HAL
open science

Exploiting NoSQL Graph Databases and In Memory Architectures for Extracting Graph Structural Data Summaries

Arnaud Castelltort, Anne Laurent

To cite this version:

Arnaud Castelltort, Anne Laurent. Exploiting NoSQL Graph Databases and In Memory Architectures for Extracting Graph Structural Data Summaries. International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems, 2017, 25 (1), pp.81-109. 10.1142/S0218488517500040 . lirmm-01381083

HAL Id: lirmm-01381083

https://hal-lirmm.ccsd.cnrs.fr/lirmm-01381083v1

Submitted on 1 Nov 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems

EXPLOITING NoSQL GRAPH DATABASES AND IN MEMORY ARCHITECTURES FOR EXTRACTING GRAPH STRUCTURAL DATA SUMMARIES

ARNAUD CASTELLTORT
LIRMM - UNIVERSITY OF MONTPELLIER
{*firstname.lastname@lirmm.fr*}

ANNE LAURENT
LIRMM - UNIVERSITY OF MONTPELLIER
{*firstname.lastname@lirmm.fr*}

Received (received date)
Revised (revised date)

NoSQL graph databases have been introduced in recent years for dealing with large collections of graph-based data. Scientific data and social networks are among the best examples of the dramatic increase of the use of such structures. NoSQL repositories allow the management of large amounts of data in order to store and query them. Such data are not structured with a predefined schema as relational databases could be. They are rather composed by nodes and relationships of a certain type. For instance, a node can represent a *Person* and a relationship *Friendship*. Retrieving the structure of the graph database is thus of great help to users, for example when they must know how to query the data or to identify relevant data sources for recommender systems. For this reason, this paper introduces methods to retrieve structural summaries. Such structural summaries are extracted at different levels of information from the NoSQL graph database. The expression of the mining queries is facilitated by the use of two frameworks: Fuzzy4S allowing to define fuzzy operators and operations with Scala; Cypherf allowing the use of fuzzy operators and operations in the declarative queries over NoSQL graph databases. We show that extracting such summaries can be impossible with the NoSQL query engines because of the data volume and the complexity of the task of automatic knowledge extraction. A novel method based on in memory architectures is thus introduced. This paper provides the definitions of the summaries with the methods to automatically extract them from NoSQL graph databases only and with the help of in-memory architectures. The benefit of our proposition is demonstrated by experimental results.

Keywords: Graph Databases; Graph Mining; Data Summaries; NoSQL; In Memory.

1. Introduction

NoSQL databases are attracting more and more attention in data science and provide an efficient framework for dealing with complex data and intensive read and/or write access. They are often linked to distributed architectures in order to scale up

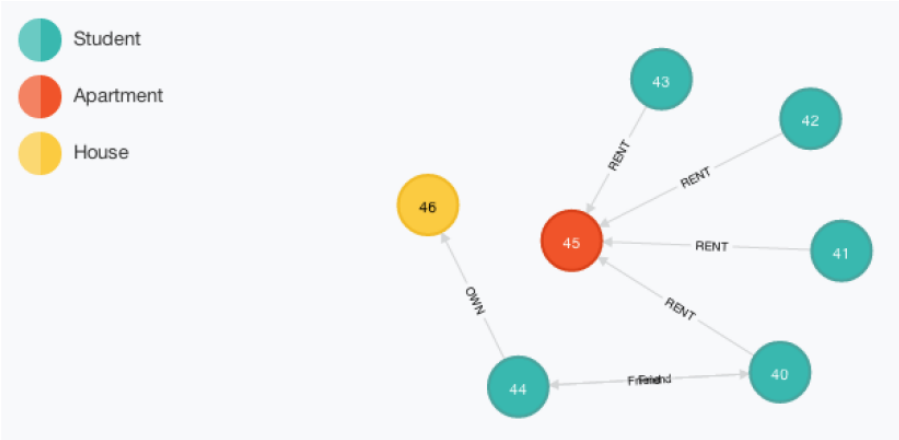


Fig. 1. Example of a Graph Database

on very large databases. Several types of NoSQL databases have been proposed (column, store, key-value,...) and are chosen depending on the type of data, type of queries etc [10,20].

In many applications, data are based on graphs, as in for instance social networks, biological interactions, semantic Web and more. In such applications, it is often the case that the connections between data matter just as much as the data itself. In such cases, extracting knowledge is a hot topic [1] as it has been the case for many years in pattern recognition [13]. However when the relationships are central and tend to evolve, both relational databases and classical NoSQL databases are no longer efficient for storing and querying data. For this reason, NoSQL databases have been extended to NoSQL Graph databases [3,29]. Several NoSQL graph engines have been developed. Neo4j is one of the most popular and is thus used in this work [28].

In the NoSQL graph database model, the objects considered are nodes and relationships. Complex information on both nodes and relationships are managed as properties with $(key, value)$ pairs. In addition to properties, nodes and relationships may be labeled with types.

For instance, Figure 1 shows the relationships between people and the places they live in. For instance, the people from nodes 44 and 40 are *friends*. The relationship types depict whether people *own* or *rent* their housing. The type of housing can be an apartment or a house: this value is known as the *node type*. Node information (e.g., person's age) and relationships (e.g., monthly rental fees) can be provided as node and relationship properties (*e.g.*, $key = age$).

Queries in graph databases are defined as traversals over the graph. They can be run at several levels, either programmatically or by using declarative languages (as done when using SQL in relational databases). This work relies on declarative

query languages. In Neo4j, this language is Cypher [27].

NoSQL databases are not populated over a predefined *schema*. The user, should (s)he be writting or reading the data, may not be aware of the content of the repository. Moreover when being accessed by machines, the databases must be easily understandable. For instance this is the case for automatic ontology alignment, for data sources recommendation, etc [7].

For this reason, the automatic extraction of the structure of the database is crucial and is the main contribution of this paper. More specifically, this paper is about automatically extracting summaries from very large NoSQL graph databases. Several summaries are considered in order to allow a comprehensive summary of the data.

Structural summaries contain all the nodes and relationship types of the source database. Structural data summary provides a more accurate summary by providing relevant information on the presence of the relationships in the database, thus allowing users to easily understand what the graph deals with.

For instance, the nodes of type *Person* could be connected to several other types of nodes like *Housing*, *Person*, etc. by relationships of type *Friend, Rents, Buys, etc.* These *Person* nodes are connected with a high number of relationships of type *Rents* and a smaller number of relationships of type *Buys*.

In order to weigh the relationships to account for their gradual presence, we propose using the framework of the fuzzy quantifiers such as *Few, Most*. For this reason, we introduce the use of two frameworks in the context of NoSQL graph databases: Fuzzy4S and Cypherf.

Fuzzy4S (standing for Fuzzy for Scala) is a fuzzy logic framework written in Scala (a language mixing functional and object-oriented paradigms). Proposed as a library, it contains membership functions, t-norms and t-conorms [23]. Upon this library, an open Domain Specific Language (DSL) has been built to define approximate queries at an abstract level. It relies both on the IEC 61131 standard (IEC61131-7) that has been developped for fuzzy control programming and on JFL that is a Java Fuzzy framework [11,12]. Fuzzy4S is used in the Cypherf language which extends the Cypher declarative language.

Using such frameworks allows us to rely on the existing work related to linguistic summaries. Summaries of relational databases have been addressed in many works, especially with linguistic summaries [6,15,22]. Linguistic summaries are based on protoforms, the first one being $Qy\ are\ P$ where Q stands for a fuzzy quantifier, y are the objects to be summarized, and P is a possible value, such as in *Most students are young*. Linguistic summaries have been extended in many works, to handle for example time series [2,21]. Linguistic summaries can be extracted with (fuzzy) pattern-mining-based algorithms.

However, these works cannot be easily applied to NoSQL graph databases. The summaries we aim to discover must indeed be transposable to linguistic summaries. Moreover such databases combine several criteria that have never been considered

altogether: relationships as the main concern, presence of node and relationship types, complex information contained in the $(key, value)$ properties, etc.

What's more is the framework of NoSQL graph databases must be able to handle very large databases. This paper shows the limits of NoSQL graph databases for such mining tasks and details how in-memory architectures can be used to scale up.

The remainder of the paper is organized as follows: Section 2 reviews works on NoSQL graph databases and linguistic summarization. Section 3 introduces the definitions of the four types of summaries proposed in this work. Section 4 details the methods to extract such summaries from large datasets and demonstrate the limits and benefits of our propositions through experiments. Final conclusions and future possibilities are discussed in Section 5.

2. Background

This work relies on NoSQL graph databases and data summarization.

2.1. *Graph Databases*

2.1.1. *General Concepts*

Graphs have been studied for a long time by both mathematicians and computer scientists. A graph can be directed or not, labeled or not. NoSQL graph databases rely on labeled directed graphs.

Definition 1. Labeled Directed Graph. A labeled oriented graph G, also known as oriented property graph, is given by a n-uplet $(V, E, \alpha, \beta, l_V, l_E)$ where V stands for a set of vertices and E stands for a set of edges with $E \subseteq (V \times V)$, α stands for the set of attributes defined over the nodes, β the set of attributes defined over the relations, $l_V : V \rightarrow \mathcal{P}(\alpha)$ is the labeling function for nodes and $l_E : E \rightarrow \mathcal{P}(\beta)$ is the labeling function for relationships.

