Data Migration in Graph-oriented Databases

Soumaya Boukettaya[1,3], Ahlem Nabli[1,2], Faiez Gargouri[1]

Sfax University, MIRACL Laboratory,
Institute of Computer Science and Multimedia,
Tunisia

Al-Baha University, Faculty of computer sciences and information technology,
Kingdom of Saudi Arabia

Faculty of Economic Sciences and Management, Sfax University,
Tunisia

faiez.gargouri@isims.usf.tn,soumayaboukettaya@gmail.com,
ahlem.nabli@fss.usf.tn

Abstract. Data is expanding at a rapid pace these days, and dealing with it has become incredibly challenging. Since they allow for the storage of various data structures, NoSQL graph databases are becoming more popular. Nonetheless, due to their schema-less nature, improper data migration and manipulation during the query phase might result in significant data loss. This paper deals with data migration within NoSQL graph databases in which we propose a graph matching algorithm based on similarity measures. We also adopt a lazy data migration approach to ensure a low cost of data migration and avoid critical data loss.

Keywords: Graph databases, data migration, similarity measures, graph matching, nodes similarity, relationships similarity.

1 Introduction

NoSQL data models are different from the relational model in terms of structure and capacity. NoSQL data models offer great flexibility due to their schema-free nature. As one of the NoSQL data models, the use of graph databases has increased significantly. Graph databases are database models that are based on the graph structure, essentially nodes and edges. NoSQL graph databases come with the benefit of handling large volumes of heterogeneous and semi-structured data. Yet, data management in such databases is a challenging task.

Data under databases with a pre-defined scheme, such as relational databases, can be migrated in a version-controlled sequence by saving each schema transformation alongside its data migration. While data under schema-free databases still require careful migration techniques.

Graph databases have a schema-free nature. Thereby, schemes are implicit and modified directly by managing data instances without any specific constraints on any such manipulations.

Primarily, a data management task of a graph database does not require a dedicated team of database administrators. It is often the responsibility of Application teams or business units.

A common practice to handle data and schema management is to write custom migration scripts to migrate data eagerly (all data migrated on one go when the database structure changes) or lazily (migrating only data being acceded to).

When migrating data eagerly, data is accessed all at once. That reduces the data latency. Nevertheless, migrating all data at once can take a long time and requires a shut down the access to the database. With lazy data migration, legacy data that no longer needs any changes is not accessed. Yet, this strategy has a high data latency.

This paper deals with data migration and evolution in graph databases. It propose an approach that helps to manage data migration taking into account not only the flexibility provided by such databases but also the nature of the graph databases.

The remainder of the paper is structured as follows: Section 2 overviews of the state-of-the-art that treated data migration in the field of NoSQL databases and specifically graph databases. Section 3 contains some required preliminary definitions. Section 4 explains our approach for data migration and details our process of lazy data migration.The proposed approach is based on similarity measures and graph matching. We performed experiments and addressed the evaluation results in section 5. Section 6 concludes the paper and presents some future works.

2 Related Works on Data Migration

Data migration has been an area of active research with a long history [10, 11, 13, 1]. There are essentially two main data migration strategies to guaranty accurate data migration and evolution and recently, three new data migration strategies are developed.

In the following, we start by presenting the data migration strategies. Then, we overview related works on data migration within the context of NoSQL databases.

2.1 Data Migration Strategies

There are essentially two data migration strategies to guaranty accurate data migration and evolution such as *Eager data migration* and *Lazy data migration*. The authors in [12] present three more migration strategies such as *incremental migration*, *predictive migration*, and *adaptive migration*.

Eager data migration: with this strategy, all entities within the database are migrated at once. Though this strategy has a low access cost (latency) , it procures a high migration cost as some of the data will be updated even though they will not be accessed in the future. This strategy is best when migrating

data from different DBMS (E.g., from a relational model to graph model). The key issue of this strategy is that the data, even data that may not be usable again in the future, must be kept up to date.

Lazy data migration: with this strategy, all data remain unchanged until being accessed. This strategy has no immediate migration costs and ensures flexibility to agile requirement changes. Lazy data migration strategy aims to minimize migration costs. Nevertheless, when compared to the eager data migration strategy, data access latency can be relatively high.

Incremental data migration: Incremental data migration strategy works similarly to lazy data migration strategy, i.e., only data that needs to be changed is accessed. Nonetheless, lazy migration periods are regularly interrupted to clean the database which reduce run-time overhead caused by updating legacy data on-the-fly. For these interruptions, eager data migration over legacy entities is performed regularly.

Predictive data migration: Predictive data migration strategy highlights the frequently accessed data. It keeps track of past data accesses while ordering the accessed entities accordingly via exponential smoothing. This established technique in time series data weighs the entities by their actuality and access frequency. [12] present a detailed study of the different data migration strategies.

