BULLETIN OF THE POLISH ACADEMY OF SCIENCES
TECHNICAL SCIENCES, Vol. 64, No. 3, 2016
DOI: 10.1515/bpasts-2016-0051

Application of graph databases for transport purposes

A. CZEREPICKI

Warsaw University of Technology, Faculty of Transport, 75 Koszykowa St., 00-662 Warsaw

Abstract. The article presents an innovative concept of applying graph databases in transport information systems. The model of a graph database has been presented together with implementation of data structures and search operations in a graph. The transformation concept of relational model to a graph data model has been developed. The schema of graph database has been proposed for public transport information system purposes. The realization methods have been illustrated by the use of search function based on the Cypher query language.

Key words: databases, graph model, information systems in transport.

1. Introduction

The paper presents the proposal for architecture of information system in transport using a graph database for storing and processing information.

Advanced databases usually constitute a central element of modern intelligent transport systems (ITS). They play the role of an integrator through combining information retrieved from such sources as vehicles, maps, timetables, stops etc. and making it available to end users or storing into data warehouse for discovering and analysing frequently used routes, bottlenecks etc. through advanced on-line analytical processing [1]. However, it should be mentioned that more and more often the recipient of this information is an ordinary user of e.g. application installed on a mobile device.

The growth in the number of users translates into increasing demands on information systems, first of all, on databases. What should be enumerated here are such features as ability to handle large numbers of users, efficiency of query processing and horizontal scalability of the system, which allows simple expansion of the system to meet its performance requirements [2].

Classic relational databases used in most ITS systems are implemented on the basis of a relational data model. In accordance to such model, abstraction of real world objects is stored in the form of records in tables. Relationships among objects are defined by means of integration constraints of primary and foreign key columns. As early as in the 1980s a certain incompatibility was noticed between a relational and an object-oriented approach, which is a more natural reflection of modelled part of reality [3].

A similar mismatch can be observed in information ITS systems which are oriented for search of optimal travel route between two points on the map or public transport stops. The task of finding optimal travel route can be brought down to the task of algorithmic path-finding in a graph conditioned by additional restrictions like time, distance, number of changes, etc.

Operational efficiency of search algorithms in a graph depends on a graph representation form in computer memory. What refers to most often used data structures are neighbourhood matrices and neighbourhood lists [4, 5]. Since traditional databases do not offer their own tools for graph storage, in practice, indirect solutions are applied. In relational databases, conversion of a graph structure into recording in the form of tables and relations is required. Whereas, realization of complex search operations is left to the user.

The alternative solution consists of extending the existing functionality of a database by the possibilities to store the graph structures together with execution of basic search operations, as it was the case with *GraphGB* system originating from object-oriented database [6]. Anyway, all solutions based on existing database systems are burdened with a series of defects, among which we can enumerate, e.g., additional expenses related to data conversion, necessity of operating at a low abstraction level, difficulties with interpretation of data obtained, problems of solution scalability and limited applicability of optimization specific for graphs.

Graph Databases belong to the category of databases named NoSQL (Not Only SQL). NoSQL databases use data models other than relational model. They are characterized by high efficiency and scalability, which are obtained by resigning from data integrity or accessibility [7]. The basis of a data model for graph databases is a graph. Most often it is a directed graph (digraph) whose nodes and edges can have attributes [8].

Depending on practical realization, graph databases differ each other data storage method and in search algorithm realization methods in a graph [9–11]. The present paper limits itself to databases using their own storage and data search methods, at implementing in graph model assumptions. So, adapting a graph model to the needs of ITS systems, can be brought down to the task of path-finding in a graph. In the case of systems using a graph database, there are ensured performance

*e-mail: a.czerepicki@wt.pw.edu.pl

increasing user query processing, indicating, by the same token, a new perspective development direction for dedicated group of ITS systems.

2. Graph database model

By definition, data model is constituted by data structures, a set of operations on data and the constraints ensuring integrity of the whole system [12].

2.1. Data structure of graph database. The basis of representation of a graph data model is graph *G = (W, K)* consisting of a set of *nodes W = {W1, W2, ... Wn}* and *edges K = {K1, K2, ... Km}* connecting them. Each edge has its start and end in the form of elements belonging to the set of nodes *W*. Both nodes and edges can have a list of *attributes*: *L[Wi] = {A1, A2, ... Ap}*, *L[Kj] = {A1, A2, ... Ar}*. The pair *(k, v)* is referred to as attribute *A*, where *k* is the *key,* whereas *v* represents a corresponding *value* for this key. The enumerated elements of a graph model have been presented in Fig. 1.

A graph data model represents:

- a *directed* graph (digraph), i.e. an edge connects *ordered* pair of nodes,
- a *disconnected* graph, i.e. not for each pair of nodes there exists a connecting path (the requirement of connection would make it impossible to add new nodes without simultaneous definition of edge, which would make it difficult to realize a graph database in practice),
- a graph included in the category of *multigraphs*, because there is a possibility of connecting two nodes by means of more than one edge,
- a graph, the *size* of which can be *zero*, because zero number of nodes *n = 0* and edges *m = 0* is admissible in a graph model, which signals zero state of the system.

The data model in Fig. 1. therefore presents the concept of data organization in a graph database. This is a general model, common for all practical graph database instances. The structure of each particular instance of database is created by its nodes – objects and edges – relations. By analogy to relational databases, we will call this construction *schema* of a graph database. In contrary to relational databases, where the schema must be known prior to using databases, schema of a graph database is a very flexible structure and allows to introduce almost any modifications during system operation.

When analysing the path *model – schema – instance*, there should be noted that in graph databases, the border between a schema and an instance is fuzzy: each instance can have its own schema or the schema can be common for several instances. The cases enumerated describe respectively a *non-structural* and *structural* data model. In the case of the structural model, introduction of additional object construction rules are required, similarly to a relational database: each object belonging to a particular *class* of objects has attributes characteristic for this class. When creating an instance of such an object, in the database one should indicate the values of all its attributes except default or automatically filled by the database. In the graph database, there are no mechanisms which would require the user to define all attributes of a node at its creation, therefore. The entire responsibility for this is the part of database application.

The construction process of a graph database schema will be described in details in Section 3.

2.2. Integrity constraints of graph data model. Integrity constraints ensure consistency of database i.e. compatibility of data model with a fragment of reflected reality. In relational databases, these are: object identity, primary and foreign key reference, the requirement of unique records in the table, domain restriction, defaults and null values, etc.