NoSQL graph databases [29] such as Neo4j consider vertices as nodes and edges as relationships. The labels are sometimes said to be the *types*. On top of this, NoSQL graph databases consider that nodes and relationships are provided with information called *properties*. These are stored using the $(key, value)$ paradigm that is very common in NoSQL databases. Figure 3 shows a graph and its structure in $(key, value)$ pairs.

Studies have shown that these technologies perform well, much better than classical relational databases at representing and querying such large graph databases.

2.1.2. *NoSQL Graph Databases*

NoSQL graph databases [29] are based on graph concepts with the following additional point: properties are defined over the nodes and relationships stored according to the $(key, value)$ paradigm, which is very common in NoSQL databases.

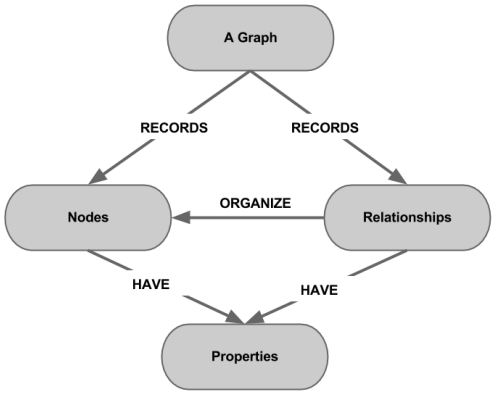


Fig. 2. Labeled Graph

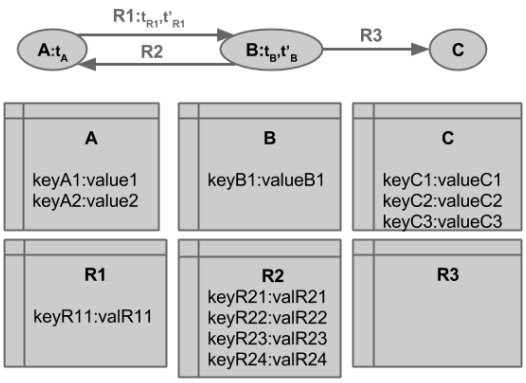


Fig. 3. Node and Relation Properties

It should be noted that types are distinguished from properties, as in NoSQL engines such as Neo4j. These types appear in Figure 1 as colors for nodes (*e.g.,* Student, House) and as labels for relationships (*e.g.,* Owns).

Generalizing the definition of labeled directed graphs, we propose a formal definition of NoSQL graph databases below.

Definition 2. NoSQL Graph Database. A NoSQL graph database G is given by a tuple $(V, E, \theta, \tau, \alpha, \beta)$ where

- α stands for the set of node properties defined by (*key*:*value*) pairs;

- β stands for the set of edge properties defined by (key:$value$) pairs;
- θ stands for the set of node types;
- τ stands for the set of edge types;
- V stands for a set of vertices with $\forall v \in V, v = (id_v, t_v, \kappa_v)$ $s.t.$ $t_v \subseteq \theta$ stands for the types of v, $\kappa_v \subseteq \alpha$ stands for the properties of v and id_v is the vertice identifier;
- E stands for a set of edges with $\forall e \in E, e = (id_e, (v_e^1, v_e^2), t_e, \lambda_e)$ $s.t.$ $(v_e^1, v_e^2) \in \{V \times V\}$, $t_e \subseteq \tau$ stands for the types of e, $\lambda_e \subseteq \beta$ stands for the properties of e and id_e is the edge identifier.

The set of properties of a node v is denoted by α_v, the set of types is denoted by θ_v. The set of properties of a relation e is denoted by β_e, the set of types is denoted by τ_e.

Figure 3 shows a graph and its structure in $(key, value)$ pairs. In this example, we have:

- $\alpha = \{(keyA1, valueA1), \ldots\}$
- $\beta = \{(keyR11, valueR11), \ldots\}$
- $V = \{(A, \{t_A\}, \{(keyA1, valueA1), (keyA2, valueA2)\}),$
  $(B, \{t_B, t'_B\}, \{(keyB1, valueB1)\}),$
  $(C, \emptyset, \{(keyC1, valueC1), (keyC2, valueC2), (keyC3, valueC3)\})$
- $E = \{(R1, (A, B), \{t_{R1}, t'_{R1}\}, \{(keyR11, valueR11)\}), \ldots\}$

Several NoSQL graph database engines exist (OrientDB, Neo4j, HyperGraphDB etc.) [3]. Neo4j is considered the best perfomer [30]. All NoSQL graph databases require developers and users to use graph concepts to query data. Queries are called traversals, referring to the action of visiting elements, i.e. nodes and relationships. There are three main ways to traverse a graph:

- programmatically: using an API;
- by functional traversal: using a traversal based on a sequence of functions applied to a graph;
- by declarative traversal: explicitly expressing the required data and letting the database engine define the best way to achieve this goal.

This paper focuses on declarative queries over a NoSQL graph database. The Neo4j language is called Cypher.

For instance, in Figure 4, a query to return the customers who have visited the *Ritz* hotel is displayed. Those customers are both displayed in the list and circled in red in the graph.

Cypher clauses are similar to SQL ones. It is based on a "ASCII art" style of writing graph elements. For example, directed relations are written using the -[]-> chain. Types and labels are written after a semi-colon (:).

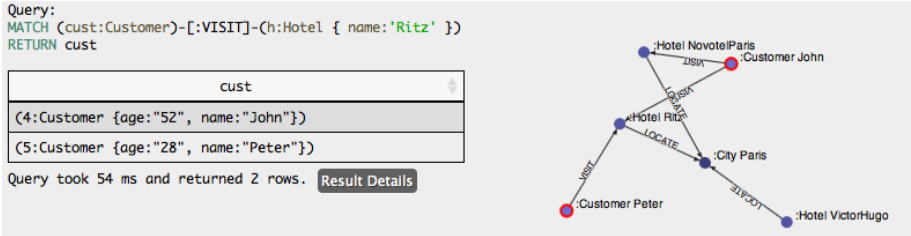


Fig. 4. Displaying the Result of a Cypher Query

Listing 1. Query example on a Graph

```
[START]
[MATCH]
[OPTIONAL MATCH WHERE]
[WITH [ORDER BY] [SKIP] [LIMIT]]
RETURN [ORDER BY] [SKIP] [LIMIT]
```

Listing 2. Cypher Clauses

```
START: Starting points in the graph, obtained via index lookups or by element IDs.
MATCH: The graph pattern to match, bound to the starting points in START.
WHERE: Filtering criteria.
RETURN: What to return.
CREATE: Creates nodes and relationships.
DELETE: Removes nodes, relationships and properties.
SET: Set values to properties.
FOREACH: Performs updating actions once per element in a list.
WITH: Divides a query into multiple, distinct parts.
```

More specifically, queries in Cypher have the following syntax[a]:

As shown above, Cypher is comprised of several distinct clauses which are listed below in Listing 2.

These operations can even be extended to fuzzy queries [8]. In this work, they are used for computing the linguistic summaries introduced below.

2.2. *Graph Summarization*

Data Summarization has been extensively studied in the last decades to produce linguistic sentences, such as *Most of the students are young* [31]. These approaches are based on the so-called *protoforms* (*e.g., Qy are P*) where Q is a fuzzy quantifier, y

[a]http://docs.neo4j.org/refcard/2.0/
http://docs.neo4j.org/chunked/milestone/cypher-query-lang.html

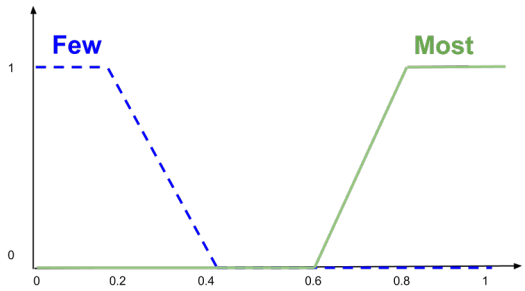


Fig. 5. Example of Fuzzy Quantifier Membership Functions

are the objects to summarize and P is a (fuzzy) predicate. They focus on relational data where source data are represented in the form of tuples defined over a schema. For instance, the tuples (John, 23, 45000), (Mary, 32, 60000) and (Bill, 38, 55000) are three tuples defined on the schema (Name, Age, Salary). The fuzzy quantifiers are defined over the $[0, 1]$ universe of proportions. We may for instance consider two quantifiers Few and $Most$ whose membership functions are displayed by Figure 5.

The quality of linguistic summaries can be assessed by many measures, the seminal one being T, the *degree of truth* that can be simply computed with a σ-count:

$$T(Qy's\ are\ P) = \mu_Q \left(\frac{1}{n} \sum_{i=1}^{n} \mu_P(y_i) \right)$$

where n is the number of objects (y_i) that are summarized and μ_P , and μ_Q are the membership functions of the summarizer and quantifier, respectively.

There are various ways to examine summaries. Researchers have focused on fields like the design of protoforms, quality measures, efficient algorithms etc [6].

