Querying heterogeneous data in graph-oriented NoSQL systems

Mohammed El Malki, Hamdi Ben Hamadou, Max Chevalier, André Péninou, Olivier Teste

Querying Heterogeneous Data in Graph-Oriented NoSQL Systems

Mohammed El Malki[2(✉)], Hamdi Ben Hamadou[1], Max Chevalier[1], André Péninou[2], and Olivier Teste[2(✉)]

[1] Université Toulouse 3 Paul Sabatier, IRIT (CNRS/UMR5505), Toulouse, France

[2] Université Toulouse 2 Jean Jaurès, UT2C, IRIT (CNRS/UMR5505), Toulouse, France

{elmalki, prenom.nom, teste}@irit.fr

Abstract. NoSQL systems are based on a "*schemaless*" approach that not does require schema specification before writing data, allowing a wide variety of representations. This flexibility leads to a large volume of heterogeneous data, which makes their querying more complex for users who are compelled to know the different forms (i.e. the different schemas) of these data. This paper addresses this issue focusing on simplifying the heterogeneous data querying. Our work specially concerns graph-oriented NoSQL systems.

Keywords:
Information systems · NoSQL data stores
Graph-oriented databases · Heterogeneous data querying

1 Introduction

The growing usage of "Not-only-SQL" storage systems, referred as NoSQL, has given ability to efficiently handle large volume of data [4]. Graph-oriented systems are among the most increasingly used NoSQL approaches. A special attention has focused on how to model data in the form of graphs [4]. In this approach, data is represented as nodes, edges and attributes, which allows the modelling of different interactions between data. Graphs modeling is ubiquitous in most social networks, semantic web and bioscience (protein interactions …) applications.

Graph-oriented systems belong within the "*schemaless*" framework [2, 7], that consists in writing data without any prior schema restrictions; i.e., each node and each edge have its own set of attributes, thus allowing a wide variety of representations [6]. This flexibility generates heterogeneous data, and makes their interrogation more complex for users, who are compelled to know the different schemas of the manipulated data. This paper addresses this issue and consider a straightforward approach for the interrogation of heterogeneous data into NoSQL graph-oriented systems. The proposed approach aims at simplifying the querying of heterogeneous data by limiting the negative impact of their heterogeneity and leads to make this heterogeneity "transparent" for users.

This paper is organized as follows. We highlight in Sect. 2 the issues addressed in this paper. Section 3 gives an overview of the related work. We present in Sect. 4 our

https://doi.org/10.1007/978-3-319-98539-8_22

approach for heterogeneous data interrogation in order to simplify data querying. The results of the experimental evaluation of our approach are presented in Sect. 5.

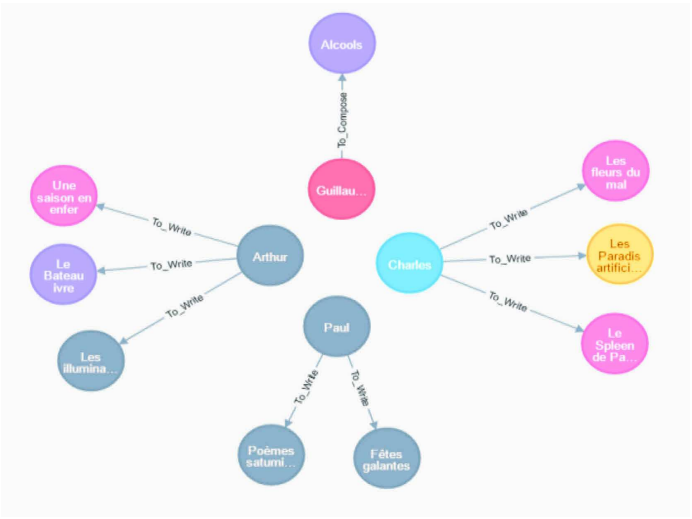
2 Problem Modeling

2.1 Notations

The data modeling in NoSQL graph-oriented systems consists in representing the database as a graph. Figure 1 illustrates an example of a simple graph $G = (V, E, \gamma)$ where $V = \{u_1, \ldots u_n\}$ denotes the set of nodes, $E = \{e_1, \ldots e_m\}$ is the set of edges connecting one node to another and $\gamma : E \rightarrow V \times V$ represents a function that determines the nodes pairs connected by the edges.

