Using Metagraph Approach for the Big Data Based Graph Processing

Valeriy M. Chernenkiy
doctor of technical sciences, professor
chernen@bmstu.ru

Ivan V. Dunin,
postgraduate student
johnmoony@yandex.ru

Yuriy E. Gapanyuk
candidate of technical sciences, associate professor
gapyu@bmstu.ru

Bauman Moscow State Technical University,
Baumanskaya 2-ya, 5, postcode 105005, Moscow, Russia

Abstract: The paper discusses the approach to solving the problem of processing metagraphs using Big Data technology. A brief review of the Big Data based graph processing is given. The formal definition of the metagraph model and its comparison with hypergraph and hypernetwork models is discussed. The metagraph textual representation is described. The principles of the metagraph storage model based on the multipartite graphs are discussed. The generalized structure of the metagraph Big Data processing engine is proposed.

Keywords: Big Data, Big Graph, Flat Graph, Hypergraph, Hypernetwork, Metagraph, Metavertex, Multipartite graph.

1 Introduction

Nowadays, Big Data frameworks are widely used for processing information in various domains, from modern industrial enterprise [1] to social networks analysis [2]. Graph data is one of the most common types of data in modern applications. In can be found both in explicit forms (like social networks, computer networking) and implicit forms (language processing, collaborative filtering, computer vision). The problem of processing Big Graphs can be solved in various ways, for example, by creating specialized high-performance hardware for processing graphs [3, 4]. However, the most common way for Big Graphs processing is to use specialized Big Data frameworks for Big Graphs processing.

The dominant graph model in such frameworks is usually a flat graph model or property graph model (which is, in fact, the multigraph model). However, the flat graph model is not flexible enough and may not be a convenient solution for modeling complex data domains with hierarchical relationships. For example, the assembly sequence problem of complex technical products requires not a flat graph but a hypergraph model [5].

One of the existing extensions of the traditional graph model is the metagraph model. In this article, we describe the metagraph model and consider possibilities of the implementation of this model with the current state of Big Data graph processing technologies.

The article is organized as follows. A brief review of the Big Data based graph processing is given. The metagraph model is formally defined and compared with hypergraph and hypernetwork models. The metagraph textual representation and the principles of the metagraph storage model are discussed. The generalized structure of the metagraph Big Data processing engine is proposed.

2 The Brief Review of the Big Data Based Graph Processing

At present, probably the most widespread technology in Big Data processing is MapReduce algorithms, which are used in various Hadoop ecosystem-based solutions. The core principle of these algorithms is to store data separated into multiple partitions (usually over multiple physical machines), perform computations individually on each partition (“map” phase), later combining them on the “reduce” phase. MapReduce solutions perform well when used for aggregate operations on array-like data, but have some severe limitations, especially when dealing with graphs.

Graph processing problems tend to have low computational locality, i.e., graph processing algorithms (like PageRank, graph clustering, triangle counting, etc.) often require dealing with many non-locally stored nodes simultaneously. There are some ways to increase locality by user-defined heuristics (like storing vertices with the same domain name on the same machine in case of web graph). However, in general, graph algorithms require the processing of remotely-stored data or even processing graph as a whole, which in the case of MapReduce creates heavy computational and network workload. Secondly, MapReduce is a functional approach, thus performing multiple operations (which is often needed in graph algorithms) requires a series of MapReduce jobs with passing the entire graph state between them.

In 2010 Google introduced Pregel architecture [6], which addresses these limitations. Pregel is based on BSP (batch synchronous parallel) model and suggests a vertex-oriented approach instead of basic graph-oriented. Graph processing is performed on the vertex level. In this model, the graph is split into multiple partitions assigned to distributed workers by master-worker. Computation occurs in so-called “supersteps.” During one superstep each worker performs compute-function on each vertex of its subgraph. Vertex computation includes processing of the incoming messages, changing the vertex value (if necessary), and emitting messages over outgoing edges. After each vertex is computed, the next superstep begins. Messages sent by vertex during computation will be received only at the following step. On initial iteration, compute-function is called for all vertices. Later they can declare themselves inactive (“vote halt”). The algorithm converges when all vertices become inactive, and there are no more incoming messages.

Pregel architecture is adopted by multiple products. Apache Spark GraphX [7] is an extension of Apache Spark resilient distributed datasets (RDD) API, which allows performing parallel in-memory computations over a cluster. GraphX includes Pregel implementation with the batch synchronous model. GraphX is built upon Spark and unifies Spark data parallelism with graph parallelism of the Pregel model, allowing to create a graph from distributed data. Vertices and edges of the graph are stored as separated RDDs, which can be co-located and co-partitioned for better performance. Unlike the default Pregel model, in GraphX, messages can access attributes of both source and destination vertices. GraphX proposes a single system that supports the entire pipeline of graph analytics, including data preprocessing, graph processing, and post data analysis.

Apache Giraph [8] is an open-source implementation of Pregel architecture used by Facebook. It is a similar project to GraphX but outperforms it according to experiments conducted by Facebook engineers [9], though it is less flexible in terms of supported data sources. Apache Giraph offers some extensions over the basic Pregel model, like sharded aggregators, which allow to perform aggregate operations on worker nodes and avoid the master bottleneck. Giraph also allows executing different computational functions on different supersteps (controlled by the master worker).

