CMST 22(4) 217-224 (2016) DOI:10.12921/cmst.2016.0000014

Visualisation of Relational Database Structure by Graph Database

Przemysław Idziaszek, Wojciech Mueller, Janina Rudowicz-Nawrocka*, Michał Gruszczyński, Sebastian Kujawa, Karolina Górna, Kinga Balcerzak

Poznań University of Life Science
ul. Wojska Polskiego 28, 60-637 Poznań

**E-mail: jrudowicznawrocka@gmail.com*

Received: 29 March 2016 ; revised: 27 October 2016 ; accepted: 28 October 2016; published online: 02 December 2016

Abstract: Most IT systems rely on dedicated databases, and most of these databases are relational. The advantages of such databases are well known and widely reported in literature. Unfortunately, attempts to identify the topology of links in the relational model produced by iterative development or administrative enhancements are often hampered by the large number of tables that make up the database and the lack of comprehensive technical documentation. Analysis of the model by someone other than its designer requires substantial effort. The aim of the presented work is therefore to develop an application for effective presentation of the database structure in the form of a directed graph. The main assumption was that a graph-oriented database environment would be used. This paper presents the RELATIONS-Graph application developed by the authors. This application automatically generates a directed graph which presents links between tables and attributes which constitute a relational database. The RELATIONS-Graph application can also scan the generated graph in order to discover links between selected tables and columns. This solution has been applied to SQL Server 2014 SP1 DBMS using the *Microsoft .NET* technology and the *Neo4j* graph database, also by .NET API. The RELATIONS-Graph application was developed in *C#*, an object-oriented programming language.

Key words: relational database, graph database, Neo4j, visualisation of relations, database relations graph

I. INTRODUCTION AND KEY TECHNOLOGIES

Ongoing development of existing applications generally involves modification of relational databases, and these modifications are usually based on an understanding of the database structure [1]. Similar understanding is also required in the scope of system administration. A visualisation of database links, for example in the form of a graph, is a prerequisite of proper database structure analysis [2].

MS SQL Server, a Relational Database Management System, supports the creation of database diagrams but does not provide any accompanying query mechanism. *MS SQL Management Studio* [3] can create entity diagrams, automatically verify links between tables and present link attributes to the user, who is then able to recognize, for example, the name of a given relation and the names of columns used to enforce that relation. It is also possible to view the properties of all tables and columns comprising the diagram [4].

The database diagram, when made up of multiple tables linked by a large number of relations, is often unreadable. In practical terms, its structure cannot be easily visualized in the editor window, and no opportunity to query the diagram with a dedicated query language is provided.

The main aim of our work was to create a solution which would visualise the links present in a relational database in the form of a directed graph, and to query this graph in order to discover links between selected tables and columns. In particular, the ability to query the graph improves the user's understanding of the database structure. The main assumption was that a graph-oriented database environment would be used. Our model assumes that each relation corresponds to a table, with columns listed as attributes and links presented as connections between tables.

Several relatively new technologies which attempt to solve these issues are currently available. They include NoSQL databases [5, 6], with graph databases as a subset.