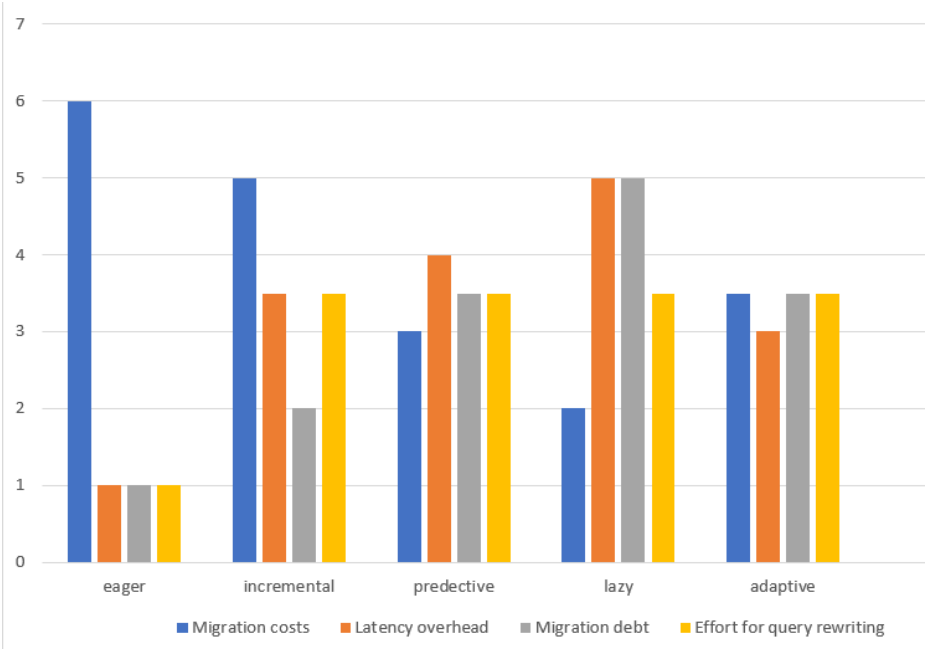


Fig. 1. Characteristics of the different data migration strategies.

Figure 1 compare the different data migration strategies with regards to the *migration costs*, *latency*, *migration debts* and *effort for query rewriting* [12].

The term *latency overhead* refers to the time needed to retrieve the data. In [12], *migration cost* is a term that refers to the charges occasioned by migrating the data, and *migration debt* refers to the changes needed to be invested to migrate data to a homogeneous data structure.

Both *eager data migration* and *incremental data migration* have a high migration cost due to migrating all entities at once. *Lazy data migration* has the lowest data migration costs. However, it has the highest latency overhead and migration cost.

2.2 Data Migration in NoSQL Databases

Data migration in NoSQL databases is a relatively new research field. Regardless, researchers conducted many works in this area, from managing JSON-based files to schema extraction and data migration.

The authors in [4, 5] present a framework τJSchema that for the definition and validation of temporal JSON documents that conform to a temporal JSON schema. The authors proposed a versioning technique that provides a complete set of low-level and high-level change operations. Both at the instance and schema levels, τJSchema fully supports temporal versioning of JSON-based Big Data.

The work presented in [14] details a study of the evolution of the domain model of applications built against a NoSQL data store. This methodology is applied to ten real-world database applications. They start by extracting the schema, then analyze the entire project history. Lastly, they analyze the evolution of the NoSQL database schema.

In [15], the authors propose a framework called *Datulation* that support *lazy* and *eager* data migration strategies using Datalog rules. Datalog is a programming language for deductive databases. The work highlights the merit of the lazy data migration strategy. The framework supports adding, renaming, or removing attributes for entities. When carrying out data migration eagerly, the framework evaluate Datalog rules bottom up. As for lazy migration, *Datulation* evaluates Datalog rules top-down, thus migrating only the legacy entities that are deployed by the application. One strong point of Datalution is that it can lazily roll forward chains of schema changes. Yet, this tool only supports data with prefixed schema and written in JSON format.

In [9], the authors demonstrate *ControVol Flex*, an Eclipse plugin for controlled schema evolution in Java applications backed by NoSQL document databases. Based on a lazy migration strategy. This tool helps to migrate the NoSQL by adding Morphia annotations to the code. It also supports an automatic version-numbering mechanism for the different stages of the schema evolution process. This tool keeps track of the various schema versions that occur in the production data store. It also support *lazy* and *eager* data migration strategies. The advantage of *ControVol Flex* is that it allows to carry out eager and lazy

data migration concurrently, which is vital for the continuous deployment of zero-downtime applications.

In [12], the authors present a methodology of self-adapting data migration which focuses on data migration itself. The framework presented is based on schema management middleware *Darwin* and a tool-based advisor *MigCast*. *Darwin* supports schema extraction, schema evolution, and data migration of data stored in a NoSQL database. *Darwin* support all four data migration strategies such as eager, lazy, incremental, and predictive strategy. *Darwin* also supports popular NoSQL database management systems, such as MongoDB, Couchbase, Cassandra, and the multi-model database ArangoDB. The choice of the best data migration to be applied is made by *MigCast*: a tool for self-adapting data migration strategy. *MigCast* helps to explore the different data migration strategies in order to examine the effects of the data migration strategies with regard to the metrics of migration costs, latency, and migration debt.

The work [6] emphasizes the importance of tracking the history of data changes within the graph database. The authors present a plug-in that delivers a novel representation of historical graph data using graph versioning techniques.This work features the specific structure of the graph database. It proposes an approach to represent history related to both (i) nodes (track down changes that occurred to the entity itself) and (ii) relationships (track down the changes to the relation including start or end node updates). The authors in this work present the different versions of data separately in another graph called *VersionGraph*. *VersionGraph* stores the history of different versions of a data graph.

The authors of [7] introduce a method based on in-memory architectures to retrieve the structural schema of a graph database. The authors focused on schema extraction in the context of semi-structured graph data. They extend the existing methods to manage large NoSQL graph databases. They introduce several types of summaries and provide the methods to extract them. The authors present four types of summaries, such as structural summaries, structural data summaries, structural data key property summaries, structural data key-value property summaries.

The work [16] presents a solution for missing data in the NoSQL graph databases by e introducing a novel approach for mining gradual patterns in the presence of missing values. The focus of this paper is the extraction of gradual patterns from property graphs on *IntraNode-Label* gradual patterns, that is the pattern extraction process is to be performed among the properties/attributes of the "same label" nodes. The work presented in [2] proposes a data definition language (DDL) schema for property graphs inspired by Cypher query language to handle schema validation and schema evolution for graph databases. The work presented a mathematical framework that allows enforcement schema and expresses propagation from schema to instance and vice versa. Table 1 summarizes data migration works in NoSQL databases.

