Representing history in graph-oriented nosql databases: A versioning system

Arnaud Castelltort, Anne Laurent

Representing History in Graph-Oriented NoSQL DataBases: A Versioning System

Arnaud Castelltort, Anne Laurent

***Abstract*— Graph databases are taking more and more importance, especially for social networking. For instance, organizations can implement graph databases to represent and query data such as *Person1 is CEO of Organization1*. NoSQL graph databases (e.g., Neo4j) have been designed to deal with such data. However, managing history is not yet possible in an easy manner while being critical in many applications. Tracking changes is indeed one of the main functionalities in databases (especially Relational BD) and should not be forsaken in NoSQL graph DB. For instance, queries like "list all the people who have been CEO of Organization1" or "list all the functions People1 has been taken in his career" are important. In this paper, we thus propose a novel representation of historical graph data and tools to implement it as a plug-in of existing NoSQL graph systems.**

I. Introduction

Graph databases are becoming more and more important as their use is increasing for managing data within applications such as social networks. In this context, they indeed propose a scalable and easy-to-use environment [1], [2]. For instance, they allow to represent organizations with their people, structures and links between these entities, as described in Fig. 1. Graph databases are becoming popular as they provide a robust solution for facing big data and as they allow to focus more on relations betweens objects rather than on the objects themselves, as it is the case in social networking.

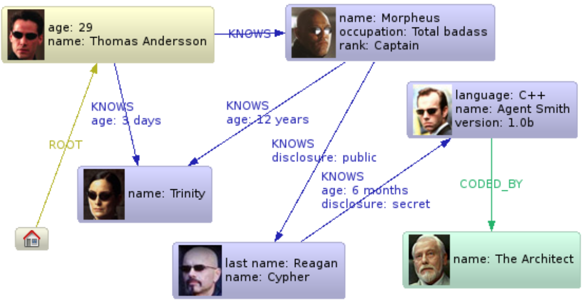

Fig. 1. Example of Graph Data: The *Matrix* Neo4j Database

Many work have been proposed in the literature of databases to deal with history [3], [4], [5]. However, very few of them have addressed historization for NoSQL databases. Some works have studied and implemented solutions for some of the products (e.g., versioning modules for MongoDB, CouchDB's "Simple Document Versioning"...). To the best of our knowlege, graph-oriented databases do not provide such modules. We thus propose a novel approach for dealing with this challenge. For this purpose, we provide in this paper the necessary definitions and methods.

The rest of the paper is organized as follows. Section II recalls the seminal definitions related to graph-oriented databases. Sections III and IV introduce our proposal for representing history in the context of graph-oriented databases, which is discussed in Section V. Section VI concludes and presents the future work.

II. Preliminary Definitions

Graph oriented databases are based on directed graphs [6], [7], [8]. Formally, a graph is a representation of a set of objects where some pairs of the objects are connected by links. The interconnected objects are represented by mathematical abstractions called vertices, and the links that connect some pairs of vertices are called edges. A set of vertices and the edges that connect them is said to be a graph.

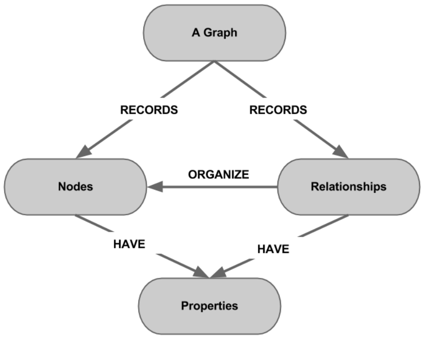

Fig. 2. Graph of Graph Organization

Fig. 2 illustrates these entities. Typically, a graph is depicted in diagrammatic form as a set of dots for the vertices, joined by lines or curves from the edges. The edges may be directed or undirected. This oriented graph is called a directed graph and the edges are called directed edges or arcs. Vertices are also called nodes or points, and edges are also called lines or arcs.