GraphLab [10] (currently incorporates another project PowerGraph [11]) is a graph processing framework also based on a vertex-oriented BSP model. Unlike GraphX and Giraph, Grahlab uses GSA (Gather-Sum-Apply) model. In this model, vertices can access (pull) neighbors data during the “gather” stage without receiving messages from them. GraphLab processing can run either synchronously with superstep barriers like in Pregel-styles system, or asynchronously. Asynchronous mode allows reducing messaging because neighbor nodes can fetch adjacent inactive node data, and avoid superstep barriers with distributed locking. However, the asynchronous mode is more prone to inconsistency, and distributed locking itself requires network exchange. Asynchronous mode is also less deterministic and fault-tolerant and is reported to be more challenging to debug.

Among graph-processing tools, it is worth mentioning Apache Flink Gelly [12], built on top of streaming framework Apache Flink. Unlike the products above, it is not a batch-processing framework. It is also a vertex-centric platform. It allows performing neighbor aggregation operations with accessing neighbors of each vertex (incoming, outcoming, or both). Flink uses its iterative streaming nature to implement superstep iteration process with messaging and vertex-oriented update operations. Flink Gelly also supports Gather-Sum-Apply model. Gather, sum, and apply are user-defined functions wrapped in the map, reduce, and join operators, respectively. In each superstep, the active vertices are joined with the edges in order to create neighborhoods for each vertex. The gather function is then applied to the neighborhood values via a map function. Finally, the outcome of the sum phase is joined with the current vertex values.

It is essential to point out that Hadoop and Spark-based systems are designed for mostly read-only analytical processing. For mutable data, it may be beneficial to use OLTP-database like Neo4j, which can be integrated with graph processing framework (Spark and GraphX natively support connectors to Neo4j). In this case, the graph processing framework would serve as external compute solution for data stored in the database.

Pregel architecture solves most problems of MapReduce, has its limitations. Firstly, the requirement of message passing, even though it is beneficial compared to whole graph shuffling, can create heavy network workload and may become the system’s bottleneck. Secondly, supersteps of the algorithm occur synchronously. The next superstep comes only when the previous is finished. It eliminates races and deadlock problems, but if some vertex or subgraph computation takes a large amount of time (which may be the case for a complex graph structure), some workers may stay idle for a long time. Finally, high degree nodes with many incoming edges are likely to receive many messages, which leads to increasing computation and network workload for these nodes, at the same time increasing idle time of other nodes [13].

Apart from the mentioned technical limitations of existing graph processing solutions, it is essential to point out that those solutions have no natural way of processing more complex graph data. Current graph processors are designed for handling conventional graphs with binary edges. In many domain areas, we can find more complex relations between data than simple peer-to-peer connections. One possible extension of the flat graph model is hypergraph. Now hypergraphs are mainly considered as auxiliary structures for indexing, partitioning [14], etc. The only well-known attempt to implement a hypergraph processing framework is the HyperX framework for the Spark platform [15]. However, gradually, the community begins to develop the idea that Big Data frameworks should work with more complex graph structures [16]. As a complex graph model, we propose to use a metagraph data model, which is discussed in detail in the following section.

3 The Metagraph Model and its Comparison to Other Complex Network Models

In this section, we will formally define the metagraph model and compare it with other well-known complex network models.

According to [17]: "a complex network is a graph (network) with non-trivial topological features – features that do not occur in simple networks such as lattices or random graphs but often occur in graphs modeling of real systems." The terms "complex network" and "complex graph" are often used synonymously. According to [18]: "the term 'complex network,' or simply 'network,' often refers to real systems while the term 'graph' is generally considered as the mathematical representation of a network." We using these terms also synonymously.

3.1 The Metagraph Model Definitions

Thus, the metagraph is a kind of complex network (or complex graph) model, proposed by A. Basu and R. Blanning in their book [19] and then adapted for information systems description in our papers [20, 21]:

$$MG = \langle V, MV, E, ME \rangle, \tag{1}$$

where MG – metagraph; V – set of metagraph vertices; MV – set of metagraph metavertices; E – set of metagraph edges; ME – set of metagraph metaedges.

Metaedge is an optional element of the metagraph model aimed for process description. Since we are talking about a data model in this article, this component is not considered.

Metagraph vertex is described by a set of attributes:

$$v_i = \{atr_k\}, v_i \in V, \tag{2}$$

where v_i – metagraph vertex; atr_k – attribute.

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

$$e_i = \langle v_S, v_E, eo, \{atr_k\} \rangle, e_i \in E, eo = true \mid false, \tag{3}$$

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:

$$MG_i = \{ev_j\}, ev_j \in (V \cup E \cup MV), \tag{4}$$

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

The metagraph metavertex:

$$mv_i = \langle \{atr_k\}, MG_j \rangle, mv_i \in MV, \tag{5}$$

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

Thus, metavertex, in addition to the attributes, includes a fragment of the metagraph. The presence of private attributes and connections for metavertex is a distinguishing feature of the metagraph. It makes the definition of metagraph holonic – 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 particular 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.