All the literature on fuzzy linguistic summaries is not recalled here as this paper focuses on the subject of graph data. In this framework, two main characteristics have to be highlighted. First, graph databases are not provided with a strict and given schema such as relational data. In fact, they are closer to semi-structured data. Second, graph databases focus on relationships.

Summarizing graph data has been considered for many years, aiming for instance at compressing such data with the use of supernodes as shown by Figure 6 from [25].

Graph summarization is related to graph mining. Graph (and tree) mining deals with the problem of extracting frequent patterns (subgraphs/subtrees) from a large graph. It is often presented as an extension of the so-called itemset mining methods. Such methods have been successfully applied to large graphs by considering efficient approaches [14,24,32].

Several works in the literature have focused on schema extraction in the context of semi-structured graph data, i.e., XML data. The schema extraction problem

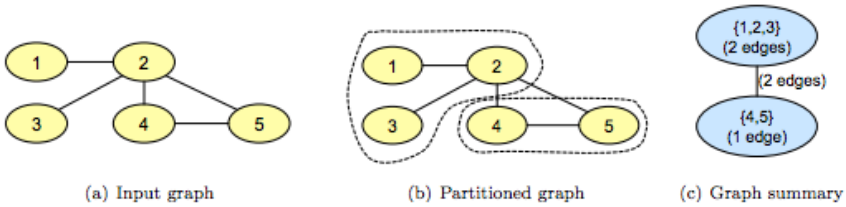


Fig. 6. Graph Summarization using Supernodes

consists in identifying a schema S from a given set of XML data documents D, such that S captures the structural information of the documents in D in the most minimal way. The schema extraction process is also referred to as schema inference [5]. The underlying structure of a given collection of XML documents can be described using Document Type Definitions (DTD), XML Schema, or via a more general representation such as tree or a graph. The structure extraction techniques in the literature aim to infer three kinds of representations: tree or graph summaries, DTD, or XML Schema.

Our work extends the existing methods in order to deal with large NoSQL graph databases by introducing several types of summaries and by providing the methods to extract them.

3. NoSQL Graph Summaries: Definitions

Several structures can be used to summarize NoSQL graph databases. This work extends that of [9] as well as taking into account the specificities of NoSQL graph databases, especially the fact that nodes and relationships are provided with properties. Four types of summaries are proposed:

- Structural Summaries;
- Structural Data Summaries;
- Structural Data Key Property Summaries;
- Structural Data Key Value Property Summaries.

3.1. *Structural Summaries*

Structural summaries are meant to retrieve the structure of the graph embedded in element types, which could in some ways be associated with relational database schema. Such summaries could thus be associated with with schema mining in the literature.

Definition 3. Structural Summary. Let $G = (V, E, \theta, \tau, \alpha, \beta)$ be a NoSQL graph database. A structural summary S of G is defined as $S = (a\text{-}[r]\text{-}>b, Q)$ where $a, b \in \theta$ (node types), $r \in \tau$ (relation type) and Q is a fuzzy quantifier.

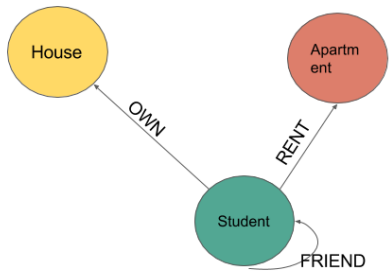


Fig. 7. Structural Summary of the Example

The structure summary can be expressed in a linguistic form as follows: *In G, Q of the a r b*

Example 1. In the toy example, the structural summary shows that students own houses or rent apartments and are connected through the friend relationship. The corresponding structural summary is depicted by $S = (students - [rend] - > apartments, Most)$ (see Figure 7).

It should be noted that such a summry is ambiguous. It may indeed consist of computing either the proportion of students who rent an apartment over the whole student population, or the proportion of students who rent an apartment over the number of students who rent their residence..

We thus consider a truth degree to assess the quality of a summary and we propose two ways of calculating this degree. These two definitions are provided below.

This weight can be compared to the degree of truth in the context of linguistic summaries, although no proposition has been made in the case of graphs. In our framework, the degree of truth determines the extent to which the relationship appearing in the summary is truthful in regards to the fuzzy quantifier. For instance, if the summary mentions that *most of the students rent an apartment,* then the degree of truth describes to what extent a high proportion of students rent an apartment.

Definition 4. Degree of Truth of NoSQL Graph Summaries. Given a graph database G and a summary $S = a -[r]-> b, Q$, the degrees of Truth of S in G are defined as:

$$Truth_1(S) = \mu_Q \left(\frac{count(distinct(S))}{count(distinct(a))} \right)$$

$$Truth_2(S) = \mu_Q \left(\frac{count(distinct(S))}{count(distinct(a -[r]-> (?)))} \right)$$

The second type of degree of truth is also called the *diversity of target source* denoted by D_T later on in this paper.

Example 2. In the toy example from Figure 1, the degree of truth of the summary "$S = Student -[rent]->apartment, Most$" is given by the membership degree to the fuzzy quantifier[b] of the ratio between the number of times a relationship appears between a Student and an Apartment (s)he rents over the number of relations of type *Rents* starting from a Student node. In this example, we thus have $Truth_2(S) = \mu_{Most}\left(\frac{4}{5}\right) = \mu_{Most}(0.8)$

The ratio appearing in the definition of the degree of truth can be compared to the $confidence$ in association-rule mining which acts as a conditional probability.

The proportion is computed with respect to the total number of relationships outgoing from the nodes of this type. For instance, the proportion of the relationship of type t_r between node types t_a and t_b is computed as:

$proportion(t_a - [t_r] - > t_b) = \frac{|t_a - [t_r] - > t_b|}{|t_a|}$

The transformation of a proportion p using the fuzzy quantifier from a set $\mathcal{Q}$ could be easily computed as $\arg\max_{q \in \mathcal{Q}}(\mu_q(p))$. However, this would falsely convey the idea that a relationship occuring at a proportion of 10% with only one other relationship at a proportion of 90% is considered in the same way as a a relationship occuring at proportion 10% with nine other relationships at proportion 10% each.

For this reason, we propose another measure based on the difference between the proportion and the proprtion corresponding to the equi-distributed situation: $m(t_a - [t_r] - > t_b) = \left| \frac{|t_a - [t_r] - > t_b|}{|t_a - [*] - > *|} - \frac{1}{|t_a - [*] - > *|} \right|$.

m can then be matched to a set of *relative quantifiers* such as *rare, regular, frequent.*

Example 3. In the toy example, the structural data summary shows that students rarely own houses (1 relationship out of 6 in the database), often rent apartments (4 relationships over 6) and are rarely connected through the friend relationship (1 relationship over 6). The corresponding structural summary is depicted by Figure 8.

This can easily be converted graphically by emphasising the relationships with high weights. It can also be easily converted to linguistic summaries, for instance in the form of *Most of the students rent an apartment.*

[b]We do not mention here the detailed membership function of the *Most* quantifier which can be defined in a very classical manner as done in the literature of fuzzy quantifiers and fuzzy summaries.

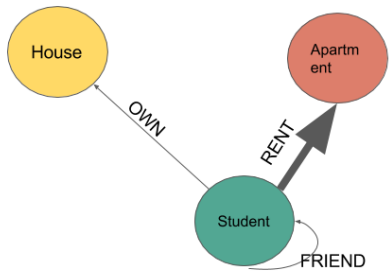


Fig. 8. Data Structure Summary of the Example

However, such summaries do not convey any information about the properties of the nodes and relations. Structural data summaries are thus proposed.

3.2. *Data Structure Summaries*

Structural data summaries are based on structural summaries and embed information about the properties, as for instance the income of the people being considered, or the rental paid for a residence. The above definition is thus extended to allow this new type of summaries.

Definition 5. Data Structure Summary. Let $G = (V, E, \theta, \tau, \alpha, \beta)$ be a NoSQL graph database. A Data Structure Summary S is defined as $S = (a.X\text{-}[r.Z]\text{->}b.Y, Q)$ with $a, b \in \theta$ (node types), $r \in \tau$ (relation type), $X, Y \subseteq \alpha$ (node properties), $Z \subseteq \beta$ (relation properties) and Q a fuzzy quantifier.

However, structural summaries and structural data summaries do not take nodes and relationships properties into account even though they are one of the main characteristics of the NoSQL graph databases.

3.3. *Data Structure Key Property Summaries*

Definition 6. Structural Data Key Property Summary. Let $G = (V, E, \theta, \tau, \alpha, \beta)$ be a NoSQL graph database. A structural data summary S_G is defined as a graph $G = (V_S, E_S, \theta, \tau, K_{\alpha_S}, K_{\beta_S})$ and a set δ_{E_S} where

- $|V_S|$ is equal to the number of distinct combinations of types and properties of nodes,

- $|E_S|$ is equal to the number of distinct combinations of types and properties of relationships,
- there are no two nodes or two relationships having the same type: $\forall v, v' \in V_S \times V_S, t_v \neq t_{v'}$ and
- δ_{E_S} corresponds to a set of weights provided for every relationship $e \in E_S$,
- K_{α_S} is a set of keys associated with the nodes,
- K_{β_S} is a set of keys associated with the relationships.