A. Defining Graphs

This paper considers the classical definitions given below.

Definition 1 (Graph): A graph G is given by a pair (V, E) where V stands for a set of vertices and E stands for a set of edges with $E \subseteq (V \times V)$.

Arnaud Castelltort and Anne Laurent are with the Univ. Montpellier 2, LIRMM, CNRS UMR5506, France (email: firstname.lastname@lirmm.fr).

Definition 2 (Directed Graph): A directed graph G is given by a pair (V, E) where V stands for a set of vertices and E stands for a set of edges with $E \subseteq \{V \times V\}$. That is E is a subset of all ordered permutations of V element pairs.

B. Defining Graph Databases: A Topic-Oriented Point of View

The structure a graph takes in the real-world determines the efficiency of the operations that are applied to it. It is exactly those efficient graph operations that yield an unconventional problem-solving style.

In this section, we introduce our conceptual representation of the different elements of a graph database independently on the implementation library which will be chosen.

Definition 3 (Property): A property p is a $(key, value)$ pair where the key is a string identifier and the $value$ is a set of elements of a given type, this element being possibly reduced to one element, the null element being forbidden.

The type of the elements can be string, int, bool, etc. Null values can be modeled by the absence of a key.

Nodes form a graph when being linked with relationships. They are often used to represent entities and can contain properties as illustrated on Fig. 3. In our vision, a node is not only defined by its properties, but also its incoming and outgoing relationships. This allows to improve traversal between nodes [9].

Definition 4 (Node): A node n is defined as a pair (id_n, P_n):

- id_n is an identifier,
- P_n is a set of properties.

It should be noted that the id does not contain any semantical information and is not unique over time within a graph. It can indeed be recycled after node's deletion to be reused for a new node.

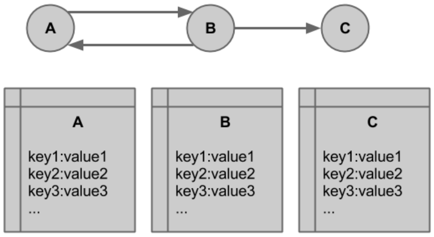

Fig. 3. Node Properties

Relationships between nodes are a key part of a graph database. A relationship connects two nodes, and is guaranteed to have valid start and end nodes. As relationships are always directed, they can be viewed as outgoing or incoming relative to a node, which is useful when traversing the graph.

Relationships are equally well traversed in any direction. This means that, you can ignore the direction where it is not useful in your application. There is no need to add duplicate relationships in the opposite direction (with regard to traversal or performance). Just like nodes, relationships have properties as illustrated on Fig 4 and defined below.

Definition 5 (Relationship): A relationship R is defined as a 5-tuple $(id_R, n_{iR}, n_{oR}, T_R, P_R)$ where:

- id_R is an identifier,
- n_{iR} is the id of the incoming node,
- n_{oR} is the id of the outgoing node,
- T_R is the type of R,
- P_R is a set of properties.

It should be noted that start (incoming) n_{iR} and endpoint (outgoing) n_{oR} nodes can be the same (a node can have relationships to itself).

A node can easily be linked to its incoming and outgoing nodes by considering the relationthips connected to it. Graph databases implement these features very efficiently [9].

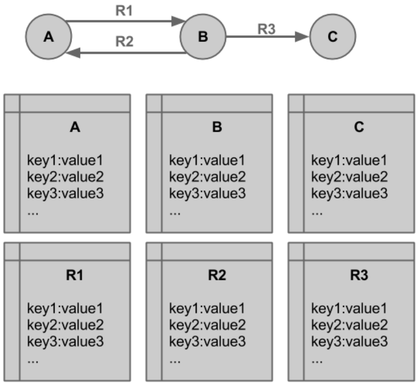

Fig. 4. Relationship Properties

III. Graph Versioning

This section presents our proposition to design and implement a versioning system on graph databases. We define the following criteria that must be met by the versioning solution:

- **Non-intrusivity criterion**
  - At the technical layer, the system must work without the need for developers to modify code unless they want to get some history support functionality.
  - At the conceptual layer, the system must not have any impact on the graph's structure.
- **Pluggability criterion**
  The system should be distributed has a library or plugin.
- **Temporal independance criterion**
  The system must be pluggable at any moment (at project launching or after a while in production) like a "Plug and Play" device.
- **History support criterion**
  - The system must provide a way to get history of a node or a relationship.
  - The system must provide a way to get the difference between two versions of a graph in time.
  - The system should provide a way to track context informations about every version (e.g., who, what, when, why) if asked.

Before introducing the system, we propose some terms and ideas.

A. Preparatory Statements

This section aims at presenting the use cases we have targeted and at defining the language and terms we are using.

1) Historical Points of View: We first claim that history must be considered as being highly linked with some point of view. We do not consider history as a stand-alone concept, but rather consider *history of objects*.

For this purpose, we distinguish three types of history, namely:

- The history from the **node point of view**, that is the entity point of view, to see the changes that have occurred on the entity itself. (e.g., on a Person point of view, searching for changes on marital status over time or on jobs held);
- The history from the **relationship point of view**, to see the changes to the relation including start or end node updates.(e.g., on the relationship “President” point of view, searching the last ten presidents);
- The history from the **Graph point of view**, to see the change on the graph itself.(e.g., on a Human Resources Graph point of view, searching for the differences between two dates to establish turnover).

2) Versioning Point of View: Our proposal is based on the idea that versioned data have not to be stored in the same modelisation way than the operational data. Thus, we provide a separation between the operational graph data and the management of the sequence of versions over time. For this purpose, we consider the following concepts.

- **DataGraph:** the current graph in use
- **Transaction:** a set of operations which either all occur, or nothing occurs on the DataGraph. A guarantee of atomicity prevents updates to the graph occurring only partially, which can cause greater problems than rejecting the whole series outright
- **Revision:** Every transaction makes a new version of the graph, termed revision
- **VersionGraph:** A VersionGraph stores the history of different versions of a data graph. The VersionGraph model does not depend on the DataGraph architecture choice, as explained later in this paper.

B. VersionGraph Structure

For understanding sake, we explain our proposal using a very simple graph as shown in Fig. 5.

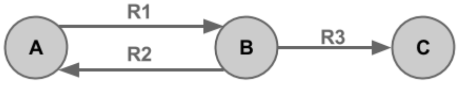

Fig. 5. Simple Graph Model

The graph from Fig. 5 is only made of 3 nodes (A, B, C) and three relations $(R1, R2, R3)$.

1) Sample Graph Model representation in Version Graph: To manage the history of the Data Graph ‘DG, a Version-Graph ‘VG is built as shown in Fig 6.

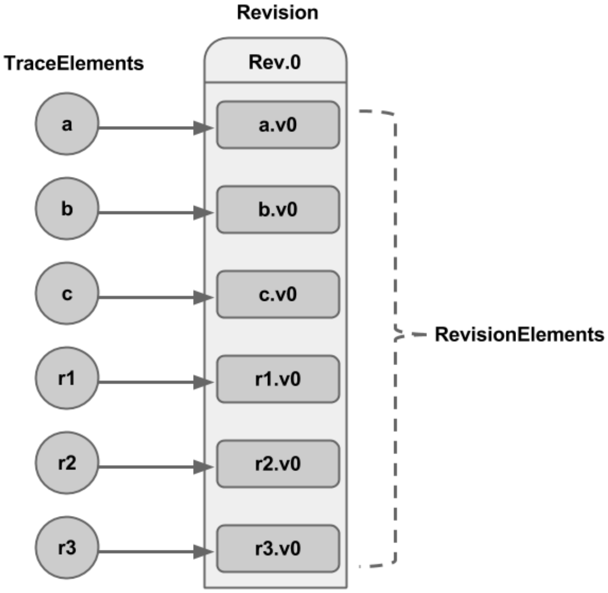

Fig. 6. Version Graph Model

Our system manages the versioning as follows:

- Every DG node is represented by a VG node (e.g., ‘A’ node is represented by ‘a’ node in VersionGraph)
- Every DG relation is represent by a VG node (e.g., ‘R1’ relation represented by ‘r1’ node in VersionGraph)
- For every DG element {node, relation}, called TraceElement, will have a linked list of RevisionElements
- Every RevisionElement has the entire set of properties of the element which referenced to and some meta-informations fields as the date of creation, and optionaly who and when
- RevisionElement for DG nodes have two more sets: one for ingoing and one for outgoing relations
- A new Version Graph Revision is created for every transaction

These elements are formally defined in Section IV.

2) Making a Graph Change: A modification on a simple node or relationship property leads to a new RevisionElement in the linkedList of the TraceElement for this node/relationship as shown in Fig 7. Of course, the transaction creates a new VersionGraph revision.

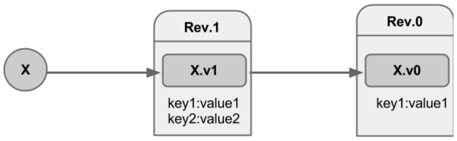

Fig. 7. Edit a node Property

Deleting a node is a different matter. To represent this case, relation R2 between A and B is deleted, as shown by Fig. 8.

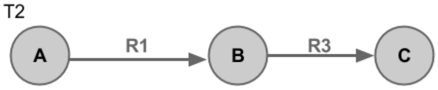

Fig. 8. Data Graph : Deleting R2

Deleting a relation between two nodes has several consequences.

1) A new RevisionElement is created for the start node of the relation where the outgoing set of relationship will be updated.
2) In the same way, the end node will have a new RevisionElement created but this time it is the ingoing set that will be updated.
3) The new revision will not track the relation R2.

When the transaction is committed, the version graph looks as shown by Fig. 9

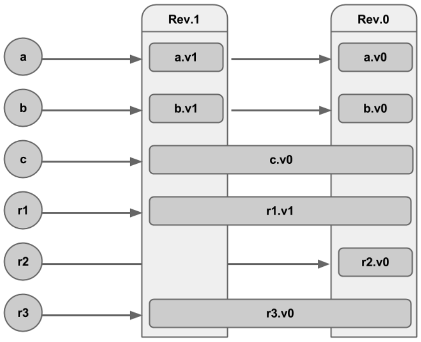

Fig. 9. VersionGraph: Deleting R2

It should be noted that there is only a new RevisionElements for the TraceElements whose tracking DG element have been modified by the transaction. The last RevisionElement is tracked for the others.

3) Tracking a Relation: In relational databases, relationships' attributes are used in flat tables to represent incoming and outgoing links. Then, this information can easily be updated. By contrast, in graph databases, only properties can be modified on the relationship. There is no update order on relationship start/end node because a relationship cannot exist without both start and endpoint. Such an update is then handled by deleting the existing relationship and creating a new one.

4) Tracking a Relationship When Changing the Start or End Point: In the transaction 3, a new relation is created between C and B of type R2. We want this relation to be tracked as a new version of the initial relation R2 from A and B of the first revision.

Here are two real world use cases:

- changing a job title (i.e. defining a new type for the relation)

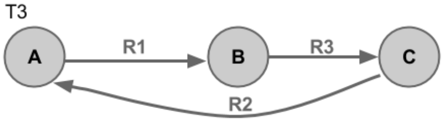

Fig. 10. DataGraph: Adding a new R2 type relation with history tracking

- Tracking the list of people who get the CEO job

Fig. 11 depicts the VersionGraph.