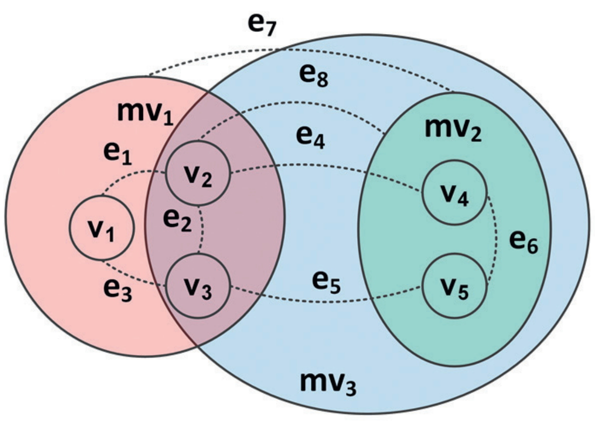


Figure 1 – The example of data metagraph

The example of data metagraph (shown in Figure 1) 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 are contained in different metavertices mv_1 and mv_2. Edge e_7 is an example of the edge connecting metavertices mv_1 and mv_2. Edge e_8 is an example of the 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 holonic nature of the metagraph structure.

3.2 The Comparison of the Metagraph and the Hypergraph Models

According to [22], the hypergraph may be defined as follows:

$$HG = \langle V, HE \rangle, v_i \in V, he_j \in HE, \tag{6}$$

where HG – hypergraph; V – set of hypergraph vertices; HE – set of non-empty subsets of V called hyperedges; v_i – hypergraph vertex; he_j – hypergraph hyperedge.

A hypergraph may be directed or undirected. A hyperedge in an undirected hypergraph only includes vertices, whereas, in a directed hypergraph, a hyperedge defines the order of traversal of vertices.

The example of an undirected hypergraph is shown in Figure. 2. The example contains thee hyperedges: he_1, he_2, and he_3. Hyperedge he_1 contains vertices v_1, v_2, v_4, v_5. Hyperedge he_2 contains vertices v_2 and v_3. Hyperedge he_3 contains vertices v_4 and v_5. Hyperedges he_1 and he_2 have a common vertex v_2. All vertices of hyperedge he_3 are also vertices of hyperedge he_1.

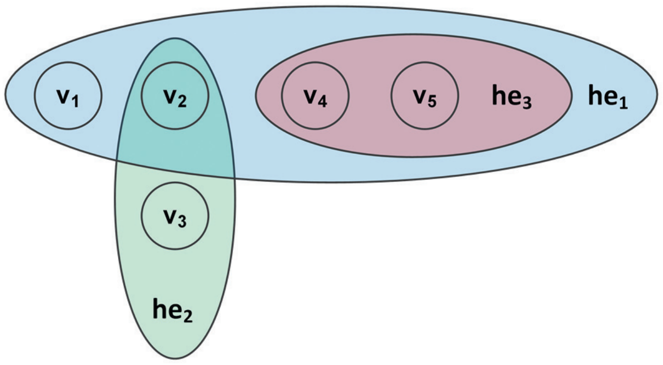


Figure 2 – The example of an undirected hypergraph

Comparing the metagraph and hypergraph models, it should be noted that the metagraph model is more expressive than the hypergraph model. It is possible to note some similarities between the metagraph metavertex and the hypergraph hyperedge, but the metagraph offers more details and clarity because the metavertex explicitly defines metavertices, vertices, and edges inclusion, whereas the hyperedge does not. The inclusion of hyperedge he_3 in hyperedge he_1 in Figure 2 is only graphical and informal because, according to hypergraph definition, a hyperedge inclusion operation is not explicitly defined.

Thus, the metagraph is a complex graph model, whereas the hypergraph is a near flat graph model that does not fully implement the emergence principle.

3.3 The Comparison of the Metagraph and the Hypernetwork Models

It should be noted that there are two versions of hypernetwork models were proposed.

The first version of the hypernetwork model was proposed by Professor Vladimir Popkov with colleagues in the 1980s. Professor V. Popkov proposes several kinds of hypernetwork models with complex formalization, and therefore only the main ideas of hypernetworks will be discussed in this subsection. According to [23], given the hypergraphs $PS \equiv WS_0, WS_1, WS_2, \ldots WS_K$. The hypergraph $PS \equiv WS_0$ is called primary network. The hypergraph WS_i is called a secondary network of order i. Also given the sequence of mappings between networks of different orders: $\{\Phi_i\} : WS_K \xrightarrow{\Phi_K} WS_{K-1} \xrightarrow{\Phi_{K-1}} \ldots WS_1 \xrightarrow{\Phi_1} PS$. Then the hierarchical abstract hypernetwork of order K may be defined as:

$$AS^K = \langle PS, WS_1, \ldots, WS_K ; \Phi_1, \ldots, \Phi_K \rangle \tag{7}$$

The emergence of this model occurs because of the mappings Φ_i between the layers of hypergraphs.

The second version of the hypernetwork model was proposed by Professor Jeffrey Johnson in his monography [24]. The main idea of Professor J. Johnson's variant of the hypernetwork model is the idea of hypersimplex (the term is adopted from polyhedral combinatorics). According to [24], a hypersimplex is an ordered set of vertices with an explicit n-ary relation, and hypernetwork is a set of hypersimplices. In a hierarchical system, the hypersimplex combines k elements at level N (base) with one element at level N+1 (apex). Thus, hypersimplex establishes an emergence between two adjoining levels. The example of hypernetwork that combines the ideas of two approaches is shown in Figure 3.