The different nodes are described with textual format as presented below:

We can notice in Fig. 1 that the name of the edges can vary (either `To_Write` or `To_Compose`). Similarly, the graph's nodes and their attributes can be heterogeneous.



Textual format

```
u1:Author{
        firstname:'Charles',
        lastname:'Baudelaire,
        birth_date:1821,
        date_of_death:1867}

u3:Author:Poet{
        firstname:'Guillaume,
        lastname:'Apollinair,
        birth:1880,
        death:1918}

u2:Writer{
        firstname:'Paul',
        lastname:'Verlaine',
        birth:1844}
```

Fig. 1. A graph example

2.2 Heterogeneity Classes

The heterogeneity can be considered from different perspectives [10] depending on the structural elements composing the graph:

Structural heterogeneity refers to data represented by variable structural elements.

Syntactic heterogeneity refers to the fact that a structural element can be refered to a variable way; *e.g., the attributes 'birth_date' and 'birth' belong to different nodes but both refer to an author's date of birth.*

Semantic heterogeneity defines the problem of two different elements corresponding to the same data, or inversely when a single element is related to two different data; *e.g., the edges 'To_Write' and 'To_Compose' both have the same meaning*. In this article, we address the different heterogeneity classes discussed in this section.

2.3 Querying Heterogeneous NoSQL Graph

We use Cypher language [4] offer by Neo4j NoSQL graph database to illustrate the problem of querying heterogeneous graphs.

Our model is based on a theoretical foundations, called algebraic core, which ensures the genericity of the approach. In this paper, we only examine the operators defined for projection and selection. This set of elementary operators compose the algebraic core.

Therefore, using naively Cypher language in the context of heterogeneous graphs may leads to produce wrong analyses and to take decisions on incomplete data. In this paper, we propose an approach that allows the user to express a query using the set of attributes, without considering the structural, syntactic, and semantic differences, without changing the original structures of the graphs. Our solution grants the possibility to obtain a "complete" results, without having to explicitly manage the various heterogeneity aspects.

Table 1. Querying heterogeneous data problem

	Projection. *Get the lastname, the firstname and the book's title of the* ***authors***	**Selection**. *Obtain the* ***books'*** *titles of the author Baudelaire*
Result	`{firstname:'Charles', lastname:'Baudelaire', title:'Les Paradis artificiels'}` `{firstname:'Charles', lastname:'Baudelaire', title:'Le Spleen de Paris'}` `{firstname:'Charles', lastname:'Baudelaire', title:'Les fleurs du mal'}` `{firstname:'Guillaume', lastname:'Apollinaire', title:'Alcools'}`	`{title:Paradis artificiels'}`
Missing	`{firstname:PAUL, lastname:VERLAINE, title:'Les saturnies}`	`{title:'Le Spleen de Paris'}` `{title:'Les fleurs du mal'}`
	It is not included in the results because the node's type (i.e. label) is not **`Author`** but rather **`Writer`**.	It is not included because labeled as **`Book`**

3 Related Work

The problem of querying heterogeneous data is an active research domain studied in several contexts such as data-lake, federated database [15], data integration, schema matching [16]. We classify the state-of-the-art as follows.

Schema Integration. The schema integration process is an intermediary step to facilitate the query execution. In [16] the authors present a survey on techniques used to

automate the schema integration process. The schema integration techniques may lead to data duplication and original structure loss, which affects the support of legacy applications.

Physical Re-factorization. Several works are conducted to enable querying data without any prior schema restrictions. Generally, they propose to flatten data into a relational form [11, 17, 18]. SQL queries are formulated based on relational views built on top of the inferred structures. This strategy suggests performing heavy physical re-factorization. Hence, this process requires additional resources such as external relational database and extra efforts to learn the new relational schema whenever new schemas are inserted.

Schema Discovery. Other works propose to infer implicit schemas. The idea is to give an overview of the different elements present in the integrated data [19, 21]. In [12] the authors propose summarizing all schema under a skeleton to discover the existence of fields or sub-schema. In [20] the authors suggest extracting all structures to help developers while designing their applications. The limitation with such logical view is the need to manual process while building queries by including all attributes and their corresponding paths.

