The Principles and the Conceptual Architecture of the Metagraph Storage System

Valeriy M. Chernenkiy, Yuriy E. Gapanyuk (✉), Georgiy I. Revunkov,
Ark M. Andreev, Yuriy T. Kaganov, Ivan V. Dunin, Maxim A. Lyaskovsky

Informatics and Control Systems Dept., Bauman Moscow State Technical University,
Baumanskaya 2-ya, 5, postcode 105005, Moscow, Russia
chernen@bmstu.ru, gapyu@bmstu.ru, revunkov@bmstu.ru,
arkandreev@gmail.com, kaganov.y.t@bmstu.ru, johnmoony@yandex.ru,
maksim_lya@mail.ru

Abstract. This paper discusses an approach for active metagraph model storage. The formal definition of the metagraph data model is proposed. The example of data metagraph model is given. The formal definition of the metagraph function and rule agents are discussed. The example of a metagraph rule agent is given. It is shown that the distinguishing feature of the metagraph agent is its homoiconicity which means that it can be a data structure for itself. Thus, the metagraph agent can change both data metagraph fragments and the structure of other metagraph agents. The definition of active metagraph is given. The possible states of active metagraph elements and transitions between them are discussed. The conceptual architecture of the metagraph storage system based on active metagraph is proposed. The approaches for mapping the metagraph model to the flat graph, document-oriented, and relational data models are proposed. The experiments result for storing the metagraph model in different databases are given. It is shown that the flat graph model is most suitable for metagraph storage.

Keywords: Metagraph, Metavertex, Metagraph agent, Metagraph Function Agent, Metagraph Rule Agent, Active Metagraph, Flat Graph, Graph Database, Document-oriented Database, Relational Database.

1 Introduction

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

The 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 mentioned above data model. But at the same time, we conducted experiments on several databases.

The results of the experiments are presented in the corresponding section.

This paper is an extended version of our paper [1]. Compared to paper [1], sections with new materials "The Active Metagraph and Principles of its Storage" and "The Conceptual Architecture of the Metagraph Storage System" have been added to this article. Section "The Metagraph Agent and its Homoiconicity" is not completely new material, but has been added for reasons of clarity. The discussion about RDF model removed from this version of the paper.

The paper [1] addressed the storage of data metagraphs. This paper offers a holistic view of the metagraph storage, designed to store both metagraph data and metagraph agents. For this, an approach based on active metagraphs is used.

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 [2] and then adapted for information systems description by the present authors [1]. According to [1]:

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

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 distin-

guishing 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 Fig. 1.

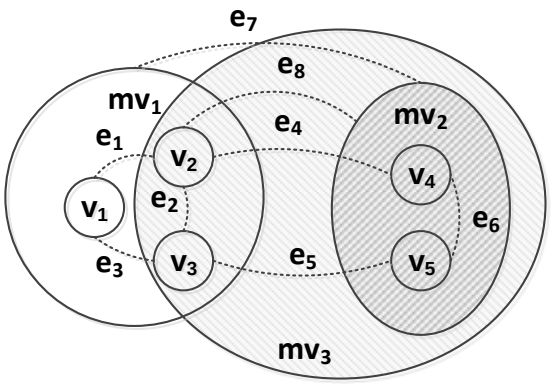

Fig. 1. The example of metagraph

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. Fig. 1 shows that the metagraph model allows describing complex data structures and it is the metavertex that allows implementing emergence principle in data structures.

3 The Metagraph Agent and its Homoiconicity

The metagraph model is aimed for data and knowledge description. But it is not aimed for data transformation. To solve this issue the metagraph agent (ag^{MG}) aimed for data transformation was proposed in our paper [3].

There are two kinds of metagraph agents: the metagraph function agent (ag^F) and the metagraph rule agent (ag^R). Thus $ag^{MG} = (ag^F|ag^R)$.

The metagraph function agent serves as a function with input and output parameter in the form of metagraph:

$$ag^F = \langle MG_{IN}, MG_{OUT}, AST \rangle,$$

where ag^F – metagraph function agent; MG_{IN} – input parameter metagraph; MG_{OUT} – output parameter metagraph; AST – abstract syntax tree of metagraph function agent in form of metagraph.

The metagraph rule agent is rule-based:

$$ag^R = \langle MG, R, AG^{ST} \rangle, \mathrm{R} = \{r_i\}, r_i : MG_j \to OP^{MG},$$

where ag^R – metagraph rule agent; MG – working metagraph, a metagraph on the basis of which the rules of the agent are performed; R – set of rules r_i; AG^{ST} – start condition (metagraph fragment for start rule check or start rule); MG_j – a metagraph fragment on the basis of which the rule is performed; OP^{MG} – a set of actions performed on metagraph.