In graph databases integrity is ensured by:

- the requirement of a unique identifier for each node within the scope of the whole database i.e. instance identity, (entity integrity),

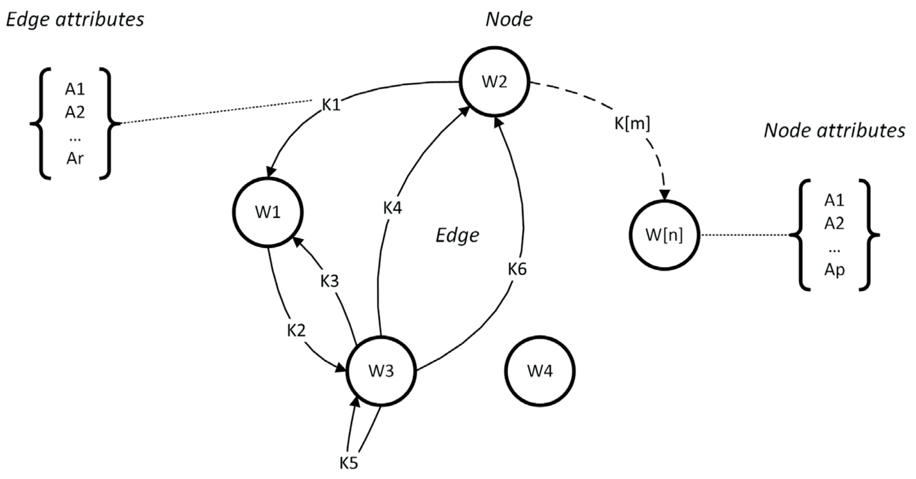


Fig. 1. Data model of graph database

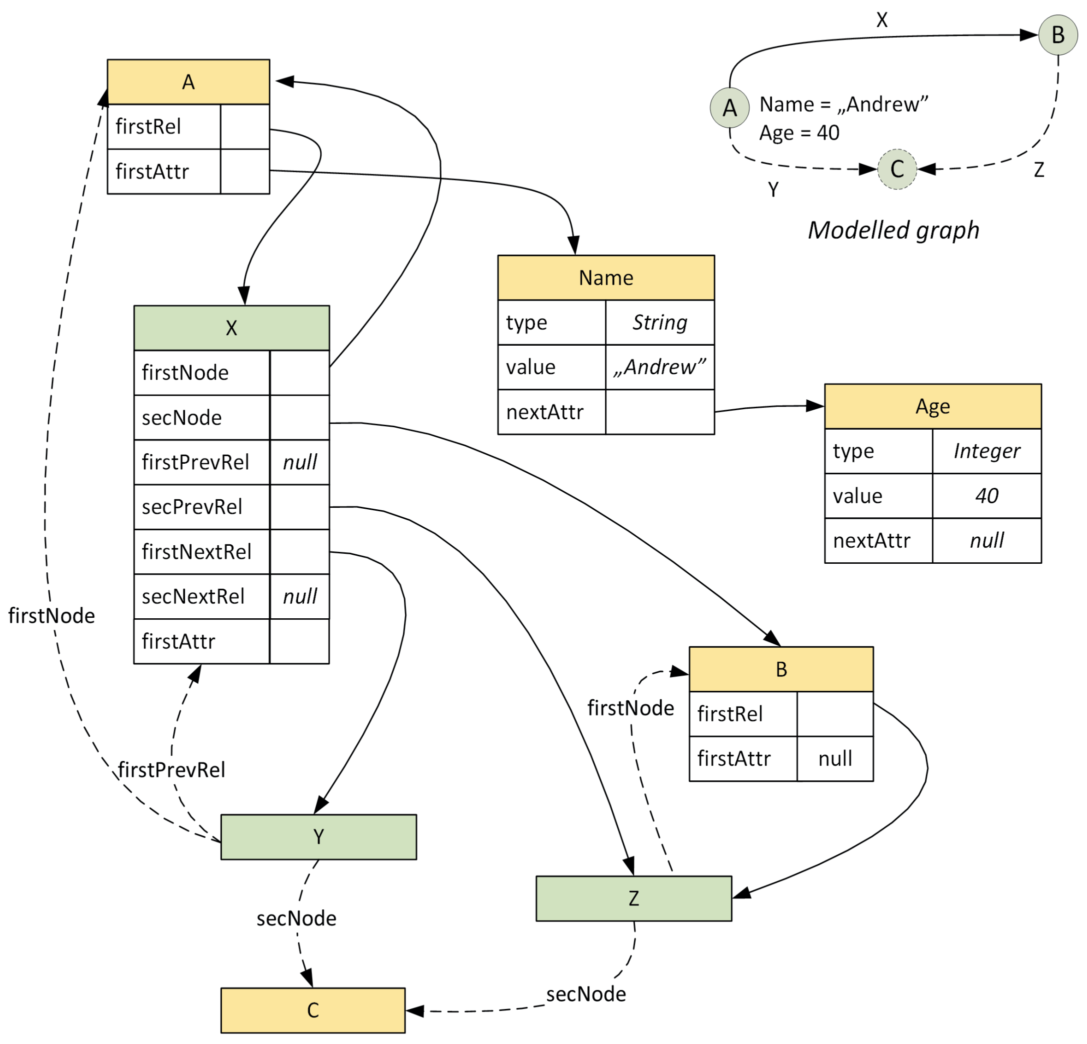


Fig. 2. Structure of graph database file

- the requirement of connecting two nodes by means of an edge (referential integrity),
- the requirement of unique attribute names within one node (attribute integrity).

2.3. Graph database operations. On the basis of the data model presented in Fig. 1, we can distinguish basic *types of data* of a graph database: *node*, *edge*, *attribute key* and *attribute value*. Nodes and edges are represented by complex data structures discussed further in the paper. Attribute key is a text value. Attribute value can be represented by an object belonging to the set of universal data types used in databases, among others, *Integer*, *Boolean*, *String*, *Date*, etc. and also by an array of elements.

Operations conducted on a graph database can have arguments in the form of variables or objects of the abovementioned types. In most databases, including graph databases, we can distinguish four basic types of operations: inserting data, their updating, deleting and searching. The cost of each type of operation directly depends on physical organization of database structure i.e. data storage method in external memory and organization of access to data.