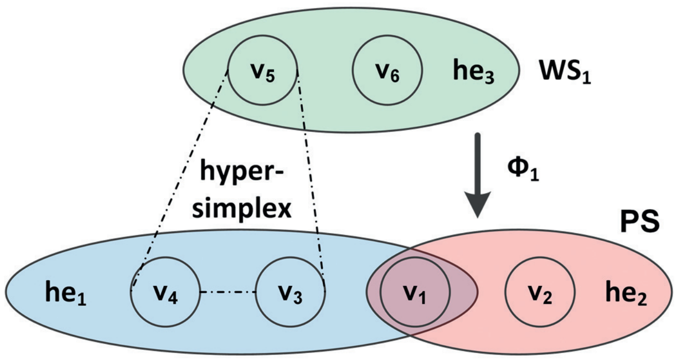


Figure 3 – The example of the hypernetwork

The primary network PS is formed by the vertices of hyperedges he_1 and he_2. The first level WS_1 of the secondary network is formed by the vertices of hyperedge he_3. Mapping Φ_1 (according to Professor V. Popkov model) is shown with an arrow. The hypersimplex (according to Professor J. Johnson model) is emphasized with the dash-dotted line. The hypersimplex is formed by the base (vertices v_3 and v_4 of PS) and apex (vertex v_5 of WS_1).

It should be noted that unlike the relatively simple hypergraph model, the hypernetwork model is a full model with emergence.

Consider the differences between the hypernetwork and metagraph models. According to the definition of a hypernetwork, it is a layered description of graphs. It is assumed that the hypergraphs may be divided into homogeneous layers and then mapped with mappings or combined with hypersimplices. The metagraph approach is more flexible because it allows combining arbitrary elements that may be layered or not using metavertices.

Comparing the hypernetwork and metagraph models, we can make the following notes:

- Hypernetwork model may be considered as “horizontal” or layer-oriented. The emergence appears between adjoining levels using hypersimplices. The metagraph model may be considered as “vertical” or aspect-oriented. The emergence appears at any level using metavertices.
- In the hypernetwork model, the elements are organized using hypergraphs inside layers and using mappings or hypersimplices between layers. In the metagraph model, metavertices are used for organizing elements both inside layers and between layers. Hypersimplex may be considered as a particular case of metavertex.
- Metagraph model allows organizing the results of previous organizations. The fragments of the flat graph may be organized into metavertices, metavertices may be organized in higher-level metavertices, and so on. Metavertex organization is more flexible then hypersimplex organization because hypersimplex assumes base and apex usage, and metavertex may include general form graph.
- Metavertex may represent a separate aspect of the organization. The same fragment of a flat graph may be included in different metavertices whether these metavertices are used for modeling different aspects.

Thus, we can conclude that the metagraph model is more flexible than the hypernetwork model. At the same time, unlike the hypergraph model, the metagraph model is a complete graph model with emergence. Therefore, in the proposed approach, we will use the metagraph model.

4 The Proposed Approach

In this section, we will consider the proposed approach for metagraph Big Data processing. Firstly, we will consider the language for the textual representation of the metagraph data model. Then we will discuss the metagraph representation via a flat graph model that is used for the storage model. In the last subsection, we will discuss the generalized structure of the metagraph Big Data processing engine.

4.1 The Metagraph Textual Representation

Probably the most common way of representing graph data in textual form is Resource Descriptive Framework (RDF), part of the Semantic Web technologies [25]. RDF model describes data via predicate triples. Each triple consists of a subject, a predicate, and an object. Triple is a natural way to represent a flat graph binary edge. There are some specialized graph engines that store and process RDF graphs efficiently, such as gStore and TrinityRDF. gStore evaluates SPARQL queries by subgraph matching, while, TrinityRDF uses graph exploration to answer SPARQL queries.

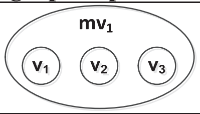
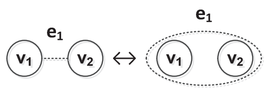
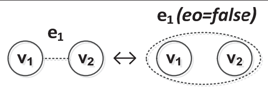
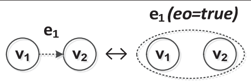
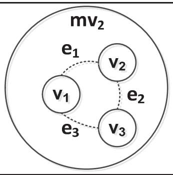
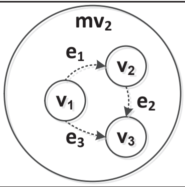
Semantic web technologies are brought to the level of industrial technologies and are used in many information systems. The standard for RDF practical implementation is SPARQL query language by the World Wide Web Consortium, explicitly designed for retrieving and manipulating data in RDF-triples format. Creators of GraphX framework also state that their graph representation adopts RDF concepts of triples view of graphs.

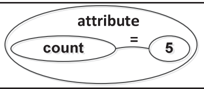
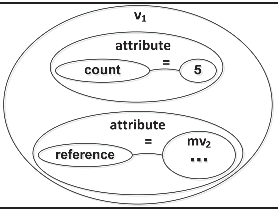
Our article [26] shows that the RDF approach has several limitations in complex situations representation. The first limitation is that the RDF data model consists of very small basic data items – "subject-predicate-object" triples. As a result, an average-sized relational database may correspond to a triple store containing billions of triples. The second limitation is the N-ary relation limitation. This limitation is that the RDF model does not allow simple ways to describe N-ary relations between vertices of the semantic graph, which worsen the description of complex situations in the semantic graph. The article [26] also shows that the metagraph model is free from these shortcomings.