The antecedent of the rule is a condition over metagraph fragment, the consequent of the rule is a set of actions performed on metagraph. Rules can be divided into open and closed.

The consequent of the open rule is not permitted to change metagraph fragment occurring in rule antecedent. In this case, the input and output metagraph fragments may be separated. The open rule is similar to the template that generates the output metagraph based on the input metagraph.

The consequent of the closed rule is permitted to change metagraph fragment occurring in rule antecedent. The metagraph fragment changing in rule consequent cause to trigger the antecedents of other rules bound to the same metagraph fragment. But incorrectly designed closed rules system can cause to an infinite loop of metagraph rule agent.

Thus, metagraph rule agent can generate the output metagraph based on the input metagraph (using open rules) or can modify the single metagraph (using closed rules).

The example of metagraph rule agent is shown in Fig. 2. The metagraph rule agent "metagraph rule agent 1" is represented as metagraph metavertex. According to the definition it is bound to the working metagraph MG_1 – a metagraph on the basis of which the rules of the agent are performed. This binding is shown with edge e_4.

The metagraph rule agent description contains inner metavertices corresponds to agent rules (rule 1 … rule N). Each rule metavertex contains antecedent and consequent inner vertices. In given example mv_2 metavertex bound with antecedent which is shown with edge e_2 and mv_3 metavertex bound with consequent which is shown with edge e_3. Antecedent conditions and consequent actions are defined in the form of attributes bound to antecedent and consequent corresponding vertices.

The start condition is given in the form of attribute "start=true". If the start condition is defined as a start metagraph fragment, then the edge bound start metagraph fragment to agent metavertex (edge e_1 in the given example) is annotated with the attribute "start=true". If the start condition is defined as a start rule, then the rule metavertex is annotated with attribute "start=true" (rule 1 in the given example). Fig. 2 shows both cases corresponding to the start metagraph fragment and to the start rule.

The distinguishing feature of the metagraph agent is its homoiconicity which means that it can be a data structure for itself. This is due to the fact that according to definition metagraph agent may be represented as a set of metagraph fragments, and this set can be combined in a single metagraph. Thus, the metagraph agent can change the structure of other metagraph agents.

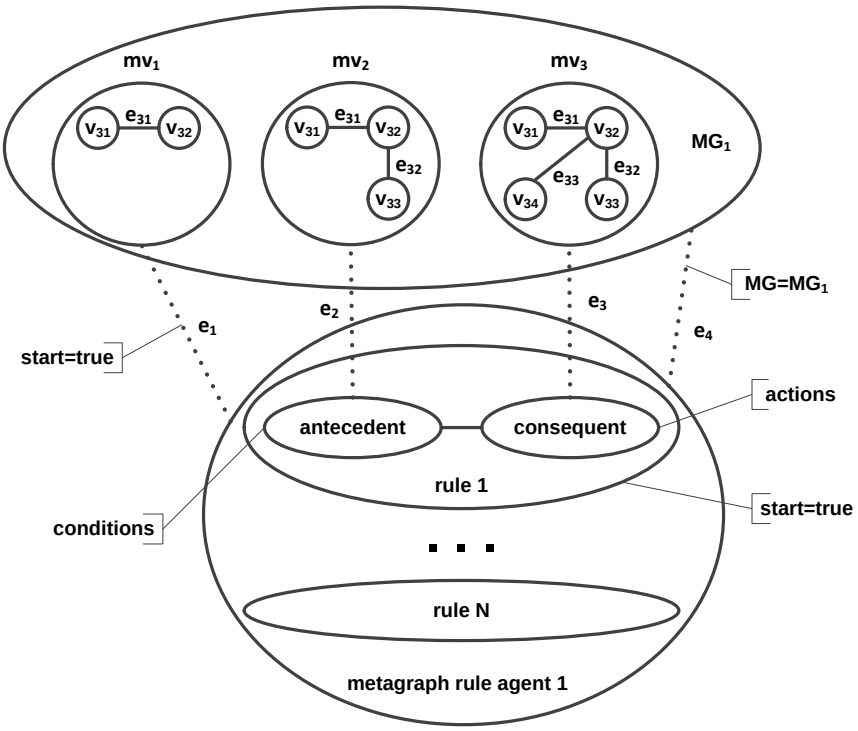

Fig. 2. Example of metagraph rule agent

It should also be noted that efficient pattern matching algorithms used in production systems often use a graph representation of production rules. A well-known example of such an algorithm is RETE. In the case of metagraph approach, the rules of the metagraph agent may be transformed into RETE-network (Alpha and Beta networks) using the higher-level metagraph agent.