As it results from the reference book analysis [9], the following general format of a graph database record can be assumed (Fig. 2). For the storage of nodes, edges and attributes, separate data files are used. In accordance to assumed classification of structures of internal data file organization, the format presented refers to the category of *unordered files* [13].

For nodes and edges, the size of record is fixed (elements *A* and *X* respectively in Fig. 2). This allows for unconditional addressing an object in a file based on its identifier, which is an integer number. Address of the object is thus defined as a product of record size and identifier number. Accordingly, the complexity of node or edge *search* operation by identifier does not depend on the number of objects in a database and it is equal to *O(1)*. The operation of node or edge *adding* consists of creating a new record at the end of a database file. The cost of such operation is linear *O(1)* and does not depend on a graph size. The operation of *deleting* consists in setting an appropriate flag in object record and does not cause the change of data file size, thus its complexity also is equal to *O(1)*. In accordance to the requirement of referential integrity of a graph database, a node can be deleted, if there are no edges linked to it, whereas, an edge can always be deleted.

Object's attributes are addressed through master objects. For example, in Fig. 2. the attributes *name* and *age* are addressed by means of indicators of master object *A*. The record of a master object indicates the first attribute, while the next attributes are addressed by indicators of attribute records. Thus, the attribute set constitutes the data structure of the *unidirectional list* type. Since attribute value can be represented by a number of types of data, including complex ones, attribute record will comprise either an indicator to another location in a data file in which data is stored (indirect addressing), or the value for basic data types (direct addressing). The operation of attribute searching by name consists of the stage of master object search and unidirectional list browsing. Its complexity therefore is equal to where – number of object attributes. Since *k* number is not large, in most cases, the operation complexity of attribute search can be assumed as *O(1)*. The same complexity is also characteristic for adding or deleting attribute operation.

As can be seen, a data file structure of graph database is optimized for graph searching: the execution time of transversal between two nodes connected by an edge does not depend on the size of a database file.

Operations presented above are related to single objects of a graph database, therefore, they can be named as *simple operations*. In accordance to the assumed approach, *path-finding operation* in a graph is classified as the category of *complex operation*, requiring iterative execution of simple operations to obtain an expected result.

A linked list of graph nodes is connected by edges in such a way that each edge appears on this list only once. This is referred to in presented paper as *path*. In graph databases this path corresponds to a distinct type of data including the list of nodes and edges connecting them in a strict sequence. From formal point of view, the path makes directed graph without a loop. A complex search operation make a set of paths, which in particular cases can be empty. Thus, for a graph in Fig. 2, a search operation of all paths will return the set $\{A \to B, A \to B \to C, A \to C, B \to C\}$.

A path-finding operation can contain constraints in the form of *conditions*, which should be fulfilled by indication of initial or final path node, edge types, middle nodes or the values of selected attributes etc. For example, a query about all paths leading from A to C (Fig. 2) will return response $\{A \to B \to C, A \to C\}$. Another example of a complex search can be an operation returning all paths of defined length, starting and ending at a given node.

Algorithmic complexity of path-finding operations in a graph database can be estimated on the basis of a simple BFS algorithm (breadth-first search) [14]. Its complexity makes $O(|W| + |K|)$, where $|W|$ is a number of nodes, and $|K|$ is a number of edges in a graph. Despite the fact that it is a large number, in practice, path-finding operation time can be significantly reduced by means of imposing additional constraints in the form of search conditions. Search time significantly depends on graph density and the characteristics of its edges. In the case of non-negative edge weights, application of Dijkstra's algorithm, for instance, [15] allows for decreasing algorithm complexity of the shortest path search in a graph weighed to the value of. However, this requires designing in a graph database a separate search algorithm, for example, in a form of a separate module written in a high level programming language.

Table 1 contains a summary of algorithmic complexities of particular operations of a graph database.

In conclusion it can be stated that a graph data model is a natural form of data representation form many algorithmic tasks, including those which have application in the field of logistics and transport. First of all, the problem of finding an optimal route directions should be mentioned here. As there has been shown above, algorithmic complexity of the transversal along the edge connecting two nodes does not depend on the total number of edges in the graph. This is a significant difference in comparison implementation of a graph structure of relational database, where such dependence is a logarithmic function of the total number of edges. This allow to formulate a thesis that the use of graph database can increase efficiency to process some functions in transport information system in

Table 1
Algorithmic complexity of particular operations of graph database

Operation	Time complexity
Adding, deleting and searching for a single object (node, edge) by an identifier	$O(1)$
Adding, deleting and searching for an attribute by a master object identifier and by attribute name	$O(\|A_i\|)$
Adding, deleting and searching for an object by name or attribute value without indexation	$O(\|W\|)$ *for a node* $O(\|K\|)$ *for an edge*
Adding, deleting and searching for an object by name or attribute value using indexing structure	$O(\log(\|W\|))$ *for a node* $O(\log(\|K\|))$ *for an edge*

Searching for all edges outgoing from a node	*for rare graphs* $>= O(1)$
	for dense graphs $<= O(\|K\|)$

Searching for a path between any two nodes	$O(\|W\| + \|K\|)$ *BFS algorithm,* $O(\|W\| \log \|K\|)$ *Dijkstra's algorithm*
Searching for all paths between any two nodes	$O(\|W^2\| + \|K\|)$

