Storing metagraph model in relational, document-oriented, and graph databases

© Valeriy M. Chernenkiy, © Yuriy E. Gapanyuk, © Yuriy T. Kaganov,
© Ivan V. Dunin, © Maxim A. Lyaskovsky, © Vadim S. Larionov
Bauman Moscow State Technical University,
Moscow, Russia
chernen@bmstu.ru, gapyu@bmstu.ru, kaganov.y.t@bmstu.ru,
johnmoony@yandex.ru, maksim_lya@mail.ru, larionov.vadim@mail.ru

Abstract. This paper proposes an approach for metagraph model storage in databases with different data models. The formal definition of the metagraph data model is given. The approaches for mapping the metagraph model to the flat graph, document-oriented, and relational data models are proposed. The limitations of the RDF model in comparison with the metagraph model are considered. It is shown that the metagraph model addresses RDF limitations in a natural way without emergence loss. The experiments result for storing the metagraph model in different databases are given.

Keywords: metagraph, metavertex, flat graph, graph database, document-oriented database, relational database.

1 Introduction

At present, on the one hand, the domains are becoming more and more complex. Therefore, models based on complex graphs are increasingly used in various fields of science from mathematics and computer science to biology and sociology.

On the other hand, there are currently only graph databases based on flat graph or hypergraph models that are not capable enough of being suitable repositories for complex relations in the domains.

We propose to use a metagraph data model that allows storing more complex relationships than a flat graph or hypergraph data models.

This paper is devoted to methods of storage of the metagraph model based on the flat graph, document-oriented, and relational data models.

We have tried to offer a general approach to store metagraph data in any database with the above-mentioned data model. But at the same time, we conducted experiments on several databases. The results of the experiments are presented in the corresponding section.

2 The description of the metagraph model

In this section, we will describe the metagraph model. This model may be considered as a "logical" model of the metagraph storage.

A metagraph is a kind of complex network model, proposed by A. Basu and R. Blanning [1] and then adapted for information systems description by the present authors [2]. According to [2]:

$$MG = \langle MG^V, MG^{MV}, MG^E \rangle,$$

Proceedings of the XX International Conference "Data Analytics and Management in Data Intensive Domains" (DAMDID/RCDL'2018), Moscow, Russia, October 9-12, 2018

where MG – metagraph; MG^V – set of metagraph vertices; MG^{MV} – set of metagraph metavertices; MG^E – set of metagraph edges.

A metagraph vertex is described by the set of attributes: $v_i = \{atr_k\}, v_i \in MG^V$, where v_i – metagraph vertex; atr_k – attribute.

A metagraph edge is described by the set of attributes, the source and destination vertices and edge direction flag:

$$e_i = \langle v_S, v_E, eo, \{atr_k\} \rangle, e_i \in MG^E, eo = true|false,$$

where e_i – metagraph edge; v_S – source vertex (metavertex) of the edge; v_E – destination vertex (metavertex) of the edge; *eo* – edge direction flag (*eo=true* – directed edge, *eo=false* – undirected edge); atr_k – attribute.

The metagraph fragment:

$$MG_i = \{ev_j\}, ev_j \in (MG^V \cup MG^{MV} \cup MG^E),$$

where MG_i – metagraph fragment; ev_j – an element that belongs to the union of vertices, metavertices, and edges.

The metagraph metavertex:

$$mv_i = \langle \{atr_k\}, MG_j \rangle, mv_i \in MG^{MV},$$

where mv_i – metagraph metavertex belongs to set of metagraph metavertices MG^{MV}; atr_k – attribute, MG_j – metagraph fragment.

Thus, a metavertex in addition to the attributes includes a fragment of the metagraph. The presence of private attributes and connections for a metavertex is a distinguishing feature of a metagraph. It makes the definition of metagraph holonic – a metavertex may include a number of lower level elements and in turn, may be included in a number of higher level elements.

From the general system theory point of view, a metavertex is a special case of the manifestation of the emergence principle, which means that a metavertex with its private attributes and connections becomes a whole that cannot be separated into its component parts. The example of metagraph is shown in Figure 1.