4 The Active Metagraph and Principles of its Storage

In order to combine the data metagraph model and metagraph agent model we propose the concept of "active metagraph":

$$MG^{ACTIVE} = \langle MG^{D}, AG^{MG} \rangle, AG^{MG} = \{ag_i^{MG}\},$$

where MG^{ACTIVE} – an active metagraph; MG^{D} – data metagraph; AG^{MG} – set of metagraph agents ag_i^{MG}, attached to the data metagraph.

Thus, active metagraph allows to combine data and processing tools for the metagraph approach.

Similar structures are often used in computer science. As an example, we can consider a class of object-oriented programming language, which contains data and methods of their processing. Another example is a relational DBMS table with an associated set of triggers for processing table entries.

The main difference between an active metagraph and a single metagraph agent is that an active metagraph contains a set of metagraph agents that can use both closed and open rules. For example, one agent may change the structure of active metagraph using closed rules while the other may send metagraph data another active metagraph using open rules. Agents work independently and can be started and stopped without affecting each other.

The possible states of active metagraph elements and transitions between them are represented in Fig. 3.

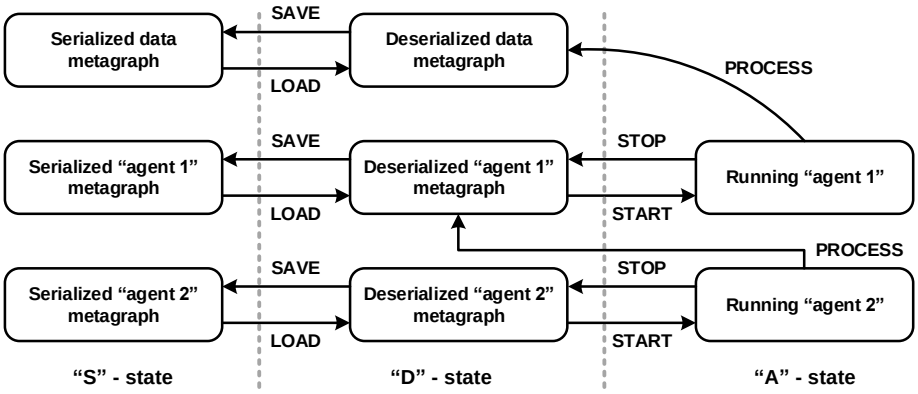

Fig. 3. Possible states of elements of the active metagraph and transitions between them

There are three possible states for active metagraph elements:

1. "S-state" is "serialized" or "stored" state. The serialized active metagraph elements are saved into the store and not ready for processing.
2. "D-state" is "deserialized" state. In this state data metagraph or metagraph agent are ready for processing as passive data structures.
3. "A-state" is "active" state. This state is possible only for metagraph agents. In active state, the metagraph agent may process any metagraph data in "D-state".

It should be noted that "S-state" does not depend on the storage "physical" data model, although different storage models are discussed in the following sections.

Consider the transitions between states. The transitions correspond to the possible operations that can be performed on elements in these states:

- SAVE – serialize metagraph data (which can be data metagraph or agent representation) from the “D-state” to the “S-state”.
- LOAD – deserialize metagraph data from the “S-state” to the “D-state” (which is the reverse action for SAVE operation).
- START – starting the execution of metagraph agent on the basis of metagraph agent data representation, i.e. transferring agent from the “D-state” to the “A-state”.
- STOP – stopping the metagraph agent (which is the reverse action for START operation).
- PROCESS – any metagraph agent in “A-state” may process any metagraph data in “D-state” (which can be data metagraph or agent representation).

Figure 3 shows an example with data metagraph and two metagraph agents. The “agent 1” changes the structure of data metagraph while “agent 2” changes the structure of “agent 1” using PROCESS operation.

Whereas several active metagraphs may share data metagraph fragments, therefore the PROCESS operation may be used for data-driven communication between agents.

It should also be noted that the active metagraph is a hierarchical structure because the hierarchical structure is the data metagraph included in the definition of the active metagraph.

5 The Conceptual Architecture of the Metagraph Storage System

The conceptual architecture of the metagraph storage system based on active metagraph is represented in Fig. 4.

