary so much. When querying using an index, a user expresses a query in Cypher and additionally provides an index to be used for the query. When creating a new index, a user specifies a graph pattern that the index should be created for by using Cypher's *MATCH* clause.

All results in this chapter are achieved by measuring within a test environment provided by GraphAware framework. 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`. Graphs in this chapter are used to visualize differences between measured elements. Unfortunately tables with actual numbers are too large for the thesis and thus they are provided within enclosed CD. 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 includ-

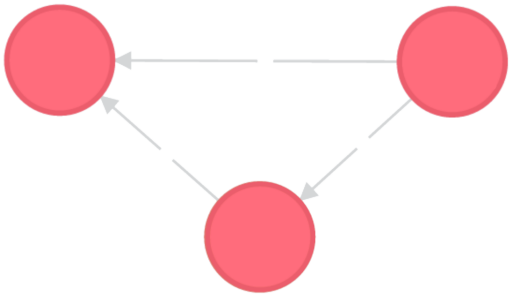


Figure 4.1: Triangle graph pattern

ing *No-cache* (Neo4j instance with no caching), *Low-level cache* and *High-level cache*. These are described in chapter 1. Because there are extreme performance differences between these cache types, the exponential scale is used for measured metrics within graphs. There are two metrics used to measure performance:

- *time*: Time in microseconds (μs).
- *DBHits*: A database hit is an abstract unit representing a single operation within Neo4j storage engine that does some work such as retrieving or updating data. Then DBHits is a total number of such database hits that a measured process needs to perform.

Three graph databases are used for measuring. For each of them appropriate graph pattern to be indexed is devised [18].

Social graph with Triangle index
Social graph is a database that consists of 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 *FRIEND_OF* type. Such database of changeable size is generated by *Erdős–Rényi model* for generating random graphs. The model is described in chapter 1.

Triangle index is used for this database. Such index is built for a triangle graph pattern shown in figure 4.1 and expressed in Cypher as:

$$((n_1) - [r_1] - (n_2) - [r_2] - (n_3) - [r_3] - (n_1)$$

[18] Names of indexes are made up for the purpose of this thesis. They are inspired by shapes of appropriate graph patterns.

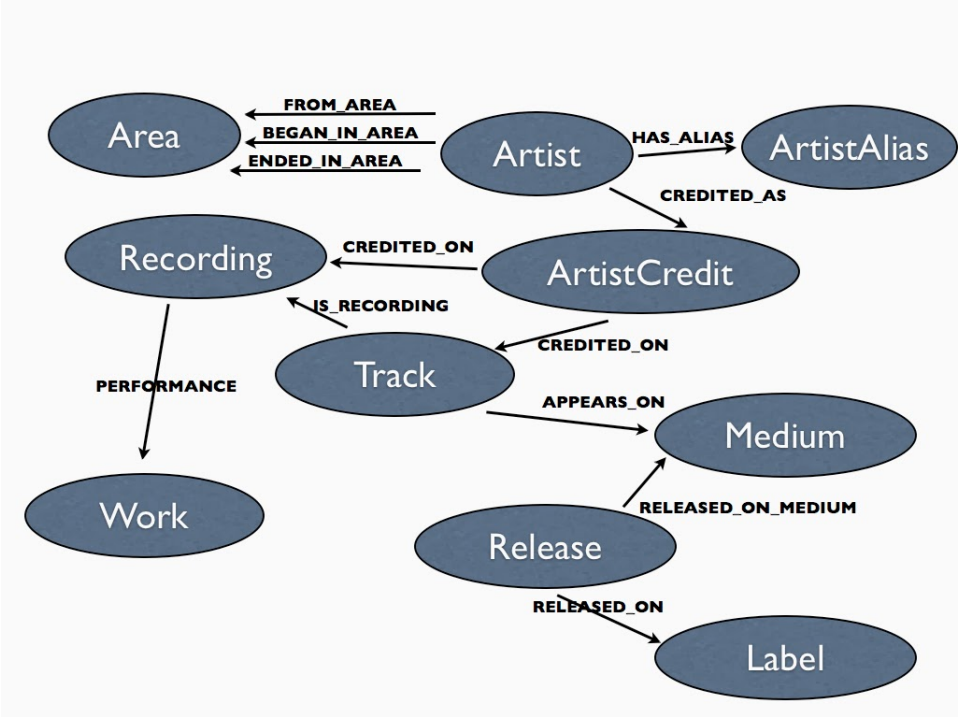


Figure 4.2: The structure of Music database

Music database with Funnel index
Music database stores data about artists, detailed information about 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 example data sets that Neo4j provides on their website [26]. Figure 4.2, taken from Tremberth [27], shows the structure of data within this database.

Funnel index is used for this database. Such index is built for a specific graph pattern shown in figure 4.3 and expressed in Cypher as:

$$(n_1) - [r_1] - (n_2) - [r_2] - (n_3) - [r_3] - (n_1) - [r_4] - (n_4)$$

Transaction database with Rhombus index
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, for example a date of transaction execution or the amount of transferred money within a transaction. Such database of changeable size is generated by a Cypher script that is created

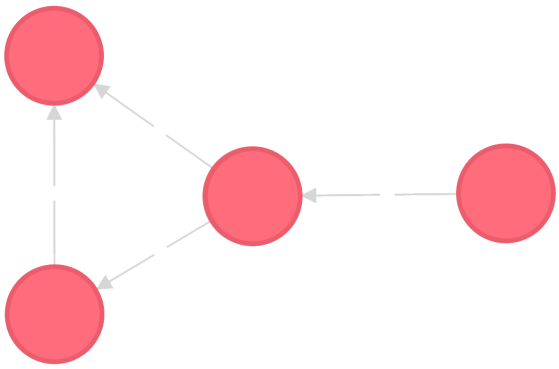


Figure 4.3: A graph pattern used for Music database

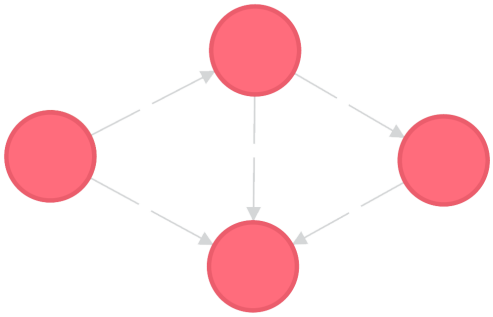


Figure 4.4: A graph pattern used for Transaction database

especially for this purpose. Such simple script creates bank accounts at first and then generates a transaction relationship for each pair of these accounts with given probability.

Rhombus index is used for this database. Such index is built for a specific graph pattern shown in figure 4.4 and expressed in Cypher as:

$$(n_1) - [r_1] - (n_2) - [r_2] - (n_3) - [r_3] - (n_1) - [r_4] - (n_4) - [r_5] - (n_2)$$

This chapter is divided into three separated sections. First section compares specific approaches for querying using an index as they are described in chapter 3. Second section presents a comparison of the process of matching graph patterns between a simple Cypher query and a query using appropriate index. Third and final section is focused on comparing two different implementations of graph pattern indexes. First one, introduced in this thesis, uses a tree structure and stores its index data within the same database where the actual data is. Second one, introduced in a concurrent master's thesis, uses a hash map structure and stores its index data within an external key-value store.