Others works suggest resolving the heterogeneity problem by working on the query side. Query rewriting is a strategy to rewrite an input query into several derivations to overcome the heterogeneity [14, 24, 25]. Most of works are designed in the context of the relational database where heterogeneity is usually restricted to the lexical level only. In NoSQL, the first papers focused on document stores and using another query language that cannot be applied to oriented graph systems [22].

4 EasyGraphQuery: Query Rewriting Engine

EasyGraphQuery differs from the conventional systems, which require a prior knowledge of the different schemas to formulate adequate queries. *EasyGraphQuery* takes for input the user's query, rewrites it using the dictionary to eventually extract similar attributes and run it into Neo4J. Figure 2 gives an overview of the *EasyGraphQuery* architecture.

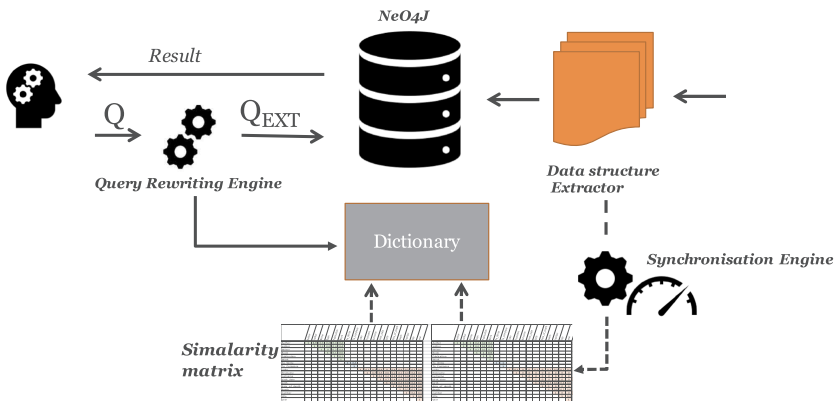


Fig. 2. The *EasyGraphQuery* architecture

The creation of the dictionary is done automatically when inserting data and is updated with each new insertion or update of the already stored data. For performance reasons, the update is continuously operated in the background. By keeping the dates of the last modifications in files, that are independent of the similarity matrix and the dictionary, it is possible to consult the updates status without affecting the queries being executed.

4.1 Data and Dictionary Modeling

Definition 1. A graph G is defined by (V, E, γ)
- $V = \{u_1,\ldots, u_N\}$ is the graph's nodes set ;
- $E = \{e_1,\ldots, e_M\}$ is the graph's edges set ;
- $\gamma : E \rightarrow V$ x V is the function that associates each edge with the connected vertices.

We denote $\mathcal{L} = \{l_1, \ldots, l_L\}$ a set of terms indicating the different nodes labels and possible relations.

Definition 2. A node u_i is defined as (L_i, S_i)
- $L_i \subseteq \mathcal{L}$ is the set of labels to which the node belongs;
- $S_i = \{a_{i,1},\ldots,a_{i,n_i}\}$ is the node schema composed of a set of attributes

We denote $S_V = \bigcup_{i=1}^{i=N} \left(\bigcup_{l_k \in L_i} \bigcup_{a_{i,j} \in S_i} l_k.a_{i,j} \right)$ the graph node schema.

Definition 3. an edge e_i is defined as (l_i, S_i, $u_{i,1}$, $u_{i,2}$)
- $l_i \in \mathcal{L}$ is the label to which the edge belongs;
- $S_i = \{a_{i,1},\ldots,a_{i,n_i}\}$ is the edge's schema composed of a set of attributes;
- $u_{i,1}$ et $u_{i,2}$ are the source and the target nodes connected by edge; $\gamma(e_i) = \{(u_{i,1}, u_{i,2})\}$.

We denote $S_E = \bigcup_{i=1}^{i=M} \left(\bigcup_{a_{i,j} \in S_i} l_i.a_{i,j} \right)$ the graph edges schema. Thus, $S_G = S_N \cup S_M$ is the graph's attributes schema.

We define $\mathcal{L}_G = \left(\bigcup_{i=1}^{i=N} \mathcal{L}_i \right) \cup \left(\bigcup_{i=1}^{i=M} l_i \right)$.

Example. Let us consider the graph illustrated in Fig. 1.