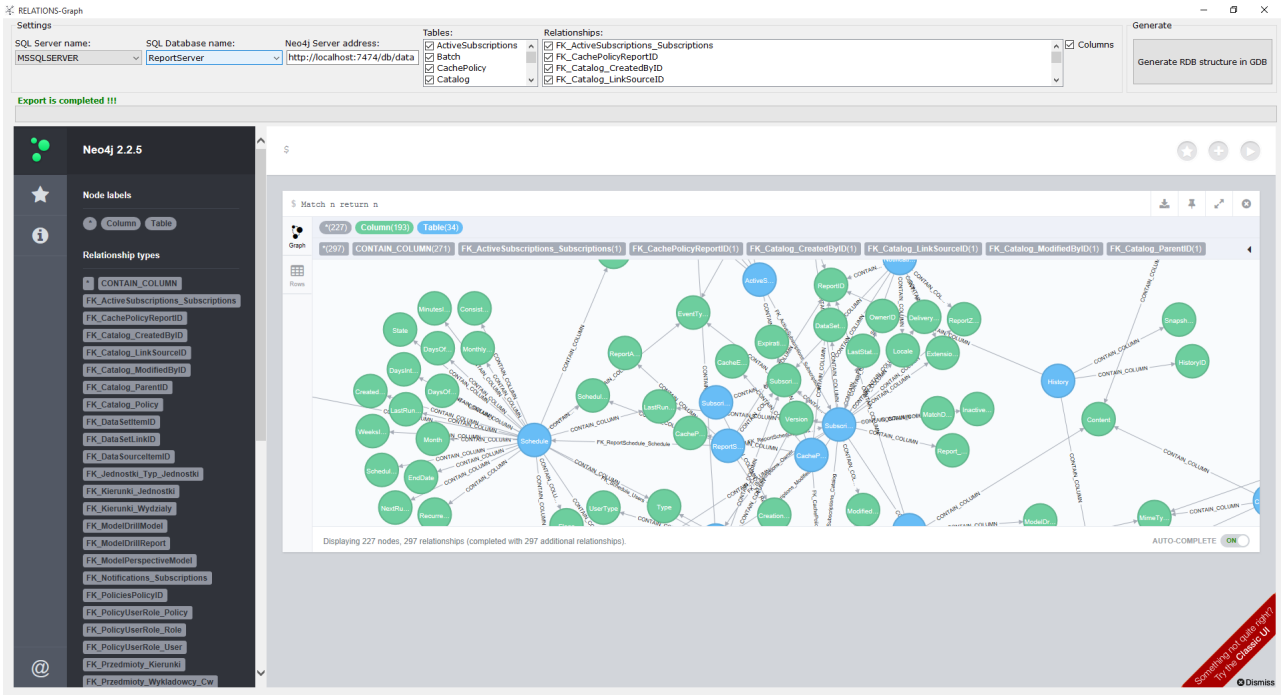


Fig. 1. RELATIONS-Graph application interface and example of a graph of links between database tables and columns [own source]

Such structures are well poised to store data comprising a large number of links. An example of a graph database is Neo4j (http://neo4j.com/), which is distributed under GPL *(General Public License)* [7].

The *Neo4j* database stores objects in the form of graph nodes, with relations between objects represented as edges [8]. In addition, both edges and nodes can be equipped with any number of properties called attributes. Apart from the widely known capability of graph databases to perform powerful JOIN operations, *Neo4j* also allows its user to create indexes [9]. Such indexes can be created on nodes' or edges' attributes, improving the efficiency of searches. Currently, two languages can be used to perform queries: *CYPHER* and *GREMLIN*. The default tool used to manage the graph structure interactively is a built-in web user interface; in addition, a dedicated *Neoclipse* application is also available [10, 11]. The latter allows users to connect to the graph structure in order to scan or modify it [12]. There is also a considerable group of dedicated *APIs (Application Programming Interface)*, including APIs for object-oriented programming languages, such as C# or Java. The *REST (Representational State Transfer) API* is an additional asset. It allows programmers to start interactions with the *Neo4j* database via HTTP *(Hypertext Transfer Protocol)*.

II. RELATIONS-GRAPH APPLICATION

The application developed by the authors and described in this article is called RELATIONS-Graph. Its primary task is to present the structure of a relational database in the form of a directed graph. The graph consists of nodes representing database tables and columns, as well as edges which represent links between tables and columns (Fig. 1). Moreover, the application can be used to query the graph in order to discover links between tables and columns. It also presents the properties of each link.

This application is an attempt to address the inconveniences and limitations experienced while viewing the structure of relations in MS SQL Server SP1 2014 (and earlier versions) databases using the MS SQL Server Management Studio [13]. The application uses Windows Forms and takes advantage of multithreading. The GUI (Graphical *User Interface*) and the backend logic of the application use separate CPU *(Central Processing Unit)* threads [14, 15]. This means that – despite the complex and time-consuming processing which takes place when dealing with a large number of tables – the application is not subject to "freezing" and its interface remains available to the user. Another advantage of this solution is the ability to inform the user about the level of progress in mapping the relational structures to a graph, visualized as a progress bar. The user can always view the total number of tables in the selected database and the name of the table being mapped to a graph node. Separation of system threads is provided by the *BackgroundWorker* class, which is part of the *System.ComponentModel* namespace. An object of this class implements three events: *DoWork*, *ProgressChanged* and *RunWorkerCompleted*, to which the user can assign appropriate actions in the code.