4.1 Approaches for querying using an index

Four approaches for querying using an index are introduced and described in chapter 3. To choose appropriate one for implementing the method of indexing graph patterns all of these were simulated to estimate their performance. Social graph of 10 000 nodes and 50 000 relationships with Triangle index is used for the measurement.

Figure 4.5 shows a graph with results of the measurement. Approach no. 3 and approach no. 4 are very time consuming compared to two other approaches. It is caused by additional costs that the extra instance of Neo4j brings. These additional costs mostly relate to the fact that data must be copied between two databases. Also no speed improvement is noticeable when increasing caching levels. The extra database is always empty before appropriate data is copied there from the main database. It means that every single Cypher query is executed on top of a database with fresh data. Thus at the time of executing a Cypher query there is no data cached yet and then no speed improvement is to be expected when enabling caching.

Approach no. 2 is the fastest approach due to results for this specific use case. However the number of subqueries within a query for each index unit grows exponentially with the number of nodes within queried graph pattern. It is the bottleneck of the approach. If indexed graph pattern consists of more then three nodes, querying using this approach will be significantly slower compared to other approaches.

Approach no. 1 proves to be enough efficient, especially if a database uses High-level caching (mostly used in practice). During measuring the approach uses random strategy to find representative nodes for appropriate index units. Some additional improvement can be expected if more efficient strategy is used. All such strategies are described in chapter 3.

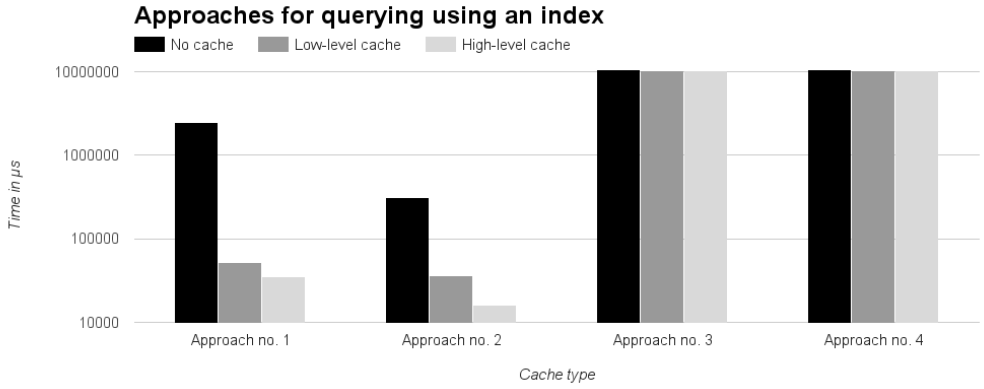


Figure 4.5: Measurement - approaches for querying using an index

4.2 Simple query versus query using index

The measurement is very important since it compares the process of matching graph patterns between a simple Cypher query and a query using appropriate index. Here it is meant to prove qualities of the new method of indexing graph patterns. As mentioned many times before, the method aims to speed up matching of general graph patterns at additional cost of creating and updating indexes.

The main intention of this measurement is to observe how effective both approaches are with growing size of a database. Social graphs of different sizes with Triangle index are used for the measurement. The size of a database scales from 50 nodes and 100 relationships to 100 000 nodes and 500 000 relationships. Matching triangle graph pattern using a simple query is nearly impossible for larger databases of this type.

There are two main graphs provided for the measurement. Graph in figure 4.6 shows measured *DBhits* metric for both approaches. On the other hand figure 4.7 shows graphs for the second measured metric, *time*. The first graph in this figure shows comparison of both approaches for all cache types together, whereas other graphs in the figure show the comparison for each cache type individually. The progression in complexity with growing size of a database is very similar for both metrics (i.e. in both main graphs). This is caused by the fact that *time* and *DBhits* are mutually dependent variables. Obviously the more operations a database must perform the more time is needed for the whole process and vice versa.

The behavior of approaches on both main graphs proves theoretical assumptions about indexes. They are most effective when used for large databases where there is a small number of appropriate graph pattern units. If a database is of a smaller size a simple query can be effective enough. Also if there are too many graph pattern units within a database and perhaps they are evenly distributed, it may be better just to scan the whole database instead of using appropriate index. There is a certain size of a database for which the usage of an index becomes to be beneficial. This specific size varies for each database and each index and it is influenced by many factors. Type of data stored within a database, a graph pattern the index is created for and also a number and distribution of appropriate graph pattern units rank among these factors. For this particular case, the usage of triangle index is beneficial for larger databases than the one with 1 000 nodes and 5 000 relationships.

There is a big difference between a simple query and a query using an index. The complexity of a simple query grows exponentially with the size of a database whereas the complexity of a query using an index grows linearly with the number of appropriate graph pattern units within a database. The process of querying using an index is almost independent of the size of a database. This can be observed on both graphs. 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. In other words using indexes may cause a huge difference in many use cases. Also some queries that could not be otherwise processed due to its high complexity can be possibly processed if appropriate indexes are used.

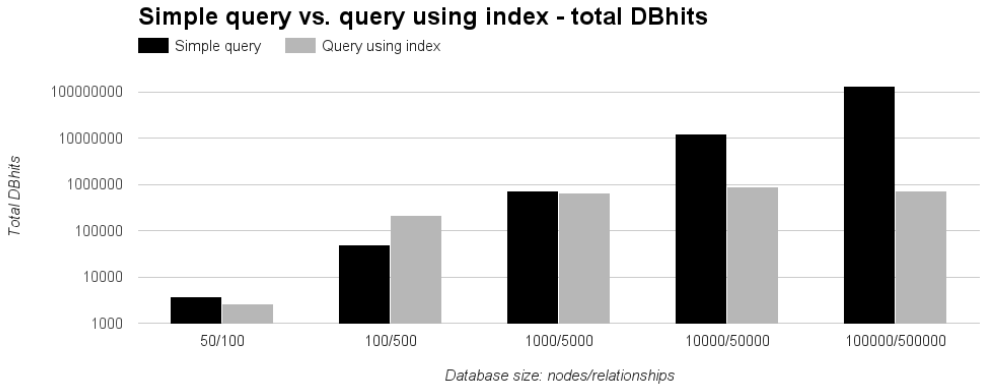


Figure 4.6: Simple query vs. query using index - total DBhits

4.3 Index implementations comparison

The last measurement is the most complex one of all. It consists of more individual measurements and covers comparison of simple Cypher and two different methods of indexing graph patterns. First index implementation, introduced in this thesis, uses a tree structure to store index data. Index created by this method will be further referred to as *tree index*. Second index implementation, presented in a concurrent master's thesis, uses a hash map to store index data. Index created by this method will be further referred to as *hash index*. The main difference between these implementations is the fact that tree index is stored within the same database where the actual data is whereas hash index is stored within a separate key-value store. Note that for each graph pattern a new index must be created.