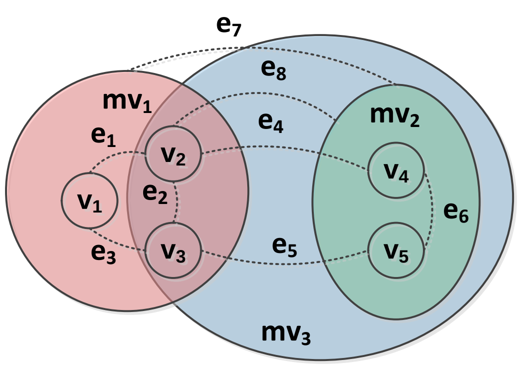

Figure 1 The example of metagraph

This example contains three metavertices: mv$_1$, mv$_2$, and mv$_3$. Metavertex mv$_1$ contains vertices v$_1$, v$_2$, v$_3$ and connecting them edges e$_1$, e$_2$, e$_3$. Metavertex mv$_2$ contains vertices v$_4$, v$_5$ and connecting them edge e$_6$. Edges e$_4$, e$_5$ are examples of edges connecting vertices v$_2$-v$_4$ and v$_3$-v$_5$ respectively and are contained in different metavertices mv$_1$ and mv$_2$. Edge e$_7$ is an example of an edge connecting metavertices mv$_1$ and mv$_2$. Edge e$_8$ is an example of an edge connecting vertex v$_2$ and metavertex mv$_2$. Metavertex mv$_3$ contains metavertex mv$_2$, vertices v$_2$, v$_3$ and edge e$_2$ from metavertex mv$_1$ and also edges e$_4$, e$_5$, e$_8$ showing the holonic nature of the metagraph structure. The Figure 1 shows that the metagraph model allows describing complex data structures and it is the metavertex that allows implementing emergence principle in data structures.

It should be noted that according to [2] the metagraph model also includes more complex elements such as metaedges and metagraph agents. However, they are derived from the considered model elements and do not affect the methods of metagraphs storage in different databases.

3 Mapping the metagraph model to storage models

The logical model described in the previous section is a higher-level model. To store the metagraph model efficiently, we must create mappings from "logical" model to "physical" models used in different databases.

In this section, we will consider metagraph model mappings to the flat graph model, document model, and relational model.

3.1 Mapping metagraph model to the flat graph model

The main idea of this mapping is to flatten the hierarchical metagraph model.

Of course, it is impossible to turn a hierarchical graph model into a flat one directly. The key idea to do this is to use multipartite graphs [3].

Consider there is a flat graph:

$$FG = \langle FG^V, FG^E \rangle,$$

where FG^V – set of graph vertices; FG^E – set of graph edges.

Then a flat graph FG may be unambiguously transformed into bipartite graph BFG:

$$BFG = \langle BFG^{VERT}, BFG^{EDGE} \rangle,$$
$$BFG^{VERT} = \langle FG^{BV}, FG^{BE} \rangle,$$
$$FG^V \leftrightarrow FG^{BV}, FG^E \leftrightarrow FG^{BE},$$

where BFG^{VERT} – set of graph vertices; BFG^{EDGE} – set of graph edges. The set BFG^{VERT} can be divided into two disjoint and independent sets FG^{BV} and FG^{BE} and there are two isomorphisms $FG^V \leftrightarrow FG^{BV}$ and $FG^E \leftrightarrow FG^{BE}$. Thus, we transform the edges of graph FG into subset of vertices of graph BFG. The set BFG^{EDGE} stores the information about relations between vertices and edges in graph FG.

It is important to note that from bipartite graph point of view there is no difference whether original graph FG oriented or not, because edges of the graph FG are represented as vertices and, orientation sign became the property of the new vertex.

From the general system theory point of view, transforming edge into vertex, we consider the relation between entities as a special kind of higher-order entity that includes lower-level entities.

Now we will apply this approach of flattening to metagraphs. In case of metagraph we use not bipartite but tripartite target graph TFG:

$$TFG = \langle TFG^{VERT}, TFG^{EDGE} \rangle,$$
$$TFG^{VERT} = \langle TFG^V, TFG^E, TFG^{MV} \rangle,$$
$$TFG^V \leftrightarrow MG^V, TFG^E \leftrightarrow MG^E, TFG^{MV} \leftrightarrow MG^{MV}.$$

The set TFG^{VERT} can be divided into three disjoint and independent sets TFG^V, TFG^E, TFG^{MV}. There are three isomorphisms between metagraph vertices, metavertices, edges and corresponding subsets of TFG^{VERT}: $TFG^V \leftrightarrow MG^V$, $TFG^E \leftrightarrow MG^E$, $TFG^{MV} \leftrightarrow MG^{MV}$. The set TFG^{EDGE} stores the information about relations between vertices, metavertices, edges in original metagraph.