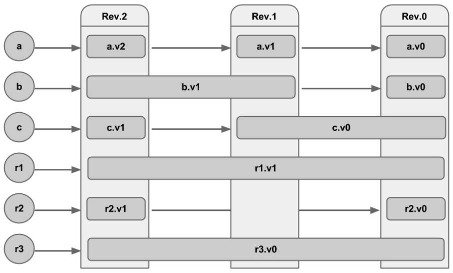

Fig. 11. VersionGraph: Adding a new R2 type relation with history tracking

The lack of time-unique id for every element makes it difficult to track history. As presented above, the data graph implementation looks like Fig. 12

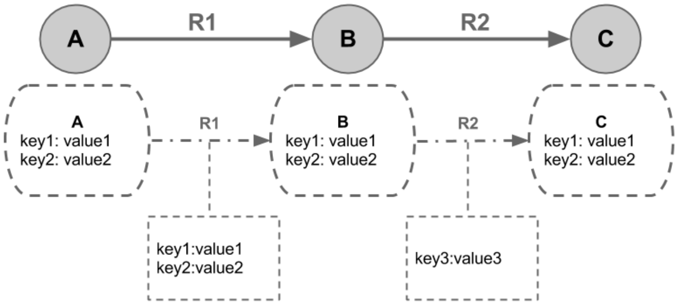

Fig. 12. Graph Properties

To enable the tracking capacity, we have to enhance the data graph with an historization property on every node and relation: the $rev–uuid$ property.

This property is a logical entites' tracer for tracking their history, so that the type of an element can change, be deleted or recreated, without the risk of losing its history. The developer does not need to implement this uuid by herself. A trigger/handler system on the database can do it for her.

The algorithm is simple, as described in Algorithm 1.

Algorithm 1: OnElementCreation(Element X)

```
if !$exists(X.rev–uuid)$ then
    $X.rev–uuid = generateUUID()$
    $VG.CreateVersionNode(X)$
end if
$VG.CreateRevisionNode(X)$
```

This algorithm creates a $rev–uuid$ for every new element except if the new element is created with an $rev–uuid$.

This case can happen when binding a new element on an already existing history. For example, if we replace a $(A) - [R1] \rightarrow (B)$ relation by a $(A) - [R1] \rightarrow (C)$ relation, the process will be taken from Algorithm 2.

Algorithm 2: Change end node of a relation R1

```
    $newRelation = newRelation(A, C)$
    $newRelation.rev–uuid = R1.rev–uuid$
    $Delete(R1)$
    $Save(newRelation)$
```

Fig. 13 depicts the Data graph enhanced by the $rev–uuid$.

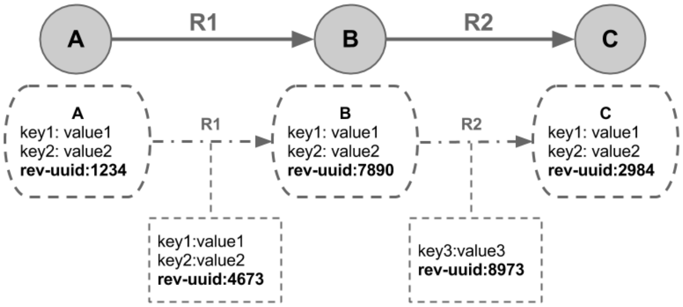

Fig. 13. Revision uuid

C. Managing Points of View in VersionGraph

Fig. 14 illustrates the three different kinds of historical points of view, namely by graph revision, by node and by relation.

IV. Anatomy of VersionGraph

A. Formal Definitions

GraphRevision. For every new revision, the VersionGraph is only increased by the modified elements. Every transaction or action (if no transaction support is implemented in the graph database) creates a new version of the Data Graph which is materialized by a revision in the VersionGraph.

The new revision contains both the newly modified elements and the ones that have not been modified from the previous revision. As a result, every revision is a complete snapshot of the Data Graph taken at a given time.

There are two ways to modelize a GraphRevision:

1) as a node pointing to all the RevisionElements included in the GraphRevision,
2) as a label stored in every RevisionElement included in the GraphRevision (labels being a new feature available in some graph databases).