Both methods of indexing graph patterns use the same approach when querying. It is the approach no. 1, as it is described in chapter 3. The approach chooses a representative node for each index unit and performs a query that involves such node for each index unit separately. Finally, results of these queries are merged to present the final result for the query given by a user. A strategy of choosing representative nodes for appropriate index units varies for both index implementations. Hash index chooses the node with the smallest internal Neo4j ID, whereas tree index chooses the node with the

Figure 4.7: Simple query vs. query using index - time

biggest number of incoming relationships from index units [19]. The strategy that is used by tree index should be more effective since it provides, in average, a smaller set of representative nodes that cover all index units of appropriate index. Thus less queries must be performed in order to get the final result for given query. Such strategy cannot be used by hash index since key-value structure does not provide necessary information about incoming relationships from index units [20].

All three databases, as they are introduced together with appropriate indexes above, are used when measuring. There are 6 graphs for each of these databases that visualize database operations that are crucial for index implementations [21]. All of them use *time* as measured metric since it is what really matters in practice.

1. **Querying** refers to the process of matching a graph pattern. A simple query is compared to queries that use indexes provided by both methods of indexing graph patterns.

2. **Creating an index** is the process of building a new index before it can be actually used. Comparison between both methods of indexing graph patterns is provided for the process.

3. **Creating a relationship** is a DML operation that may affect some graph pattern units. Thus all indexes must be updated after such operation is applied to a database. Delete operation and subsequent process of updating indexes is done within the same transaction. Thus time that is needed to successfully commit such transaction is measured for a database instance with no indexes and also for database instances with tree index respectively hash index.

4. **Deleting a relationship** is a DML operation that may, again, affect some graph pattern units. Thus time needed to successfully commit a transaction with such operation is measured in the same way as described for the previous operation.

5. **Deleting a node** is a DML operation that, by itself, cannot affect any graph pattern units. However in this case it is applied to a database as a part of more complex transaction. Such transaction consists of operations to delete relationships of a node together with an operation to delete the node itself. Such complex transaction can affect some graph pattern units. It is used to find out how effectively index implementations handle updating indexes within more complex transactions. It is measured in the same way as described for previous operations.

[19] Strategies for choosing representative nodes are described in chapter 3.

[20] Detailed structure of hash index is described in concurrent master's thesis.

[21] Elementary DML operations that may affect a graph pattern index are described in chapter 2.

6. **Deleting a label of a node** is a DML operation that belongs to a group of operations that in some way update nodes or relationships. Any operation of such group may affect some graph pattern units. The operation of deleting a label of a node is chosen to represent the group since all operations within such group are handled in the same way when updating indexes. The method of indexing graph patterns presented in concurrent master's thesis does not support updating of its indexes after any of operations within such group is applied to a database at the time of writing this thesis. Thus time that is needed to commit a transaction with the operation of deleting a label of a node is measured only for a database instance with no indexes and for an instance with tree index.

Measurement no. 1: Social graph with Triangle index

- Social graph of 10 000 nodes and 50 000 relationships is used.
- An index is created for a triangle graph pattern.
- Triangle index maps 183 graph pattern units.
- Figure 4.8 shows the process of querying in a graph. Figure 4.9 shows the process of creating a new index in a graph. Figures 4.10, 4.11, 4.12 and 4.13 show graphs for individual DML operations as they are mentioned above.

There is a noticeable difference between a simple Cypher query and both index implementations. Query using tree index is, in average across different cache types, 15 times faster than a simple Cypher query. Query using hash index is slightly faster than tree index, but both are very competitive. For 183 graph pattern units there are 179 representative nodes to cover all appropriate index units chosen during querying using hash map and 166 representative nodes chosen during querying using tree map. Thus the strategy of choosing representative nodes based on the number of incoming relationships from index units proves to be better than the one that chooses them based on their internal Neo4j IDs. However key-value stores are in general faster than graph databases in performing read and write operations. Thus it is slightly faster to use hash index when querying in this case even though there must be more queries performed in order to get the final result.

Building of hash index is a little faster then building of tree index especially in a database that uses no caching. To build a tree structure within a graph database is more complex process than storing index data within a key-value store. The size of hash index is 86 kB whereas the size of tree index is 0,1 MB. Data stored within a graph database typically require more memory space than key-value pairs within a key-value store.

DML operations, including deleting and creating a relationship and mainly an operation of deleting node with its relationships, seem to be processed faster by the method of indexing graph patterns introduced in this thesis compared to the method from concurrent master's thesis. Tree index structure within the database where the actual data is provides direct links to indexed data. Also tree structure in general provides richer information then a hash map. These facts are useful when optimizing the whole process of updating indexes especially when multiple DML operations are applied at once [22].

The process of updating indexes after one or more DML operations are applied to a database brings additional costs. Fortunately there is no significant slowdown when applying DML operations compared to a database with no indexes.

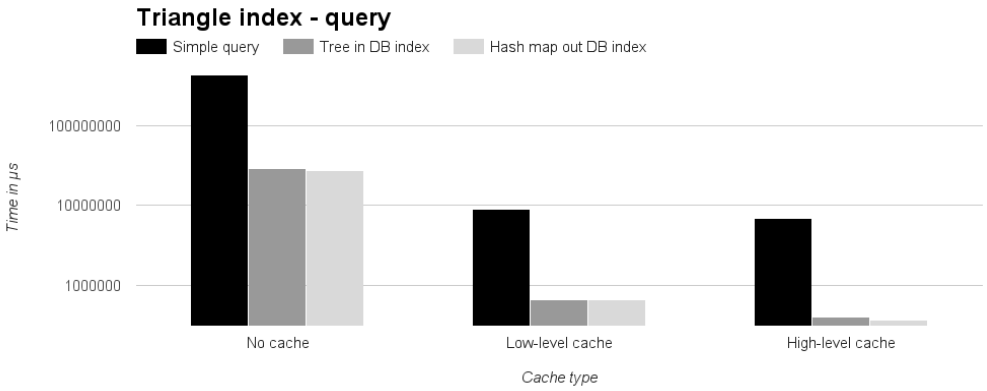


Figure 4.8: Triangle index - query

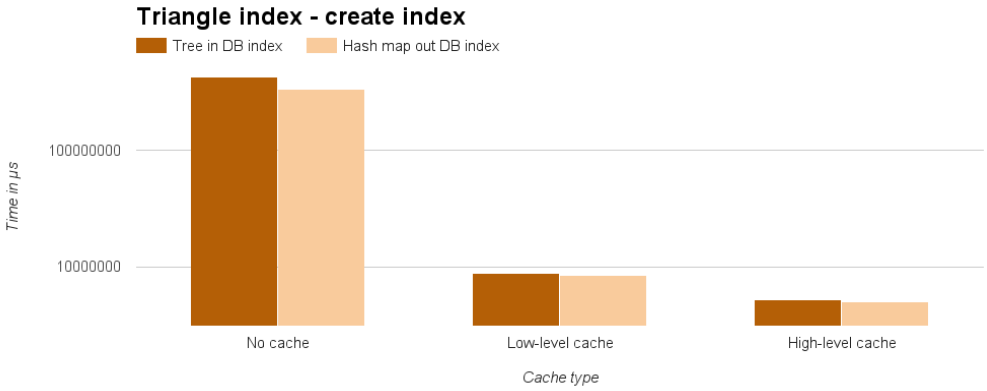


Figure 4.9: Triangle index - create index

[22]The whole process of updating indexes is described in documentation of index implementation.