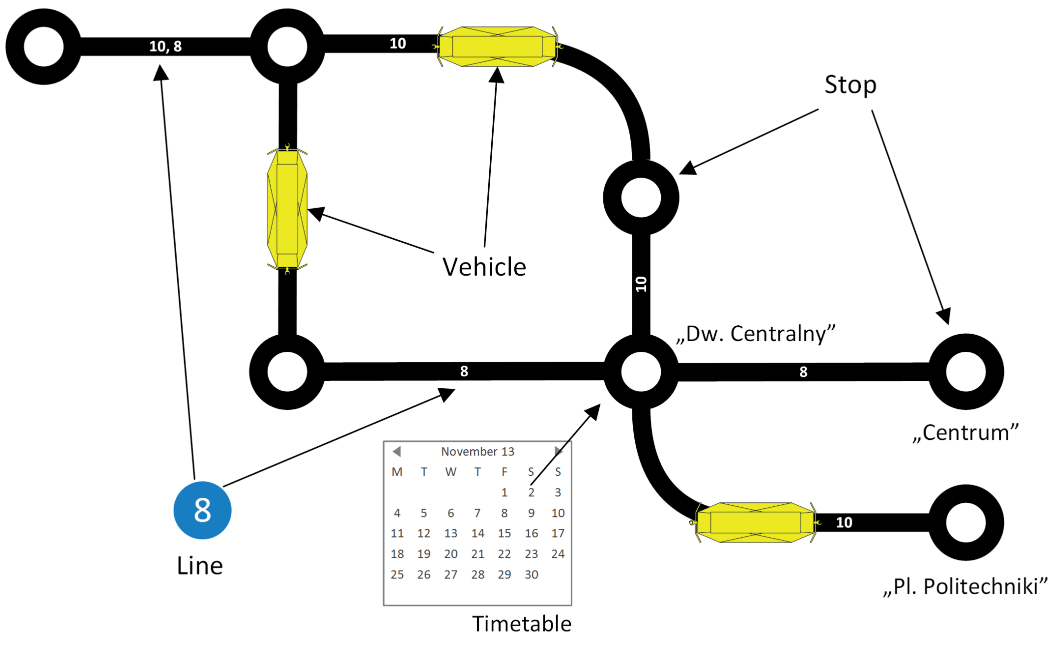


Fig. 3. Model of public transport connections

comparison to systems based on classic relational databases. Especially when a necessary condition is possibility to build an appropriate data schema in which exploited the full potential of databases.

3. Application of graph database for description of public transport connections

3.1. System model. The proposed model of public transport connections consists of elements presented in Fig. 3:

- *stop*: public transport stop which has its own name, number, GPS coordinates,
- *line*: public transport line is a directed list of stops which belong to it,
- *vehicle*: means of transport running on a given line according to its own timetable,
- *timetable*: table with times of departure of a given vehicle from each stop belonging to a given line.

Let $S = \{S_1, S_2, \dots S_k\}$ constitute a set of all stops in the city and let the set $L = \{L_1, L_2, \dots L_n\}$ constitute a set of all urban connections, where each line is presented as a directed list of stops $L_i = \{S_a \to S_b \to \dots \to S_z\}$, where $S_i \in S \ \forall \ i$. The task of the system will be to find the optimal travel route from the initial stop S_p to the final stop S_k. If the optimization criterion is to be adopted as minimization of time of travel, the goal function can be written down as

$$\mathrm{T}(S_p, S_k) = \min\left(\sum_{i=0}^{n-1} (T_i + P_i)\right) \qquad (1)$$

where T is a general travel time which is a sum of constituent times T_i of travelling parts of the route of one line $L_i \in L$, waiting times P_i connected with a change to another line at the same stop. P_o corresponds to waiting time to start a journey, $P_{n-1} = 0$. Journey time can be expressed by one line as $T_i = \sum_{j=0}^{m-1} t_i$ where T_i – is the travel time between two neighbouring stops belonging to the same line L_i, which depends on three factors: line number, stop and departure time resulting from the timetable: $t_i = R(L_i, S_j, \tau)$. The timetable R can be then represented by a three-dimensional matrix. In a model not addresses to timetables, we should assume $P_i = 0 \ \forall \ i$.

3.2. Entity relationship diagram. On the basis of analysis of the domain presented in Fig. 3, the following *entities* have been distinguished which are components of a logical model: *line*, *stop*, *vehicle*, *timetable*.

Line entity characterizes public transport lines. Its key attribute is the line number. Relationships with *stop* entity indicates the initial and final stop of the line. *Stop* entity represents an public transport stop of certain name and location. The name of stop is a key attribute. The entity takes part in a compulsory relationship with a *timetable*, which is individual for each of the stops and in an optional relationship with a *vehicle* indicating the current position of vehicles. *Vehicle* entity represents a specified means of transport, which operates a selected line according to a certain timetable and has a current position. Vehicle registration number is a key attribute.

The entities *line*, *stop* and *vehicle* are strong entities existing independently of a *timetable*. *Timetable* entity is a weak entity existing exclusively in the context of relationships *stop – timetable* (a timetable valid for a given stop), *line – timetable* (a line number for which a timetable is valid) and *vehicle – timetable* (a vehicle operating within a line, which should appear at a stop at a given time).

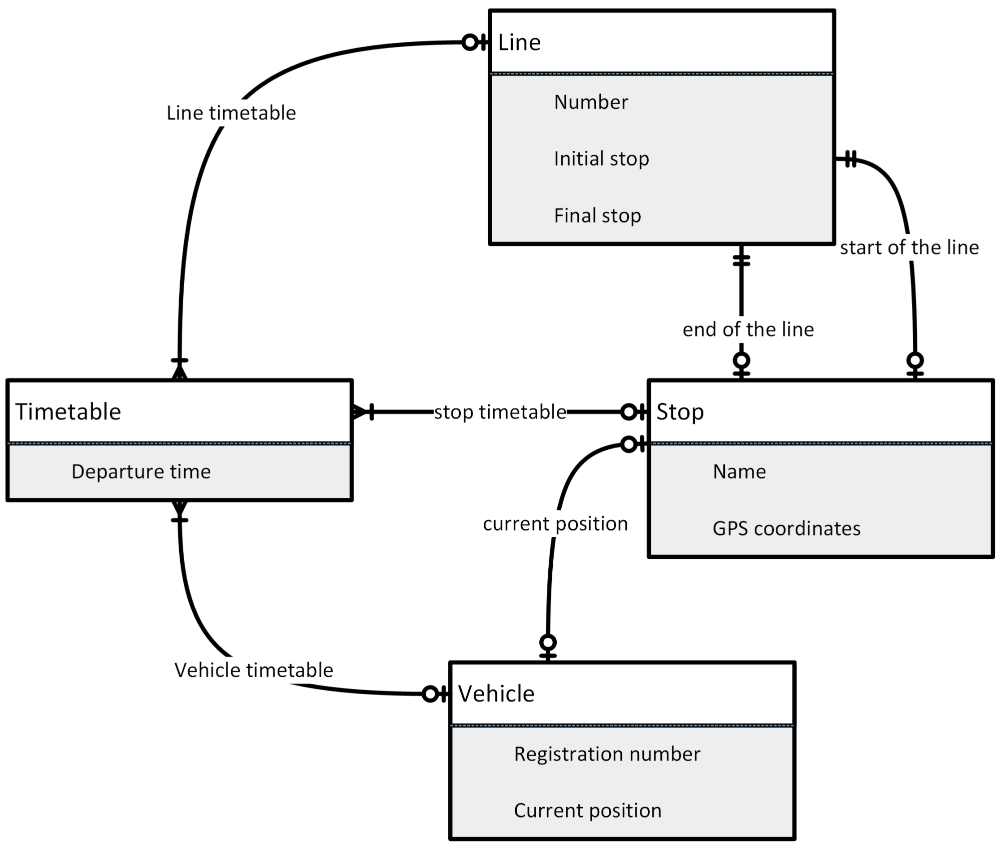


Fig. 4. Entity relationship diagram of public transport connections

Significant attributes of each entity and their associations are presented in Fig. 4. in the form of an entity relationship diagram created in crow's foot notation [16].