3.4. *Data Structure Key Value Property Summaries*

Definition 7. Structural Data Key Value Property Summary. Let $G = (V, E, \theta, \tau, \alpha, \beta)$ be a NoSQL graph database. A structural data summary S_G is defined as a graph $G = (V_S, E_S, \theta, \tau, K_{\alpha_S}, K_{\beta_S})$ and a set δ_{E_S} where

- $|V_S|$ is equal to the number of distinct combinations of types and properties of nodes,
- $|E_S|$ is equal to the number of distinct combinations of types and properties of relationships,
- there are no two nodes or two relationships having the same type: $\forall v, v' \in V_S \times V_S, t_v \neq t_{v'}$ and
- δ_{E_S} corresponds to a set of weights provided for every relationship $e \in E_S$,
- α_S is a set of $(key : value)$ pairs associated to the nodes,
- β_S is a set of $(key : value)$ pairs associated to the relationships.

These summaries can be expressed as linguistic summaries.

Definition 8. Structural Data Key Property Linguistic Summary. Let $G = (V, E, \theta, \tau, \alpha, \beta)$ be a NoSQL graph database. A Data Structure Summary S is defined as $S = (a.X -[r.Z]\text{–}>b.Y, Q)$ with $a, b \in \theta$ (node types), $r \in \tau$ (relation type), $X, Y \subseteq \alpha$ (node properties), $Z \subseteq \beta$ (relation properties) and Q a fuzzy quantifier.

Example 4. $(Student(Age : 28) -[rent(fees : 1200)]\text{–}> apartment, Few)$ is an example of a structural data key property linguistic summary.

Such summaries are extended in order to allow fuzzy linguistic labels in the refinement. Indeed, it would be both difficult and useless to define summaries on single values such as "the age is 28", as in fuzzy data mining for fuzzy association rule mining. Using fuzzy linguistic labels makes it possible to retrieve fuzzy linguistic summaries where young students and low rental fees are considered.

Definition 9. Structural Data Key Property Fuzzy Linguistic Summary. Let $G = (V, E, \theta, \tau, \alpha, \beta)$ be a NoSQL graph database. Let F_α and F_β be sets of fuzzy properties. A Fuzzy Data Structure Summary S is defined as $S = (a.X -[r.Z]\text{–}> b.Y, Q)$ with $a, b \in \theta$ (node types), $r \in \tau$ (relation type), $X, Y \subseteq \alpha \bigcup \mathcal{F}_\alpha$ (node

Listing 3. Retrieving structure of a graph

```
1 MATCH (a)-[r]->(b)
2 RETURN DISTINCT labels(a), type(r), labels(b)
```

properties and node fuzzy properties), $Z \subseteq \beta \bigcup F_\beta$ (relation properties and relation fuzzy properties) and Q a fuzzy quantifier.

Example 5. $(Student(Age : young) -[rent(fees : low)]\text{–>}apartment, Most)$ is an example of a structural data key property fuzzy linguistic summary.

4. Extracting NoSQL Graph Summaries: Methods and Experiments

Most of the analytical treatments we propose can be expressed by defining queries over the NoSQL graph database, so as to obtain a more declarative than procedural way to extract summaries. This property is based on the fact that NoSQL graph databases provide powerful pattern-matching features, as intended in inductive relational databases [16].

In this section, we show how to run such extractions by using native operations defined in the declarative Cypher language. These extractions are extended to a more expressive model by using the Fuzzy4S and Cypherf frameworks.

However, as shown by our experiments, NoSQL graph database engines cannot process very large databases for analytical processing and extracting the summaries. We thus introduce an original method based on the use of in-memory graph processing systems.

Experiments have been run on synthetic and real databases. The real dataset is introduced in Section 4.2.

4.1. *Graph Database Engine-based method*

In this section, a graph database engine will be used to process graph databases and extract summaries such as those defined in Section 3.

4.1.1. *Native Method*

To retrieve a graph summary, graph database engines can be used. For instance, the strutural summary can be retrieved by considering Cypher queries such as those presented in Listing 3.

Line 1 of this query asks for all oriented relations “r” between an incoming node “a” and an outgoing node “b”. Line 2 asks the engine to return all the distinct labels of incoming nodes a, type of relations r and labels of outcoming nodes b. The output result will be all distinct triplets (labels, type, labels).

Listing 4. Retrieving Structure Summaries

```
MATCH (a)-[r]->(b)
WITH DISTINCT labels(a) AS labelsA, type(r) AS typeR, labels(b) AS ↩
    labelsC, toFloat(count(*)) AS countS
MATCH (a1)-[r2]->(m)
WHERE labels(a1)= labelsA AND type(r2)= typeR
WITH DISTINCT labelsA, typeR, labelsC, countS, labels(a1) AS labelsA1, ↩
    type(r2) AS typeR2, count(*) AS count2
RETURN labelsA, typeR, labelsC,
       tofloat(countS)/ count2 AS Truth
```

In order to retrieve the structural data summaries, it is necessary to compute the measures for evaluating the weights of the relationships. The query from Listing 4 extracts the summaries from a graph and calculates the weight of every relationship.

The degree of Truth that is calculated in Listing 4 does not inform the end-user to what extent it is "Truth": few? most? To do so, fuzzy quantifiers have to be used. The next section will introduce how to use fuzzy queries to enhance graph summarization.

4.1.2. *Extending Graph database engine to fuzzy queries*

Introducing Fuzzy4S and Cypherf

As said before, Cypher is a an external Domain Specific Language (DSL) for expressing queries on NoSQL graph databases. A Domain Specific Language is a language tailored to a specific application domain [18] such as SQL for relational databases, Make for building softwares or CSS for styling description.

There are several ways to extend Neo4j to support fuzzy queries. We chose to enhance Cypher with functions that allow the user to express fuzzy concerns with a "Fuzzy DSL". This new extension of Cypher is called *Cypherf*, standing for Cypherfuzzy.

It allows the end-user to use fuzzy features such as defining membership function with the use of a fuzzy DSL, using t-norms and t-conorms functions as defined in [23]. The main available functions are listed bellow:

- Fuzzy(μ_f, value): returns the degree of membership to μ_f function
  - μ_f is expressed as a String that describes the membership function with the Fuzzy DSL
  - value is expressed as a Double
- FuzzyLT(fuzzyVariable, value): returns a collection that contains for every fuzzy term of the fuzzy linguistic variable two properties: the name of the term and the degree of membership of "value" to the term. For instance, for a value X and a fuzzy linguistic variable $Age = (Age, [0, 130], \{young, middleaged, old\}, \{\mu_{young}, \mu_{middle}, \mu_{old}\})$ the result will be:

```
{
    {name:"young",degree:μ_young(x)},
    {name:"middle",degree:μ_middle(x)},
    {name:"old", degree:μ_old(x)}
}
```

Membership function	Scala code	DSL
PieceWise	PieceWiseMembershipFunction (List(ValuePoint(0,1), ValuePoint(2,1), ValuePoint(4,0)))	(0,1) (2,1) (4,0)
Triangular	TriangularMembershipFunction(0,2.5,5)	trian 0 2.5 5
Trapezoidal	TrapezoidalMembershipFunction(0,2,3,4)	trape 0 2 3 4
Sigmoidal	SigmoidalMembershipFunction(-4, 3)	sigm -4 3
Singleton	SingletonMembershipFunction(SingletonPoint(2))	2
...	...	...

Fig. 9. Examples of membership function using Scala4S

  - fuzzyVariable: is expressed as a String that defines the fuzzy variable. This definition is composed of the name of the fuzzy variable and a set of fuzzy-terms. Every fuzzy term is defined by its name and its membership function
  - value is expressed as a Double

- TNorms(tnormName, expression1, expression2): applies a TNorms of name "tnormName" on expression1 and expression2
- TCoNorms(type, expression1, expression2): same as TNorms but this time for t-conorms.

Cypherf is processed by an underlying fuzzy framework called "Fuzzy4S" that implements the fuzzy logic. Fuzzy4S has been developed explicitly as part of Cypherf implementation but can be used separately in other projects. Upon this library an open Domain Specific Language (DSL) has been built to define approximate queries at an abstract level.

Figure 9 provides some examples of using membership functions of Fuzzy4S both in Scala code and with the external DSL.