Table 1 compares the presented works with regard to the data storage type, the different data migration strategies they support, and whether or not they

Table 1. Data migration and schema extraction in NoSQL databases.

Works	Data type	Migration Strategy	schema extraction	schema versioning
[4, 5]	JSON documents	-	✓	✓
[14]	NoSQL databases	Eager, Lazy, Incremental, Predictive	✓	-
[15]	NoSQL databases	Lazy	✓	-
[9]	NoSQL databases	Lazy	✓	-
[12]	NoSQL databases	Eager, Lazy, Incremental, Predictive, Self-adapting	✓	-
[6]	Graph databases	-	✓	✓
[7]	Graph databases	-	✓	-
[16]	Graph databases	-	✓	-
[2]	Graph databases	-	✓	✓

support schema extraction and schema versioning. being an important step of data management, all works presented offer schema extraction techniques. even though most of them support different types of NoSQL databases, yet only a few of them emphasize the graph database structure in terms of nodes and relationships.

2.3 Discussions

Most works use data stored in document-store databases or data of JSON format as input. Adopting this strategy when dealing with graph data excludes the benefits of using graph theory and graph algorithms that may help reduce the run time overhead. Moreover, they treat data entities similarly and don't consider the graph structure (nodes, and relationships). That is explained due to the structure of links (relationships) between different entities as they are different with each NoSQL data model. Thus, most works support data only and exclude the relationships between entities.

Additionally,they emphasize schema extraction or schema validation. Though schema proved to be relevant when dealing with data history or data migration, depending on schema extraction or schema validation to deal with data migration goes against the fundamental idea of having a schema-free database nature.

Graph matching methods are widely used for subgraph matching or extraction. They also proved to be very useful for pattern recognition and biological and biomedical database where graph representation of data is used. NoSQL graph databases are already based on graph theory. To make sure to highlight the structure of the graph database, proposing a solution based on the graph matching technique seems very promising. By definition, graph matching is the problem of finding a similarity between graphs. Thus, we propose a solution based on string similarity measures and graph matching to help migrate data correctly within the graph database.

3 Preliminary Definitions

A graph database (GDB) is formed by a set of nodes, relationships, properties, and labels. Both nodes and their relationships are named and can store properties. These properties are represented by key/value pairs. Nodes and relationships can be labeled. The edges between two graphs representing the relationships have two qualities: they always have a start node and an end node and are directed making the graph a directed graph. Relationships can also have properties.

Definition1 (graph): A graph G is defined by a pair (V, E), where V is a set of vertices and E is a set of edges with $E \subseteq (VxV)$

Definition2: (directed graph is defined by a pair (V, E), where V is a set of vertices and E is a set of edges with $E \subseteq (VxV)$ and where all the edges are directed from one vertex to another.

Definition 3 (Graph database): A graph database is a couple (N, R), where N is the total set of nodes and which form the entities of GDB and R as the set of relations that join the different nodes.

Definition 4 (Nodes): Each node n is composed of its identifier id_n, a set of properties P_n, and a set of labels L_n. It should be noted that the identifier does not contain any semantic information. Semantic is usually expressed through one or more labels and a single property is a couple of as (key/ value) pair. A node can be written as follows: $\forall n \in N; n = (id_n, P_n, L_n)$.

Definition 5 (Relationships): A relation r is defined as $(id_n, Strn, Endn, T, P)$ which contains the identifier id, the start node Strn/ end node Endn, the type of relation T, and its set of properties P.

4 LD-MIG: Graph Data Migration Approach

In order to control data migration under graph databases, we propose a lazy data migration approach based on graph matching. The data migration approach under graph database is a process composed of four phases (i): Analysis phase, (ii): Graph matching, (iii): MOs identification, and (iv): Graph merging as shown in figure 2. In the following, We assume that we have two graphs GDB (the database that is currently deployed) and the *operation-graph*(the new graph to add) where N (respectively M) are the sets of nodes of GDB (respectively the *operation-graph*) and R (respectively W) are the sets of relationships of GDB (respectively the *operation-graph*).

4.1 Analysis Phase

The first step in our approach is to examine the input queries responsible for any database changes. The aim of this phase is to generate the *Operation-graph* from the query to be applied on the database or the source code. This step intend to analyze the composition of different CRUD(create, read, update and delete) queries, extract the input data, and build a graph model based on the operations

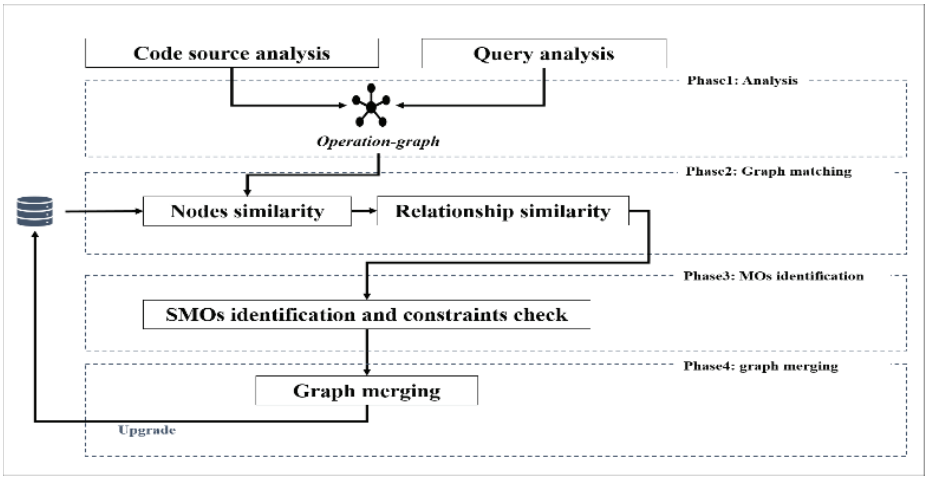


Fig. 2. Overall approach.

to be performed on the graph database (GDB). The generated graph is called an *operation-graph (OPG)*.

