Managing and Querying Historical NoSQL GraphDatabases: The HNTP Criteria

Arnaud Castelltort, Anne Laurent

Managing and Querying Historical NoSQL GraphDatabases:

the HNTP Criteria

Arnaud Castelltort[1], Anne Laurent[2]

[1]LIRMM, University Montpellier 2, CNRS UMR5506
Montpellier, France
castelltort@lirmm.fr

[2]LIRMM, University Montpellier 2, CNRS UMR5506
Montpellier, France
laurent@lirmm.fr

Abstract
Graphs allow to represent many cases from the real world, for instance ontologies, social networking or chemical databases. Graph databases, and especially NoSQL graph databases (e.g., Neo4j) have been designed to deal with such data. This new generation of databases focus on relations rather than on the objects themselves. NoSQL graph databases provide efficient tools to implement robust solutions against big data. In such databases, managing historical data may be important in order to store and analyze evolutions and to understand how relations within and between graph entities evolve. Many works have been done in information systems to keep track of actions and successive states of the data and actions applied on it (updates, deleting, creating, archiving, ...). However, it is hard to handle in NoSQL graph database systems. In this paper, we thus propose to discuss the challenges of implementing historical features in graph databases by introducing and discussing the HNTP criteria solutions should meet.
Keywords: *NoSQL Graph Databases, Historical Data.*

1. Introduction

Operational applications often focus on the current state of the system without concentrating efforts on the management of history and versions. When modelling an organisation for instance, the objects (e.g., people, units), their associated information (e.g., name, address) and the relationships between them (e.g., membership, leadership) are considered without focusing on the timeline and history. Systems only capture the current state as a snapshot without considering their dynamic.

When dealing with data, maintaining history is yet one of the key points for understanding and managing systems. It has thus been integrated from thebeginning of database systems, providing tools for storing and querying both the data and the changes. History can be considered to deal with many goals.
In some cases, change management in databases aim at dealing with legal requirements, for instance to be able to retrieve which people did an action, the versions of a document (e.g., forensics, legal archives) [21].

In other cases, change management aims at maintaining systems and being able to restore them
in case of tampered data and breakdowns (Disaster Recovery Plans). Moreover, the failures having caused the breakdown can sometimes be identified. In many other cases, change management systems are designed to understand how systems work. This is for instance the case for Custumer-Relationship Management (CRM) systems, medical systems or scientific data [7].
In these cases, history is intrinsic. Evolutions are then studied in order to detect trends, to retrieve the causes for some behaviour, etc. As shown in the case of data warehouses in [30], data may not be historical by nature although it is important to store and query temporal views.

Change management have been studied all over the history of databases and data representation and query (e.g., relational model, temporal logic, temporal databases, versions of ontologies, XML or RDF repositories).

In this paper, we focus on the framework of graph data.

Graph data are ubiquitous. In such databases, the focus is put on the relationships between entities, such as social networks, media networks, biological networks, etc. Many tools and systems have been built to deal with such data, ranging from theoretical models to popular implemented systems.
As well theoretically as in implemented systems, it has been shown that NoSQL graph databases are much more efficient on linked data than relational model [4].

Several query methods and languages are proposed withing the graph NoSQL database [15]. We thus consider this type of database systems. However, very few work have focused on change management in such NoSQL graph databases [16, 25].

We thus discuss below how such systems should manage temporal changes, by providing and discussing the main criteria to be fulfilled.

The paper is organised as follows. Section 2 recalls the main work from the literature dealing with history management and what NoSQL graph databases are. Section 3 introduces the criteria we propose for managing change in graph NoSQL datatabases. Section 4 discusses these criteria and their links, while Section 5 concludes and presents some parts of our further work.

2. Related Work

2.1 NoSQL Graph Databases

Graphs have been considered for many years by mathematicians and computer scientists. They represent entities and their relations, as nodes and vertices. The design of dedicated database engines goes back to the early 80's [2]. They were then surpassed by XML models, before getting back in the recent years, especially with NoSQL models. NoSQL models have emerged in the early 2000's and are often classified into several categories (key-value, column, document, graph NoSQL databases) [9, 14].