The RDF model may be represented in several textual notations: XML (using namespace rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"), N3, Turtle.

To represent the metagraph in the textual form, we propose to use the predicate model close to logical programming languages, e.g. Prolog. The classical Prolog uses the following form of the predicate: $predicate(atom_1, atom_2, \ldots, atom_N)$. We propose to use an extended form of predicate where along with atoms predicate can also include key-value pairs and nested predicates: $predicate(atom, \ldots, key = value, \ldots, predicate(\ldots), \ldots)$. The mapping of metagraph model fragments into predicate representation is shown in Table 1, according to our paper [20].

Table 1 – The textual representation of the metagraph model

No	Metagraph representation	Textual representation
1		Metavertex(Name=mv$_1$, v$_1$, v$_2$, v$_3$)
2		Edge(Name=e$_1$, v$_1$, v$_2$)
3		Edge(Name=e$_1$, v$_1$, v$_2$, eo=false)
4		1. Edge(Name=e$_1$, v$_1$, v$_2$, eo=true) 2. Edge(Name=e$_1$, v$_S$=v$_1$, v$_E$=v$_2$, eo=true)
5		Metavertex(Name=mv$_2$, v$_1$, v$_2$, v$_3$, Edge (Name=e$_1$, v$_1$, v$_2$), Edge(Name=e$_2$, v$_2$, v$_3$), Edge(Name=e$_3$, v$_1$, v$_3$))
6		Metavertex(Name=mv$_2$, v$_1$, v$_2$, v$_3$, Edge(Name=e$_1$, v$_S$=v$_1$, v$_E$=v$_2$, eo=true), Edge(Name=e$_2$, v$_S$=v$_2$, v$_E$=v$_3$, eo=true), Edge(Name=e$_3$, v$_S$=v$_1$, v$_E$=v$_3$, eo=true))

7		Attribute(count, 5)
8	 v1 attribute count = 5 attribute reference = mv2 ...	Vertex(Name=v_1, Attribute(count, 5), Attribute(reference, mv2))

Case 1 shows the example of metavertex mv_1, which contains three nested disjoint vertices v_1, v_2, and v_3. The predicate corresponds to metavertex, nested vertices are isomorphic to atoms that are parameters of the predicate. As the name of the predicate "Metavertex" is used as the corresponding element of the metagraph model. The key-value parameter "Name" is used to set the name of metavertex. This case is simplest, since nested vertices are disjoint, and metavertex, in this case, is isomorphic to the hypergraph hyperedge.

Case 2 shows the metagraph edge, which may be represented as a particular case of metavertex containing source and destination vertices. This case is also isomorphic to the hypergraph hyperedge. The metagraph edge is represented as a predicate with the name "Edge." The source and destination vertices are represented as predicate atom parameters.

Case 3 also shows the metagraph edge, which fully complies with the formal definition of undirected edge, including direction flag parameter.

Case 4 shows an example of a directed edge. The direction flag parameter is also used. The source and destination vertices may be represented as predicate atom parameters (case 4.1) or as predicate key-value parameters (case 4.2).

Case 5 shows an example of metavertex mv_1, which contains three nested vertices v_1, v_2, and v_3 joined with undirected edges e_1, e_2, and e_3. Edges are represented with separate predicates that are nested to the metavertex predicate. Case 6 is similar to case 5 unless edges e_1, e_2, and e_3 are directed.

The attribute may be represented as a particular case of metavertex containing name and value. Case 7 shows a simple numeric attribute representation. Case 8 shows an example of vertex v1 containing numeric attribute and reference attribute that refers to the metavertex mv2. The attribute is represented as a predicate with the name "Attribute".

Thus, we have defined a predicate description of all the main elements of the metagraph data model.

The textual representation of the metagraph model may be used for storing metagraph model elements in relational or NoSQL databases.

Also, it is well compatible with the classical Big Data approach. Nowadays, the lambda architecture described in [27] is considered to be a classic approach. The textual representation of the metagraph model is the base for processing metagraph data on all layers of the lambda architecture. On the batch layer, the textual representation is used for storing in the master dataset. On the serving layer, the textual representation helps to construct the batch views. On the speed layer, the textual representation helps to construct the realtime views.

4.2 The Principles of the Metagraph Storage Model

The metagraph model discussed in section 3 may be considered as a "logical" metagraph model. To store the metagraph data efficiently, we must create mappings from the "logical" model to "physical" models used in different databases. Our paper [26] discusses the metagraph model mappings to the flat graph model, document model, and the relational model. The experiment results show that the flat graph model mapping is the most efficient, and this approach will be discussed in the current subsection.