In order to remain in the most general level, our modelisation relies on the first case.

Definition 6 (GraphRevision): The GraphRevision node is defined as a node $n = (id_n, P_n)$ where:

- P_n contains:
  - a date of creation,
  - an author,
  - a comment property
- there exists at most one relation starting from n and ending to a GraphRevision node representing the previous version (if it exists),
- there exists at most one relation ending to n and starting from a GraphRevision node representing the next version if it exists or the revision root node otherwise,
- there exists a set of relationships starting from n and ending to all RevisionElements that are included in the revision.

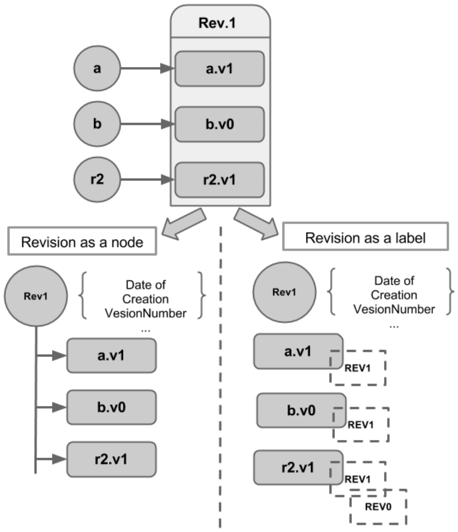

Fig. 15. Revision implementation as label or node

The revision root node is the startpoint to traverse data graph by revision. Every revision can be seen as a subgraph that has the semantic information available in Data Graph at a time.

TraceElement. Every element of a Data Graph is represented by a TraceElement in the VersionGraph.

Definition 7 (TraceElement): Given an element e of the DataGraph, a TraceElement is a node $n_e = (id_n, P_n)$ where:

- P_n contains meta-information such as tracking launch date, last modified date, number of revision nodes, last graph revision number to enable fast access to some statistical information and usefull data;
- P_n contains a property $rev–uuid$ mapped with $e.rev–uuid$;
- there exists a unique relation starting from n and ending to the last RevisionElement.

The set of properties of a node may change over time, this information is not set on the TraceElement but on RevisionElement.

RevisionElement. There are differencies between the implementation of RevisionElement's nodes and relations.

Definition 8 (RevisionElementNode): Given a TraceElementNode $t = (id_t, P_t)$ tracking a DataGraph node, a