We choose to work with Neo4j as one of the most popular graph databases management systems that use Cypher a SQL- inspired query language that describes visual patterns in graphs using ASCII-Art syntax. Figure 3 presents an example of an *operation-graph* created based on a "Merge" query expressed by *Cypher* in *Neo4j*.

Fig. 3. Merge query with its corresponding *operation-graph*.

4.2 Graph Matching

In this step, we aim to extract a sub-graph from GDB that is most similar to the operation graph. The extracted sub-graph will serve as the entry of the third phase. A graph database presents the data as it is conceptually viewed in the form of nodes linked by relationships. Therefore, we propose a process composed of node-based similarity measures, relation-based similarity measures, and graph matching step based on Levenshtein edit distance to detect the most

similar sub-graph. As mentionned above, the Graph matching phase is composed by three main steps ***(i): nodes similarity, (ii): relationships similarity, (iii): sub-graph matching***.

Given the nature of the graph database, we can safely consider that node's labels, properties, and relationship types are presented as strings. Therefore, it is suitable to propose a string-based similarity, embedded in the process of structure graph matching, to identify the similarity between both graphs.

Nodes Similarity We assume that we have two graphs GDB (the database that is currently deployed) and the *operation-graph*. N (respectively M) are the sets of nodes of GDB (respectively the *operation-graph*) and R (respectively W) are the sets of relationships of GDB (respectively the *operation-graph*). The similarity between two nodes (m,n) depends on the similarity between their labels and their properties.

Labels similarity. To compute the similarity between two labels we simply apply the Levenshtein distance between them. Taking into account that a node may have more than one label, the similarity of the labels between two nodes is calculated by comparing each label of m to each label of n and then taking the maximum value obtained. Formula 1 determines the similarity between a label $l_m \in L_m$ of a node $n \in N$ with all the labels of a given node n and returns the maximum:

$$sim(l_m, L_n) = \max_{j=1}^{k} \big(1 - \frac{lev(l_m, l_{nj})}{max(length(l_m), length(l_{nj})}\big) \tag{1}$$

To determine the similarity between all labels L_m of a node m with a given node $n \in N$ we apply algorithm 1.
As input for algorithm 1we have two node's labels L_m and L_n and the overall

Algorithm 1: Simlabels.

Input: L_n, L_m
Output: simls
; ▷ `With` L_n `as the labels of the node n,` L_m `as the labels of the node m and simls as the similarity value between the labels of m and the labels of n`

1 $k \leftarrow \min\big(\mid L_m \mid, \mid L_n \mid\big)$
2 $x \leftarrow$ *the node with the minimum set of labels*
3 $y \leftarrow$ *the remaining node*
4 $simls \leftarrow simls\big(L_m, L_n\big) = \frac{\sum_{i=1}^{k} SimL(l_{xi}, L_y)}{k}$

similarity *simls* as an output, (line4) calculate the average similarity using the *simls* formula.

Properties similarity. Node property is composed of a key/ value pair. To measure the similarity between two properties we compute the similarity of

their keys and then for the maximum value obtained, we measure the similarity between their values. To calculate the similarity between two pairs of keys k_{mi} as a key of a property of the node m (respectively k_{nj} as a key of a property of the node n) we apply the following formula:

$$SimK(k_{mi}, k_n) = \max_{j=1}^{h} \big(1 - \frac{lev(k_{mi}, k_{nj})}{\max(length(k_{mi}), length(k_{nj}))}\big) \tag{2}$$

$$SimValue(v_{mi}, v_{nj}) = \big(1 - \frac{lev(v_{mi}, v_{nj})}{\max(length(v_{mi}), length(v_{nj}))}\big) \tag{3}$$

Formula 3 computes the similarity of property key k_{mi} of the node m to all properties of the node n. Formula 4 computes the similarity between the value of k_{mi} and the most similar key from the node n.

The global similarity between both of the properties is computed by the average of the keys and values similarity. Algorithm 2 describes the process of computing the properties similarity.

Algorithm 2: Simproperties.

Input: P_m, P_n
; ▷ P_m and P_n are the properties of the nodes m and n
Output: simps
; ▷ simps the similarity value between the properties of m and the properties of n

1 $z \leftarrow \min\big(\mid P_m \mid, \mid P_n \mid\big)$
2 $x \leftarrow$ *the node with the minimum set of labels*
3 $y \leftarrow$ *the remaining node*
4 **foreach** $i \in P_x$ **do**
5 **foreach** $j \in P_y$ **do**
6 $dictkey \leftarrow \{xkey : i, ykey : j, simkey : SimK(k_{xi}, k_y)\}$; ▷ dictkey is a dictionary that contains i (current key of x), j (current key of y) and their similarity value using SimK(kxi,ky)
7 $maxdictkey \leftarrow \{xkey : i, ykey : j, simkey : MaxSimK(k_{xi}, k_y)\}$; ▷ maxdictkey is a dictionary that contains the maximum similarity of i (current key of x) to all the keys of y
8 $maxdictkey \leftarrow \{xkey : i, ykey : j, simkey : MaxSimK(k_{xi}, k_y), simvalue : SimValue(v_{xi}, v_{yj}), prop_sim : maxpro\}$; ▷ add the value similarity (SimValue) and the overall similarity (max pro) to maxdict
9 $simps(P_m, P_n) \leftarrow \frac{\sum_{i=1}^{z} maxdict['maxpro']}{z}$

As an input for algorithm 2, we have the sets of properties of the nodes m and n. The output will be simps of all properties (similarity value between the properties of m and the properties of n). After determining the node with the minimum set of properties (lines:1,2,3) we calculate the maximum keys similarity and store

them in a dictionary where *(dictionaries are unordered, changeable collections that have key/value pairs used in the python programming language.)* (line 4 to 7) Then, for each key in Maxdict, the corresponded value similarity is computed (line 8). Finally (lines 9) computes the overall properties similarity.