As shown in Listing 5, a point can be expressed as a pair of two numbers, the first one defining the value in the universe and the second one the membership degree. A membership function is provided with an optional function name and some parameters (point or number). A Term defines a fuzzy set; it is composed of a name and a membership function. A fuzzy linguistic variable is composed of a name, for instance *age*, and of all the terms that compose this fuzzy linguistic variable (*young*, *old*, etc.) such as those defined in Listing 5 and represented by

Listing 5. Fuzzy variable example: fuzzy linguistic definition of VAge

```
FUZZIFY Vage
        TERM young:=  (25,1) (40,0);
        TERM middle:= trape 25 40 50 65;
        TERM old:= (50,0) (65,1);
END_FUZZIFY
```

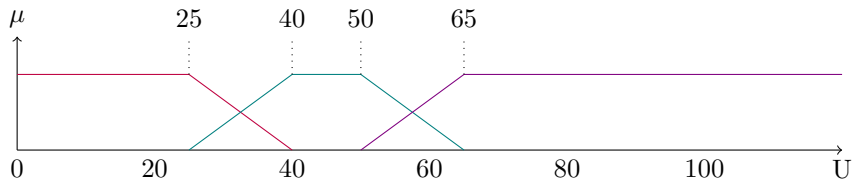


Fig. 10. Result of listing 5

Listing 6. Fuzzy Partition of Quantifiers

```
FUZZIFY quantifiers
        TERM veryFew :=  trape 0 0 10 20;
        TERM few := trape 0 0 20 40;
        TERM almostHalf :=  trape  20 40 60 80;
        TERM most :=  trape  60 80 100 100;
END_FUZZIFY
```

Figure 10.

Adding fuzzy quantifier to structural summary

A fuzzy variable can be defined as in Listing 6 and represented by Figure 11.

This method have been tested on several real databases[c], showing its limits on large datasets that it cannot process successfully, as described below and discussed in the next section:

[c]http://neo4j.com/developer/example-data

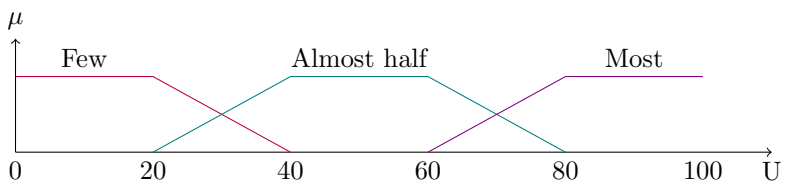


Fig. 11. Fuzzy Quantifiers

Listing 7. Retrieving Structure Summaries with fuzzy query

```
MATCH (a)-[r]->(b)
WITH DISTINCT labels(a) AS labelsA, type(r) AS typeR, labels(b) AS ↩
    labelsC, toFloat(count(*)) AS countS
MATCH (a1)-[r2]->(m)
WHERE labels(a1)= labelsA AND type(r2)= typeR
WITH DISTINCT labelsA, typeR, labelsC, countS, labels(a1) AS labelsA1, ↩
    type(r2) AS typeR2, count(*) AS count2
RETURN labelsA, typeR, labelsC,
       tofloat(countS)/ count2 AS Truth,
       fuzzyLT("fuzzyQuantifier.fl" , tofloat(countS) / count2) as ↩
           quantifiedTruth
```

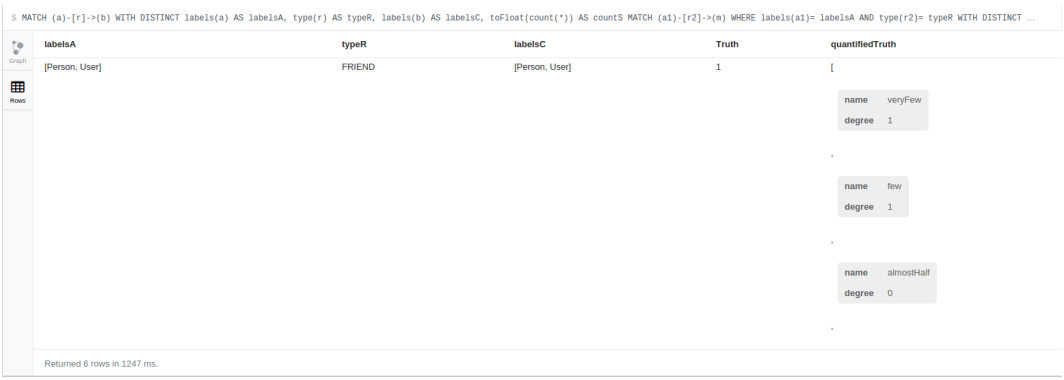
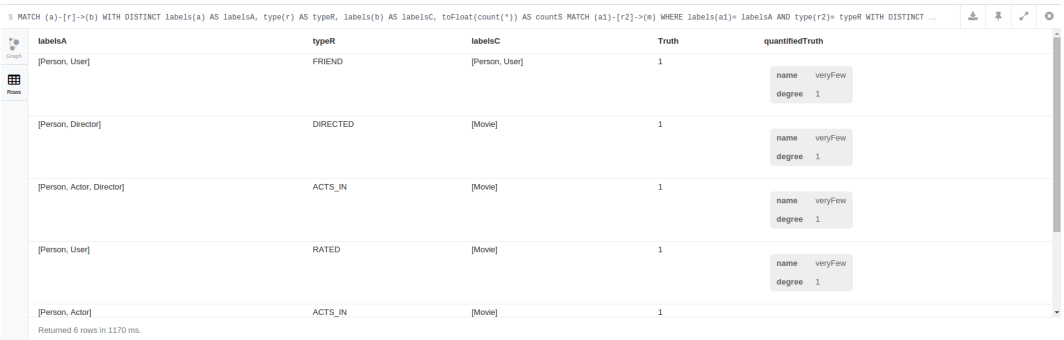


Fig. 12. Neo4j execution of Listing 7

Listing 8. Retrieving Structure Summaries with fuzzy query

```
MATCH (a)-[r]->(b)
WITH DISTINCT labels(a) AS labelsA, type(r) AS typeR, labels(b) AS ↩
    labelsC, toFloat(count(*)) AS countS
MATCH (a1)-[r2]->(m)
WHERE labels(a1)= labelsA AND type(r2)= typeR
WITH DISTINCT labelsA, typeR, labelsC, countS, labels(a1) AS labelsA1, ↩
    type(r2) AS typeR2, count(*) AS count2
RETURN labelsA, typeR, labelsC,
       tofloat(countS)/ count2 AS Truth,
       head(fuzzyLT("fuzzyQuantifier.dsl" , tofloat(countS) / count2)) as↩
           quantifiedTruth
```

Dataset	Size	Number of Nodes	Number of Relationships	Result
Dr.Who	0.05MB	1,060	2,286	Passed
Cinema	12.3MB	63,042	106,651	Passed
Musicbrainz	5.4GB	35,778,712	73,433,369	Failed



$ MATCH (a)-[r]->(b) WITH DISTINCT labels(a) AS labelsA, type(r) AS typeR, labels(b) AS labelsC, toFloat(count(*)) AS countS MATCH (a1)-[r2]->(m) WHERE labels(a1)= labelsA AND type(r2)= typeR WITH DISTINCT ...

labelsA	typeR	labelsC	Truth	quantifiedTruth
[Person, User]	FRIEND	[Person, User]	1	name veryFew, degree 1
[Person, Director]	DIRECTED	[Movie]	1	name veryFew, degree 1
[Person, Actor, Director]	ACTS_IN	[Movie]	1	name veryFew, degree 1
[Person, User]	RATED	[Movie]	1	name veryFew, degree 1
[Person, Actor]	ACTS_IN	[Movie]	1	

Returned 6 rows in 1170 ms.

Fig. 13. Neo4j execution of Listing 8

4.1.3. *Limitation of Graph database engine in summaries extraction*

Graph databases solve problems that relational databases struggle to solve, such as retrieving friends of friends. Graph databases have some great features: transactions, query language, incremental update and persistence, etc., but they are designed to work with small pieces of graphs. Indeed, Graph databases are OLTP systems and are not designed to run an analytical algorithm on an entire graph.

The queries of Listings 3, 4, 7 and 8 are executable in a graph database engine with datasets of thousands of nodes and relations but not with a real life dataset that contains millions of relationships and nodes.

Running graph analysis or processing on a massive dataset requires some other kind of system called "Graph Processing Systems". These systems are designed for graph analysis and running algorithms on the entire graph.

There are several Graph Processing systems. The most popular ones are GraphX, Giraph, and GraphLab, which are implementations of the ideas expressed in the Google Pregel paper [26] and on Map/Reduce papers [17,19].

In the next section, GraphX, a graph processing system based on Apache Spark In-Memory BigData processing library, is used to show how graph summarization can be performed with a Big Data In-Memory approach.

4.2. *In Memory-based method*

The goal of this section is to explain the different steps that lead to the production of a graph summary using In-Memory process and data from a graph database engine. For the sake of simplicity, the structural summary schema will be used as the running example of this section. The dataset used is taken from MusicBrainz[d]. Some basic statistics of this dataset are given in Table 1.

[d]http://musicbrainz.org - http://neo4j.com/developer/example-data

Number of Artists	1,025,077	Number of URLs	3,240,462
Number of Release Groups	1,250,415	Number of Areas	115,245
Number of Releases	1,546,965	Number of Places	14,453
Number of Mediums	1,733,205	Number of Series	3,895
Number of Recordings	15,511,654	Number of Instruments	759
Number of Tracks	19,439,391	Number of Events	9,523
Number of Labels	101,621	**Number of Nodes**	**35,778,712**
Number of Works	657,796	**Number of Relationships**	**73,433,369**

Table 1. Statistics on the MusicBrainz Dataset

4.2.1. *Overview*

Figure 14 is a high level representation of the steps that must be applied to extract a summary from a graph database:

- Step 1 exports data from Neo4j
- Step 2 imports data to In-Memory based graph processing system
- Step 3 distributes data on workers (node of the cluster that will treat the data) and executes the treatments
- Step 4 exports the result as some output depending on the way it will be used (vizualisation, analytics, import back into a graph database engine to be queried etc.)