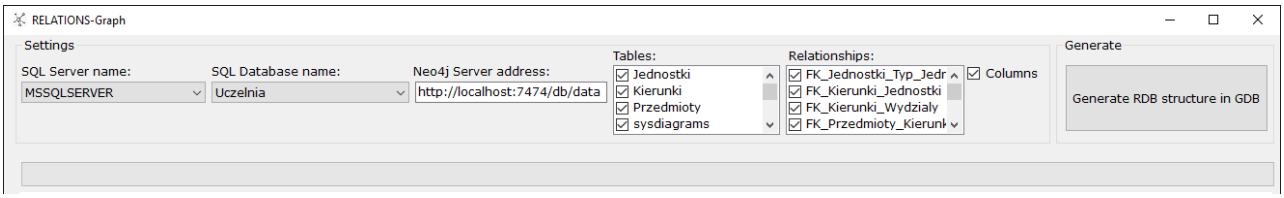


Fig. 2. Choice of relational database in RELATIONS-Graph application [own source]

The selection of the relational structure to be presented in the form of a graph is performed by the user with a graphical interface (*Settings* block – see Fig. 2).

Creating the graph of RDBMS databases relations

The first step to create a graph of database links is defining by user an instance of the *MS SQL Server* by selecting it from a list of available instances within the operating system (OS). The information contained in the drop-down list is retrieved directly from the OS registry. This functionality is provided by the *RegistryValueDataReader* class. Once a relational database server instance has been selected, the names of all databases available within the given instance are automatically loaded into the next drop-down list. This is done by using a *foreach* loop to scan the database collection provided by the *Server* class, instantiated by supplying the name of the previously selected instance to its constructor [16]. Both classes (*server* and *database*) belong to the *Microsoft.SqlServer.Management.Smo* namespace. Next, all tables from the selected database are loaded into the "Tables" drop-down list. At the same time, the RELATIONS-Graph Application look through all the tables for foreign keys. Each foreign key is added to the "Relationships" drop-down list. The last step is taking a decision whether the columns defined within the tables will take part in a process of generating graph or not.

The address of the *Neo4j* graph database server to which the selected relational structure is mapped is typed into the *Neo4j Server Address* textbox, if different from the default value [17]. *NeoTechnology* bases on the principle that the server address is nearly invariant. It is important to note, however, that when using *Neo4j Community,* the user should also indicate the folder containing the files of the graph database [18].

A link to the *Neo4j* graph database is passed to the GraphClient class constructor that comes from Neo4jClient namespace. This space becomes available after the application is extended with a reference to *.NET API* libraries provided by *NeoTechnology*. The main class of the said space is *GraphClient*, which is not only used to implement the link but also to build queries. To establish the link, the *Connect* method is used. This method does not require any input parameters to connect as the server address had previously been transferred to the *GraphClient* class constructor.

The next step is to select the graph generation mode. The user selects tables and links, and then decides which additional information will be presented in the graph. By default, all tables and relations are selected, because we assume that the required adjustments will be made while querying the graph database. The current version of the application does not permit it to be executed from the command line with appropriate parameters. The application checks whether all needed options are selected and starts generating the graph with clearing any existing nodes in the graph database. The method is shown in Fig. 3.

```
void cClear_Graph_DB(GraphClient graphClient)
{
   graphClient.Cypher
   .Match("b")
   .OptionalMatch("(b)-[r]-()")
   .Delete("r, b")
   .ExecuteWithoutResults();
}
```

Fig. 3. Clearing graph database [own source]

The graph structure is built by using suitably prepared queries in the *CYPHER* language [19]. A dedicated API of the *Neo4j* graph database supports two ways of creating queries. The first way, recommended by the supplier, involves running suitable methods provided by the *GraphClient* class. These methods *(Match, AndWhere, Create, ExecuteWithoutResults)* are available by inheriting the *ICypherFluentQuery* interface by the GraphClient class. The second approach consists of creating a query as a string (sequence of characters). The first approach is described in details below, and some comments about the second approach are later.

```
public class Node_Table_Class
  {
    public string Name { get; set; }
    public string ID { get; set; }
  }
```