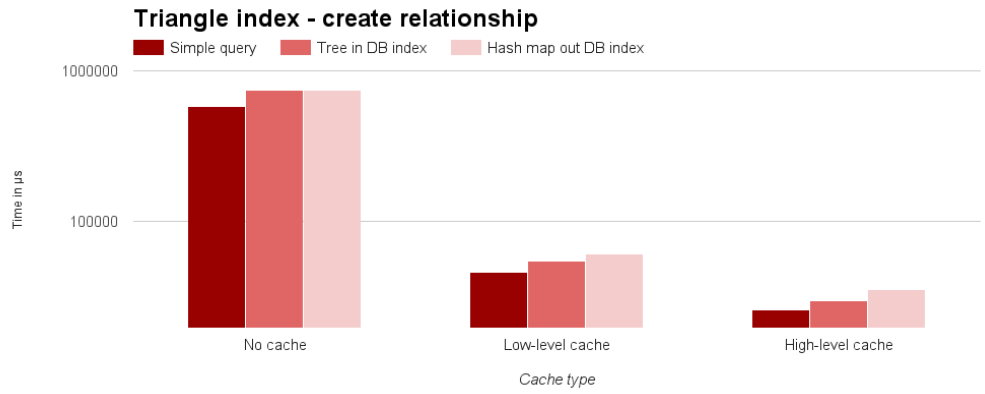


Figure 4.10: Triangle index - create relationship

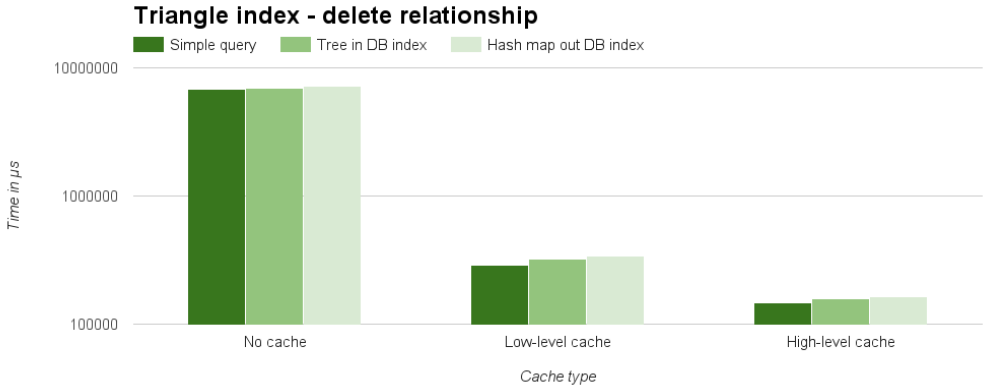


Figure 4.11: Triangle index - delete relationship

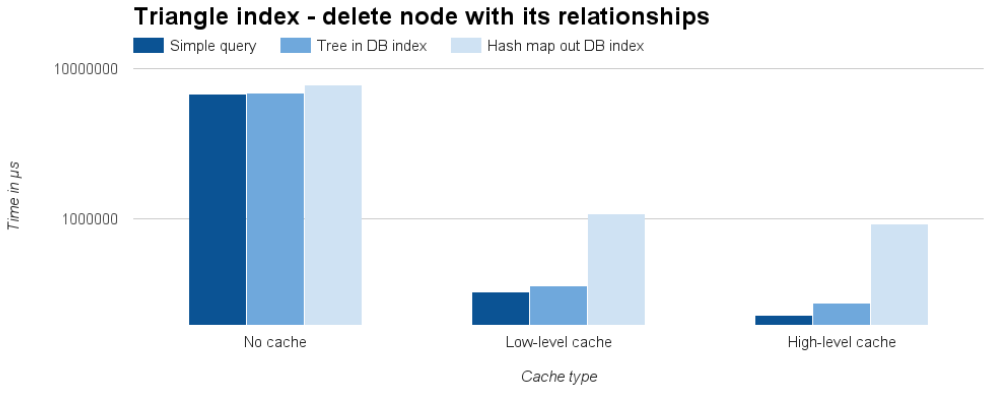


Figure 4.12: Triangle index - delete node with its relationships

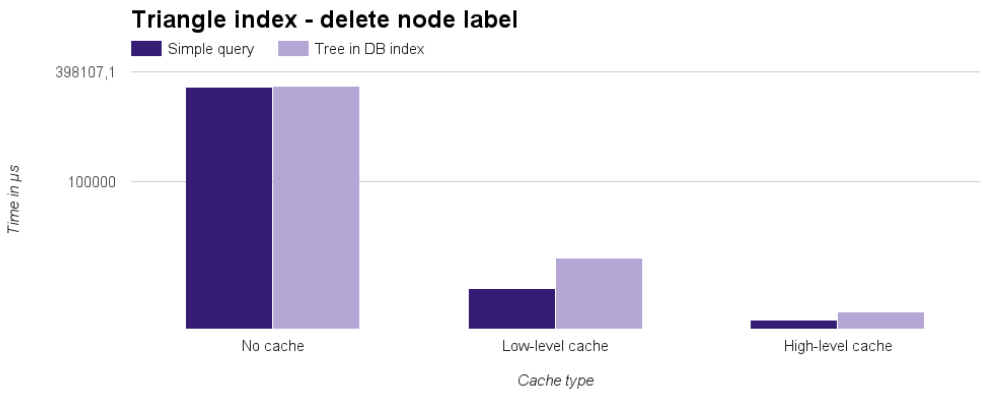


Figure 4.13: Triangle index - delete node label

Measurement no. 2: Music database with Funnel index

- Music database of 12 000 nodes and 50 000 relationships is used.
- An index is created for a funnel graph pattern.
- Funnel index maps 86 graph pattern units.
- Figure 4.14 shows the process of querying in a graph. Figure 4.15 shows the process of creating a new index in a graph. Figures 4.16, 4.17, 4.18 and 4.19 show graphs for individual DML operations as they are mentioned above.