4.2.2. *Step 1: Retrieving a dataset*

Dataset can be retrieved from different datasources. Sometimes it will be retrieved from text files such as RDF text files of CSV and other times from systems like relation/graph databases.

In this case study, data will be extracted from a graph database engine, more specifically Neo4j database. Using Graph Processing Systems in conjunction with a graph databases gives the best of both worlds: transactions and analytics. There are two types of extraction that can be done: a "raw" extraction or a specific subset of the database.

The first solution is to dump the entire database in a specific format (e.g CSV files). It can be used when all the data is required.

The second solution is to query the graph database engine to retrieve a subset of the data (a subgraph) that contains the required data. For instance, it is possible to extract the subgraph of all the nodes that are connected together with a "FOLLOW" relationship and calculate the Betweeness centrality indicator on them.

Betweenness centrality is an indicator of a node's centrality in a network. It is equal to the number of shortest paths from all vertices to all others that pass through that node. The higher the indicator is, the larger the vertex has influence

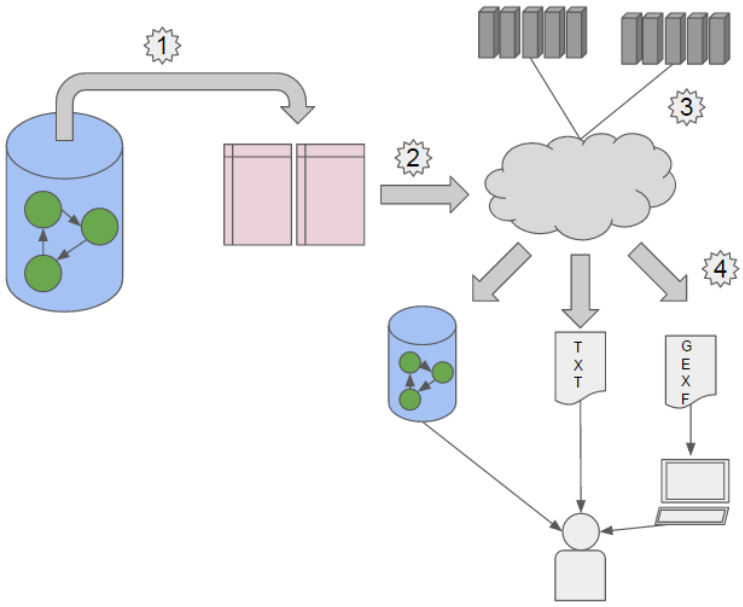


Fig. 14. Overview of In-Memory based Summary process

Listing 9. Export Vertex table

```
import-cypher -d" " -o .cypher match n return id(n), labels(n)
```

on the transfer of items through the network (under some assumptions that will not be covered in this paper).

In GraphX, a graph is a directed multigraph with user-defined objects attached to each vertex and edge. A directed multigraph is a directed graph with potentially multiple parallel edges sharing the same source and destination vertex. Such kind of graph is called "Property Graph" in GraphX terminology.

To build a property graph, two datasets are required:

(1) The first one should contain for each vertex an identifier and a property bag (if it exists)
(2) The second should contain the incoming and outgoing node id and property bag (if it exists) of each edge.

The extractions creating a structural graph summary are shown in Listings 9 and 10.

In Listings 9 and 10 the "-o" parameter is used to define the output file and the "-d" parameter to define the type of delimiter between each value of a result row

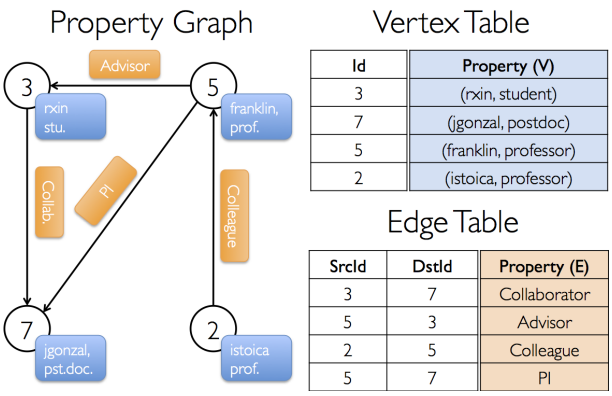


Fig. 15. GraphX Property Graph representation (official documentation)

Listing 10. Export Edge table

```
import-cypher -d" " -o edges.cypher match (a)-[r]->(b) return id(a), id(b), type(r)
```

(in this case a space). The data of each extraction are provided by the execution of a Cypher query[e] on the Neo4j database engine. Results are stored in two text files: vertices.cypher and edges.cypher.

```
Extract of vertices.cypher:
...
42886 "[Artist, Group, Entry]"
42887 "[Artist, Group, Entry]"
42888 "[Artist, Person, Female, Entry]"
42889 "[Artist, Person, Male, Entry]"
42890 "[Artist, Person, Female, Entry]"
42891 "[Artist, Person, Male, Entry]"
...
Extract of edges.cypher:
...
16553747 10 "RELEASED_IN"
16553684 10 "RELEASED_IN"
16553679 10 "RELEASED_IN"
16552723 10 "RELEASED_IN"
...
```

4.2.3. *Step 2: Import data as Resilient Distributed Datasets (RDDs)*

A Resilient Distributed Dataset (RDD) consists of data partitions distributed across the memory of the cluster machines as illustrated in Figure 16. Every RDD is

[e]using the Neo4j-shell tools available at https://github.com/jexp/neo4j-shell-tools

Listing 11. Create RDD from vertices and edges files

```
val efile = sc.textFile("edges.cypher") //A
  .persist(StorageLevel.MEMORY_AND_DISK)
val vfile = sc.textFile("vertices.cypher") //B
  .persist(StorageLevel.MEMORY_AND_DISK)

val vertexRDD: RDD[(VertexId, String)] = vfile.map(line => {
  line.split(" ", 2) match { //C
    case Array(idVertex, labels) => (idVertex.toLong, labels)
  }
})

val edgeRDD: RDD[Edge[String]] = efile.map(line => {
  line.split(" ", 3) match { //D
    case Array(vIn, vOut, rel)
      => Edge(vIn.toLong, vOut.toLong, rel.toString) //E
  }
})
```

immutable (data in RDD never changes) however a new RDD can be created from an existing one.

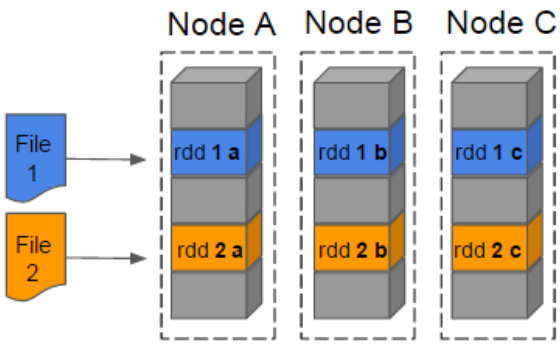


Fig. 16. Resilient Distributed Datasets

Listing 11 shows how to create two RDD from the generated two files of Step1.

```
#A Load as a RDD[String], each line is a String: "idVertex labels"
#B Load as a RDD[String], each line is a String: "idVin idVOut relType"
#C Split the line in an Array of two strings: Array(idVertex, labels)
#D Transform the array in a Tuple of two elements: idVertex as a Long and labels as a String.
   The returned type RDD[(VertexId, String)] because idVertex is cast as a VertexId
#E Split the line in an Array of three strings: Array(vIn, vOut, rel)
```

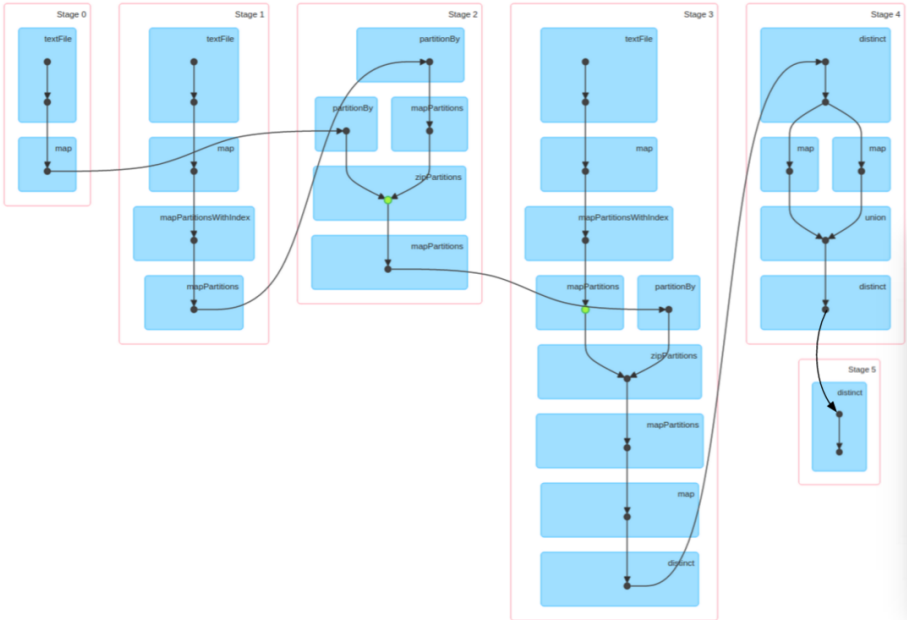


Fig. 17. Stages of GraphX Job to Summarize a Graph Database

4.2.4. *Step 3: Extracting a Graph summary*

In Spark/GraphX, operations that can be executed on the data within each partition without the need to access data in other partitions are called "stage". A job consists of one or more stages to be performed on the data.