Fig. 4. Class which describes table's attributes in C# [own source]

After clearing the graph database the nodes with label "Tables" are generating. These nodes map tables from database,

each node one table and its attributes. The class describing table's attributes at level of C# and the method of creating the table with Neo4j API are shown in Fig. 4 and 5.

```
void cAdd_Table_To_Graph(GraphClient graphClient,
    Node_Table_Class cTab)
  {
    graphClient.Cypher
      .Create("(a:Table␣{newTab})")
      .WithParam("newTab", cTab)
      .ExecuteWithoutResults();
  }
```

Fig. 5. Method of creating the table with Neo4j API [own source]

Next, the links between nodes "Tables" are generated. These links present the foreign key relations. Each link, also called relation according to the graph database theory, has the same name as the foreign key at the level of MS SQL Server DBMS. In the graph database the relation is directed from the node representing the main key table to the node representing the foreign key table. At the level of Neo4j graph database relations can also have attributes, but in case of the RELATIONS-Graph Application they have only the name. The method of creating the relation at the level of Neo4j is presented in Fig. 6.

```
void cAdd_Relationship_Foreign_Key(GraphClient
    graphClient, string cName_PK_Table, string
    cName_FK_Table, string cFK_name)
  {
    graphClient.Cypher.Match("(a:Table)", "(b:
        Table)")
      .Where((Node_Table_Class a) =$>$ a.Name ==
          cName_PK_Table)
      .AndWhere((Node_Table_Class b) =$>$ b.Name
          == cName_FK_Table)
      .Create("a-[:" + cFK_name + "]-$>$b")
      .ExecuteWithoutResults();
    var query = new CypherQuery("MATCH␣(a:Table)␣
        WHERE␣a.Nazwa='Przedmioty'␣RETURN␣a", new
         Dictionary$<$string, object$>$(),
        CypherResultMode.Set);
    var result = ((IRawGraphClient)graphClient).
        ExecuteGetCypherResults$<
        $Node_Table_Class$>$(query).ToList();
  }
```

Fig. 6. Method of creating relation with Neo4j API [own source]

In the last stage our application checks whether the columns are selected by the user and, if yes, nodes with labels "Columns" are attached to the nodes "Tables". The class of describing a column at level of C# and the method of adding a column node to the graph with Neo4j API are shown in Fig. 7 and 8. The application assures that each column is added only once. The method of creating relations between the Table node and the Column node is shown in Fig. 9.

```
public class Node_Column_Class
  {
    public string Name { get; set; }
    public string ID { get; set; }
    public string Data_type { get; set; }
    public bool Identity {get;set;}
    public bool Primary_key { get; set; }
    public bool Foreign_key { get; set; }
    public bool Allow_NULL { get; set; }
  }
```

Fig. 7. Class which describes the column node in C# [own source]

```
void cAdd_Column_To_Graph(GraphClient graphClient,
     Node_Column_Class cCol)
  {
    graphClient.Cypher
      .Create(``(a:Column {newKol})'')
      .WithParam(``newKol'', cCol)
      .ExecuteWithoutResults();
  }
```

Fig. 8. Method of adding column node to the graph with Neo4j API [own source]

```
void cAdd_Relationship_Table_To_Column(GraphClient
     graphClient, string cRelationship, string
    cTab_name, string cCol_name, string
    cData_type)
  {
    graphClient.Cypher.Match("(a:Table)", "(b:
        Column)")
      .Where((Node_Table_Class a) =$>$ a.Name ==
          cTab_name)
      .AndWhere((Node_Column_Class b) =$>$ b.Name
          == cCol_name)
      .AndWhere((Node_Column_Class b) =$>$ b.
          Data_type == cData_type)
      .Create("a-[:" + cRelationship + "]-$>$b")
      .ExecuteWithoutResults();
  }
```

Fig. 9. Method of creating relation between table node and Column node with Neo4j API [own source]

The abovementioned multistage procedure is used to create the graph of relations based on the given relational database. The example of the result as the graph's node with relations is shown in Fig. 10.