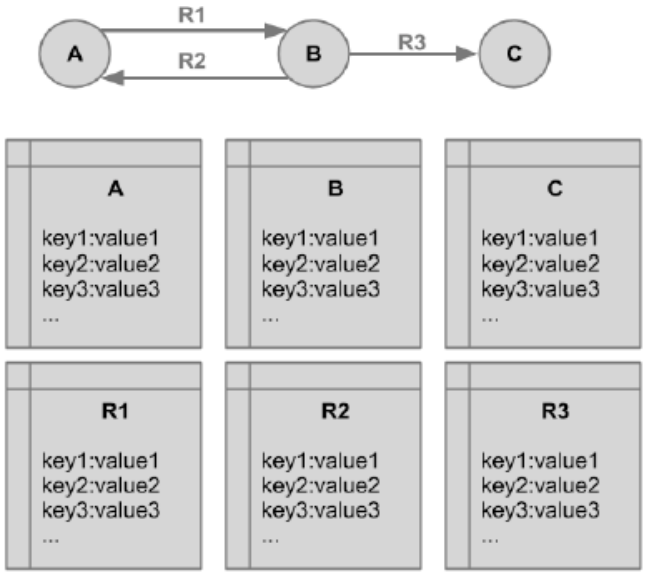

Figure 1: Example of Key-Value Properties on Nodes and Relationships

NoSQL graph databases [26] have been proposed to manage the large networks that can be found in all domains. When comparing NoSQL and relational models in the context of graph data, it is shown that NoSQL models overpass relational ones [15,27].

The current leader of NoSQL graph databases is Neo4j [5].

In NoSQL graph databases, entities and relationships are equipped with a set of properties that are represented by key-value pairs. Keys allow to retrieve in a very e#cient manner the associated values. Figure 1 displays an example of such key-value pairs both on nodes and relationships.

In this Figure, A, B and C are node identifiers and R1, R2, R3 are relationship identifiers. All these entities have properties represented as key-value pairs. The values can be of di#erent forms (numeric, strings, date, etc.), including collections. For instance, nodes may represent people and units from an organisation and which are linked via relationships and properties may store information on these people and units, as shown in Figure 2.

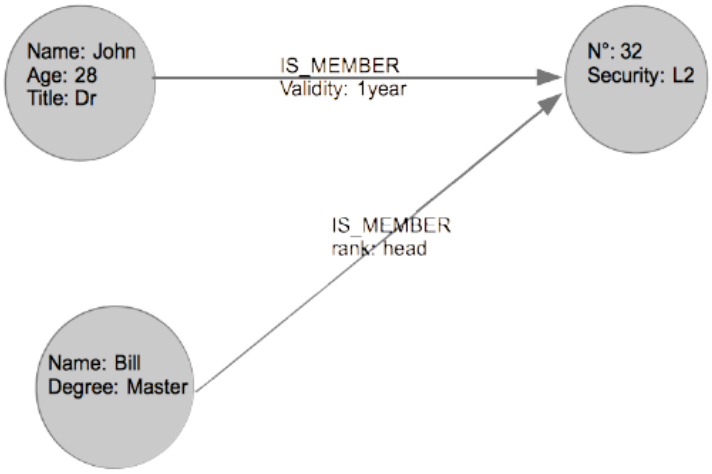

Figure 2: Database Example

The tools to traverse such data can be classified into three mains categories, depending on the level which queries are given at, from high-level declarative levels to low level:

- API: the data can be accessed through programmatic APIs (e.g., Java for orientDB, Neo4j databases);
- functional: the data can be accessed through functions that can be combined (e.g., gremlin language);
- declarative: the data can be accessed through a declarative query language (e.g., cypher for Neo4j databases).

Cypher can be seen as being close to SPARQL. It can indeed also manage pattern mining, close to an intuitive query by example manner. However, SPARQL has not been fully adopted by developers and has been designed for the particular data model of RDF. Cypher has thus be proposed as a new tool. It has been shown that the choice of the query tool impacts the performances [4].

We claim that these works must be extended in order to meet the requirements of history management for NoSQL graph databases. In the next section, we introduce change management concepts and related works about it.

2.2 Change Management

Change management can be viewed at several levels, both in terms of model being addressed and in terms of semantics. Regarding the latter point, managing changes can be simply seen at considering only two low-level operations (i.e., adding and deleting) which are the basis for all changes. It can also embed very sophisticated operations, namely high-level operations that cannot be listed in an exhaustive manner [17].