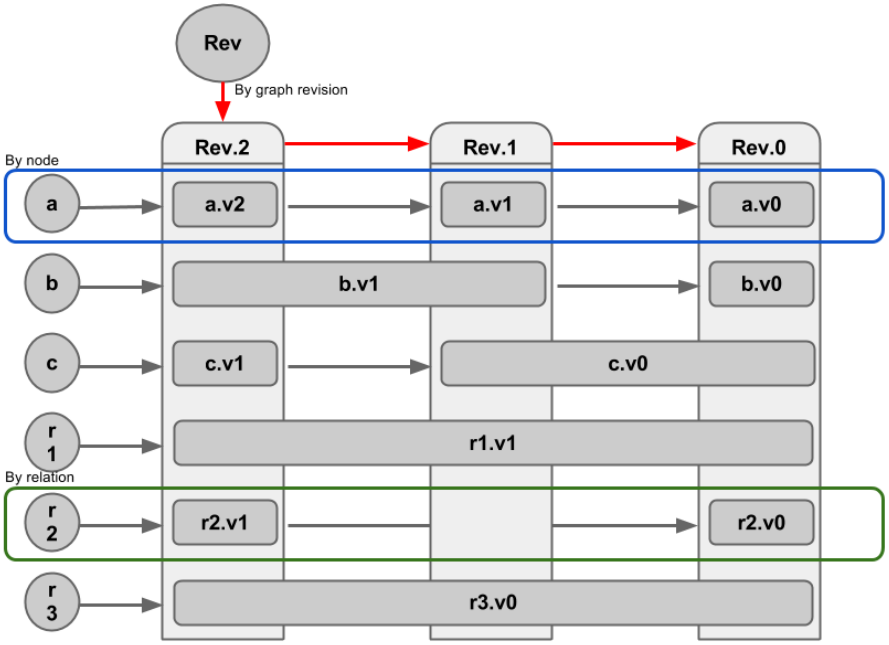

Fig. 14. Point of view in VersionGraph

RevisionElementNode is a node $n_t = (id_n, P_n)$ where:

- P_n contains all the properties of the DataGraph element related to t;
- P_t also contains:
  - the date of creation of n;
  - a version number, the value is RevisionElement.revisionNumber + 1;
  - the set of all pairs $(rev\text{–}uuid, relation\_type)$ from the relations ending to or starting from of the DataGraph element related to t;
- there exists at most one outgoing relationship from n that points to another RevisionElementNode;
- there exists a unique ingoing relation on the version graph Revision.

Definition 9 (RevisionElementRelation): Given a TraceElementNode $t = (id_t, P_t)$ tracking a DataGraph relation, a RelationElementRelation is a node $n = (id_n, P_n)$ where:

- P_n contains all the properties of the DataGraph element related to t;
- P_n contains:
  - the date of creation of n;
  - a version number, the value is RevisionElement.revisionNumber + 1;
  - the $rev\text{–}uuid$ of the incoming node of the DataGraph element related to t;
  - the $rev\text{–}uuid$ of the outgoing node of the DataGraph element related to t.

It should be noted that the startNode and endNode are in the RevisionElement and not in TraceElement because they can change over time. Also, the root element of a chain of RevisionElements is always the TraceElement they are related to.

B. Deployment Architecture

The VersionGraph system proposed in this paper can be deployed in two main different ways:

1) as a full graph in a separate graph database,
2) as a subgraph of the graph database.

1) VersionGraph as a Separate Graph Database: Separating the operational and historical graph databases has several advantages.

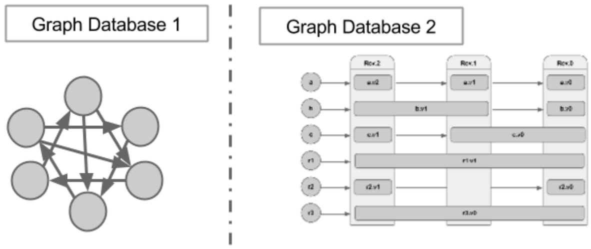

Fig. 16. Version Graph as a separate graph database

First, Graph Database can have a huge size and can be hosted on high avaibility infrastructures with a high cost to support a huge traffic. VersionGraph will be bigger than DataGraph. If the use of Version data needs less traffic, it can be hosted on a lower cost infrastructure with higher disk capacity.

Second, VersionGraph holds all the versions of the Data Graph including the current Data Graph revision. Based on this, all the queries on historization can be computed independently on the main data system.

The main drawback is that the consistency between the two graphs has to be maintained. This means that for every transaction on the Data Graph database, the new graph revision on the VersionGraph must be created in the other database.

On the one hand, this can be managed in a synchronous way. However, performance issues may occur.

On the other hand, if managing this as a failover asynchronous way, the system complexity may increase, and consistency may not be guaranteed between DataGraph and VersionGraph at any time.

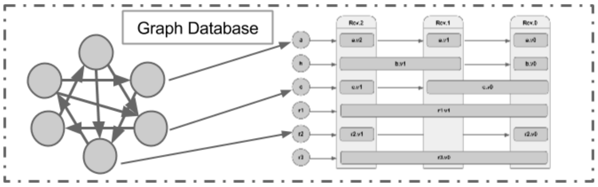

Fig. 17. Version Graph as a subgraph of DataGraph

2) Version Graph as a Subgraph of DataGraph: In this case, only one database is used. This solution has the advantage to allow mixing common queries with version queries and to ensure consistency. However, all version queries will have an impact on the load of the overall system.