Nodes similarity. The nodes similarity is calculated by invoking the two algorithms *Algorithm1* simlabels (L_m, L_n) and *Algorithm2* simproperties (P_m, P_n) as follows:

$$nodessim(m,n) = \frac{simlabels(L_m, L_n) + simproperties(P_m, P_n)}{2}. \quad (4)$$

4.3 Relationship Similarity

As stated previously, a graph database is characterized by its nodes and relationships. Relationships are the key entities used in the database to express semantic. A relationship is declared by a directed edge and defined by its type, start/end nodes, and eventually properties. We measure the similarity between the two relationships by measuring the similarity between their types (names) and properties.

Relationship type similarity. We assume that we have two relationships r = (id_r, $Strn_r$,$Endn_r$, T_r, P_r)and w = (id_w, $Strn_w$,$Endn_w$, T_w, P_w) respectively belongs to GDB and the *operation-graph*. The first step is to verify the existence of the relationship by measuring the similarity between both their types. We simply apply the Levenshtein distance function to compute the similarity between their types (names). It is to note that a relationship can have one type presented as a string. Formula 6 calculates the relationship-types similarity.

$$Simtyperelation(T_w, T_r) = \big(1 - \frac{lev(T_w, T_r)}{\max(length(T_w), length(T_r))}\big). \quad (5)$$

Properties similarity. To compute the similarity of the relationship's properties, we reuse *algorithm2* of the node's similarity.

In this section, we mainly focus on proposing a subgraph matching algorithm for specifying the similarity between the graph database (GDB) and *Operation-graph*. This algorithm 3 employs the similarity measures we previously suggested. Algorithm 3 have as input both GDB (the current database in use) and OPG (the *Operation-graph*. The algorithm go through every node in OPG and compare it with all nodes of GDB. Everytime a high similarity between two nodes m and n is detected (lines 3 and 4), we compute the similarity between their relationships (line 5 to line 12). The output is a dictioanary *SIMDICT* containing the entities of GDB that are most similar to OPG.

nodes similarity and graph matching phases are published recently in [3]. Further details concerning the nodes and relationships similarity are discussed in [3].

Algorithm 3: Graph matching.

Input: GDB, OPG
; ▷ with GDB as the database in use and OPG as the opertation-graph
Output: $SIMDICT$
; ▷ $SIMDICT$ is a dictionary contaning the diffrent nodes and relationships of OPG and the most similar nodes and relationships of GDB a long with their similarity values

foreach $m \in OPG$ **do**
  foreach $n \in GDB$ **do**
    $NSIM \leftarrow nodessim(m, n)$;
    while *(*$NSIM > threashold$*)* **do**
      $R \leftarrow$ extract all relationships of n and assign them to a list R;
      $W \leftarrow$ extract all relationships of m and assign them to a list W;
      foreach $w \in W$ **do**
        foreach $r \in R$ **do**
          $TRSIM \leftarrow Simtyperelation(T_r, T_w)$;
          $PSIM \leftarrow simproperties(P_r, P_w)$;
          ********** $ENSIM \leftarrow nodessim(Endn_r, Endn_w)$;
          $RelSim \leftarrow \frac{TRSIM+PSIM+ENSIM+NSIM}{4}$;
          while *(*$RelSim > threashold$*)* **do**
            $SIMDICT \leftarrow$
            $SIMDICTid_{mi}, w, id_n, r, NSIM(n, m), RelSim(w, r)$

4.4 MO's Identification Phase and Data Migration Process

Having a schema-free nature, while managing data in a graph database, we implicitly manage its schema. The most fundamental data manipulation that a graph database DBMS offers are:

- **Create/Add**: refers to create a new entity in a database.
- **Update**: refers to updating an existing entity in the database.
- **Delete**: refers to deleting an existing entity from the database.

GDBMS are error-prone due to the lack of schema restrictions. In the following section, we propose a set of operations for correctly migrating data in GDBMS and a global data migration algorithm. This step aims to identify the set of operations needed to convert an OPG entity to a GDB entity and ensure a correct data migration. Let OPG and GDB be two graphs where (M, W) are the sets of nodes and relationships belongs to OPG and (N, R) are the sets of nodes and relationships belongs to GDB.

A homomorphism $h : OPG \longrightarrow GDB$ is a function $h_\eta : M \longrightarrow N$ and a function $h_\varepsilon : W \longrightarrow R$, mapping nodes and relationships (M,W) of OPG to nodes and relationships (N,R) of GDB. It is to be noted that h_η and h_ε are the sets of operations required to safely migrate M and W to N and R. In the following, we present in detail a set of modification operations. In the scope of this paper, we cover the basic CRUDE operations such as Add and Update.

Add Operation. This Operation requires the non-existence of the entity in GDB. Let e be the entity from OPG to migrate e can be a node m or a relationship w.