Change management has been considered in the relational model in many works for both modelling the data structures [11, 13, 22] and querying such databases, as introduced in temporal SQL [18, 28].

It has also been studied in the context of object-oriented databases [3]. In a more general manner, temporal databases have attracted many works and discussions [29].

Change management has been considered for semi-structured databases and XML [10, 19].

Regarding change management when dealing with graph data and ontologies, it has also attracted research as in [20, 23, 24]. In this framework, work have been proposed, mainly aiming at helping to discover trends and detect changes.

Moreover, works do not often provide implemented and scalable systems. When dealing with implementation strategies, [16] introduces an original model for temporal graph change study. [25] proposes an approach for e#fficiently computing graph queries (such as shortest path) through several snapshot graphs.

[12] explores the possibility to use NoSQL databases for processing large RDF repositories and shows the interest of such frameworks. In these works, authors focus on the indexing of historical information stored in successive snapshots in order to optimise query performance but do not provide a full system for storing and maintaining historical views.

By analyzing the different types of history management systems in databases, we have extracted some criteria that we believe a historisation management system on a graph databases system should have to work and more over be adopted in the NoSQL database framework.

For this purpose, we both rely on the very rich literature and on our knowledge in software architecture and implemented industrial systems. In the next section, we detail the main characteristics a system for NoSQL graph databases must fulfil to be considered as managing history.

3. HNTP Criteria

We aim at proposing a framework for representing historical NoSQL graph databases that allows to maintain and query the system. We thus propose to extend the model. In this section, we discuss the main criteria such a solution must fulfil.

The main criteria we propose are the HNTP ones, detailed below,

- H standing for History,
- N standing for Non-Intrusivity,
- T standing for Time-independence,
- P standing for Pluggability.

3.1 History

History refers to the maintenance of successive states of the system.

In NoSQL graph databases, changes can occur on both nodes and relationships, on the levels of their existence and of their properties. This can thus result in changes on both the level of the graph structure (by adding/deleting nodes and relations) and the level of values (by adding/deleting key-value information).

For example, Fig. 3 reports the following changes between two transactions, T1 and T2 representing the state of an organization:

- the age of the node has moved from 28 to 29;
- the node of entity number 32 has been deleted together with the two incoming relationships;
- the node of entity number 28 has been added together with a relationship from John who is acting as a member at the rank of Director in this entity;

- a relationship between John and Bill as been added as Friend.