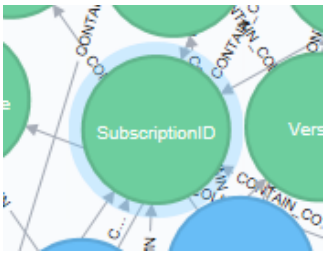


Fig. 10. Example of the graph's node features, depicting columns from a relational database in the RELATIONS-Graph application [own source]

```
1 reference
void cAdd_Relationship_Foreign_Key(GraphClient graphClient, string cName_PK_Table, string cName_FK_Table, string cFK_name)
{
    CypherQuery query = new CypherQuery(
    "MATCH (a:Table),(b:Table) WHERE a.Name = '" + cName_PK_Table + "' AND b.Name = '"
    + cName_FK_Table + "' CREATE (a)-[:" + cFK_name + "]->(b)",
    new Dictionary<string, object>(), CypherResultMode.Set);
    ((IRawGraphClient)graphClient).ExecuteCypher(query);
}
```

Fig. 11. Construction of a Cypher query in text form [own source]

The second approach of creating the graph of relations consists of creating a query as a *string* (sequence of characters), which is then sent to the *CypherQuery* class constructor (see Fig. 11). Such an approach is strongly discouraged, and the documentation compares it to performing SQL queries with the use of the *ADO.NET* technology, without parameters [20]. In this case, an incorrectly constructed query sent to the graph database server will be treated as a false request for execution because the compiler is unable to assess its syntactic correctness. Other threats connected with the second approach are similar to those involved in SQL injection [21]. Despite the potential dangers of poorly constructed queries, the second approach also has some advantages. Creating queries in a text form is extremely flexible. Carefully selected arguments in the *CypherQuery* class constructor enable users to perform queries that both return data and modify the graph structure.

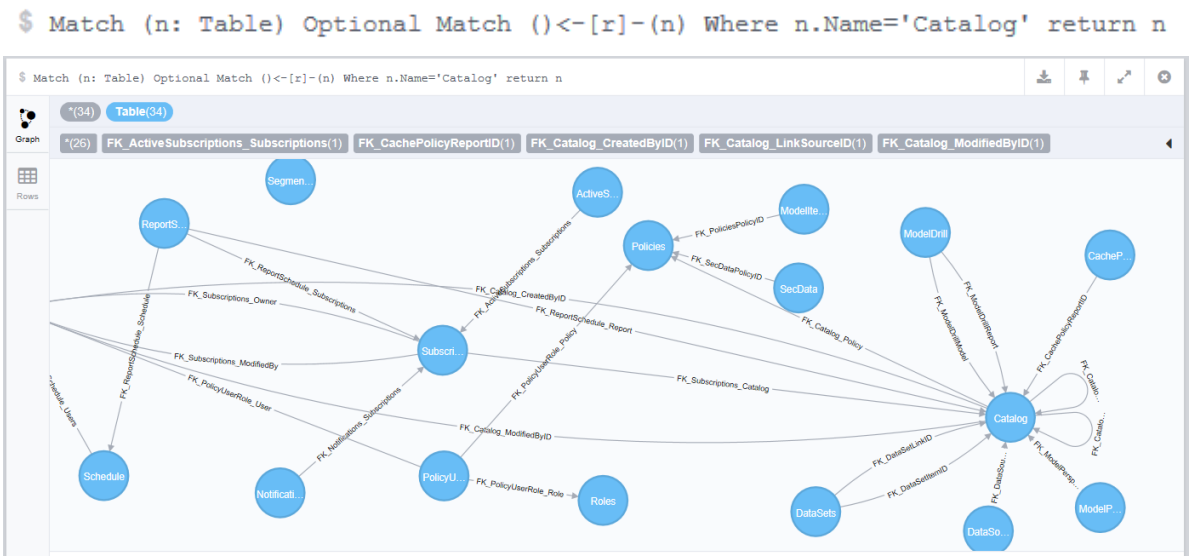


Fig. 12. The effect of searching a graph database for related tables [own source]

The result of the abovementioned queries is a graph that can be viewed in the main part of the RELATIONS-Graph application. To achieve this goal – from the programming perspective – a WebBrowser indicator was used, as supplied by the *Visual Studio Professional 2013* Integrated Development Environment (IDE)AAutorIntegrated Development Environment, IDE. This operation can be performed because a client program of the *Neo4j* graph database is made available via the *HTTP* protocol [22]. The only action required from the user in order to obtain the desired graph is to construct a query directly in the command line of the *Neo4j Community* client program [23]. The construction of this query can be carried out according to two scenarios, the first of which is more difficult and requires good knowledge of the *CYPHER* language, and the query is generally constructed in the command line made available by the client program [24]. The alternative involves the use of a menu in the graphical user interface of the *Neo4j Community* client program, which supports generation of simple queries. The result of a sample query concerning relational structures in the form of graph is shown in Fig. 12.