It should be noticed that in the model there is no clear mechanism indicating the sequence of stops belonging to the same line, and the travel time between them. The missing data can be calculated based on the timetable. During implementation of information system for the sake of system efficiency, we can consider introduction of additional weak entity *connection* including already calculated data.

3.3. Transformation of entity- relationship model to graph model. The rules of transformation of an entity- relationship model to the schema of a relational database are presented in the reference book [17]. Such a transformation can be conducted in most cases automatically: entities are transformed to the form of a table, entity attributes become table columns, while entity relationship is realized through connections between table keys. Transformation from an entity-relationship model to a graph model is not explicit, because the ultimate form of a final model depends on the input data structure and their connections.

General demands for transformation of an entity-relationship model to a graph model can be formulated in the following way:

- *entity* is transformed to the form of a graph *node*; all nodes representing a given entity have the same set of attributes, the graph, therefore, can consist of nodes of several types,
- *entity attribute* is transformed to the form of *node attribute* of a graph,
- *entity relationship* is transformed to the form of a graph *set of edges* connecting nodes representing the entity.

The transformation method of entity relationships depends on several factors: multiplicity of a relationship (*one-to-one, one-to-many* or *many-to-many*), optionality of a relationship (*optional* or *compulsory*) and directionality of a relationship. Examples of transformations of selected entity relationships are presented in Fig. 5.

Multiplicity of entity relationship affects the number of edges connecting nodes corresponding to entities. *One-to-one* relationship in a graph structure reflects itself explicitly (Fig. 5a). *One-to-many* and *many-to-one* relationships are identical in execution, which consists in connecting a node representing an entity on the side *one*, with all (or selected – in the case of an optional relationship) nodes on the side *many* (Fig. 5b). A unique feature of a graph model in comparison to a relational model is possibility of direct implementation of relationship *many-to-many* (Fig. 5c). In a relational database, such relationship would require introduction of an additional table in the schema of data.

Optionality of entity relationships is translated into node connection: a compulsory relationship *A-to-B* means that a node corresponding to an entity *A* must have connections to all nodes representing an entity *B*. In this context *optionality* means that part of enumerated edges may not exist (Fig. 5c).

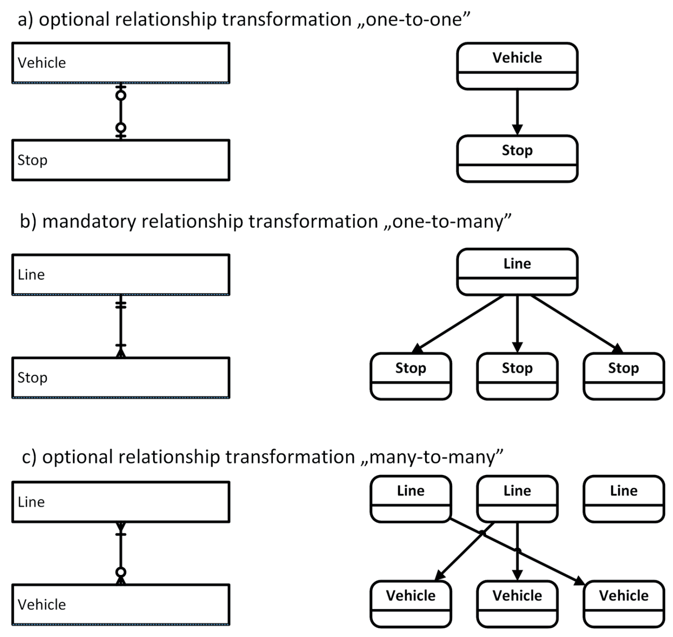

a) optional relationship transformation „one-to-one"
Vehicle
Stop
Vehicle
Stop
b) mandatory relationship transformation „one-to-many"
Line
Stop
Line
Stop
Stop
Stop
c) optional relationship transformation „many-to-many"
Line
Vehicle
Line
Line
Line
Vehicle
Vehicle
Vehicle

Fig. 5. Transformation methods of entity relationship to graph model

A graph data model introduces a notion of *directionality of relationship.* In a relational model, an entity relationship is a bilateral one, while in a graph model there are only directed relationships. In other words, representation of relationship *A-to-B* in a graph model requires execution of relations $A \to B$ and $B \to A$. In practice, relationship implementation in both directions depends on the logic of business application and it is not always necessary.

Following the above mentioned rules, the result of an entity relationship model transformation of the public transport connections system to a graph model can be presented in the form of fig 6. Integrity of the data model is ensured by the following constraints:

- two nodes of the same type cannot have the same value of a key attribute i.e. *stop* name, *line* number and *vehicle* registration number are unique within the system,
- from one node of *stop* type there cannot outgo two edges of identical values of *line* attribute i.e. for each line from one stop there is a departure in only one direction because lines $A \to B$ and $B \to A$ and constitute separate instances of the entity *line*,
- a graph does not have a loop i.e. the next *stop* for *vehicle* running according to a certain *line* cannot be the same *stop*.

3.4. Construction of graph database structure. In the case of graph databases there is no clear division into data *schema* and *data* themselves, as it takes place in relational databases. In general the structure of relational database is rigid and its change is associated with the necessity of modification of large data sets. In contrary to relational database, the structure of graph database depends directly on data processed within the system and can be flexibly changed. Thus, an empty graph database does not include either nodes or edges, whereas a table within a relational database always exists.