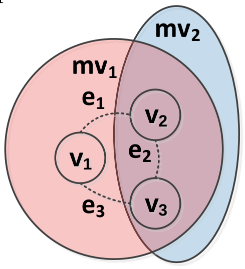

Figure 2 The example of metagraph for flattening

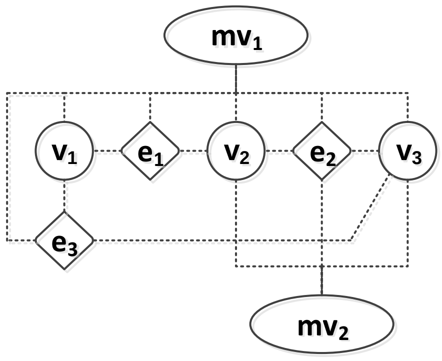

Figure 3 The example of flattened metagraph

Consider the example of flattening metagraph model. The original metagraph is represented in Fig. 2 and corresponding flat graph is represented in Fig. 3. The vertices, metavertices and edges of original metagraph are represented with vertices of different shapes.

From the general system theory point of view, emergent metagraph elements such as vertices, metavertices, edges are transformed into independent vertices of the flat graph.

The proposed mapping may be used for storing metagraph data in graph or hybrid databases such as Neo4j or ArangoDB.

It is important to note that flattening metagraph model does not solve all problems for graph database usage. Consider the example of a query using the Neo4j database query language "Cypher":

```
(n1:Label1)-[rel:TYPE]->(n2:Label2)
```

One can see that used notation is RDF-like and suppose that graph edges are named. But flatten metagraph model does not use named edges because metagraph edges are transformed into vertices.

Thus, query languages of flat graph databases are not suitable for the metagraph model because they blur the semantics of the metagraph model.

3.2 Mapping metagraph model to the document model

From the general system theory point of view, emergent metagraph elements such as vertices, metavertices, edges should be represented as independent entities.

In the previous subsection, we use flat graph vertices for such a representation. But instead of graph vertices, we can also represent independent entities as documents for the document-oriented database. Flat graph edges are represented as relations via id-idrefs between documents.

For the sake of clarity, we use the Prolog-like predicate textual representation. This representation may be easily converted into JSON or XML formats because it is compliant with JSON semantics and contains nested key-value pairs and collections.

The classical Prolog uses the following form of the predicate: $predicate(atom_1, atom_2, \cdots, atom_N)$. We used extended form of predicate where along with atoms predicate can also include key-value pairs and nested predicates: $predicate(atom, \cdots, key = value, \cdots, predicate(\cdots), \cdots)$. The mapping of metagraph model fragments into predicate representation is described in details in [2].

The proposed textual representation may be used for storing metagraph data in a document-oriented database or text or document fields of the relational database using JSON or XML formats.

3.3 Mapping metagraph model to the relational model

Nowadays NoSQL databases are very popular. But traditional relational databases are still the most mature solution and widely used in information systems. Therefore, we also need the relational representation of the metagraph model. There are two ways to store metagraphs in a relational database.

The first way is to use a pure relational schema. In this case, the proposed metagraph model may be directly or with some optimization transformed into the database schema. The tables vertices, metavertices, edges may be used. The Figure 4 contains a graphical representation of such a schema using PostgreSQL database. The table "metavertex" contains the representation of vertices and metavertices. The table "relation" contains the representation of edges.

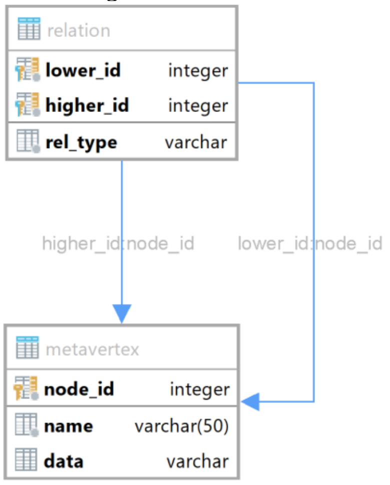

Figure 4 The database schema for pure relational metagraph representation

The second way is to use document-oriented possibilities of a relational database. For example, the latest versions of PostgreSQL database provide such a possibility. The Figure 5 contains a graphical representation of such a schema.

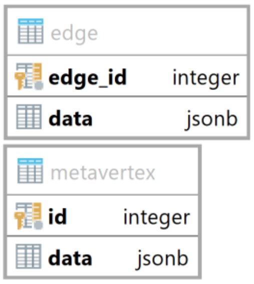

Figure 5 The database schema for document-relational metagraph representation

In this case, vertices, metavertices, and edges are stored as XML or JSON documents in relational tables. The drawback of this approach is id-idrefs storage between documents. In a relational database, we have to do this programmatically which decrease the overall system performance.