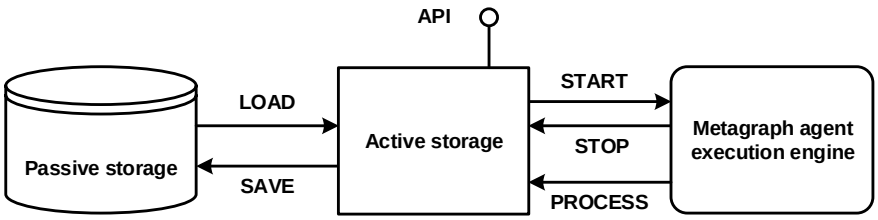

Fig. 4. The conceptual architecture of the metagraph storage system

The passive storage is aimed to store serialized metagraph data that cannot be processed directly. It corresponds to the “S-state” of the active metagraph.

The active storage is aimed to deals with ready for processing metagraph data. It corresponds to the “D-state” of the active metagraph. Let us consider in more detail the features of active storage:

- First of all, active storage is an API for metagraph data processing.

- Active storage makes it possible to process the metagraph data programmatically, which is the basis for the PROCESS operation.
- The metagraph representation of agent stored in active storage may be transformed into an executable format and send to the metagraph agent execution engine (START operation).

Several ways to implement active storage can be suggested:

1. Active storage may be implemented on the basis of the in-memory database. This case has an advantage in the processing speed of metagraph data, but also has a limit on the amount of in-memory data processed. In this case LOAD and SAVE are the physical operations for reading/writing metagraph data from passive storage into the in-memory database.
2. Active storage can be implemented as an add-on over the passive storage without using the intermediate database. In this case, active storage API calls will be translated into the corresponding operations on the passive storage. The LOAD and SAVE operations mean the passive storage reading and writing according to API calls.

The detailed implementation of the active storage is the subject of further research.

The metagraph agent execution engine is aimed to run metagraph agents. It corresponds to the "A-state" of the active metagraph. The running metagraph agent may process any metagraph data in active storage (PROCESS operation). The execution of agent may be stopped, and metagraph representation of agent is saved to the active storage (STOP operation).

The proposed conceptual architecture deals with the homoiconic nature of metagraph agents. Agents may be processed as metagraph data structures and executed as programs. Thus, both data metagraph and metagraph agents may be stored in active and passive storages.

It should be noted that the proposed architecture does not depend on the passive storage data model. The experiments with different data models are discussed in the following sections.

6 Mapping the Metagraph Model to Storage Models

The logical models described in the previous sections are higher-level models. To store the data metagraph or metagraph agent representation efficiently, we must create mappings from "logical" model to "physical" models used in different databases.

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

6.1 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 [4].

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 a 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 vertices entities.

Now we will apply this approach of flattening to metagraphs. In the 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.