Algorithm 4: Procedure: addEntity.

```
   Input: (e, GDB)
1  ∀n, r ∈ GDB,  e ∈ OPG  NodesSim(m, n) = 0 or RelSim(w, r) = 0 while
   (NodesSim(e,n)=0) do
2  |  Add m(L_m, P_m) to GDB.
3  while (RelSim(e,r)=0) do
4  |  if (Strn_w and Endn_w∃ GDB) then
5  |  |  Add only T_w and P_w to OPG
6  |  else
7  |  |  Add both nodes and the relationship to GDB
```

The addEntity aims to add a new entity from OPG that does not exist in GDB. If the entity to add is a relationship then, we verify the existence of its start/ end nodes first (lines 3 to 7).

Update operation : Updating an entity (a node or a relationship) can be done by adding, updating, retyping, or even deleting its elements (node labels, node properties,relationship properties or relationship types). Algorithm 5 illustrate an example of updating a node property.

Algorithm 5: Procedure: updateEntity.

```
   Input: (m, GDB)
1  ∀n, r ∈ GDB,  e ∈ OPG  NodesSim(m, n) >= threashold or
   RelSim(w, r) >= threashold while (NodesSim(m, n) >= threashold) do
2  if (p_m P_n) then
3  |  add the new property p_m to the existing node n
4  else if (p_n P_m) then
5  |  delete the property p_m from the existing node n
6  else
7  |  replace the existing property p_m with p_n
```

Data Migration Process : the data migration process takes as input the most similar entities from GDB to OPG, then it apply the diffrent modification operations over them. As it only deals with entities being modified, our data migration process is concidered to a lazy data migration process. algorithm 6 detilas the process to migrate data correctly.

Algorithm 6: Procedure: Lazy data migration.

Input: $(SIMDICT, GDB, OPG$
; ▷ `with GDB as the database in use, OPG as the opertation-graph and SIMDICTis a dictionary contaning the diffrent nodes andrelationships of OPG and the most similar nodes andrelationships of GDB a long with their similarity values`

1 **while** *(*$NodesSim(m,n) < threashold$*)* **do**
2 addEntity(m,GDB)
3 **while** *(*$RelSim(w,r) < threashold$*)* **do**
4 addEntity(w,GDB)
5 **while** *(*$NodesSim(m,n) >= threashold$*)* **do**
6 **if** *(*$\{P_m\} \cap \{P_n\}=\emptyset$*)* **then**
7 updateEntity(p_m, GDB)
8 **else if** *(*$\{L_m\} \cap \{L_n\}=\emptyset$*)* **then**
9 updateEntity(l_m, GDB)
10 **else**
11 break;
12 **while** $RelSim(w,r) >= threashold$ **do**
13 **if** *(*$NodesSim(Strn_w, Strn_r) < threashold)OR$ *(*$NodesSim(Endn_w, Endn_r) < threashold$*)* **then**
14 addEntity($Endn_m$,GDB)
15 **else**
16 updateEntity($Strn_w$,GDB);
17 updateEntity($Endn_w$,GDB)
18 **if** *(*$\{P_w\} \cap \{P_r\}=\emptyset$*)* **then**
19 updateEntity(p_w, GDB)
20 **else**
21 updateEntity(P_w,GDB)

Algorithm6 takes as input the simiarity dictioanry SIMSICT that contains the most similar entities of GDB and its similarity measures. Then, it compares each measure to a prefixed threshold. In case that the similarity between two entities is low, we then add the entity from OPG to GDB (lines 1 to 4). In case two entities have a high similarity values we then update the existed entity in GDB depending on the dissimilarity that both entities may have.

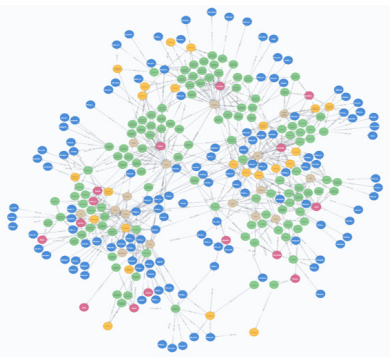
5 Evaluations: LDBC-SNB Benchmark

To evaluate the proposed approach, we carried out a set of experiments on the LDBC-SNB benchmark. First, we present an overview about the LDBC-SNB, we then present our evaluation results.

5.1 LDBC-SNB: Overview

LDBC's Social Network Benchmark (LDBC-SNB) is an effort intended to test various functionalities of systems used for graph-like data management. For this, LDBC SNB uses the recognizable scenario of operating a social network, characterized by its graph-shaped data.

A detailed description of the benchmark can be found in the initial publications [8, 17]. The model features individuals and their actions in a social network over time. It describes the structure of the data in terms of nodes and their relationship. The initial databases contain 29,192 nodes and 39,800 relationships.



Nodes	29192
Relationships	39800
Labels	18
Relationship Types	15
Property Keys	20

Fig. 4. Snapshot of the LDBC-SNB database.

5.2 Evaluations

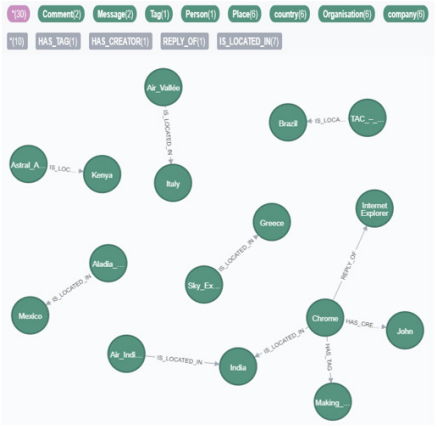
We carried out two evaluations: ***(i) an expremental evalutaion*** in which we extracted randomly ten relationships with their nodes (OPG1). We first applied some changes to the extracted entities (OPG1), then we carried off our program, and finally, we verified whether the changes made are correct in the GDB. And ***(ii) a second evaluation*** in which we variate the number of entities of the *Operation-Graph*. The evaluation is done based on the run time. the datasets used as *Operation-graphs* are:

- The first dataset (OPG1): contains 30 entities randomly extracted from the database GDB.
- The second, THird and fouth datasets (OPG2, OPG3,OPG4): contains respectively 60,150 and 200 entities.
- For the fifth, sixth and seventh datasets (OPG5, OPG6,OPG7), we increased the number of entities and alter more modifications to them. The datasets contains respectively 500, 700,1000 entities.