$V = \{ u_1, \ldots, u_{13} \}$; $E = \{ e_1, \ldots, e_9 \}$;
- $\gamma = \{ (e_1,(u_1, u_5)), (e_2,(u_1, u_6)), (e_3,(u_1, u_7)), (e_4,(u_2, u_8)), (e_5,(u_2, u_9)), (e_6,(u_3, u_{10})), (e_7,(u_4, u_{11})), (e_8,(u_4, u_{12})), (e_9,(u_4, u_{13}))\}$.

Node u_1 = ({*Author*}, {*firstname*, *lastname*, *birth_date*, *date_of_death*}) and node u_7 = ({*Book*}, {*number*, *title*, *year*}). Edge e_3 = (*Book*, { }, u_1, u_7).

In order to consider the different heterogeneity aspects of the graph (structural, syntactic and semantic), we introduce two data dictionaries that allow to determine for each element of the graph (label, attribute), similar elements.

Definition 4. The data dictionary $dict_{label}$ is defined by

$$dict_{label} = \{ (l_i, \nabla_i) \}$$

- $l_i \in \mathcal{L}_G$ is a graph's label;
- $\nabla_i = \bigcup_{\forall l_j \in \mathcal{L}_G | sim(l_i, l_j) \geq \delta} l_j \subseteq \mathcal{L}_G$ is the set of similar labels. The similarity metric, called *sim*, measures a normalized score ranged in [0..1] expressing the degree of similarity (the more l_i and l_j are similar, the closer the score is to 1). $\delta \in$ [0..1] is the threshold according to which labels l_i and l_j are considered similar.

Definition 5. The data dictionary $dict_{attribut}$ is defined as

$$dict_{attribut} = \{ (l_i.a_{i,k}, \Delta_{i,k}) \}$$

- $l_i.a_{i,k} \in S_G$ is a graph's attribute ;
- $\Delta_{i,k} = \bigcup_{\forall l_j.a_{j,l} \in S_G | sim(a_{i,k}, a_{j,l}) \geq \delta} l_j.a_{j,l} \subseteq S_G$ is the set of similar attributes.

Similarity is calculated between the eventual heterogeneous elements of the graph. In this paper, we only consider the labels, attributes of nodes and edges. The heterogeneity aspects considered are structural, syntactic and semantic. Two matrices are constructed to determine the similarities between the graph's elements: the syntactic similarity matrix is based on the *Leivenshtein* measure while the semantic similarity matrix is based on the *Lin* measure. We do not detail the pre-processing applied during multi-terms like `To_Write` or `birth_date`, when calculating matrices. We can consider extending the approach with other similarity measures, and improve the process by a weighted combination of these various measures [8, 10].

In this paper, we only consider the attributes structural heterogeneity. The labels structural heterogeneity is not examined. That means that an attribute can be located at various positions in the graph, marked by the labels that prefix the **property**.

4.2 The Algebraic Core of Operators

The interrogation is based on a set of elementary operators forming a closed minimum core. We denote G_{in} the queried graph and G_{out} the resulting graph. These elementary operators allow projection and selection (restriction) operations.

Definition 6. Projection allows reducing the graph to the structural elements specified in the structural pattern along with the projected list of attributes

$$_{Pattern}\pi_{Attribute}(G_{in}) = G_{out}$$

- *Pattern* is a path defined as l_0–l_1–…–l_p where every $l_i \in \mathcal{L}_G$ stands for the class of a node or an edge.
- *Attribute* is a set (possibly empty) of attributes $l_i.a_{i,k}$ where $l_i \in Pattern$ and $a_{i,k} \in S_i$.

Definition 7. The selection operator restricts the structures' elements to only those satisfying a selection predicate; we denote:

$$_{Pattern}\sigma_{Predicate}(G_{in}) = G_{out}$$

– *Pattern* is a path defined as l_0–l_1–…–l_p where every $l_i \in \mathcal{L}_G$ stands for the class of a node or an edge.
– *Predicate* is a selection condition. A simple predicate is expressed as $l_i.a_{i,k}\ \omega_k\ v_k$ where $a_{i,k} \subseteq S_i$ is an attribute, $\omega_k \in \{=;>;<;\neq;\geq;\leq\}$ is a comparison operator, and v_i is a value. Predicates can be combined with operands $\Omega = \{\vee, \wedge, \neg\}$ forming a complex predicate.

The predicates of a complex selection, combining several predicates, are represented in their conjunctive normal form: $Predicate = \bigwedge_x \left(\bigvee_y p_{x,y} \right)$.