Figure 17 shows the stages needed to import, treat, and output structural graph summary. The main steps of the algorithm are:

- Create a graph from the vertex and edges RDD
- For each triplet (incoming vertex, edge, outcoming vertex) create a tuple of three elements: the labels of the incoming vertex, the relation type, the labels of the outgoing vertex
- remove the duplicates tuples, meaning the tuples with the same incoming labels, relation type and outgoing labels
- create a new verticesRDD and edgesRDD from the tuples
- create the summary graph

Listing 12 shows the code to extract a summary of the graph database and to turn it in a new graph called summary graph. Some explanations are provided below the listing.

```
#F->I: Transforms all graph relationships in a set of uniques tuples
-----------------------------------------------------
#F Create a Graph from the vertexRDD and the edgeRDD
```

Listing 12. Create a Structural Graph Summary

```
 val graph: Graph[String, String] = Graph(vertexRDD, edgeRDD) //F

val triplets = graph.triplets.map { //G
  t => (t.srcAttr, t.attr, t.dstAttr) //H
}.distinct //I

val schemaVertices = triplets.map(t => t._1)
  .union(triplets.map(t => t._3)) //J
  .distinct //K
  .zipWithIndex //L

val schemaEdges = triplets.map(t => (t._1, (t._2, t._3))) //M
  .join(schemaVertices) //N
  .map(t => (t._2._1._2, (t._2._2, t._2._1._1))) //O
  .join(schemaVertices) //P
  .map(t => new Edge(t._2._1._1, t._2._2, t._2._1._2)) //Q

val summaryGraph = Graph(schemaVertices.map(_.swap), schemaEdges) //R
```

```
#G graph.triplets return an object of type EdgeTriplet[VD,ED] that is the representation of a relation.
   EdgeTriplets contains severals fields:
   - attr Attribute data for the edge
   - srcId Vertex id of the edge's source vertex
   - srcAttr Attribute data for the edges source vertex
   - dstId Vertex id of the edge's destination vertex
   - dstAttr Attribute data for the edges destination vertex
#H  Transforms an EdgeTriplets in a tuple of three elements: labelsVin, relType, labelsVout
#I Distinct operation to ensure unique tuple (labels, relType, labels)


#J->L: create a vertexRDD from triplets
---------------------------------------------------------------
#J Transforms (labelsVin, relType, labelsVout) => (labelsVIn)
#K Transforms (labelsVin, relType, labelsVout) => (labelsVOut)
   Union of #J and #K => (labels)
#L Distinct operation to ensure unique labels
   zipWithIndex add a generated index value to labels => (labels, idVertex)
   Return a RDD[String] called schemaVertices that contains all the labels of the graph summary

#M->Q: create an EdgeRDD from triplets
---------------------------------------------------------------
#M Transforms (labelsVin, relType, labelsVout) => (labelsVin, (relType, labelsVout))
#N Join on key labelsVin with schemaVertices => (labelsVin, ((relType, labelsVout), idVIn))
#O Transforms (labelsVin, ((relType, labelsVout), idVertex)) => (labelsVout, (idVIn, relType))
#P Join on key labelVout with schemaVertices => (labelsVout, ((idVIn, relType),idVout)
#Q Transforms (labelsVout, ((idVIn, relType),idVout) => (idVin, idVout, relType)

#R Create a new graph representing the summary of the original graph
```

The resulting graph summary is composed of:

- 157 nodes (0.0004388 % of the 35,778,712 original nodes)
- 5948 relationships (0.00000081 % of the 73,433,369 original relationships)

Listing 13. Example of a minimal GEXF file

```
<?xml version="1.0" encoding="UTF-8"?>
<gexf xmlns="http://www.gexf.net/1.2draft" version="1.2">
    <meta lastmodifieddate="2009-03-20">
        <creator>Gexf.net</creator>
        <description>A hello world! file</description>
    </meta>
    <graph mode="static" defaultedgetype="directed">
        <nodes>
            <node id="0" label="Hello" />
            <node id="1" label="Word" />
        </nodes>
        <edges>
            <edge id="0" source="0" target="1" />
        </edges>
    </graph>
</gexf>
```

4.2.5. *Step 4: Exporting and Visualizing the summary*

The summary graph can be exported in many formats. It can be output as a serialized text file that can be processed again in a Graph processing system for futher treatment or it can be output as a textual human readable text file such as:

```
(a:Artist)-[FROM_AREA]->(Country)
```

This section introduces two more possibilities: output the summary graph as it can be vizualized and analyzed in a vizualisation platform or inject the summary in a graph database engine.

Summary as a GEXF file

GEXF (GraphX Echange XML Format) is an open language for describing complex network structures, their associated data and dynamics. A basic example for a static graph containing 2 nodes and 1 edge between them is shown in Listing 13.

The aim of transforming a graph into a gexf file is to provide a format exploitable by an interactive visualization exploration platform such as Gephi.

Gephi [4] is an open-source network visualization platform that helps data analysts make hypotheses, intuitively discover patterns, and isolate structure singularities or faults during data sourcing.

The algorithm that transforms a Graph(verticesRDD, edgesRDD) into a GEXF file is shown in Listing 14.

The visualization from Gephi of the result of the transformation of the Structural summary of MusicBrainz in the GEXF file is shown in Figure 18.

Summary as a Neo4j Graph database

Step 3 allows the user to generate a new graph, which is a summary graph, from the inital graph. This summary graph can be parsed to generate a new graph database or to be inserted in the original graph database. Importing a graph summary in a graph database engine such as Neo4j can be useful for several practical

Listing 14. Export a GraphX graph to GEXF

```
def toGexf[VD,ED](g:Graph[VD,ED]) =
  "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n" +
  "<gexf xmlns=\"http://www.gexf.net/1.2draft\" version=\"1.2\">\n" +
    " <graph mode=\"static\" defaultedgetype=\"directed\">\n" +
    " <nodes>\n" +
    g.vertices.map(v => " <node id=\"" + v._1 + "\" label=\"" +
    v._2 + "\" />\n").collect.mkString +
    " </nodes>\n" +
    " <edges>\n" +
    g.edges.map(e => " <edge source=\"" + e.srcId +
    "\" target=\"" + e.dstId + "\" label=\"" + e.attr +
    "\" />\n").collect.mkString +
    " </edges>\n" +
    " </graph>\n" +
  "</gexf>"
```

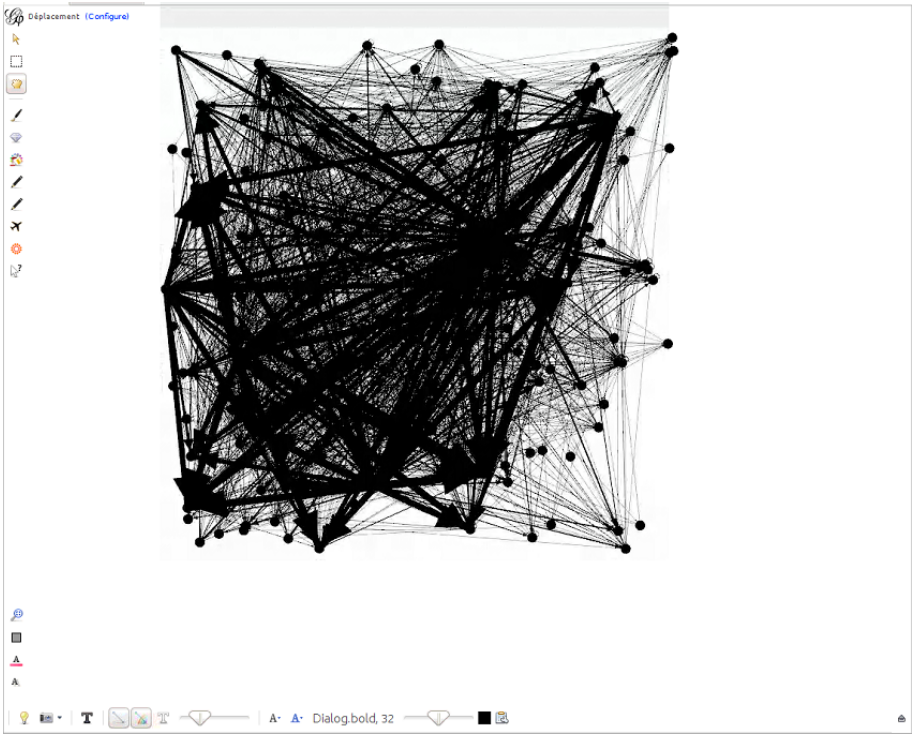


Fig. 18. Summary visualization from Gephi

applications such as running pattern matching queries, searching for intelligence, etc.