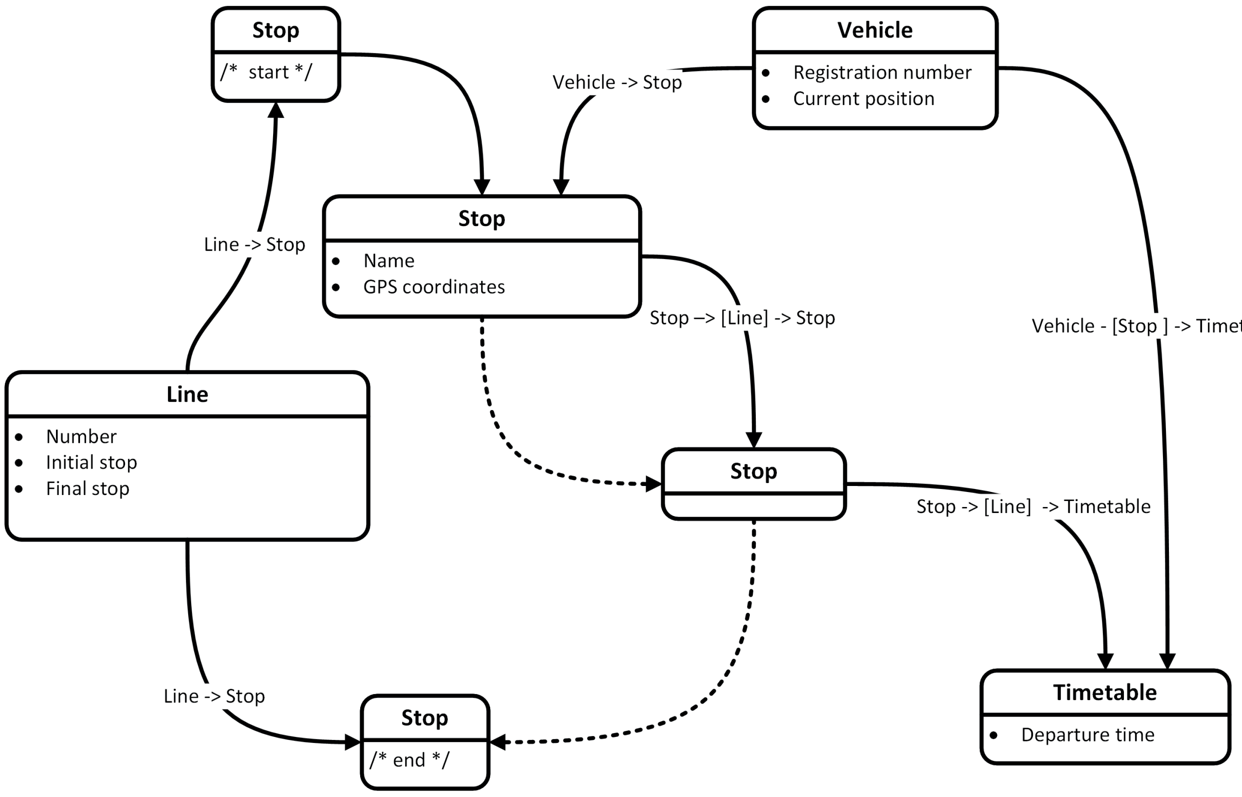


Fig. 6. Graph data model of public transport connection system

Figure 7 illustrates a part of the structure of sample graph database built according to the graph model presented in Fig. 6. The model system comprises two lines L_1 and L_2 connecting seven stops $S_1..S_7$. The lines are related to vehicles $P_1..P_3$ in certain locations. Connections are executed respectively by relations of the type *operates* and *location*. A timetable is represented by a relation of the type *timetable,* which connects subsequent stops $P_i \rightarrow P_{i+1}$ belonging to one line L. Relation of this type has attributes {*departure*, *time*} and represents respectively departure time from the stop and the time of travelling the route to the next stop. In order to simplify the figure, in the described database schema, unidirectional lines have been used.

The algorithm for creating a graph database structure consists of two stages. In the first stage, the nodes of the types *stop* (S), *vehicle* (P) and *line* (L) to the database are added. Then, the connections between the nodes of the type *timetable* $S \rightarrow S$, *location* $P \rightarrow S$, *begin* and *end* $L \rightarrow S$ are defined. For communication with a graph database, a Cypher query language has been used. It is a universal declarative language of communication with a database designed and developed as an integral part of the graph database Neo4j [18]. Figure 8 presents the syntax of commands defining a node of the type *stop* (a) and an edge of the type *timetable* (b). In order to simplify the calculation, departure time can be expressed as an integer.

It should be noticed that the dynamic element of the structure is the location of the vehicle, changed in the time. The remaining elements of structure are static. This allows to minimizing the cost of updating the data in the system and focusing on optimization of connection search task, as well as accounting for delay or

```
create ( a {name : 'S1'} )
create ( b {name : 'S2'} )
```

a)

```
start a=node(2), b=node(3)
create a-[r:L1 {
       name : 'timetable',
       departure:1000, time:5 }]->b
```

b)

Fig. 8. Defining stops and connections in Cypher language

failure of a vehicle. By this quality evaluation of journey system can be increased and adapted to actual conditions.

3.5. Execution of search operations in graph database. Search operations in a graph database can be executed by means of specialized graph query language, such as the previously introduced Cypher language, or programming in a high level programming language e.g. Java. Both approaches have their advantages and disadvantages. There is easier to formulate simple queries in a graph database. In this case database server is responsible for interpretations and execution of commands. A database user does not have to execute complex algorithms of search in a graph. The ability is however, required to formulate queries correctly and to interpret the results presented in the form of plain text. When the high level language is used,