To simplify query conception, a link can be added between every node and the equivalent VersionElement and then the history can easely be traversed: $X.version.previous$ returns the first RevisionElement of the node X

Also, as presented previously, the storage space must be provisioned to support both DataGraph and VersionGraph. This should be at least more than two times DataGraph requirement size.

V. Criteria Matching

Section III-B introduces the criteria that should be matched by a graph versioning system. In this section, we discuss our proposal regarding its complience with all these criteria.

A. Non Intrusivity criterion

The version graph proposal is independent of the DataGraph structure, then it is not necessary to adapt the graph's structure to support historization. Futhermore, the system will work unobtrusively without any change to the existing projet code as it is using a system of trigger/handler to build the VersionGraph without the need for developers to change requests nor to update code.

B. Pluggable criterion

Depending on graph implementation system, this system will be packaged as a plugin or as a set of trigger orders to be stored in the graph database engine. Deploying the system as a self package plugin seems to be a far better approach whenever possible.

C. Temporal independance criterion

The version graph system proposal could be installed at any moment. When starting the version graph system the first time, it will scan all nodes and relations. For each element $(node, relation)$ it will add a $rev\text{-}uuid$ in DataGraph and will make a TraceElement and a first RevisionElement to make GraphRevision 0 in VersionGraph. Starting from this point, every transaction will lead to a new revision .

Our system can be plugged at anytime:

- at project launch, there will be no available element (if an empty database does not require a root Node). As a consequence, the first Revision in the VersionGraph will not have any element.
- at any moment, the VersionGraph can be constructed and after it, the VG Revision 0 will have every element at the installation time of the version graph system.

D. Historization features criterion

As shown in Fig.14, three kinds of traversal can be adressed by the version graph system. We will go deeper on this subject in a futher work.

VI. Conclusion and Further Work

In this paper, we address data historization in the context of graph-oriented NoSQL databases. Although being crucial, changes tracking is not yet available in such databases. We thus propose a novel approach, by both studying what historization means in such a context, and by proposing representations and methods to manage history.

Further work include the implementation of our solution, which should be realized using a Neo4j graph database. Moreover, queries over historical graph databases will be studied. We are also studying the reduction of the number of revisions by a squashing system.

References

[1] R. Cattell, "Scalable SQL and NoSQL data stores," *SIGMOD Record*, vol. 39, no. 4, pp. 12–27, 2010.

[2] J. Han, E. Haihong, G. Le, and J. Du, "Survey on nosql database," in *Proc. of the 6th International Conference on Pervasive Computing and Applications (ICPCA)*, 2011, pp. 363–366.

[3] A. Kupfer, S. Eckstein, K. Neumann, and B. Mathiak, "Keeping track of changes in database schemas and related ontologies," in *7. Int. Baltic Conference on Databases and Information Systems*. IEEE, 2006, pp. 63–68.

[4] S. S. Chawathe, S. Abiteboul, and J. Widom, "Representing and querying changes in semistructured data," in *In Proc. of the ICDE Conference*, 1998.

[5] S. S. Chawathe and H. Garcia-Molina, "Meaningful change detection in structured data," in *SIGMOD*, 1997.

[6] R. Angles and C. Gutiérrez, "Survey of graph database models," *ACM Comput. Surv.*, vol. 40, no. 1, 2008.

[7] J. L. Reutter and T. Tan, "A formalism for graph databases and its model of computation," in *AMW*, ser. CEUR Workshop Proceedings, P. Barceló and V. Tannen, Eds., vol. 749. CEUR-WS.org, 2011.

[8] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases*, O'Reilly, Ed., to appear.

[9] M. A. Rodriguez and P. Neubauer, "The graph traversal pattern," *CoRR*, vol. abs/1004.1001, 2010.