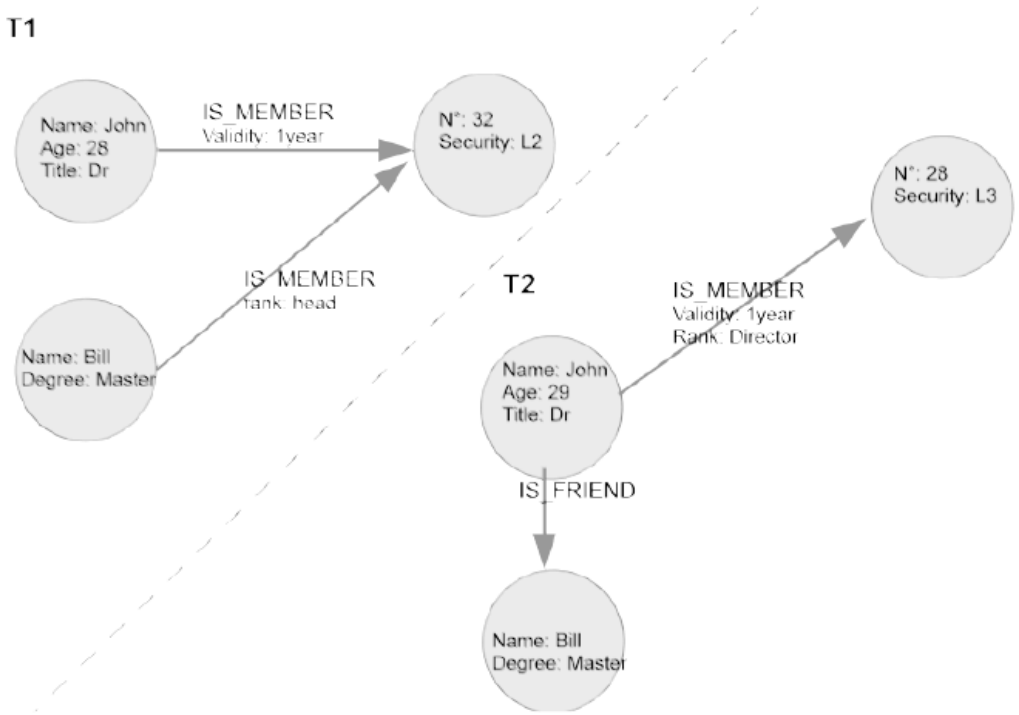

Figure 3: Update vs. Delete/Insert

For efficiency reasons, both at the semantic and performance levels, it is more suitable to avoid only keeping successive timestamped snapshots. We thus suggest that timelines are maintained to follow every node and relation. On the previous example, it is indeed not the same to consider that the node of the entity number 32 has been deleted and entity of number 28 has been added or to consider that the node of entity number 32 has changed to number 28, as described in Fig. 4.

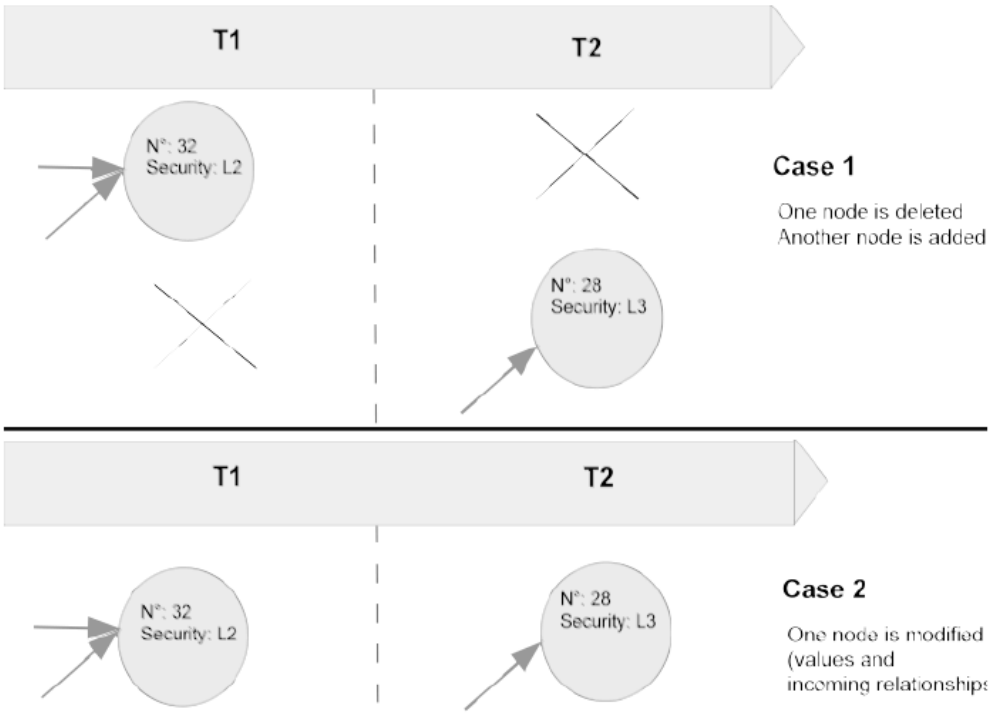

Figure 4: Maintaining Timelines

This question has already been studied in databases [1], but is still important in our context.

Such timelines are important. Managing unique identifiers in graph databases is a key challenge for this purpose. In Neo4j for instance, node identifiers are unique but the system reuses its internal identifiers when nodes and relationships are deleted, which means that we cannot yet easily rely on identifiers.

On top of such basic operations, high level changes must be considered. For instance, a system must provide a way to get the difference between two versions of a graph in time. Although it is not possible to list in an exhaustive manner all the high level operations, this type of history tracking relies on graph diff operations as studied in the field of graph theory and change detection.

To sum up, history must be maintained at the three levels of node, relationship and graph. The system should also provide a way to track context informations about every version (e.g., who, what, when, why).

This point is important as it is essential to answer to some legal queries, or to study the dynamic of networks.