The main idea of this mapping is to flatten the hierarchical metagraph model. 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 [28]. Consider there is a flat graph:

$$FG = \left\langle FG^V, FG^E \right\rangle, \tag{8}$$

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:

$$\begin{gathered} BFG = \left\langle BFG^{VERT}, BFG^{EDGE} \right\rangle, \\ BFG^{VERT} = \left\langle FG^{BV}, FG^{BE} \right\rangle, \\ FG^V \leftrightarrow FG^{BV}, FG^E \leftrightarrow FG^{BE}, \end{gathered} \tag{9}$$

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 the 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 a 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*:

$$TFG = \left\langle TFG^{VERT}, TFG^{EDGE} \right\rangle,$$
$$TFG^{VERT} = \left\langle TFG^{V}, TFG^{E}, TFG^{MV} \right\rangle, \qquad (10)$$
$$TFG^{V} \leftrightarrow V, TFG^{E} \leftrightarrow E, TFG^{MV} \leftrightarrow MV.$$

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 $TFG^{VERT} : TFG^{V} \leftrightarrow V, TFG^{E} \leftrightarrow E, TFG^{MV} \leftrightarrow MV$. The set TFG^{EDGE} stores the information about relations between vertices, metavertices, edges in the original metagraph.

Consider the example of the metagraph model flattening represented in Figure. 4. 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.

From a practical point of view, each vertex of the resulting flat graph remembers its own stereotype (metavertex, vertex, edge) in the original metagraph. This allows restoring the metagraph back from a flat graph. The stereotype can be stored as a vertex attribute directly in the graph database or in separate storage. It can be linked to the vertex unique “id.”

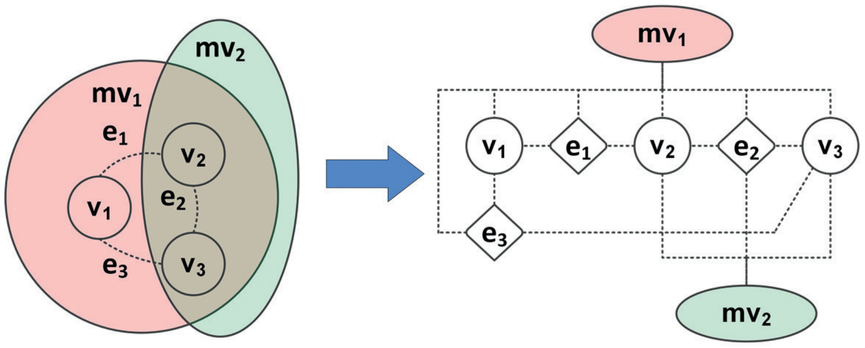


Figure 4 – The example of metagraph flattening, the metagraph for flattening shown on the left and the example of flattened metagraph shown on the right

It is important to note that the metagraph model flattening 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)*

The used notation is RDF-like and supposes that graph edges are named. However, the 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.

It should be noted that metagraph physical representation via a flat graph can be mapped to textual predicate representation. Individual entities of predicate description (vertices, edges, and metavertices) are all represented by individual flat graph vertices. Because of that metagraph processing framework can read data stored in the form of textual predicate representation and import it into flat graph data structure handled by programming tools, similarly to how modern graph processing frameworks import data from RDF representation.

Such representation will require additional memory to store extra edge vertices, but at the same time, it will allow using high-performance approaches, used in flat graph storages [29]. The essential element of graph storage is the optimization of graph traversal. Traditional techniques for this optimization are hash indexes for starting and ending vertices of edges, which allow traversing a path in O(1)-time, and some sort of routing tables. By routing table, we mean vertex adjacency list, a data structure with all incoming or outcoming edges of a vertex. This approach can be found in popular graph database Neo4j and GraphX processing framework. These optimizations make it possible to quickly perform a series of traversals and quickly retrieve adjacent vertices (both optimizations are essential for most graph algorithms, from PageRank to graph clustering). Though the metagraph is aimed to describe complex domains with hierarchical relations, it is still a form of a graph. Thus demand for effectively performing common graph tasks over metagraph storage still exists. Such benefits would not be achieved in the relational database or with tree-like indexing.

Using common graph model on a physical level also makes it possible to save metagraph in external storage and create hybrid transactional-analytical systems. An example of such a hybrid model for flat graphs is the integration of ACID-database Neo4j and Spark processing framework, which suggests using the Spark framework as an analytical tool for data imported from Neo4j storage.

The metagraph model transforms into the document and relational models in slightly different ways. However, the main principle remains the same. The emergent metagraph elements such as vertices, metavertices, edges are transformed into independent elements of the target model.

4.3 The Generalized Structure of the Metagraph Big Data Processing Engine

Based on the ideas discussed in the previous sections, in this subsection, we describe a generalized structure of the metagraph Big Data processing engine, represented in Figure 5.