4 Why not using RDF model?

Nowadays the semantic web approach for knowledge storage is widely used. In this case, the Resource Description Framework (RDF) is used as the data model, and SPARQL is used as the query language. RDFS (RDF Schema) and OWL (OWL2) are used as ontology definition languages, built on the base of RDF. Using RDFS and OWL, it is possible to express various relationships between ontology elements (class, subclass, equivalent class, etc.) [4]. For RDF persisting and SPARQL processing, special storage systems are used, e.g., Apache Jena.

But unfortunately, the RDF approach has several limitations for complex situation description. In this section, we will consider these limitations according to our paper [5]. The root of limitations is the absence of the emergence principle in the flat graph RDF model.

4.1 The reification limitation

The reification is used to define RDF statements about other RDF statements. According to the RDF Primer [6]: 'the purpose of reification is to record information about when or where statements were made, who made them, or other similar information (this is sometimes referred to as "provenance" information)'. Thus, reification is considered as an auxiliary technique to "log" provenance information about statements.

RDF contains reified triple construction to describe reification in the following form:

```
   StatementID subject predicate object
```

Consider the example of the complex statement: 'James noted that Paul noted at 4 p.m. that John arrived in London'. In the reified triples form, this example may be represented as follows:

```
1.StatementID_1 John arrived_in London
2.StatementID_2 StatementID_1 has_author Paul
3.StatementID_3 StatementID_1 has_time "4p.m."
4.StatementID_4 StatementID_2 has_author James
5.StatementID_5 StatementID_3 has_author James
```

In statements 2 and 3, StatementID_1 is used as the subject. Statements 2 and 3 contain provenance information about the author and time of statement 1. Statements 4 and 5 contain provenance information about the author of statements 2 and 3. The RDF graph form of this example is shown in Fig. 6.

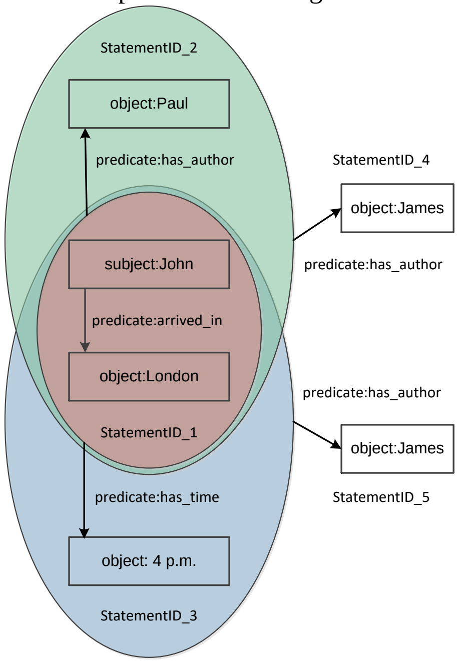

Figure 6 The example of RDF reification

In Fig. 6 statements 1, 2, 3 are highlighted, whereas statements 4 and 5 are not highlighted in order not to confuse visualization of the figure. Fig. 6 shows that a reified triple may be considered as a metavertex but in very restrictive form, containing only one subject, predicate, and object.

The problem shown in this example is emergence loss because of the artificial splitting of the whole situation into a few RDF statements. Statements 4 and 5 are represented by separate RDF statements, but they would more intuitively be represented by a single unit containing the whole situation.

The metagraph approach helps to represent this example more naturally and holistically. From the metagraph point of view, this example contains three nested situations:

- Situation 1. John arrived in London;
- Situation 2. Paul noted at 4 p.m. situation 1;
- Situation 3. James noted situation 2.

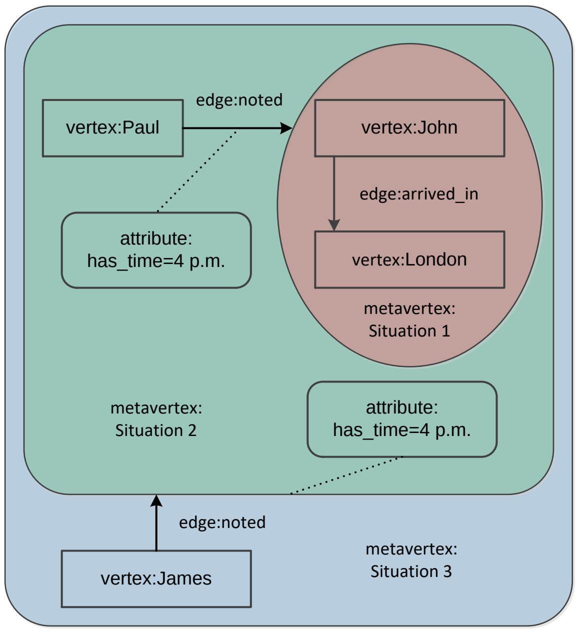

Figure 7 The metagraph representation of RDF reification

Each situation is represented by a metavertex as shown in Fig. 7. Attribute "has_time=4 p.m." may be bound either to edge "noted" or to metavertex "Situation 2" (Fig. 7 shows both cases).