3.2 Non-intrusivity

Although we aim at tracking successive states of the system, the system itself may not be historical by nature. For instance, when managing an organization, the important information for daily life is embedded in the current snapshot:

- who is managing whom?
- who is heading what?
- what is the position of X?
- who is working with Y?
- etc.

We thus aim at keeping the developers, designers and end-users out of the process of managing history.

For this purpose, we consider that the system must not interfere:

- neither at the technical level,
- nor at the conceptual level.

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 graphs conceptual structure.

3.3 Temporal Independence

Non-intrusivity guarantees that the users do not have to take care of the management of history.

The user must also be able to decide when changes must be tracked or not. For this purpose, the system must be able to run with and without history management.

We introduce thus the so-called temporal independence criterion. The more representative illustration of such a criterion is when the system runs for a while without tracking changes and must at some time start managing history.

3.4 Pluggability

As discussed in [8], the history can be either stored as a separate graph or as a subgraph, depending on the choices to maintain the current version in a smaller and faster system or to maintain the whole information (current and past versions) in a single equally-performant system.

Whatever the choice, the system should be distributed as a library or plugin, thus avoiding the management of history and avoiding to oblige developers to change existing source codes.
This library will be responsible for managing all events leading to an action of historization.

As described in the non-intrusivity criterion, the existing system must not be changed. The library will thus rely on hooks in the existing system so as to trigger actions in its inner code. Such hooks are commonly implemented in the NoSQL graph databases, as for instance the ones from Neo4j.

The links between the criteria are discussed in the next section.

4. Discussion

The criteria proposed above have been introduced in an implemented architecture described in [8].

Fig. 5 shows how successive versions of nodes, relationships and graph are maintained.

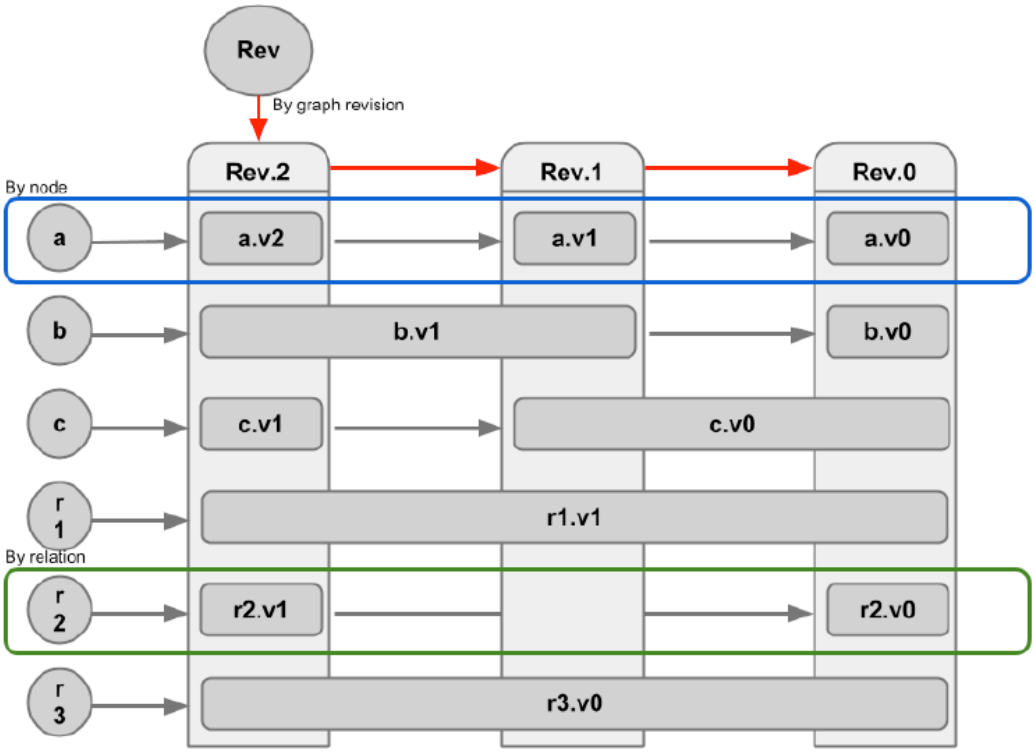

Figure 5: Maintaining Successive Versions of Objects

Every node and relation is displayed as a line in this Figure. For instance a is a node and r1 is a relationship.