There is a huge difference between a simple Cypher query and both index implementations. Query using tree index is, in average across different cache types, 190 times faster than a simple Cypher query. Query using hash index is slower than tree index this time. For 86 graph pattern units there are only 6 representative nodes to cover all appropriate index units chosen during querying using hash index and only 3 representative nodes chosen during querying using tree index. The are just few representative nodes to cover all index units because of the fact that appropriate graph pattern units within the database are not evenly distributed but rather organized around a few "central" nodes . The strategy of choosing representative nodes used by the method introduced in this thesis is very effective. It leads to the fact that querying using tree index is faster than querying using hash index. Using index in this case is extremely helpful.

Index implementations are very competitive when handling the process of creating indexes and also when handling updates after DML operations. The method introduced in this thesis seems to be a little faster in all of them in this case. Bigger difference in performance is noticeable for the operation of deleting a node together with its relationships. Index implementation from

this thesis again proves to be more effective when handling such operation. Additional costs associated with the process of updating indexes do not cause significant slowdowns in this case.

The size of hash index is 86 kB whereas the size of tree index is 0,1 MB. This is again caused by the fact that graph databases require more memory space to store data than key-value stores.

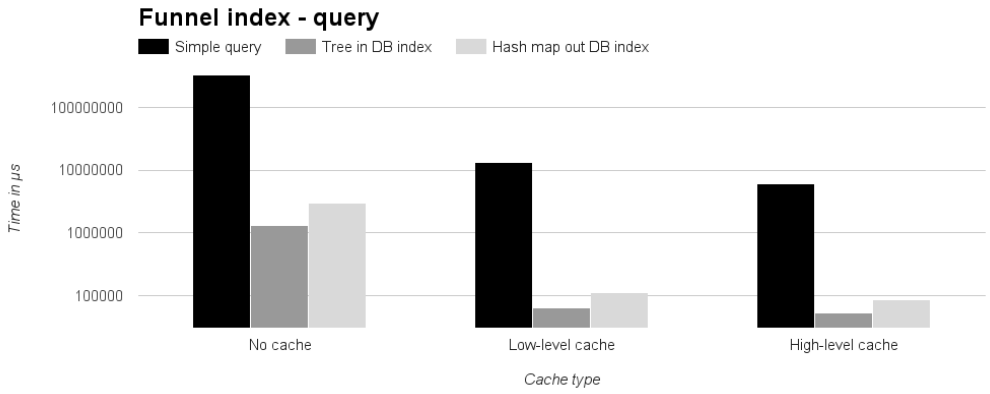


Figure 4.14: Funnel index - query

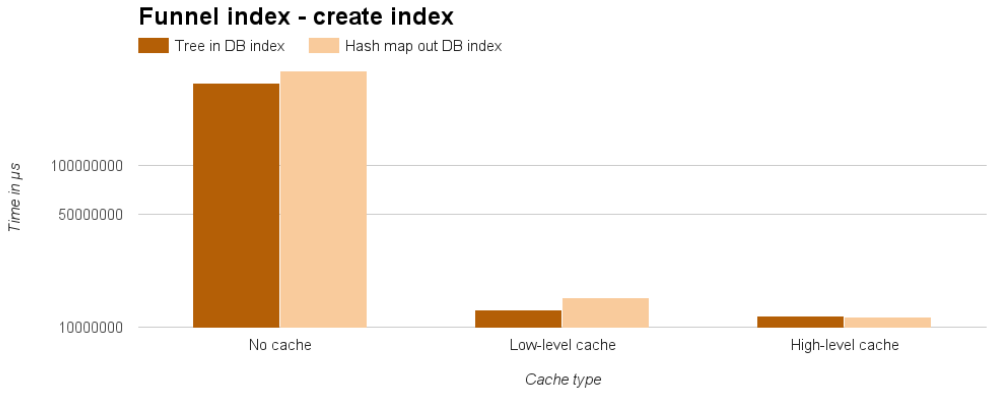


Figure 4.15: Funnel index - create index

Measurement no. 3: Transaction graph with Rhombus index

- Transaction database of 10 000 nodes and 100 000 relationships is used.
- An index is created for a rhombus graph pattern.
- The index maps 70 graph pattern units.