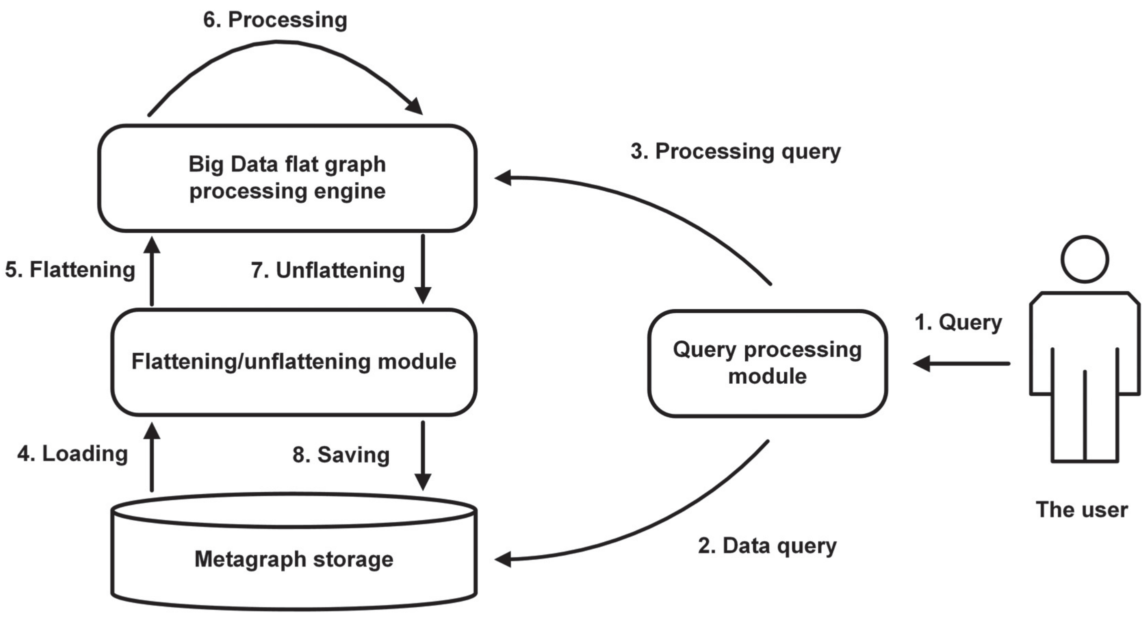


Figure 5 – The generalized structure of the metagraph big data processing engine

Step 1. Query. The user is sending a query to the "query processing module." The query consists of two parts: the "data query" and the "processing query." The "data query" selects the data to be processed. The "processing query" specifies how to process the data.

Step 2. Data query. The "query processing module" translates the "data query" and sending the appropriate commands to the "metagraph storage."

Step 3. Processing query. Simultaneously with step 2, the "processing query" is translated into the processing program for the corresponding "Big Data flat graph processing engine," according to the processing model used by the engine. The processing models were discussed in section 2 (Pregel model, Gather-Sum-Apply model, and so on).

Step 4. Loading. The metagraph data is loaded from the "metagraph storage" into the "flattening/unflattening module."

Step 5. Flattening. The "flattening/unflattening module" transforms metagraph data to the flat graph data. This module may be considered as a separate software unit or as a part of the complex metagraph storage. The flattened graph data is sent into the "Big Data flat graph processing engine."

Step 6. Processing. Based on the data from step 5 and the processing program from step 3, the "Big Data flat graph processing engine" processes the graph data.

Step 7. Unflattening. The processed flat graph data is transformed into the metagraph data using the "flattening/unflattening module."

Step 8. Saving. The query results in the metagraph form are saved to the "metagraph storage."

In order not to overload the Figure 5, it does not show various options for providing the user with the results of the query. The user can receive the results of the query from the "metagraph storage" or directly from the "flattening/unflattening module."

Two main tasks of further research follow from the proposed structure: the development of the languages for the metagraph "data query" and "processing query" and development of translators (interpreters or compilers) for these languages. These tasks open a broad scope for research. In particular, the development of the "data query" language and its translator depends on different variations of the physical model of the metagraph storage. Also, the development of the "processing query" language depends on the processing model and implementation features of the Big Data flat graph processing engine.

Thus, the proposed approach makes it possible to process the metagraph data using the "Big Data flat graph processing engine."

5 Conclusions

Nowadays, Big Data frameworks are widely used for processing graph information. The dominant graph model in such frameworks is usually a flat graph model or property graph model (which is, in fact, the multigraph model). However, to describe complex situations, flat graph models may not be enough. To describe complex situations, we propose the use of a metagraph model.

The key element of the metagraph model is metavertex. From the general system theory point of view, a metavertex is a particular 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 metagraph model is a complex graph model, whereas the hypergraph is a near flat graph model that does not fully implement the emergence principle. Also, the metagraph model is more flexible than the hypernetwork model.

The main elements of the metagraph data model may be represented in textual form using the predicate description.

The metagraph model may be transformed into a flat graph model based on the multipartite graphs.

The generalized structure of the metagraph Big Data processing engine makes it possible to process the metagraph data using the traditional "Big Data flat graph processing engine."

References

[1] D. Reut, S. Falko, E. Postnikova. About scaling of controlling information system of industrial complex by streamlining of big data arrays in compliance with hierarchy of the present lifeworlds. International Journal of Mathematical, Engineering and Management Sciences, 4(5):1127-1139, October 2019.

[2] V. Chesnokov. Overlapping community detection in social networks with node attributes by neighborhood influence. In: Proceedings of the 6th International Conference on Network Analysis, NET 2016, SPMS, vol. 197, pp. 187-203. Springer, 2017.

[3] B. Rasheed, A.Yu. Popov. Network graph datastore using DiSc processor. In: Proceedings of the 2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering, ElConRus 2019, pp. 1582-1587.

[4] C. Abdymanapov, A.Yu. Popov. Motion planning algorithms using DISC. In: Proceedings of the 2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering, ElConRus 2019, pp. 1844-1847.

[5] A.N. Bozhko. Hypergraph model for assembly sequence problem. In: IOP Conference Series: Materials Science and Engineering, vol. 560, Issue 1, 2019.