III. QUERYING THE GRAPH

The RELATIONS-Graph Application allows to generate a graph of links in a database and also to query this graph in order to find relations between given tables and columns. Querying, searching the graph is based on the queries written in the CYPHER language and is realised as Neo4j server-side. Examples of queries are shown in Fig. 13.

a) `Query: MATCH (n:Table) OPTIONAL MATCH ()-[r]->(n) WHERE n.Name= 'Catalog' return n`
Translation: „Show all incoming relationships of table 'Catalog' "
b) `Query: MATCH (n:Table) OPTIONAL MATCH ()<-[r]-(n) WHERE n.Name= 'Catalog' return n`
Translation: „Show all outgoing relationships of table 'Catalog' "

Fig. 13. Examples of CYPHER queries for searching in generated graph of relations [own source]

Querying the generated graph is also possible directly in the internet browser with the web interface of the Neo4j graph database. The user must only know the names of the graph nodes.

IV. SUMMARY

As the tools that make up the MS SQL Server 2014 SP1 package (as well as its earlier versions) have a number of limitations in identifying, searching and presenting the complex structures which make up a relational database, the authors have proposed a new IT solution [25]. The RELATIONS-Graph application relies on technologies associated with graph databases (specifically Neo4j) along with the Visual Studio environment and available libraries. This application enables the user to visualize relational structures in the form of a graph [26]. This, in turn, enables the use of the *Neo4j Community* client program and the *CYPHER* language to perform an extensive graph database querying of the existing links between the tables. Preliminary tests demonstrated the tool's usefulness for MS SQL Server 2014 SP1. It therefore seems desirable to undertake further efforts to make the presented system more versatile, covering a larger group of RDBMSAAutorRDBMS.

References

[1] A. Oppel, H. McGraw, *Data Modeling*, 2009.
[2] Sideris Courseware Corp., *Data Modeling: Logical Database Design*, 2011.
[3] A. Kreigel, Discovering SQL, Wrox, 2011.
[4] I. Robinson, J. Webber, E. Eifrem, *Graph Databases* Second Edition, O'Reilly Media, 2015.
[5] J. Celko, M. Kaufman, *Joe Celko's Complete Guide to NoSQL*, 2013.
[6] P. J. Sadalage, M. Fowler, *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*, Pearson Education, 321826626 2013.
[7] S. Raj, *Neo4j High Performance*, Packt Publishing, 2015.
[8] A. Fowler, *NoSQL For Dummies*, Wiley, 2015.
[9] S. Gupta, *Neo4j Essentials*, Packt Publishing, 2015.
[10] E. Redmon, J. R. Wilson, *Seven Databases in Seven Weeks*, O'Reilly Media, 2012.
[11] S. Tiwari, *Professional NoSQL*, Wrox, 2011.
[12] A. Vucotic, N. Watt, T. Abedrabbo, D. Fox, J. Partner, *Neo4j in Action*, Manning, 2014.
[13] R. Dewson, *Beginning SQL Server for Developers*, 4th Edition, Apress, 2014.
[14] J. Powell, *A Librarian's Guide to Graphs, Data and the Semantic Web*, Chandos Publishing, 2015.
[15] M. Schmalz, C# Database Basics, O'Reilly Media, 2012.
[16] G. Ellis, *Getting Started with SQL Server 2014 Administration*, Packt Publishing, 2014.
[17] G. Vaish, *Getting Started with NoSQL*, Packt Publishing, 2013.
[18] G. Jordan, *Practical Neo4j*, Apress, 2015.
[19] O. Panazarino, *Learning Cypher*, Packt Publishing, 2014.
[20] E. Johnson, J. Jones, *A Developer's Guide to Data Modeling for SQL Server*, Addison-Wesley, 2008.
[21] B. A. Masood-Al.-Faroog, *SQL Server 2014 Development Essentials*, Packt Publishing, 2014.
[22] R. Van Bruggen, *Learning Neo4j*, Packt Publishing, 2014.
[23] M. Lal, *Neo4j Graph Data Modeling*, Packt Publishing, 2015.
[24] A. Goel, *Neo4j Cookbook*, Packt Publishing, 2015.
[25] I. Robinson, J. Webber, E. Eifrem, *Graph Databases*, O'Reilly Media, 2013.
[26] Z. Naboulsi, S. Ford, *Coding Faster: Getting More Productive with Microsoft Visual Studio*, Microsoft Press, 2011.