Example. Let us consider the queries from Sect. 2.3. We can express these queries with an algebraic representation (internal) as follows:

Projection. « `Get the lastname, the firstname, and the title of the books written by the authors` »: $(\ _{\texttt{Author--l}}\pi_{\texttt{Author.firstname,Author.lastname,l.title}}\ )$.

We present the obtained results in the Table 1. When the attributes are projected, the identifiers of the nodes (u_i) and the edges (e_i) are lost; this breaks the closure principle of the algebraic core, thus not allowing to combine these results with a new operation.

Projection and Selection. « Get the titles of the books for author having name = 'Baudelaire' »

$$_{\texttt{Author--l}}\pi_{\texttt{Author.firstname,Author.lastname,l.title}}\ (\ _{\texttt{Author--l}}\sigma_{\texttt{Author.firstname='Charles'}\wedge\texttt{Author.lastname='Baudelaire'}}\ )$$

The obtained results are given in the Table 2.

Table 2. Results of the selection and projection operations combination.

G_{out}
`firstname:'Charles',lastname:'Baudelaire',title:'Le Spleen de Paris'`
`firstname:'Charles',lastname:'Baudelaire',title:'Les fleurs du mal'`

The use of this internal representation, of the interrogation operations on graphs, does not support the heterogeneity of the graph's elements. Therefore, these queries' results remain incomplete regarding the information represented in the graph.

We present, in the following, the rewriting process of these internal queries allowing to obtain a complete result that is transparent to the user and dynamic (without data transformation).

4.3 Queries Rewriting Algorithm

Neo4J does not provide native operators to manage the graphs heterogeneity; *e.g.*, the `match` operation is very case-sensitive and does not automatically allow comparisons of labels and attributes that are syntactically, semantically or structurally similar. This sensitivity leads to ignore several data that are relevant to the result. Our approach aims at assisting users with querying, by automatic query reformulation. This process makes use of the dictionary and indirectly the similarity matrix to reformulate the query by determining similar elements (nodes, edges, and attributes). The following algorithm describes the automatic rewriting process of the user's query.

The function *exists*(A, L) verifies if L includes the pattern constituted from the labels resulting from A. The union operation, denoted $\cup$, allows unifying two graphs

$$G_1 \cup G_2 = G_{out} | V_{out} = V_1 \cup V_2; E_{out} = E_1 \cup E_2; \gamma_{out} : E_{out} \rightarrow V_{out} \, \mathrm{x} \, V_{out} | e_{out} \in \gamma_1 \mathrm{v} \, e_{out} \in \gamma_2$$

Algorithm : Automatic extension of the user's query

Input : Q
Output : Q_{ext}

begin
  $Q_{ext} \leftarrow id$
  foreach $q_x \in Q$ **do**
    switch q_x **do**
      case ${}_{Pattern}\pi_{Attribute}$ **do** *// projection*
        $L_{ext} \leftarrow \prod_{i=1}^{p} \nabla_i$
        $A_{ext} \leftarrow$
        $q_{ext} \leftarrow id$
        foreach $L \in L_{ext}$ **do**
          foreach $A \in A_{ext}$ **do**
            if *exists*(A,L) **then** $q_{ext} \leftarrow q_{ext} \cup {}_{L}\pi_{A}$
            end
          end
        end
        $Q_{ext} \leftarrow Q_{ext}$ o (q_{ext})
      end
      case ${}_{Pattern}\sigma_{Predicate}$ **do** *// selection*
        $L_{ext} \leftarrow \prod_{i=1}^{p} \nabla_i$
        $P_{ext} \leftarrow \wedge_x \left(\vee_y \left(\vee_{a_{l,k} \in \Delta_{x,y}} l_x . a_{l,k} \varpi_{l,k} v_{l,k} \right) \right)$
        $q_{ext} \leftarrow id$
        foreach $L \in L_{ext}$ **do**
          foreach $P \in P_{ext}$ **do**
            if *exists*(P,L) **then** $q_{ext} \leftarrow q_{ext} \cup {}_{L}\sigma_{P}$
            end
          end
        end
        $Q_{ext} \leftarrow Q_{ext}$ o (q_{ext})
      end
    end
  end
end.