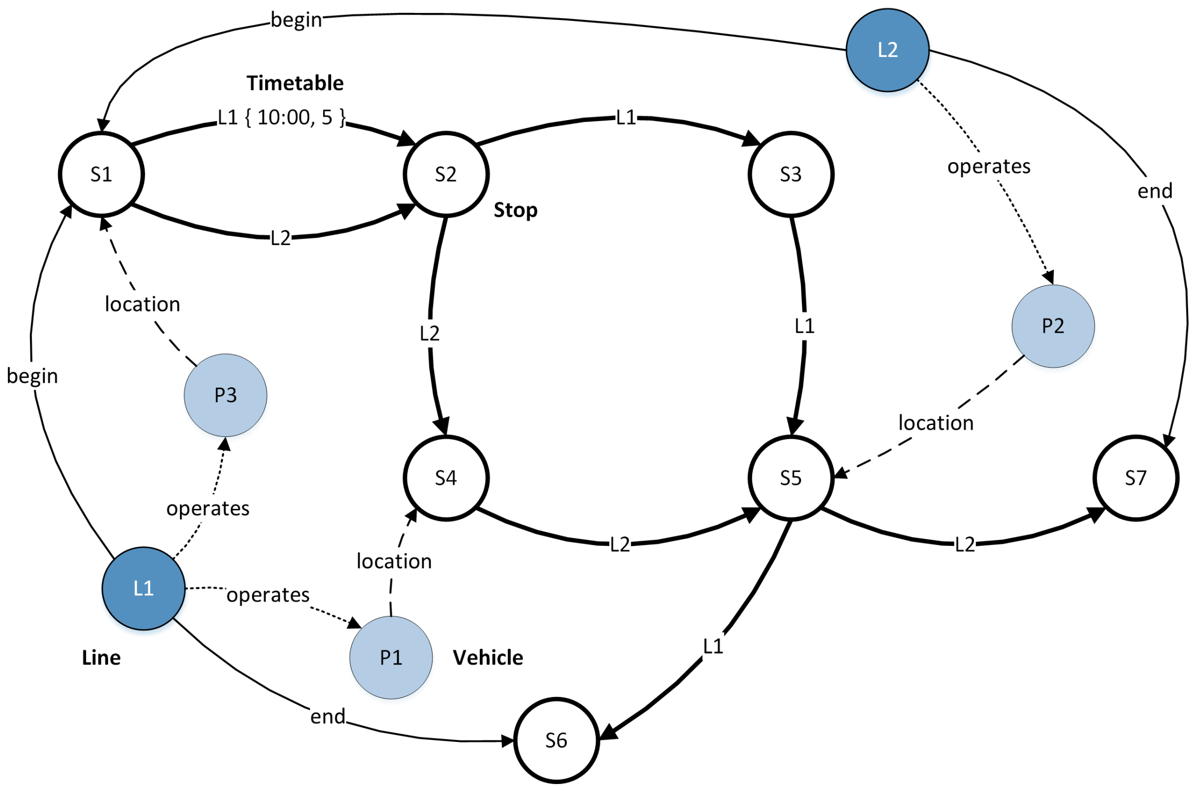


Fig. 7. Part of sample graph database structure

the stage of formulating a query can be omitted by acting directly on database objects – nodes, edges and paths. One can implement one's own algorithm of graph traversal if a built-in implementation is ineffective for a given case.

In Fig. 9 a simplified syntax diagram of a query in Cypher language executing the function of graph transversal are presented. This is equivalent to *select* operation in the SQL language. The keyword *start* specifies the conditions for starting a search operation, for example, it indicates an initial node in a graph. Section *match* includes a pattern of path transversal in the form of nodes and edges list. Instruction *where* serves the purpose of filtering results returned by the query and it can contain logical conditions operating on attributes of nodes or edges. The query results are returned in section *return*.

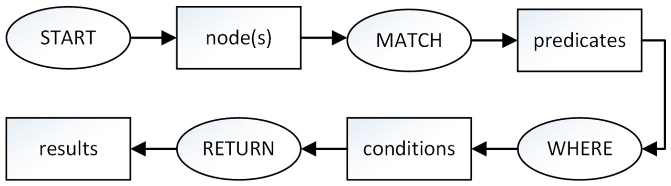


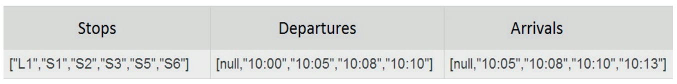
Fig. 9. Syntax diagram of graph transversal instruction in Cypher language

The timetable of a given line can be downloaded from a graph database by means of the instruction (Fig. 10). Nodes represented by variables a and b are respectively a line number and its final stop. In the example, addressing the nodes by their names has been used by means of an index of the name *nodes*. Regular expression in the section *match* limits the set of paths returned to the subset fulfilling the condition of belonging to the line L_1 and of the length not smaller than 2. The second condition serves the purpose of elimination of auxiliary edge connecting node L_1 with a final stop S_6.

Variable p comprises the query result in the form of paths in a graph. In order to display the information obtained in a legible form for the user, instruction *extract* has been used which for each of the stops on the path displays its name, departure time and travel time to the next stop. The query result in the form of text has been presented in Fig. 11.

```
start
    a=node:nodes(name="L1"),
    b=node:nodes(name="S6")
match
    p=(a)-[r:L1*2..]->(b)
return
    extract ( x in nodes(p) : x.name )
           as stops,
    extract (
    k in relationships(p) : k.departure)
           as departures,
    extract ( k in relationships(p) :
    k.departure + k.time)
           as arrivals
```

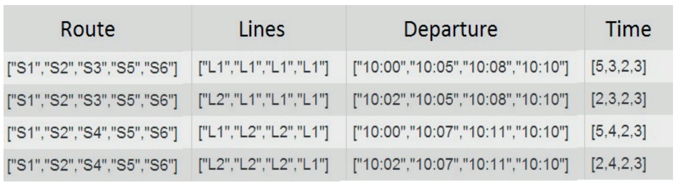
Fig. 10. Query that returns timetable of a given line



Stops	Departures	Arrivals
["L1","S1","S2","S3","S5","S6"]	[null,"10:00","10:05","10:08","10:10"]	[null,"10:05","10:08","10:10","10:13"]

Fig. 11. Query result returning the timetable of a given line

In order to obtain all possible connections between two selected stops, the query to the database should be modified. The potential changes can be accounted for by including line in the form of condition $[r{:}L1|L2*]$ in section *match*. For the connection $S_1 \to S_6$ the query will return 4 possible journey variants (Fig. 12).



Route	Lines	Departure	Time
["S1","S2","S3","S5","S6"]	["L1","L1","L1","L1"]	["10:00","10:05","10:08","10:10"]	[5,3,2,3]
["S1","S2","S3","S5","S6"]	["L2","L1","L1","L1"]	["10:02","10:05","10:08","10:10"]	[2,3,2,3]
["S1","S2","S4","S5","S6"]	["L1","L2","L2","L1"]	["10:00","10:07","10:11","10:10"]	[5,4,2,3]
["S1","S2","S4","S5","S6"]	["L2","L2","L2","L1"]	["10:02","10:07","10:11","10:10"]	[2,4,2,3]

Fig. 12. All possible connections between stops $S_1 .. S_6$

There is easy to notice that among four possible connections, two are not correct from the point of view of the current timetable because time for the line change cannot be negative i.e. departure time cannot be earlier than arrival time. Cypher language currently does not have mechanisms making possible to operate on edge attributes [19]. The problem can be solved by conducting additional verification of the result obtained $t_{departure}(P_{i+1}) \geq t_{departure}(P_i) + t_{journey}(P_i \to P_{i+1})$ for all pairs of connected stops. In the case of the search algorithm implementation by the programming interface *API* , query execution time does not change.