Przemysław Idziaszek graduated in Computer Sciences and Agri-engineering at the Faculty of Agriculture and Bio-engineering from the Poznan University of Life Sciences (PULS) in 2014. After graduate studies he began further education at the doctoral studies, and he is currently at the third year. His Scientific Supervisor is Prof. dr hab. Wojciech Mueller. PhD thesis is realized at the Department of Applied Informatics, Institute of Biosystems Engineering at PULS. The aim of this work is to check if graph databases can be applied in advisory systems supporting agriculture. His scientific work is focused on: application (web, desktop, web-services) development in Microsoft .NET technologies, database management systems (relational, graph), geospatial data, agriculture production, distributed systems and cloud computing. Since the last general meeting, he has been member of the Board of Polish Society for Information Technology in Agriculture (POLSITA).

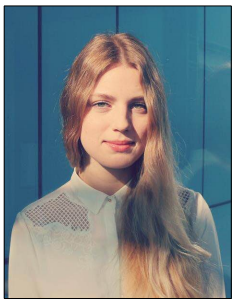
Wojciech Mueller professor at the Department of Applied Informatics of the Institute of Biosystems Engineering of the PULS. Research domain: agri-food engineering, specialization: advanced internet applications supporting analysis, design and management of agri-food systems, in particular: systems analysis and data modeling, business intelligence, cloud computing and advanced technologies for database management systems. He was involved in several research projects as a project leader. He is the member of the Board of Polish Society for Information Technology in Agriculture (POLSITA).

Janina Rudowicz-Nawrocka, senior researcher at the Department of Applied Informatics of the Institute of Biosystems Engineering of the PULS. Research domain: agri-food engineering, development of ICT technologies for agri-food systems and sustainable development, in particular GIS, monitoring. She is involved in several research projects as a researcher (European Projects agrixchange, SmartAgriFood) and also as a project leader (National Project for pastures and meadows aerial monitoring). She is a board member of the Polish Society for ICT in Agriculture, Forest and Food Production (POLSITA) and a member of the Polish Association of Spatial Information (PASI).

Michał Gruszczyński is a PhD student at Institute of Biosystems Engineering. His scientific interests are focused on Objective Oriented Programming, graph databases and data visualization. The main subject of his research is analysis of thermodynamic processes and models in a packed stone bed.

Sebastian Kujawa is employed as Assistant Professor in the Institute of Biosystems Engineering at Poznań University of Life Sciences. In 2003 he graduated in Applied Informatics in Agriculture Engineering from the August Cieszkowski Agricultural University of Poznań. In 2009 he received a PhD degree in the field of agricultural engineering (speciality: applied informatics) from PULS. His scientific activity is focused on applications of modern ICT technologies in solving scientific problems of broadly defined biosystems engineering. In particular, it includes the application of digital image processing and analysis, and also neural modeling, in development of methods for assessment of condition of dynamic biosystems. He is member of the Board of Polish Society for Information Technology in Agriculture (POLSITA).

Karolina Górna is a third year PhD student at the Poznań University of Life Sciences. Her research field includes image analysis and processing, artificial intelligence - especially neural networks. Her PhD thesis is related to use of neural modelling in classification of bovine ovaries USG images.



Kinga Balcerzak is a PhD student at the Faculty of Agriculture and Bioengineering at the Poznań University of Life sciences. She is currently during the third year of her PhD degree education, along with leading classes for the students. Her research interests concern 3D modelling and analysis of the agriculture products based on mathematical methods like finite element mesh generation. She is both programmer and graphic designer.