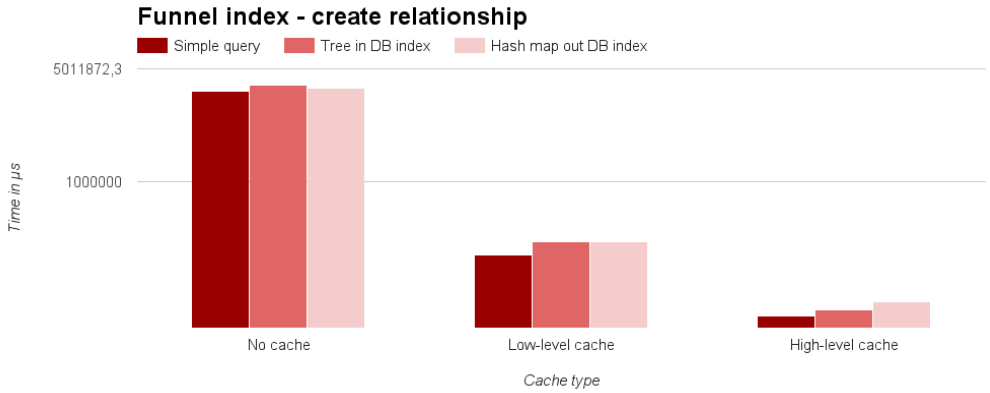


Figure 4.16: Funnel index - create relationship

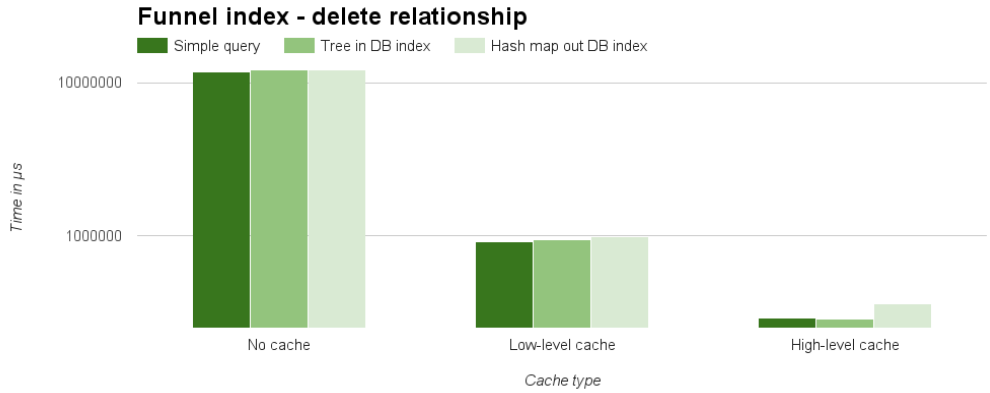


Figure 4.17: Funnel index - delete relationship

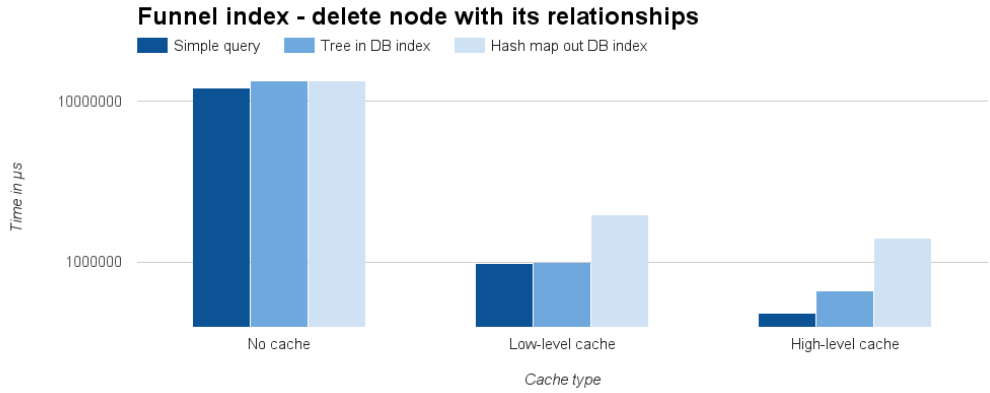


Figure 4.18: Funnel index - delete node with its relationships

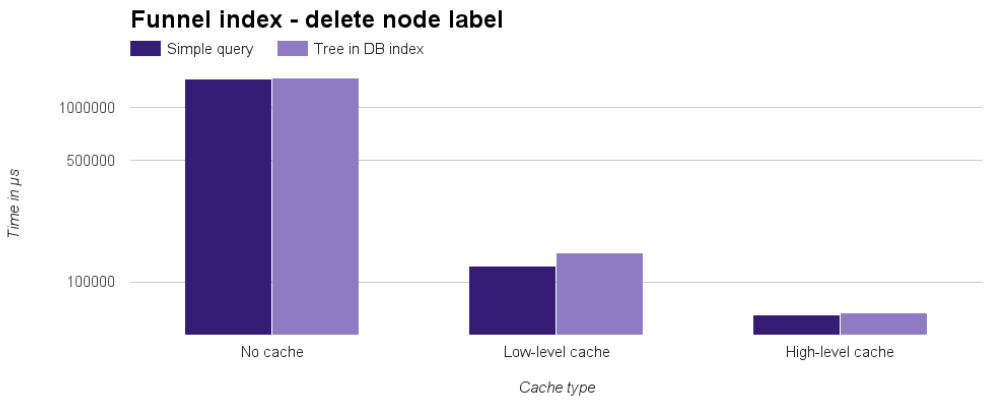


Figure 4.19: Funnel index - delete node label

- Figure 4.20 shows the process of querying in a graph. Figure 4.21 shows the process of creating a new index in a graph. Figures 4.22, 4.23, 4.24 and 4.25 show graphs of individual DML operations as they are mentioned above.

There is again a big difference between a simple Cypher query and both index implementations, mostly for a database with High-level and Low-level caches in this case. Also there is a noticeable difference between queries using both indexes. Query using tree index is in average more than 30 times faster than a simple Cypher query while query using hash index is "only" 25 times faster. For 70 graph pattern units there are 66 representative nodes to cover all appropriate index units chosen during querying using hash index and 64 representative nodes chosen during querying using tree index. The strategy of choosing representative nodes used by the method introduced in this thesis proves to be better again.

Again, both implementations are very competitive when handling DML operations. As it can be already observed from measurements for other two databases, the method introduced in this thesis updates indexes after a node gets deleted together with its relationships faster then the method from concurrent master's thesis.

The size of hash index is 70 kB whereas the size of tree index is 3 MB. Size difference is more significant here than within two other databases. It is probably caused by the fact that indexed graph pattern consists of more nodes than graph patterns within previous databases. Thus there are many relationships between nodes that form specific graph pattern units and appropriate index units within a tree index structure. Thus more data must be stored. However additional space of 3 MB for a database of 10 000 nodes and 100 000 relationships is still not significant if the index is used in practice.