– The eighth, ninth and tenth datasets (OPG8, OPG9,OPG10) contains respectively 50, 100 and 200 entities.

It is to note that the entities of datasets (OPG1 to OPG7) are randomly extracted from the GDB. to be able to test our approach, we altered modifications to each of these datasets. Datasets (OPG8, OPG9 and OPG 10) contains entities that are very dissimilar to the original database GDB. We used Neo4j as a graph database management system and python 3.7 as a programming language. In this paper, we have carried out two types of experimentation. The *(i): first evaluation* in which we used the first dataset as *operation-graph* to evaluate the correctness of our data migration approach. And the *(ii): second evaluation* in which we used all the datasets and for each set, we compute the run time of graph matching and the data migration process.

Experimental Evaluation We extracted ten random relations with theirs corresponding nodes from the initial database GDB (OPG1-1). Figure 5 displays the nodes and relationships of OPG1-1.



Nodes	16
Relationships	10
Labels	8
Relationships types	4

Fig. 5. Snapshot of OPG1-1.

Then we altered some modifications over some of the nodes and relationships (OPG1-2). We changed about 30% of the dataset. An example of modifications are presented as follows:

– changing the label "Person" of some nodes to "User".
– changing the value of the property "creationDate" of some nodes.
– changing the relationship type "HAS-TAG" to "H-TAG".

We then performed our data migration process to check the data correctness and migration run time after migrating the data. Figure 6 is a showcase of the database after the data migration.

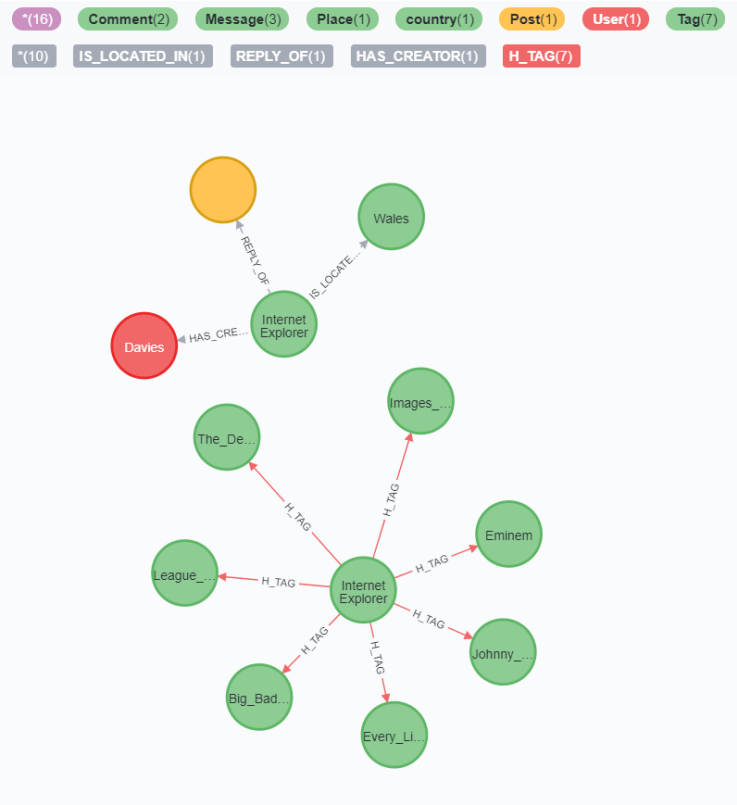


Fig. 6. Snapshot of OPG1-1 after data migration.

Table 3 summarizes the obtained results. The performance of our data mi-

Table 2. Results of the first evaluation.

OPG	Changes	latency	Data correctness	Run time	similarity value
OPG1-1	0%	0.017 s	100%	07.639049 s	1
OPG1-2	30%.	0.016 s	100%	0.783 s	0.93566120387549

gration approach shows that, though the runtime is relatively higher when the average similarity is ¡1 , legacy data is migrated correctly and no data loss is generated. it is not that data latency is almost the same for each dataset.

Second Evaluation For the second type of evaluation, we used all the datasets and for each set, we computed the run time of graph matching and the data migration process. We also computed data latency (number of data access for

each data migration). Table 2 represents a quantitative study for all datasets. Table 3 summarizes the results we obtained. We noticed that even though our

Table 3. Results of the second evaluation.

OPG	OPG entities	nodes	relations	properties
OPG1	30	20	10	128
OPG2	60	40	20	198
OPG3	150	100	50	630
OPG4	300	200	100	1180
OPG5	1500	1000	500	3750
OPG6	2100	1400	700	5009
OPG7	3000	2000	1000	630
OPG8	150	100	50	274
OPG9	300	200	100	563
OPG10	600	400	200	1142

Table 4. Results of the second evaluation.

OPG	Similarity value	Run time
OPG1	0.93566120387549	00:07.639049 m
OPG2	0.9549847478673998	00:17.390528 m
OPG3	0.9356612038754876	00:19.007823 m
OPG4	0.9549847478674043	00:18.727151 m
OPG5	0.8501378090918807	03:25.444671 m
OPG6	0.9281139995680916	04:59.606046 m
OPG7	0.8491987906273264	07:33.561546 m
OPG8	0.5036420203001845	00:12.009441 m
OPG9	0.5036420203001861	01:18.471618 m
OPG10	0.8491987906273264	07:33.561546 m

approach ensure a correct data migration aver small datasets, checking both graph similarity and running the data migration process take an important time. In fact, the runtime depends greatly on entities number and the modification rate, i.e., the more dissimilarity the higher the run time. In fact whenever the dissimilarity is higher, the more data to be migrated. Figure 7 illustrates the runtime of executing our program in regards to the number of entities and the average similarity.