There has been one initial version rev:0 and 2 successive states rev:1 and rev:2, all these 3 labels pointing on the whole graph version. The Rev node has only one outgoing relationship pointing out to the current version of the graph.
In this proposition, timelines are respected. Graph versions can be directly queried by following one red path, while nodes can be queried by following some blue path, and relationships by following some green path.

4.1 Links with Other Criteria

The management of history is not new, even for graph data. However, no system has yet been implemented in the specific context of NoSQL graph databases, making our proposition original.

NoSQL databases are indeed quite different from regular relational database engines and concepts. Data are modelled and indexed in a different manner.
ACID criteria have first been mentioned as being non mandatory in NoSQL systems, leading to the co-called CAP theorem (Consistency, Availability, Partition Tolerence) [6].

However, some current NoSQL engines are not restricted to CAP criteria.In particular the NoSQL Neo4j engine offer ACID properties.

As mentioned above, the HNTP criteria lead to a transparent system of historization. This transparency must thus not interfere with any other criterion such as those from the ACID and CAP concepts. They thus are compatible with ACID and CAP criteria.

4.2 Links between Criteria

Some of the HNTP criteria may appear as being linked or even redundant, which is not the case.

For instance, non-intrusivity and pluggability are not linked and may occur in a system in an independent manner (one without or with the other one).

However, history is mandatory and transparency should be guaranteed to the users at the condition that the three criteria N, T and P are fulfilled.

4.3 Queries

The H-History criterion guarantees that the information is kept to follow the graph successive versions from three points of view (node, relationship and graph).

On top of this, we argue that all types of queries can be run. Low-level queries will be easily answered by following basic operations on nodes or relationships. Such queries are for example:

- What are the successive Degrees of People X?
- Who were the friends of the friends of X last year?

High-level queries will be as well treated by looking down at the node and relationship level. Such queries may for instance aim at studying dynamic networks, to detect trends.

It should be noted that the fulfilment of the four HNTP criteria implies to manage history independently from the query types. It is indeed impossible to design a data structure devoted to historical queries because of the Non-intrusivity criterion.

Regarding NoSQL Neo4j graph databases, the impact on the three ways of querying the data are somehow comparable:

- either the languages (should it be declarative or functional, or even rely on programmatic functions through API) are extended ;
- or the user is aware of the implementation of history tracking in order to explicitly traverse historical data.

We recommend that the languages are extended in order to offer better performances with underlying efficient implementations and to fulfil the non-intrusivity criterion.

5. Conclusion

In this paper, we discuss the criteria a system must fulfill in order to manage history in NoSQL graph databases. Graphs are pervasive and established in various and numerous areas. Such data have attracted

much attention in the last years, especially because of the Web 2.0 (XML, social and media networks etc.) and the arriving Web 3.0 concepts. We focus in this paper on the management of change.

In further work, we aim at study the extension of query languages for traversing graph NoSql databases, including extensions to fuzzy queries.

References

[1] Serge Abiteboul and Victor Vianu. Procedural and declarative database update languages. In Proceedings of the seventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, PODS '88, pages 240{250, New York, NY, USA, 1988. ACM.

[2] Renzo Angles and Claudio Gutierrez. Survey of graph database models. ACM Comput. Surv., 40(1), 2008.

[3] J. Barnerjee, W. Kim, H. Kim, and H. Korth. Semantics and implementation of schema evolution in object-oriented databases. In Proceedings of the International Conference on Management of Data, 1987.

[4] Shalini Batra and Charu Tyagi. Comparative analysis of relational and graph databases. International Journal of Soft Computing and Engineering (IJSCE), 2(2):509-512, 2012.

[5] ThoughtWorks Technology Advisory Board. Technology radar, http://thoughtworks.fileburst.com/assets/technology-radar-may-2013.pdf, May 2013.

[6] E. A. Brewer. Towards robust distributed systems. In Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing, PODC00. ACM, 2000.

[7] P. Buneman, S. Khanna, K. Tajima, and W. Tan. Archiving scientific data. ACM TODS, 29(1):2{42, 2004.

[8] A. Castelltort and A. Laurent. Representing history in graph-oriented nosql databases: A versioning system. In Proc. of the Int. Conf. on Digital Information Management, 2013.