Listing 15 presents the algorithm used to produce a Cypher creation query that can be executed on a Neo4j engine. Parts of the cypher query are presented in the

Listing 15. Export a GraphX graph to Cypher

```
def toCypher[VD, ED](g: Graph[VD,ED]) = "create "+
  g.vertices.map(v => "(node" + v._1 + ":" + v._2+ "),\n").collect.↩
      mkString +
  g.edges.map(e => "(node"+e.srcId+")"+"-[:"+e.attr+"]->(node"+e.dstId+"↩
      )").collect.mkString(",\n")"
```

Listing 16. Cypher file for importing graph Summary into Neo4j

```
create (node134:"[Work, Partita, Entry]"),
(node0:"[Recording, Entry]"),
(node1:"[Other, Release, Official, Entry]"),
(node135:"[Work, Concerto, Entry]"),
(node136:"[Other, Release, Promotion, Entry]"),
(node2:"[Label, Distributor, Entry]"),
(node137:"[Work, Madrigal, Entry]"),
(node3:"[Release, SlimJewelCase, Bootleg, Entry]"),
(node4:"[Work, Entry]"),
..
(node140)-[:"CONDUCTOR_POSITION"]->(node155),
(node155)-[:"HAS_ALIAS"]->(node154),
(node155)-[:"INSTRUMENTAL_SUPPORTING_MUSICIAN"]->(node155),
(node155)-[:"COLLABORATION"]->(node155),
(node155)-[:"CATALOGUED"]->(node155),
(node155)-[:"SUPPORTING_MUSICIAN"]->(node155),
(node155)-[:"VOCAL_SUPPORTING_MUSICIAN"]->(node155),
(node155)-[:"PARENT"]->(node155),
(node155)-[:"IS_PERSON"]->(node155),
(node155)-[:"MEMBER_OF_BAND"]->(node155),
(node152)-[:"CREDITED_ON"]->(node156)
```

Listing 16.

5. Conclusion and Perspectives

This paper concerns NoSQL graph databases that are attracting more and more interest and are used in many companies and organizations dealing with large real databases. In this framework, we define four types of summaries for the purpose of assisting the users in their understanding of the data content and semantic. These summaries can be extracted using the native declarative query languages available in the NoSQL graph engines. In our work, such queries are extended using two frameworks (the Fuzzy4S DSL and the Cypherf declarative query language) to facilitate the user experience. However, experimental results run on a real database containing more than 35 million nodes and 73 million relationships show that NoSQL engines cannot manage the efficient extraction of such summaries. We thus introduce an original method based on in-memory graph processing systems.

This article opens up numerous possibilities. Future work includes experiments on several datasets. Moreover, the quality measures will be explored together with data visualization challenges. Protoforms may also be extended in the case of our

complex summaries in order to contain several relationships. Finaly, we aim to develop algorithms to retrieve the most significant "contradictory" summaries, such as *Most of the students rent a one-room studio in the city but the youngest ones rent a room in a student residence.*

1. Aggarwal, C.C., Wang, H. (eds.): Managing and Mining Graph Data, Advances in Database Systems, vol. 40. Springer (2010)
2. Almeida, R.J., Lesot, M., Bouchon-Meunier, B., Kaymak, U., Moyse, G.: Linguistic summaries of categorical series for septic shock patient data. In: FUZZ-IEEE 2013, IEEE International Conference on Fuzzy Systems, Hyderabad, India, 7-10 July, 2013, Proceedings. pp. 1–8. IEEE (2013), `http://dx.doi.org/10.1109/FUZZ-IEEE.2013.6622581`
3. Angles, R., Gutiérrez, C.: Survey of graph database models. ACM Comput. Surv. 40(1) (2008)
4. Bastian, M., Heymann, S., Jacomy, M.: Gephi: An open source software for exploring and manipulating networks (2009), `http://www.aaai.org/ocs/index.php/ICWSM/09/paper/view/154`
5. Bex, G.J., Neven, F., Vansummeren, S.: Inferring XML schema definitions from XML data. In: Koch, C., Gehrke, J., Garofalakis, M.N., Srivastava, D., Aberer, K., Deshpande, A., Florescu, D., Chan, C.Y., Ganti, V., Kanne, C., Klas, W., Neuhold, E.J. (eds.) Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007. pp. 998–1009. ACM (2007)
6. Bouchon-Meunier, B., Moyse, G.: Fuzzy linguistic summaries: where are we, where can we go ? In: IEEE Conf. on Computational Intelligence for Financial Engineering & Economics (CIFEr). pp. 317–324. CIFEr 2012, IEEE (2012)
7. Cao, J., Xing, C.: Data source recommendation for building mashup applications. In: Web Information Systems and Applications Conference (WISA) (2010)
8. Castelltort, A., Laurent, A.: Fuzzy queries over nosql graph databases: Perspectives for extending the cypher language. In: International Conference on Processing and Management of Uncertainty in Knowledge-Based Systems. Springer (2014)
9. Castelltort, A., Laurent, A.: Extracting fuzzy summaries from nosql graph databases. In: International Conference on Flexible Query Answering Systems. Springer (2015)
10. Cattell, R.: Scalable SQL and NoSQL data stores. SIGMOD Record 39(4), 12–27 (2010)
11. Cingolani, P., Alcala-Fdez, J.: jfuzzylogic: a robust and flexible fuzzy-logic inference system language implementation. In: FUZZ-IEEE. pp. 1–8. Citeseer (2012)
12. Cingolani, P., Alcalá-Fdez, J.: jfuzzylogic: a java library to design fuzzy logic controllers according to the standard for fuzzy control programming. International Journal of Computational Intelligence Systems 6(sup1), 61–75 (2013)
13. Conte, D., Foggia, P., Sansone, C., Vento, M.: Thirty Years Of Graph Matching In Pattern Recognition. International Journal of Pattern Recognition and Artificial Intelligence (2004)
14. Cook, D.J., Holder, L.B.: Mining Graph Data. John Wiley & Sons (2006)
15. D., R., R.R., Y.: Finding fuzzy and gradual functional dependencies with summarysql. Fuzzy Sets and Systems 106, 131–142 (1999)
16. De Raedt, L.: A perspective on inductive databases. SIGKDD Explor. Newsl. 4(2), 69–77 (Dec 2002)
17. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Communications of the ACM 51(1), 107–113 (2008)
18. Fowler, M.: Domain Specific Languages. Addison-Wesley Professional, 1st edn. (2010)

19. Ghemawat, S., Gobioff, H., Leung, S.T.: The google file system. In: ACM SIGOPS operating systems review. vol. 37, pp. 29–43. ACM (2003)
20. Han, J., Haihong, E., Le, G., Du, J.: Survey on nosql database. In: Proc. of the 6th International Conference on Pervasive Computing and Applications (ICPCA). pp. 363–366 (2011)
21. Kacprzyk, J., Wilbik, A., Zadrozny, S.: An approach to the linguistic summarization of time series using a fuzzy quantifier driven aggregation. Int. J. Intell. Syst. 25(5), 411–439 (2010), http://dx.doi.org/10.1002/int.20405
22. Kacprzyk, J., Zadrozny, S.: Linguistic database summaries and their protoforms: towards natural language based knowledge discovery tools. Inf. Sci. 173(4), 281–304 (2005), http://dx.doi.org/10.1016/j.ins.2005.03.002
23. Klement, E.P., Pap, E., Mesiar, R.: Triangular norms. Trends in logic, Kluwer Academic Publ. cop., Dordrecht, Boston, London (2000)
24. Kuramochi, M., Karypis, G.: Frequent subgraph discovery. In: Proceedings of the 2001 IEEE International Conference on Data Mining. pp. 313–320. ICDM '01, IEEE Computer Society, Washington, DC, USA (2001), http://dl.acm.org/citation.cfm?id=645496.658027
25. LeFevre, K., Terzi, E.: Grass: Graph structure summarization. In: Proceedings of the SIAM International Conference on Data Mining, SDM 2010, April 29 - May 1, 2010, Columbus, Ohio, USA. pp. 454–465. SIAM (2010)
26. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. pp. 135–146. ACM (2010)
27. Neo4j: Cypher - refcard, http://docs.neo4j.org/refcard/1.9/
28. Partner, J., Vukotic, A., Watt, N.: Neo4j In Action. Manning (2013)
29. Robinson, I., Webber, J., Eifrem, E.: Graph Databases. O'Reilly (2013)
30. ThoughtWorks: Technology advisory board (May 2013), http://thoughtworks.fileburst.com/assets/technology-radar-may-2013.pdf
31. Yager, R.R.: A new approach to the summarization of data. Information Sciences 28(1), 69 – 86 (1982)
32. Yan, X., Han, J.: gspan: Graph-based substructure pattern mining. In: Proceedings of the 2002 IEEE International Conference on Data Mining. pp. 721–. ICDM '02, IEEE Computer Society, Washington, DC, USA (2002), http://dl.acm.org/citation.cfm?id=844380.844811