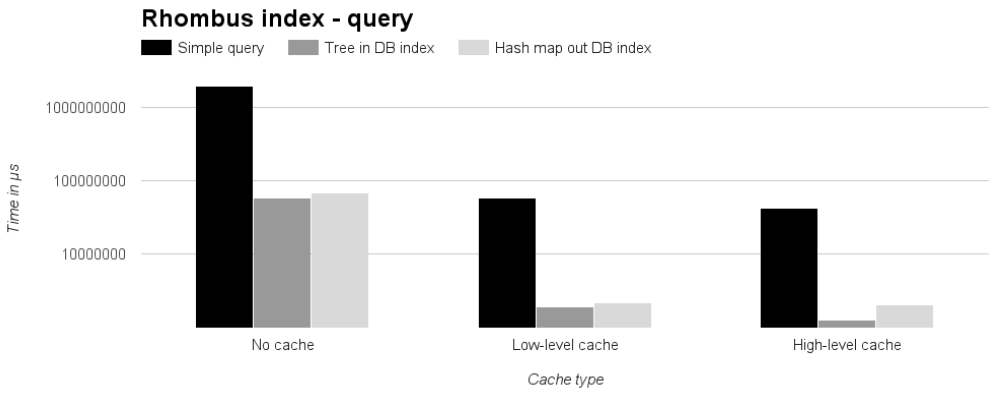


Figure 4.20: Rhombus index - query

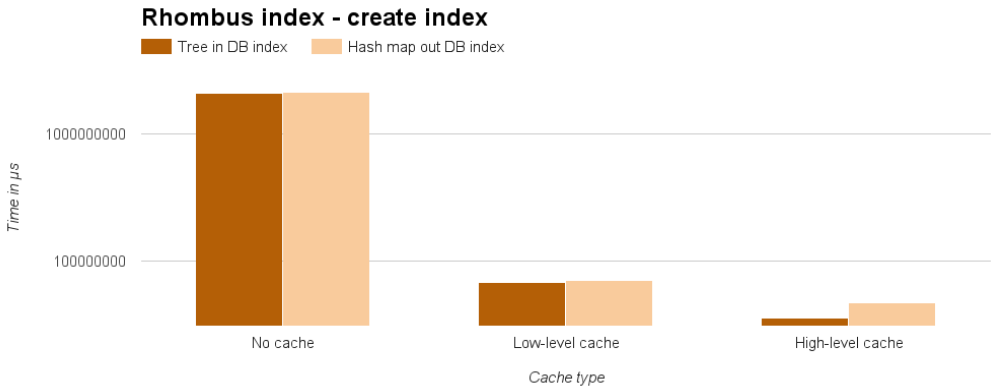


Figure 4.21: Rhombus index - create index

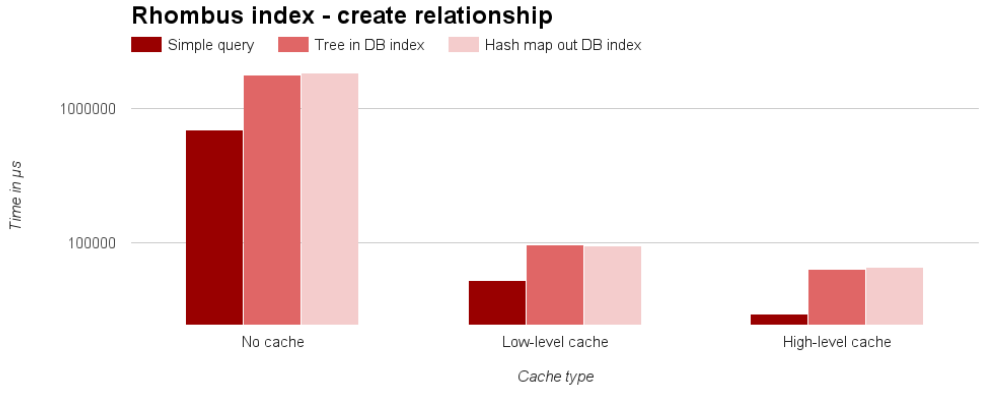


Figure 4.22: Rhombus index - create relationship

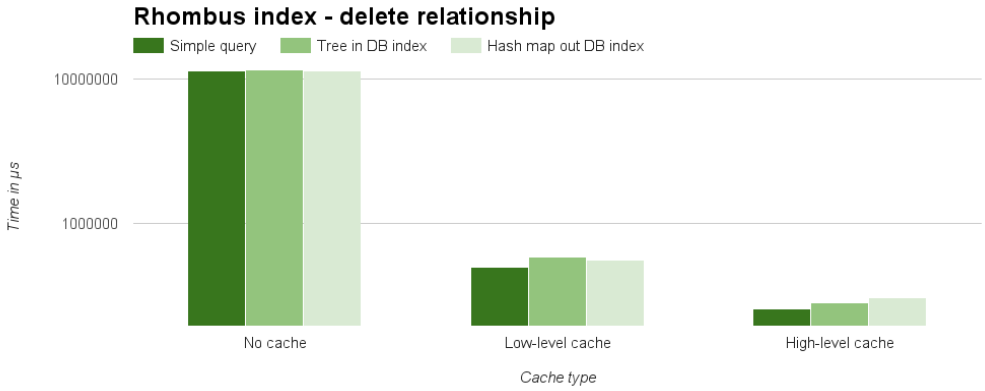


Figure 4.23: Rhombus index - delete relationship

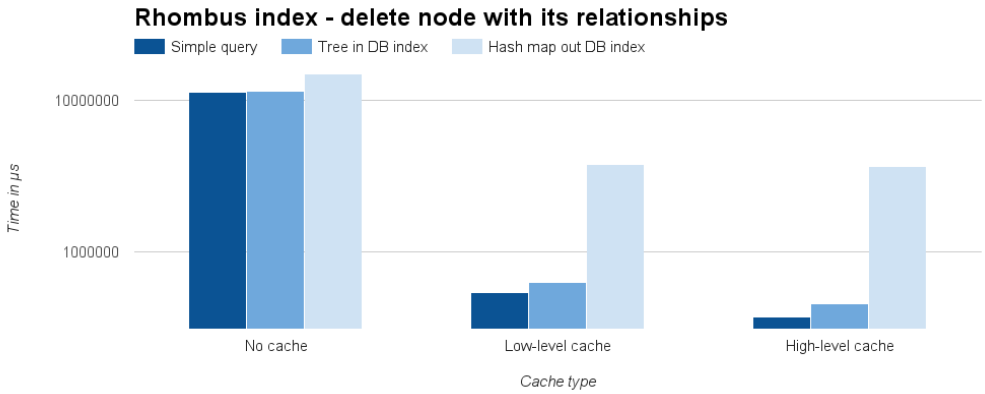


Figure 4.24: Rhombus index - delete node with its relationships

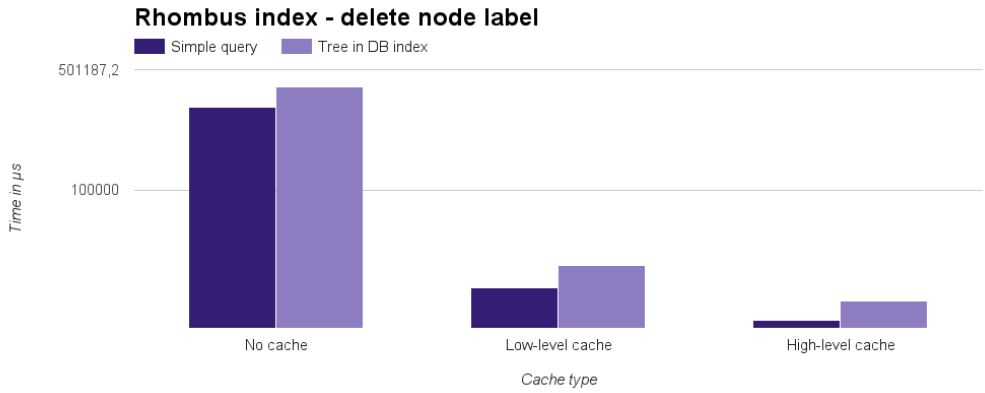


Figure 4.25: Rhombus index - delete node label

Overall evaluation

The measurement proves that both methods for indexing graph patterns can speed up the process of matching graph patterns. The effectiveness of these methods depends on many factors including a choice of graph pattern to be indexed, the size of a database, the number of graph pattern units of appropriate index and also a distribution of these graph pattern units within a database.

Creating an index is very complex process especially for complicated graph patterns and large databases. However it needs to be done only once before the index can be used. Additional costs caused by the process of updating indexes after DML operations are, on the other hand, considered not to be very significant. Both index implementations are very competitive when handling such operations.

The method for indexing graph patterns introduced in this thesis proves to have slightly better results than the method from concurrent master's thesis. The main advantage of the method is that a tree structure with direct links to the actual data provides richer information than a hash map. Such information that cannot be obtained if hash map is used is useful when choosing representative nodes of index units during the process of querying or when optimizing the process of updating indexes. The main disadvantage of the method is the fact that tree index stores its data within the same database where the actual data is. Querying using such index can be too slow if there are too many matched graph pattern units within a database or if an indexed graph pattern is too large. In both cases tree index requires to store much index data. Too much additional data within a database may seriously affect statistics that are used when evaluating execution plans of Cypher queries which may further cause slowdowns if executing them [23].

[23] Such information was achieved during personal consultation with Cypher team in London.

Chapter 5

Future work

Even though the implementation of the method for indexing graph patterns introduced in this thesis works pretty well for many graph patterns there is still space for possible improvements. There are some advantages and disadvantages of using proposed tree structure of index. Some other possible structures that could be used to store index data would be worth exploring. An index structure plays an important role in the process of speeding-up querying graph patterns.