[9] Rick Cattell. Scalable SQL and NoSQL data stores. SIGMOD Record, 39(4):12-27, 2010.

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

[11] Sudarshan S. Chawathe and Hector Garcia-Molina. Meaningful change detection in structured data. In Proceedings of the 1997 ACM SIGMOD international conference on Management of data, SIGMOD'97, pages 26-37, New York, NY, USA, 1997. ACM.

[12] Philippe Cudr#e-Mauroux, Iliya Enchev, Sever Fundatureanu, Paul T. Groth, Albert Haque, Andreas Harth, Felix Leif Keppmann, Daniel P. Miranker, Juan Sequeda, and Marcin Wylot. Nosql databases for

rdf: An empirical evaluation. In Harith Alani, Lalana Kagal, Achille Fokoue, Paul T.Groth, Chris Biemann, Josiane Xavier Parreira, Lora Aroyo, Natasha F. Noy, Chris Welty, and Krzysztof Janowicz, editors, International Semantic Web Conference (2), volume 8219 of Lecture Notes in Computer Science, pages 310-325. Springer, 2013.

[13] C. Date, H. Darwen, and N. Lorentzos. Temporal data and the relational model. Elsevier, 2002.

[14] Jing Han, E. Haihong, Guan Le, and Jian Du. Survey on nosql database. In Proc. of the 6th International Conference on Pervasive Computing and Applications (ICPCA), pages 363-366, 2011.

[15] Florian Holzschuher and Ren#e Peinl. Performance of graph query languages: comparison of cypher, gremlin and native access in neo4j. In Proceedings of the Joint EDBT/ICDT 2013 Workshops, EDBT '13, pages 195-204, New York, NY, USA, 2013. ACM.

[16] Udayan Khurana and Amol Deshpande. Efficient snapshot retrieval over historical graph data. In ICDE, pages 997-1008, 2013.

[17] M. Klein. Change management for distributed ontologies. PhD thesis, Vrije University.

[18] Krishna Kulkarni and Jan-Eike Michels. Temporal features in sql:2011. SIGMOD Rec., 41(3):34-43, October 2012.

[19] A. Marian, S. Abiteboul, G. Cobena, and L. Mignet. Change-centric management of versions in an xml warehouse. In In Proceedings of the International Conference on Very Large Data Bases (VLDB 01), page 581-590, 2001.

[20] B. Motik. Representing and querying validity time in rdf and owl: A logic-based approach. In Proceedings of the International Semantic Web Conference (ISWC), 2010.

[21] M.S. Olivier. On metadata context in database forensics. Digital Investigation, 5(3-4):115{123, 2009.

[22] G. Ozsoyoglu and R.T. Snodgrass. Temporal and real-time databases: a survey. IEEE TKDE, (4), 1995.

[23] V. Papavasileiou, G. Flouris, I. Fundulaki, D. Kotzinos, and V. Christophides. High-level change detection in rdf(s) kbs. ACM Trans.Datab. Syst, 38(1), 2013.

[24] P. Plessers, O. De Troyer, and S. Casteleyn. Understanding ontology evolution: A change detection approach. Web Semantics: Science, Services and Agents on the WWW: Selected Papers from the International Semantic Web Conference, 2007.

[25] Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. On querying historical evolving graph sequences. PVLDB, 4(11):726-737, 2011.

[26] Ian Robinson, Jim Webber, and Emil Eifrem. Graph Databases. O'Reilly, to appear.

[27] Marko A. Rodriguez and Peter Neubauer. The graph traversal pattern. CoRR, abs/1004.1001, 2010.

[28] Richard T. Snodgrass, editor. The TSQL2 Temporal Query Language. Kluwer, 1995.

[29] Abdullah Uz Tansel, James Clifford, Shashi Gadia, Sushil Jajodia, Arie Segev, and Richard Snodgrass, editors. Temporal databases: theory, design, and implementation. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1993.

[30] Jun Yang and Jennifer Widom. Maintaining temporal views over non- temporal information sources for data warehousing. In Hans-Jrg Schek, Flix Saltor, Isidro Ramos, and Gustavo Alonso, editors, EDBT, volume 1377 of Lecture Notes in Computer Science, pages 389{403. Springer, 1998.