Example. Let us consider the following query

$$\texttt{Author}\text{--}1\pi_{\texttt{Author.firstname,Author.lastname,1.title}}(\texttt{Author}\text{--}1\sigma_{\texttt{Author.firstname='Charles'}\wedge\texttt{Author.lastname='Baudelaire'}})$$

The projection operator is rewritten according to the different similar labels of the pattern, ∇_{Author} = {Author, author, Writer} and ∇_{Book} = {Book, book, Publication}, and the different similar attributes, $\Delta_{Author.firstname}$ = {Author.firstname, Writer.firstname, author.firstname} et $\Delta_{Author.lastname}$ = {Author.lastname, Writer.lastname, author.lastname}.

The algorithm constructs the following sets, according to which the operator is rewritten.

L_{ext} = { Author, author, Writer } x {} x { Book, book, Publication }
= { {Author,,Book}, {Author,,book}, {Author,,Publication}, {Writer,,Book}, {Writer,,book}, {Writer,,Publication}, {author,,Book}, {author,,book}, {author,,Publication} }

A_{ext} = { Author.firstname, Writer.firstname, author.firstname } x { Author.lastname, Writer.lastname, author.lastname } x { Book.title, book.title, Publication.title }
= { {Author.firstname,Author.lastname,Book.title}, ..., {author.firstname,author.lastname,Publication.title} }

The selection operator is rewritten according to the different labels of the selection pattern, ∇_{Author} = {Author, author, Writer}, and to the different similar attributes of the predicate, $\Delta_{Author.lastname}$ = {Author.firstname, Writer.firstname, author.firstname} and $\Delta_{Author.lastname}$ = {Author.lastname, Writer.lastname, author.lastname}.

Then, the operator is rewritten according to the set constructed by the algorithm.

L_{ext} = { Author, author, Writer } x {} x { Book, book, Publication } = { {Author,,Book}, {Author,,book}, {Author,,Publication}, {Writer,,Book}, {Writer,,book}, {Writer,,Publication}, {author,,Book}, {author,,book}, {author,,Publication} }

P_{ext} = { { Author.firstname='Charles', Writer.firstname='Charles', author.firstname='Charles' }, { Author.lastname='Baudelaire', Writer.lastname='Baudelaire', author.lastname='Baudelaire' } }

5 Experiments

We use the *EasyGraphQuery* tool to evaluate the query rewriting algorithm proposed in this article as well as the construction of the defined dictionary.

Dataset. To validate our approach, we consider ontology data because of their strong structural heterogeneity. We used the *Conference Track* collection made available by OAEI 2017[1]. These ontologies lack instances; so we had to generate synthetic instances. The goal is to evaluate the cost of interrogation; the generation and the loading times are not evaluated [26–28].

Tests Environment. We use a cluster composed of a machine (i5-4 core, 8Go RAM, 2To hard drive, 1 Gb/s network) in which we have installed a Neo4J instance – version 3.2.

The Queries Set. We defined a set of 10 queries: three queries for selection, three queries for projection, and four queries to evaluate the selection-projection combination. The set of queries based on the different comparison operators supported by Cypher language. We employed the classical comparison operators, i.e. {< , > , $\leq$, $\geq$, = ,} for numerical values as well as classical logical operators, i.e. {and, or, exists} between query predicates. Also, we employed a regular expression to deal with string values.

5.1 Setting up the Dictionary and the Similarity Matrix

In these experiments, we study the time needed to create and update the similarity dictionary. Table 3 shows the maintenance time of the dictionary as the ontologies are brought in. The results are clearly influenced by the number of elements (labels, edges, attributes) already present in the graph. Indeed, the log file is regularly analyzed by our parser, but it is not cleaned at each pass.

Table 3. Maintenance time of the syntactical dictionary according to the number of ontologies

Number of ontologies	2	4	6	8
Creation/update time (in seconds)	1.3 s	4.2 s	13.5 s	18.7 s
The dictionary size (KB)	2.7	2.9	3.4	3.5
the parsed logs file size (KB)	1333	14534	17602	21265

5.2 Evaluation of the Query Rewriting Module

In this experiment, we study the additional cost of our proposed approach, i.e. an interrogation with a rewritten query via our similarity algorithm, compared to the cost of a non-rewritten query, called initial query. We also compare the cost of the reformulated query against the sum of the costs of the subqueries, obtained from the decomposition of the reformulated queries.