The textual representation of Fig. 7 is shown below:

```
Metavertex(Name=Situation3,
  Vertex(Name=James),
  Metavertex(Name=Situation2,
    Attribute(has_time,"4 p.m."),
    Vertex(Name=Paul),
    Metavertex(Name=Situation1,
      Vertex(Name=John),
      Vertex(Name=London),
      Edge(Name=arrived_in, vS=John, vE=London,
        eo=true)),
    Edge(Name=noted, vS=Paul, vE=Situation1,
eo=true,
      Attribute(has_time,"4 p.m."))),
  Edge(Name=noted, vS=James, vE=Situation2,
eo=true))
```

This considered example shows that the metagraph approach allows representing reification without emergence loss, keeping each nested situation in its own metavertex.

4.2 The N-ary relationship limitation

An N-ary relationship is a situation where a predicate combines several subjects or objects or has nested predicates. Such a situation is a problem from an RDF point of view. To address this problem, the W3C Working Group Note was published [7].

Consider the example of the complex statement: 'John arrived to London at 4 p.m. by train in order to meet his classmates James and Paul'. This is a typical example of an N-ary relationship as shown in Fig. 8. Both problems shown in Fig. 8 cannot be represented by a pure RDF triplet model.

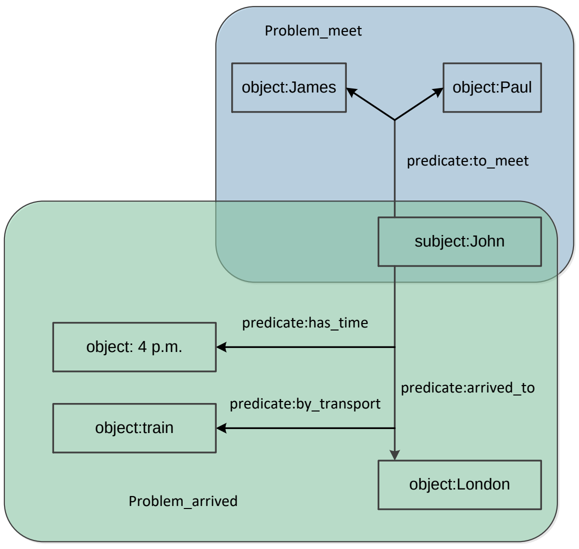

Figure 8 The example of N-ary relationship

The "Problem_arrived" is that the predicate "arrived_to" has nested predicates "has_time" and "by_transport". According to [7] we are adding a supporting subject to "Problem_arrived" representing an instance of a relation.

The "Problem_meet" is that the predicate "to_meet" has two objects "James" and "Paul". According to [7] we have several ways to solve this problem. We may use the list construct of RDF or we may join object "James" and "Paul" into the classmate's group. We do the latter in this example.

The solution is shown in Fig. 9. We have added supporting vertices "Classmates_group" and "Problem_arrived", which are shown in rounded boxes. In predicate "to_meet" the "Classmates_group" is an object while in predicate "includes" it is a subject. In predicate "has_person", "John" is an object while in predicate "to_meet" he is a subject.

Since we do not use reification, this may be represented in the RDF triple form "`subject predicate object`" as follows:

```
1. Problem_arrived has_person John
2. Problem_arrived arrived_to London
3. Problem_arrived by_transport train
4. Problem_arrived has_time "4p.m."
5. John to_meet Classmates_group
6. Classmates_group includes James
7. Classmates_group includes Paul
```

As in the reification example, the problem here is in emergence loss due to the artificial splitting of the situation. The "Problem_arrived" vertex is added not because it describes the situation in a natural way, but because it is required to keep a consistent triplet structure. In a large RDF graph, many supporting vertices may obscure meaningful understanding of the situation.