[6] G. Malewicz, M.H. Austern, A.J.C. Bik, J. Dehnert, I. Horn, N. Leiser, G. Czajkowski. Pregel: A System for Large-scale Graph Processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data SIGMOD '10, ACM, pp. 135–146. doi: 10.1145/1582716.1582723

[7] R.S. Xin, D. Crankshaw, A. Dave, J.E. Gonzalez, M.J. Franklin, I. Stoica. GraphX: Unifying Data-Parallel and Graph-Parallel Analytics. arXiv preprint arXiv:1402.2394, February 2014. [Online]. Available: https://arxiv.org/pdf/1402.2394.pdf

[8] R. Shaposhnik, C. Martella, D. Logothetis. Practical Graph Analytics with Apache Giraph. Apress, 2015.

[9] A comparison of state-of-the-art graph processing systems. [Online]. Available: https://code.fb.com/core-data/a-comparison-of-state-of-the-art-graph-processing-systems

[10] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, J.M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning in the Cloud. arXiv preprint arXiv:1204.6078, April 2012. [Online]. Available: https://arxiv.org/pdf/1204.6078.pdf

[11] J.E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In: Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation OSDI'12, pp. 17–30, 2012.

[12] Introducing Gelly: Graph Processing with Apache Flink. [Online]. Available: https://flink.apache.org/news/2015/08/24/introducing-flink-gelly.html

[13] M. Han, K. Daudjee, K Ammar, M.T. Özsu, X. Wang, T. Jin. An Experimental Comparison of Pregel-like Graph Processing Systems. In: Proceedings of the VLDB Endowment, 7(12):1047-1058, 2014. [Online]. Available: https://www.vldb.org/pvldb/vol7/p1047-han.pdf

[14] C. Mayer, R. Mayer, S. Bhowmik, L. Epple, K. Rothermel. HYPE: Massive Hypergraph Partitioning with Neighborhood Expansion. arXiv preprint arXiv:1810.11319, October 2018. [Online]. Available: https://arxiv.org/pdf/1810.11319.pdf

[15] W. Jiang, J. Qi, J.X. Yu, J. Huang, R. Zhang. HyperX: A Scalable Hypergraph Framework. IEEE Transactions on Knowledge and Data Engineering, 31(5):909-922, May 2019. doi: 10.1109/TKDE.2018.2848257

[16] F. Colace, M. Lombardi, F. Pascale, D. Santaniello. A Multilevel Graph Representation for Big Data Interpretation in Real Scenarios. In: Proceedings of the 3rd International Conference on System Reliability and Safety (ICSRS), Barcelona, Spain, 2018, pp. 40-47. doi: 10.1109/ICSRS.2018.8688834

[17] Complex network. [Online]. Available: https://en.wikipedia.org/wiki/Complex_network

[18] V. Chapela, R. Criado, S. Moral, M. Romance. Intentional Risk Management through Complex Networks Analysis. SpringerBriefs in Optimization, Springer, 2015.

[19] A. Basu, R.W. Blanning. Metagraphs and Their Applications. Springer, 2007.

[20] V.M. Chernenkiy, Yu.E. Gapanyuk, A.N. Nardid, A.V. Gushcha, Yu.S. Fedorenko. The Hybrid Multidimensional-Ontological Data Model Based on Metagraph Approach. In: A.K. Petrenko, A. Voronkov (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

[21] V.M. Chernenkiy, Yu.E. Gapanyuk, G.I. Revunkov, Yu.T. Kaganov, Yu.S. Fedorenko, S.V. Minakova. Using metagraph approach for complex domains description. In: Selected Papers of the XIX International Conference on Data Analytics and Management in Data Intensive Domains (DAMDID/RCDL 2017). Moscow, Russia, October 9-13, 2017. [Online]. Available: http://ceur-ws.org/Vol-2022/paper52.pdf

[22] V.I. Voloshin. Introduction to Graph and Hypergraph Theory. Nova Science Publishers, Inc, 2009.

[23] A.T. Akhmediyarova, J.R. Kuandykova, B.S. Kubekov, I.T. Utepbergenov, V.K. Popkov. Objective of Modeling and Computation of City Electric Transportation Networks Properties. In: International Conference on Information Science and Management Engineering (Icisme 2015), Destech Publications, Phuket, 2015, pp. 106–111.

[24] J. Johnson. Hypernetworks in the Science of Complex Systems. London, Imperial College Press, 2013.

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

[26] V.M. Chernenkiy, Yu.E. Gapanyuk, Yu.T. Kaganov, I.V. Dunin, M.A. Lyaskovsky, V.S. Larionov. Storing Metagraph Model in Relational, Document-Oriented, and Graph Databases. In: Selected Papers of the XX International Conference on Data Analytics and Management in Data Intensive Domains (DAMDID/RCDL 2018). Moscow, Russia, October 9-12, 2018. [Online]. Available: http://ceur-ws.org/Vol-2277/paper17.pdf

[27] N. Marz, J. Warren. Big Data. Principles and best practices of scalable realtime data systems. Manning, New York, 2015.

[28] G. Chartrand, P. Zhang. Chromatic graph theory. Discrete mathematics and its applications. Chapman & Hall/CRC, Boca Raton, 2009.

[29] I. Robinson, J. Webber, E. Eifrem. Graph Databases: New Opportunities for Connected Data. O'Reilly Media, 2015.