6 Conclusion

Graph databases are widely used in recent years. For that, it is crucial to study its evolution. However, migrating data in such databases can be difficult given

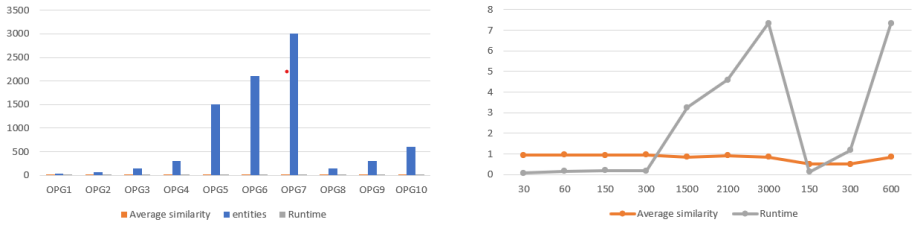


Fig. 7. Runtime by number of entities and average similarity.

their schema free nature. In the scope of this paper, we proposed an approach to safely migrate the graph database.

In this paper, We presented a Graph matching algorithm based on similarity measures in which we extracted a sub-graph from the initial database that is most similar to the *Operation-graph*.

We also presented a process based on the lazy data migration strategy in order to minimize data migration costs and reduce unnecessary data migration procedures (migrating data that will not be accessed in the future) and finally, we detailed the MO's identification phase in which we described specific scripts that help control the migration of legacy data within the graph database. As future work, we aim to propose more optimization by developing an approach based on deep learning algorithms in order minimize the runtime of our approach as it has an exponential complexity.

References

1. Abbes, H., Gargouri, F.: Modular ontologies composition: Levenshtein-distance-based concepts structure comparison. International Journal of Information Technology and Web Engineering (IJITWE) 13(4), 35–60 (2018)
2. Bonifati, A., Furniss, P., Green, A., Harmer, R., Oshurko, E., Voigt, H.: Schema validation and evolution for graph databases. In: International Conference on Conceptual Modeling. pp. 448–456. Springer (2019)
3. Boukettaya, S., Nabli, A., Gargouri, F.: Graph matching in graph-oriented databases. In: International Conference on Intelligent Systems Design and Applications. Springer (2020)
4. Brahmia, S., Brahmia, Z., Grandi, F., Bouaziz, R.: Temporal json schema versioning in the jschema framework. Journal of Digital Information Management 15(4) (2017)
5. Brahmia, S., Brahmia, Z., Grandi, F., Bouaziz, R.: Managing temporal and versioning aspects of json-based big data via the τjschema framework. In: International Conference on Big Data and Smart Digital Environment. pp. 27–39. Springer (2018)
6. Castelltort, A., Laurent, A.: Representing history in graph-oriented nosql databases: A versioning system. In: Eighth International Conference on Digital Information Management (ICDIM 2013). pp. 228–234. IEEE (2013)

7. Castelltort, A., Laurent, A.: Exploiting nosql graph databases and in memory architectures for extracting graph structural data summaries. International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems 25(01), 81–109 (2017)
8. Erling, O., Averbuch, A., Larriba-Pey, J., Chafi, H., Gubichev, A., Prat, A., Pham, M.D., Boncz, P.: The ldbc social network benchmark: Interactive workload. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. pp. 619–630 (2015)
9. Haubold, F., Schildgen, J., Scherzinger, S., Deßloch, S.: Controvol flex: Flexible schema evolution for nosql application development. Datenbanksysteme für Business, Technologie und Web (BTW 2017) (2017)
10. Herrmann, K., Voigt, H., Behrend, A., Rausch, J., Lehner, W.: Living in parallel realities: Co-existing schema versions with a bidirectional database evolution language. In: Proceedings of the 2017 ACM International Conference on Management of Data. pp. 1101–1116 (2017)
11. Herrmann, K., Voigt, H., Rausch, J., Behrend, A., Lehner, W.: Robust and simple database evolution. Information Systems Frontiers 20(1), 45–61 (2018)
12. Hillenbrand, A., Störl, U., Nabiyev, S., Klettke, M.: Self-adapting data migration in the context of schema evolution in nosql databases. Distributed and Parallel Databases pp. 1–21 (2021)
13. Mesiti, M., Celle, R., Sorrenti, M.A., Guerrini, G.: X-evolution: A system for xml schema evolution and document adaptation. In: International Conference on Extending Database Technology. pp. 1143–1146. Springer (2006)
14. Scherzinger, S., Sidortschuck, S.: An empirical study on the design and evolution of nosql database schemas. In: International Conference on Conceptual Modeling. pp. 441–455. Springer (2020)
15. Scherzinger, S., Sombach, S., Wiech, K., Klettke, M., Störl, U.: Datalution: a tool for continuous schema evolution in nosql-backed web applications. In: Proceedings of the 2nd International Workshop on Quality-Aware DevOps. pp. 38–39 (2016)
16. Shah, F., Castelltort, A., Laurent, A.: Handling missing values for mining gradual patterns from nosql graph databases. Future Generation Computer Systems 111, 523–538 (2020)
17. Szárnyas, G., Prat-Pérez, A., Averbuch, A., Marton, J., Paradies, M., Kaufmann, M., Erling, O., Boncz, P., Haprian, V., Antal, J.B.: An early look at the ldbc social network benchmark's business intelligence workload. In: Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA). pp. 1–11 (2018)