Consider the example of flattening metagraph model represented in Fig. 5. 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)
```

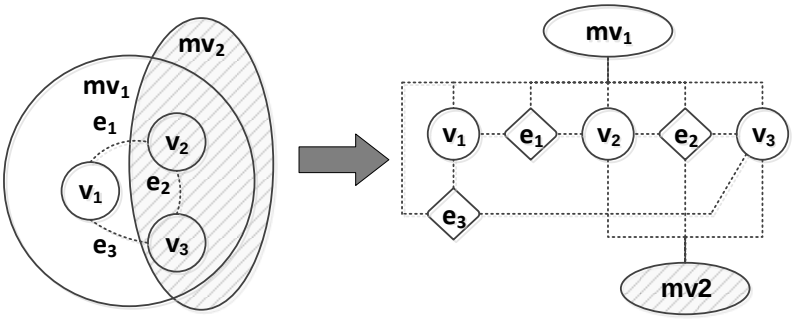

Fig. 5. The example of metagraph for flattening (shown on the left) and the example of flattened metagraph (shown on the right)

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.

6.2 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 an 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 [3].

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.

6.3 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. Fig. 6 contains a graphical representation of such a schema using a PostgreSQL database. The table "metavertex" contains the representation of vertices and metavertices. The table "relation" contains the representation of edges.

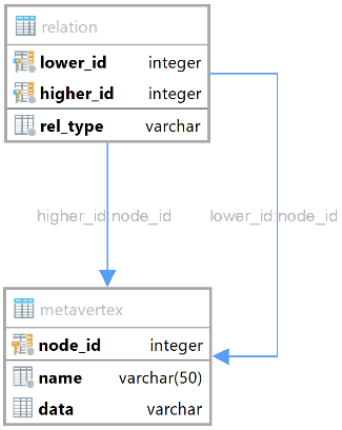

Fig. 6. The database schema for pure relational metagraph representation

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 Fig. 7 contains a graphical representation of such a schema.

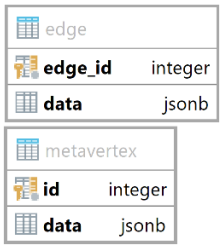

Fig. 7. The database schema for document-relational metagraph representation

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.

Figure 7 shows an example where id-idrefs are stored inside a binary JSON “data” field. In the case of a relational database, we have to parse data fields and process id-idrefs links programmatically which decrease the overall system performance.

7 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 storage system.

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

- Neo4j – the Neo4j database using flat graph model (according to subsection 6.1);
- ArangoDB(graph) – the ArangoDB database using flat graph model (according to subsection 6.1);
- ArangoDB(doc) – the ArangoDB database using document-oriented model (according to subsection 6.2);
- PostgreSQL(rel) – the PostgreSQL database using pure relational schema (according to subsection 6.3);
- PostgreSQL(doc) – the PostgreSQL database using document-oriented possibilities of relational database (according to subsections 6.2 and 6.3).

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 10 reads.

The results of tests are represented in the Table 1. This is the test time in milliseconds; the less value is better. The best results are marked in bold. If the best result is approximately the same for several databases, then all these cases are marked in Table 1.

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 flat 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 the metagraph passive storage based on a relational or document-oriented database, 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.

Table 1. The results of tests (test time in milliseconds, the less value is better).

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
The number of best results	**0/7**	**6/7**	**4/7**	**4/7**	**3/7**

8 The Related Work

Nowadays, there is a tendency to complicate the graph database data model. An example of this tendency is the HypergraphDB [5] 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 [6] 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 a data model. The Graql query language is used both for data manipulation and reasoning.

It was shown in our paper [7] 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.

But in fact, HypergraphDB and GRAKN.AI use a hierarchical hypergraph model which is suitable for complex networks description.

Nowadays the semantic web approach for knowledge representation 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.) [8]. 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 which are considered in details in [1]. The root of limitations is the absence of the emergence principle in the flat graph RDF model. The metagraph model addresses RDF limitations in a natural way without emergence loss. Therefore, despite the prevalence of the RDF model, we consider the development of a storage system for the metagraph model as an important task.

9 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 and RDF model.

The metagraph agents are aimed for metagraph data transformation. The distinguishing feature of the metagraph agent is its homoiconicity. The metagraph agent can change the structure of other metagraph agents.

The active metagraph is aimed to combine the data metagraph model and the metagraph agent model.

The proposed conceptual architecture deals with the homoiconic nature of metagraph agents. Agents may be processed as metagraph data structures and executed as programs. Thus, both data metagraph and metagraph agents may be stored in active and passive storages.

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 the flat graph may be represented as a 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.

10 References

1. Valeriy Chernenkiy, Yuriy Gapanyuk, Yuriy Kaganov, Ivan Dunin, Maxim Lyaskovsky, Vadim Larionov: Storing Metagraph Model in Relational, Document-Oriented, and Graph Databases. In: Proceedings of the XX International Conference on Data Analytics and Management in Data Intensive Domains (DAMDID/RCDL 2018), Moscow, Russia, October 9-12, 2018, pp. 82–89 (2018), http://ceur-ws.org/Vol-2277/paper17.pdf
2. Basu, A., Blanning, R.: Metagraphs and their applications. Integrated series in information systems, vol. 15. Springer, New York (2007)
3. Chernenkiy, V.M., Gapanyuk, Y.E., Nardid, A.N., Gushcha, A.V., Fedorenko, Y.S.: The Hybrid Multidimensional-Ontological Data Model Based on Metagraph Approach. In: Petrenko, A.K., Voronkov, A. (eds.) Perspectives of systems informatics. 11th International Andrei P. Ershov Informatics Conference, PSI 2017, Moscow, Russia, June 27-29, 2017, Revised Selected Papers / edited by Alexander K. Petrenko, Andrei Voronkov, vol. 10742. Lecture notes in computer science, 0302-9743, vol. 10742, pp. 72–87. Springer (2018). doi: 10.1007/978-3-319-74313-4_6
4. Chartrand, G., Zhang, P.: Chromatic graph theory. Discrete mathematics and its applications. Chapman & Hall/CRC, Boca Raton (2009)
5. HyperGraphDB website. http://hypergraphdb.org/
6. GRAKN.AI website. https://grakn.ai/
7. Valeriy Chernenkiy, Yuriy Gapanyuk, Georgiy Revunkov, Yuriy Kaganov, Yuriy Fedorenko, Svetlana Minakova: 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, October 9-13, 2017, pp. 342–349 (2017), http://ceur-ws.org/Vol-2022/paper52.pdf
8. Allemang, D., Hendler, J.A.: Semantic Web for the working ontologist. Effective modeling in RDFS and OWL, 2nd edn. Morgan Kaufmann/Elsevier (2011)