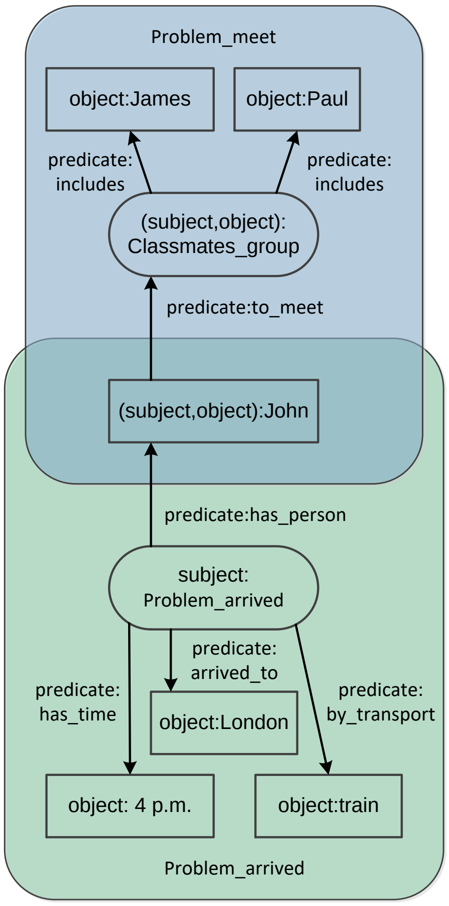

Figure 9 The RDF representation of N-ary relation example

As in the reification example, the metagraph approach helps to represent this example in a more natural and holistic way as shown in Fig. 10.

The "Problem_arrived" is solved by binding attributes "has_time=4 p.m." and "by_transport=train" to the edge "arrived_to". The "Problem_meet" is solved by using metavertex "Classmates_group" which includes vertices "James" and "Paul".

The implicit knowledge about "Classmates_group" living in London may be shown either by the edge "living" or by the inclusion of metavertex "Classmates_group" into metavertex "London" (Fig. 10 shows both cases).

The textual representation of Fig. 10 is shown below:

```
Metavertex(Name=London,
  Metavertex(Name=Classmates_group,
    Vertex(Name=James),
    Vertex(Name=Paul),
    Edge(Name=living, vS=Classmates_group,
vE=London,
      eo=true)))
Vertex(Name=John)
Edge(Name=to_meet, vS=John,
vE=Classmates_group,
  eo=true)
Edge(Name=arrived_to, vS=John, vE=London,
eo=true,
  Attribute(has_time,"4 p.m."),
  Attribute(by_transport, train))
```

This considered example shows that the metagraph approach allows the representation of N-ary relations without emergence loss, keeping each nested situation in its own metavertex.

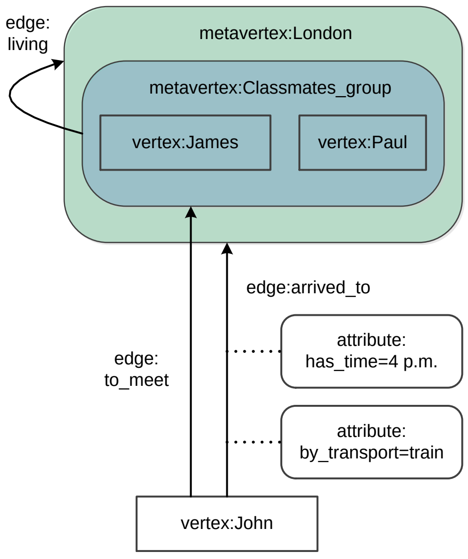

Figure 10 The Metagraph representation of N-ary relation example

Summing up this section, it should be noted that the metagraph model addresses RDF limitations in a natural way without emergence loss. Proposed textual representation of the metagraph allows clear and emergent description of examined problems.

Therefore, despite the prevalence of the RDF model, we consider the development of a storage system for the metagraph model as an important task.

5 The experiments

In this section, we present experiments results for storing the metagraph “logical” model in several databases with different “physical” data models.

It should be noted that these are just entry-level experiments that should help to choose the right data model prototype for the metagraph backend storage.

The experiments were carried out with the following “physical” data models:

- Neo4j – the Neo4j database using flat graph model (according to subsection 3.1);
- ArangoDB(graph) – the ArangoDB database using flat graph model (according to subsection 3.1);
- ArangoDB(doc) – the ArangoDB database using document-oriented model (according to subsection 3.2);
- PostgreSQL(rel) – the PostgreSQL database using pure relational schema (according to subsection 3.3);
- PostgreSQL(doc) – the PostgreSQL database using document-oriented possibilities of relational database (according to subsections 3.2 and 3.3).

The characteristics of test server: Intel Xeon E7-4830 2.2 GHz, 4 Gb RAM, 1 Tb SSD, OS Ubuntu 16.04 (clean installation on a server). Python 3.5 was used for running test scripts. Scripts are connected to Neo4j and ArangoDB via official Python drivers. Queries to these databases were written in query languages (Cypher and AQL respectively) without ORM and executed by Python drivers. However, queries for PostgreSQL were made with SQLAlchemy ORM in order to simplify database manipulations from the python script. In all cases, the database was generated by scripts in csv-format. The database was reloaded from the dump after every test, which modified the state of the database.