Unfortunately Neo4j does not support meta data at the time of writing the thesis. Index data, as introduced in this thesis, is stored within the same database where the actual data is. Thus index data is mixed with the actual data. Adding such additional data to a database may affect statistics that are used to evaluate execution plans for queries. Also it may slowdown these queries. Such problem can be solved by coming up with the way of separating index data and the actual data on the physical level.

The method of indexing graph patterns currently works with Cypher. Cypher is a high-level tool that brings some additional costs since it must be parsed before required operations are performed. Thus the method can be further optimized by replacing Cypher with low-level API such as Core API, or possibly Traversal API.

An important component of the method is the process of checking if given index can be used for given query. As it is mentioned in the thesis, there are specific rules defining which index can be used for which query. Such validation is not implemented within the method at the time of writing the thesis and would be certainly necessary if the method is used in practice.

Also the method can be tested for more different databases with more different indexes. There is for sure a space for possible optimizations of the method. By testing it with multiple different databases one can find out weaknesses of the method and react on them in order to improve the method.

Conclusion

Graph databases are, at the time of writing the thesis, one of very fast growing categories of database management systems. They are also very young and thus still provide space for possible improvements. The process of matching graph patterns is one of their main features. A graph pattern can be referred to as a subgraph of a database graph. Then the process of matching graph patterns can be referred to as the process of finding appearances of such subgraph within a database graph.

Neo4j, the world's leading graph database at the moment of writing the thesis, uses mostly Cypher for matching graph patterns. It is a graph query language that enables among other things to express a graph pattern whose matches are then found within a database. The complexity of finding matches for appropriate graph pattern depends on the size of a database, the shape of given graph pattern but mostly on how much information is provided about such graph pattern. If a user provides some detailed information about nodes or relationships that form queried graph pattern, it is not difficult to find its matches within a database. But if only structure of queried graph pattern is known, the whole process of matching graph patterns is very complex. It involves scanning of the whole database in order to find matches for such graph pattern. For this reason graph pattern indexes are introduced.

A database index is a data structure that improves speed of retrieving data by providing the database with quick jump points on where to find the full references, much like book indexes. Indexes already exist for nodes within graph databases. Although no indexes were yet introduced for graph patterns. In this thesis a new method for indexing graph patterns was analyzed, designed and implemented for Neo4j graph database engine 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 is organized in a tree structure and stored within the same database where the actual data is.

The new method for indexing graph patterns is tested by using multiple

different databases with multiple different graph patterns. It is also compared to competitive method that is introduced in a concurrent master's thesis. It is proved that using indexes which are created by the method introduced in this thesis is beneficial for the process of matching graph patterns. In some cases queries using such indexes are extremely faster than simple Cypher queries.

The thesis 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 graph database.

Bibliography

[1] Jan Paredaens, Paul De Bra, Marc Gyssens, Dirk van Gucht. *The Structure of the Relational Database Model*. Springer, 1989, ISBN 3540137149.

[2] Edgar F. Codd. A Relational Model of Data for Large Shared Data Banks. Technical report, IBM Research Laboratory, 1970.

[3] Shashank Tiwari. *Professional NoSQL*. Wrox, 2011, ISBN 047094224X.

[4] Ian Robinson, Jim Webber, Emil Eifrem. *Graph Databases*. O'Reilly Media, 2013, ISBN 1449356265.

[5] John Adrian Bondy, U.S.R. Murty. *Graph Theory With Applications*. Elsevier Science Ltd/North-Holland, 1976, ISBN 0444194517.

[6] Bin Xiong. *Graph Theory*. World Scientific Publishing Company, 2010, ISBN 9814271128.

[7] Reinhard Diestel. *Graph Theory (Graduate Texts in Mathematics)*. Springer, 2000, ISBN 0387989765.

[8] Michal Bachman. *GraphAware: Towards Online Analytical Processing in Graph Databases*. Master's thesis, Imperial College London, 2013.

[9] Jonathan L. Gross, Jay Yellen. *Graph Theory and Its Applications*. CRC Press, 1998, ISBN 0849339820.

[10] Nathan W Lemons, Bin Hu, William S Hlavacek. Hierarchical graphs for rule-based modeling of biochemical systems. *BMC Bioinformatics*, 2011.

[11] Wenfei Fan, Xin Wang, Yinghui Wu. Incremental Graph Pattern Matching. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, ACM, 2011, ISBN 978-1-4503-0661-4.

[12] Prof. Ing. Pavel Tvrdík, C. *Parallel algorithms and computing*. ČVUT, 2005.

[13] Dexter C. Kozen. *The Design and Analysis of Algorithms (Monographs in Computer Science)*. Springer, 2011, ISBN 146128757X.

[14] Wikipedia. Depth-first search. `https://en.wikipedia.org/wiki/Depth-first_search`, 2015.

[15] Wikipedia. Breadth-first search. `https://en.wikipedia.org/wiki/Breadth-first_search`, 2015.

[16] Mark Newman. *Networks: An Introduction*. Oxford University Press, 2010, ISBN 0199206651.

[17] Réka Albert, Albert-László Barabási. Statistical mechanics of complex networks. Technical report, Department of Physics, University of Notre Dame, 2002.

[18] MongoDB. NoSQL Databases explained. `http://www.mongodb.com/nosql-explained`, 2015.

[19] Eric Redmond, Jim R. Wilson. *Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement*. Pragmatic Bookshelf, 2012, ISBN 1934356921.

[20] solid IT. DB-Engines Ranking of Graph DBMS. `http://db-engines.com/en/ranking/graph+dbms`, 2015.

[21] Neo Technology. What is a Graph Database? `http://neo4j.com/developer/graph-database`, 2015.

[22] Neo Technology. The Neo4j Manual v2.2.1. `http://neo4j.com/docs/2.2.1/`, 2015.

[23] Nigel Small. Neo4j Index Confusion. `http://nigelsmall.com/neo4j/index-confusion`, 2015.

[24] GraphAware. GraphAware Neo4j Framework. `https://github.com/graphaware/neo4j-framework`, 2014.

[25] Michal Bachman. Introducing GraphAware Neo4j Framework. `http://graphaware.com/neo4j/2014/05/28/graph-aware-neo4j-framework.html`, 2014.

[26] Neo Technology. Example Dataset. `http://neo4j.com/developer/example-data/`, 2015.

[27] Paul Tremberth. Musicbrainz in Neo4j. `http://neo4j.com/blog/musicbrainz-in-neo4j-part-1/`, 2015.

Appendix A

Acronyms

ASCII American Standard Code for Information Interchange

SQL Structured Query Language

ACID atomicity, consistency, isolation, durability

REST Representational State Transfer

API Application Programming Interface

HTTP Hypertext Transfer Protocol

MVC model view controller

OLTP Online Transaction Processing

DML Data Manipulation Language

LIFO last in first out

FIFO first in first out

BFS breadth-first search

DFS depth-first search

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `measurements` ..... results in tables for chapter 4
- `src` ..... the directory of source codes
  - `impl` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format