Limiting of departure time from an initial stop can be introduced by the *where* condition (Fig. 13).

Query result presented in Fig. 12 will be narrowed down to the second and fourth position, which will fulfil basic conditions. The only possible journey in this case starts L_2 with a change to L_1 at stop S_2. The variant with a change at stop

```
start
    a=node:nodes(name="S1"),// initial stop
    b=node:nodes(name="S6") // final stop
match
    p=(a)-[r:L1|L2*]->(b) // route L1 or L2
where
    all(rel in r WHERE rel.departure >= 1001)
    // departure time 10:01
return
  extract( n in nodes(p) : n.name )
     as route,
  extract( k in relationships(p) : type(k) )
     as lines,
  extract( k in relationships(p) : k.departure )
     as departure,
  extract( k in relationships(p) : k.time )
     as time
```

Fig. 13. Query that returns all possible routes with limiting of departure time

S_5 does not fulfill the condition of non-negative time for a change. In the case of returning several correct route variants, the optimal solution of the task will be a variant where the value $t_{departure}(P_i) + t_{journey}(P_i \rightarrow P_{i+1})$ for the last edge will be minimal i.e. the shortest travel time. If for the assessment criterion of route quality minimal number of changes has been selected, the query should be accompanied by a clause returning the number of changes *count*(*distinct lines*) as *changes* and then the variant should be selected where the variable *changes* will be minimal.

4. Conclusions

The paper presents the idea of graph database and its implementation in the task solution of optimal route search connecting two stops in public transport environment. Graph databases make a rapidly growing segment of modern non-relational databases and refer itself to the category of NoSQL database. They are characterized by high efficiency, scalability and ability to handle a large number of users. A graph data model is a natural form of abstraction of transport tasks addressed to optimal travel route search. Therefore, the use of a graph database in the above mentioned task-oriented information system improved the quality of system by shortening the response time of the system and simplifying the implementation of given system functions.

On the example of public transport information system, the transition method from a classic relational model to a graph model has been demonstrated. Basic rules have been designed for entities and their relationships transform process to objects and relations of graph database. A general graph database model has been developed and on its basis a detailed schema of a graph database of information system has been proposed. Using Cypher query language of a graph database, basic commands have been given to search the vehicle routes and the connections between indicated stops.

The idea of the system presented in the paper has innovative features as:

- modern technology is applied in the form of graph database solving the problems traditionally belonging the relational databases,
- flexible structure is used which can be easily adapted to the actual transport requirements for accounting location of transport means, delays, failures, etc.,
- efficiency of basic task solutions is done due to adapting domain model to a database model.

The system model proposed in the paper has been tested using the data corresponding actual situation data. All the discussed queries have been checked in terms of correctness of returned data. In some cases, it was shown that the possibilities of the query language used in the work of database system are not sufficient for execution of all required functions. The solution proposed by the author is using libraries of direct access to database objects using the high level programming language e.g. Java.

The subject of a separate study is an experimental assessment of efficiency of graph transversal algorithms built in the graph database management system and a possible implementation of one's own algorithm. Calculation of complexity and an average time of algorithm implementation are also subject to practical verification together with a comparison of results obtained for relational and graph databases.

References

[1] B. Bębel, T. Morzy, Z. Królikowski and R. Wrembel, "Formal model of time point-based sequential data for OLAP-like analysis", *Bull. Pol. Ac.: Tech* 62, 331–340 (2014).

[2] A. Czerepicki, "Perspectives of using NoSQL databases in intelligent transportation systems", *Transport* 92, 29–38 (2013).

[3] C. J. Date and H. Darween, *Foundation for Object/Relational Databases: The Third Manifesto*, Addison Wesley, 2001.

[4] A. Malikov, "Directed graphs in relational database", *Proceedings of Tomsk State University of Control Systems and Radioelectronics* 2 (18), 100–104 (2008).

[5] D. Knuth, *The Art of Computer Programming*, Addison- Wesley, 2011.

[6] R. Güting, "GraphDB: modeling and querying graphs in databases", *Proceedings of the 20th International Conference on Very Large Databases*, 297–308 (1994).

[7] E. Brewer, "A certain freedom: thoughts on the CAP theorem", *Proceeding of the XXIX ACM SIGACT-SIGOPS symposium on Principles of distributed computing* (2010).

[8] "Neo4j: What is a Graph database?", (http://www.neo4j.org/learn/graphdatabase , accessed 2013.11.15).

[9] I. Robinson, J. Webber and E. Eifrem, *Graph Databases*, O'Reilly Media, 2013.

[10] S. Salihoglu and J. Widom, "GPS: a Graph processing system", *Proceedings of the 25th International Conference on Scientific and Statistical Database Management* (2013).

[11] N. Martınez-Bazan, S. Gomez-Villamor and F. Escale-Claveras, "DEX: A high-performance graph database management system", *Data Engineering Workshops (ICDEW), IEEE 27th International Conference* (2011).

[12] P. Beynon-Davies, *Database Systems,* third edition, Palgrave Macmillan, NY, 2003.

[13] R. Elmasri and S. Navathe, *Fundamentals of Database Systems.* 6th Edition, Addison-Wesley, 2011.

[14] M. Kurant, A. Markopoulou and P. Thiran, "On the bias of BFS (Breadth First Search)", *International Teletraffic Congress (ITC 22), Cornell University Library* (2010) (http://arxiv.org, article id: 1004.1729, accessed 2013.11.15).

[15] E. Dijkstra, "A note on two problems in connection with graphs", *Numerische Mathematik* 1, 269–271 (1959).

[16] T. Halpin and T. Morgan, *Information Modelling and Relational Databases*, Elsevier Science, 2008.

[17] T. Connolly, C. Begg and R. Holowczak, *Business Database Systems*, Addison-Wesley, 2008.

[18] "Cypher Query Language" (http://docs.neo4j.org/ chunked/milestone/ cypher-introduction.html, accessed 2013.11.15).

[19] "Limiting a Neo4j cypher query results by sum of relationship property" (http://stackoverflow.com/ questions/12449295/limiting- a-neo4j-cypher-query-results-by-sum-of-relationship-property, acce