Each operation was repeated several times to get the average time of execution.

The experimental dataset consisted of 1 000 000 vertices, randomly connected with 1 000 000 edges. Each vertex of the dataset included one random integer attribute and one random string attribute of fixed length. For read operations (selecting hierarchy), additional ten vertices of fixed structure (100 nested levels) were added to the dataset to get an average time of ten reads.

The numerical axis of the charts contains operation execution time in milliseconds. The less value is better.

The main test results are represented in the following figures and summed up in Table 1. If the best result is approximately the same for several databases, then all these variants are marked in Table 1.

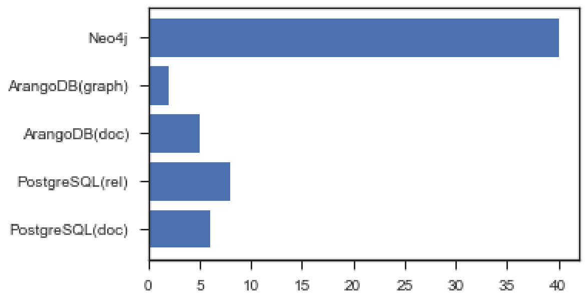

Figure 11 The test results for “Inserting vertex to the existing metavertex”

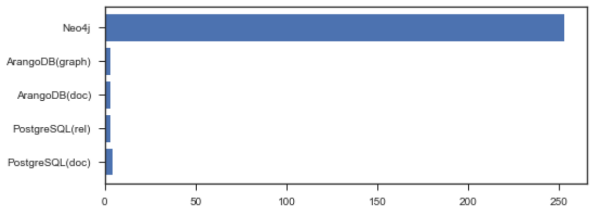

Figure 12 The test results for “Inserting vertex to the metagraph”

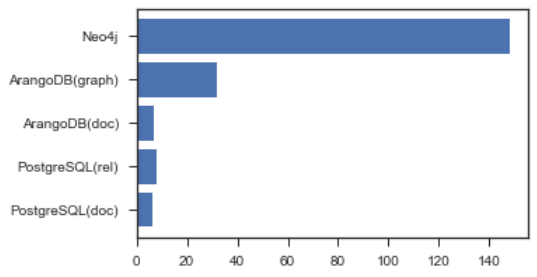

Figure 13 The test results for "Inserting edge to the metagraph"

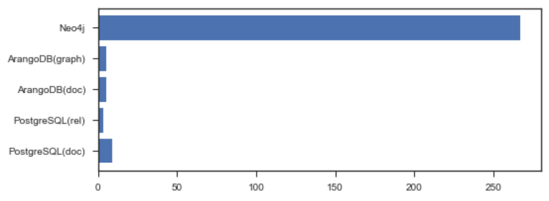

Figure 14 The test results for "Updating existing vertex value"

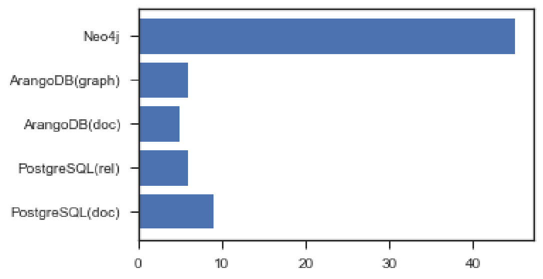

Figure 15 The test results for "Deleting vertex from the existing metavertex"

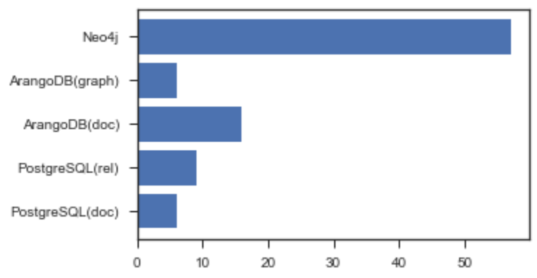

Figure 16 The test results for "Deleting edge from the existing metavertex"

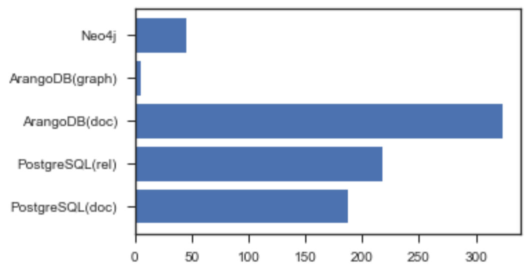

Figure 17 The test results for "Selecting hierarchy of 100 related metavertices"