The Table 4 reports the execution time of the rewritten queries, the initial queries, and the subquery cumulative execution times. A first comparison addresses the execution time of the rewritten queries and the cumulative execution time of the

[1] http://oaei.ontologymatching.org/2017/

sub-queries. We can note that the execution time of our approach is, at worst, equal to the cumulative execution time of sub-queries, and it can go up to 2 times faster (in the case of combined queries, for example in the case of our dataset). On the other hand, it is at best equal to the execution time of an initial query.

Table 4. Comparison of the execution time (in seconds) of the reformulated queries and the initial queries (without reformulation)

		Reformulated query	Initial query	Cumulative resulting queries
Projection	Q1.1	316	222	316
	Q1.2	0.160	0.008	0,166
	Q1.3	0.027	0.0013	0.027
Selection	Q2.1	4.05	2.98	4.8
	Q2.2	0.77	0.77	0.85
	Q2.3	2.34	1.73	3.73
Combination (Projection - Selection)	Q3.1	0.2734	0.2082	0.4062
	Q3.2	0.0055	0.0059	0.0073
	Q3.3	0.434	0.0431	0.9342
	Q3.4	0.324	0.324	0.659

To get a deeper understanding of these results, we have plotted the execution of our queries. For example, during the execution of the query Q1.2 where we can notice that during a reformulated query (where our algorithm uses the operator 'Union'), Neo4J starts the execution of the two '`Match`' simultaneously; and the union of the two results is consolidated only after the completion of the last '`Match`' (the one with the most rows). More precisely, in this projection query Q2.1, two types of labels are evaluated: the first corresponds to the label of the initial query, which processes 35054 lines; the second corresponds to the label added by our rewriting algorithm and which deals with 10000 lines. The number of lines explains the results illustrated in Table 2 and shows why our solution is at most equal to the execution time of the slowest sub-query and at best is equal to the initial query.

6 Conclusion

In this paper, we have defined an approach based on the construction of data similarity dictionaries allowing a rewriting of the users' queries without transforming the stored data. Our method is able to overcome the interrogation problem caused by the data heterogeneity in graph-oriented NoSQL storage systems. Our approach computes for each attribute, the set of its similar attributes (syntactic, semantic, and structural heterogeneity) and it transparently rewrites users' queries. Rewritten queries allow to enrich initial queries and return the complete set of data.

As a perspective for this work, we intend to extend our mechanisms to support more heterogeneity aspects; for example, consider the entities heterogeneity. We will

also expand the algebraic core of operators supported by the rewrite engine; for example, by integrating aggregation operations.