Table 1 The tests results (test time in milliseconds)

Test case	Neo4j	ArangoDB (graph)	ArangoDB (doc)	PostgreSQL (rel)	PostgreSQL (doc)
Inserting vertex to the existing metavertex	40	**2**	5	8	6
Inserting vertex to the metagraph	253	**3**	**3**	**3**	**4**
Inserting edge to the metagraph	148	32	**7**	**8**	**6**
Updating existing vertex value	267	**5**	**5**	**3**	9
Deleting vertex from the existing metavertex	45	**6**	**5**	**6**	9
Deleting edge from the existing metavertex	57	**6**	16	9	**6**
Selecting hierarchy of 100 related metavertices	45	**5**	323	218	187

Let's make intermediate conclusions on the basis of the considered results of experiments.

It is necessary to recognize the Neo4j implementation as inefficient compared to other cases. But this is not a disadvantage of the graph model itself, because the graph implementation in ArangoDB is quite efficient.

The inserting, updating and deleting operations are very efficient in PostgreSQL (both relational and document-oriented schemas) and ArangoDB (document-oriented schema), but this is not the case for hierarchical selecting which is typical metagraph operation.

The time for hierarchical selecting for graph databases (both Neo4j and ArangoDB) is comparable to the time of other tests while the time for hierarchical selecting for relational and document-oriented databases is several times longer than the time of other tests.

Thus, if the system architect is forced to use a relational or document-oriented database as a metagraph storage backend, then hierarchical selecting queries should be the subject of careful optimization.

Summarizing, we can say that, provided an effective graph database is used, the flat graph model is most suitable for metagraph storage.

6 The related work

Nowadays, there is a tendency to complicate the graph database data model. An example of this tendency is the HypergraphDB [8] database. As the name implies, HypergraphDB uses the hypergraph as a data model. The reasoning capabilities are implemented via integration with TuProlog.

Another interesting project is a GRAKN.AI [9] aimed for AI purpose that explicitly combines graph-based and ontology-based approach for data analysis. The flat graphs and hypergraphs may be used as data model. The Graql query language is used both for data manipulation and reasoning.

The drawbacks of both projects can be attributed to the fact that the most complex data model for them are hypergraphs. It was shown in the paper [2] that the metagraph is a holonic graph model whereas the hypergraph is a near flat graph model that does not fully implement the emergence principle. Therefore, the hypergraph model doesn't fit well for complex data structures description.

7 Conclusions

The models based on complex graphs are increasingly used in various fields of science from mathematics and computer science to biology and sociology.

Nowadays, there is a tendency to complicate the graph database data model in order to support the complexity of the domains.

We propose to use a metagraph data model that allows storing more complex relationships than a hypergraph data model.

The metagraph model may be mapped to the flat graph model, the document model and the relational model. The main idea of this mapping it the flattening metagraph to the flat multipartite graph. Then flat graph may be represented as document model or relational model.

The experiments results show that flat graph model is most suitable for metagraph storage.

In the future, it is planned to develop a metagraph data manipulation language and design a stable version of the metagraph storage based on a flat graph database.

References

[1] Basu, A., Blanning, R. Metagraphs and their applications. Springer, New York (2007)

[2] Chernenkiy, V., Gapanyuk, Yu., Revunkov, G., Kaganov, Yu., Fedorenko, Yu., Minakova, S. Using metagraph approach for complex domains description. In: Proceedings of the XIX International Conference on Data Analytics and Management in Data Intensive Domains (DAMDID/RCDL 2017), Moscow, Russia, pp. 342-349 (2017)

[3] Chartrand, G., Zhang, P. Chromatic Graph Theory. CRC Press, New York (2008)

[4] Allemang, D., Hendler, J. Semantic Web for the working ontologist: effective modeling in RDFS and OWL – 2nd ed. Elsevier, Amsterdam (2011)

[5] Chernenkiy, V., Gapanyuk, Yu., Nardid, A., Skvortsova, M., Gushcha, A., Fedorenko, Yu., Picking, R. Using the metagraph approach for addressing RDF knowledge representation limitations. In: Proceedings of Internet Technologies and Applications (ITA'2017), Wrexham, United Kingdom, pp. 47-52 (2017)

[6] RDF Primer. W3C Recommendation 10 February 2004. https://www.w3.org/TR/2004/REC-rdf-primer-20040210/

[7] Defining N-ary Relations on the Semantic Web. W3C Working Group Note 12 April 2006. http://www.w3.org/TR/swbp-n-aryRelations/

[8] HyperGraphDB website. http://hypergraphdb.org/

[9] GRAKN.AI website. https://grakn.ai/