References

1. Beyer, K.S., Ercegovac, V., Gemulla, R., Balmin, A., Eltabakh, M., Kanne, C.-C., Ozcan, F., Shekita, E.J.: Jaql. In: Proceedings of VLDB Conference (2011)
2. Radic, D.: Influence of schemaless approach on database authorization. In: Hadžikadić, M., Avdaković, S. (eds.) IAT 2017. LNNS, vol. 28, pp. 243–248. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-71321-2_21
3. Floratou, A., Teletia, N., DeWitt, D.J., Patel, J.M., Zhang, D.: Can the elephants handle the NoSQL onslaught? Proc. VLDB Endow. **5**(12), 1712–1723 (2012)
4. Holzschuher, F., Peinl, R.: Performance of graph query languages: comparison of cypher, gremlin and native access in Neo4J. In: EDBT/ICDT (2013)
5. Getoor, L., Machanavajjhala, A.: Entity resolution: theory, practice & open challenges. VLDB Endow. **5**(12), 2018–2019 (2012)
6. Chevalier, M., El Malki, M., Kopliku, A., Teste, O., Tournier, R.: How can we implement a multidimensional data warehouse using NoSQL? In: Hammoudi, S., Maciaszek, L., Teniente, E., Camp, O., Cordeiro, J. (eds.) ICEIS 2015. LNBIP, vol. 241, pp. 108–130. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-29133-8_6
7. Cattell, R.: Scalable SQL and NoSQL data stores. SIGMOD Rec. **39**(4), 12–27 (2011)
8. Megdiche, I., Teste, O., Trojahn dos Santos, C.: An extensible linear approach for holistic ontology matching. In: Groth, P., Simperl, E., Gray, A., Sabou, M., Krötzsch, M., Lecue, F., Flöck, F., Gil, Y. (eds.) ISWC 2016. LNCS, vol. 9981, pp. 393–410. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46523-4_24
9. Melnik, S., Garcia-Molina, H., Rahm, E.: Similarity flooding: a versatile graph matching algorithm and its application to schema matching, pp. 117–128 (2002)
10. Shvaiko, P., Euzenat, J.: A Survey of schema-based matching approaches. In: Spaccapietra, S. (ed.) Journal on Data Semantics IV. LNCS, vol. 3730, pp. 146–171. Springer, Heidelberg (2005). https://doi.org/10.1007/11603412_5
11. Tahara, D., Diamond, T., Abadi, D.J.: Sinew: a SQL system for multi-structured data. In: 2014 SIGMOD, pp. 815–826. ACM (2014)
12. Wang, L., Zhang, S., Shi, J., Jiao, L., Hassanzadeh, O., Zou, J., Wangz, C.: Schema management for document stores. Proc. VLDB Endow. **8**(9), 922–933 (2015)
13. Lin, C., Wang, J., Rong, C.: Towards heterogeneous keyword search. In: Proceedings of the ACM Turing 50th Celebration Conference-China, p. 46. ACM (2017)
14. Papakonstantinou, Y., Vassalos, V.: Query rewriting for semistructured data. In: ACM SIGMOD Record, vol. 28, pp. 455–466. ACM (1999)
15. Sheth, A.P., Larson, J.A.: Federated database systems for managing distributed, heterogeneous, and autonomous databases. ACM Comput. Surv. (CSUR) **22**(3), 183–236 (1990)
16. Rahm, E., Bernstein, P.A.: A survey of approaches to automatic schema matching. VLDB J. **10**(4), 334–350 (2001)
17. Chasseur, C., Li, Y., Patel, J.M.: Enabling JSON document stores in relational systems. In: WebDB, vol. 13, pp. 14–15 (2013)
18. DiScala, M., Abadi, D.J.: Automatic generation of normalized relational schemas from nested keyvalue data. In: Proceedings of the 2016 ICM, pp. 295–310. ACM (2016)
19. Baazizi, M.-A., Lahmar, H.B., Colazzo, D., Ghelli, G., Sartiani, C.: Schema inference for massive JSON datasets. In: EDBT (2017)

20. Herrero, V., Abelló, A., Romero, O.: NOSQL design for analytical workloads: variability matters. In: Comyn-Wattiau, I., Tanaka, K., Song, I.-Y., Yamamoto, S., Saeki, M. (eds.) ER 2016. LNCS, vol. 9974, pp. 50–64. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46397-1_4
21. Sevilla Ruiz, D., Morales, S.F., García Molina, J.: Inferring versioned schemas from NoSQL databases and its applications. In: Johannesson, P., Lee, M.L., Liddle, Stephen W., Opdahl, Andreas L., López, Ó.P. (eds.) ER 2015. LNCS, vol. 9381, pp. 467–480. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25264-3_35
22. Ben Hamadou, H., Ghozzi, F., Péninou, A., Teste, O.: Towards schema-independent querying on document data stores. In: DOLAP 2018 (2018)
23. Clark, J., DeRose, S., et al.: XML path language (XPath) version 1.0 (1999)
24. Boag, S., Chamberlin, D., Fernandez, M.F., Florescu, D., Robie, J., Simeon, J., Stefanescu, M.: XQuery 1.0: an XML query language (2002)
25. Bourhis, P., Reutter, J.L., Suarez, F., Vrgoč, D.: JSON: data model, query languages and schema specification. In: SIGMOD, pp. 123–135. ACM (2017)
26. Chevalier, M., El Malki, M., Kopliku, A., Teste, O., Tournier, R.: Document-oriented data warehouses: models and extended cuboids. In: RCIS 2016, pp. 1–11 (2016)
27. Chevalier, M., Malki, M.E., Kopliku, A., Teste, O., Tournier, R.: implementation of multidimensional databases in column-oriented NoSQL systems. In: Morzy, T., Valduriez, P., Bellatreche, L. (eds.) ADBIS 2015. LNCS, vol. 9282, pp. 79–91. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23135-8_6
28. El Malki, M., Ben Hamadou, H., El Malki, N., Kopliku, A.: MPT: suite tools to support performance tuning in NoSQL systems. In: CEIS 2018 (2018)