



HAL
open science

An embedding driven approach to automatically detect identifiers and references in document stores

Manel Souibgui, Faten Atigui, Sadok Ben Yahia, Samira Si-Said Cherfi

► To cite this version:

Manel Souibgui, Faten Atigui, Sadok Ben Yahia, Samira Si-Said Cherfi. An embedding driven approach to automatically detect identifiers and references in document stores. *Data and Knowledge Engineering*, 2022, 139, pp.102003. 10.1016/j.datak.2022.102003 . hal-03617510

HAL Id: hal-03617510

<https://inria.hal.science/hal-03617510>

Submitted on 22 Jul 2024

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.



Distributed under a Creative Commons Attribution - NonCommercial 4.0 International License

An Embedding Driven Approach to Automatically Detect Identifiers and References in Document Stores

Manel Souibgui^{a,b,**}, Faten Atigui^{a,*}, Sadok Ben Yahia^{b,c,*} and Samira Si-Said Cherfi^{a,*}

^aConservatoire National des Arts et Métiers, CEDRIC-CNAM, France

^bUniversity of Tunis El Manar, Faculty of Sciences of Tunis, LIPAH, Tunisia

^cDepartment of Software Science, Tallinn University of Technology, Estonia

ARTICLE INFO

Keywords:

Business Intelligence and Analytics

ETL

Document stores

Join

Identifier discovery

Reference discovery

ABSTRACT

NoSQL stores have become ubiquitous since they offer a new cost-effective and schema-free system. Although NoSQL systems are widely accepted today, Business Intelligence & Analytics (BI&A) wields relational data sources. Exploiting schema-free data for analytical purposes is a challenge since it requires reviewing all the BI&A phases, particularly the Extract-Transform-Load (ETL) process, to fit big data sources as document stores. In the ETL process, the join of several collections, with a lack of explicitly known join fields is a significant dare. Detecting these fields manually is time and effort-consuming and infeasible in large-scale datasets. In this paper, we study the problem of discovering join fields automatically. We introduce an algorithm that aims to automatically detect both *identifiers* and *references* on several document stores. The *modus operandi* of our approach underscores three core stages: (i) global schema extraction; (ii) discovery of candidate *identifiers*; and (iii) identifying candidate pairs of *identifier* and *reference* fields. We use scoring features and pruning rules to discover true candidate *identifiers* from many initial ones efficiently. To find candidate pairs between several document stores, we put into practice *node2vec* as a graph embedding technique, which yields significant advantages while using syntactic and semantic similarity measures for pruning pointless candidates. Finally, we report our experimental findings that show encouraging results.

1. Introduction

For more than a decade, NoSQL datastore became commonly used to store big data. These systems are schema-free and built upon distributed systems, which makes them easy to scale and shard. However, in a rush to solve the challenges of big data and large numbers of concurrent users, NoSQL abandoned some of the core features of relational databases, which make them highly scalable and easy to use [1; 2; 3]. Although the use of NoSQL systems is widely accepted today, Business Intelligence & Analytics (BI&A) wields relational data sources [4].

In fact, from the earliest days of data warehousing, the qualities of the relational model have been highly valued in the quest for data consistency and quality. Exploiting NoSQL stores for analytical purposes requires reviewing all the BI&A phases.


NoSQL systems are based on four different families of models: key-value, column-oriented, document-oriented, and graph-oriented [5]. According to DB-Engines Ranking¹, MongoDB, which is a document database, is the most well-known among NoSQL databases.

In our previous work [6], we have proposed a hybrid BI&A approach that considers both schemaless data sources and analytical needs to explore over one document store efficiently. The ETL, i.e., extract, transform and load, is the cornerstone of our approach. However, carrying out primary ETL operations, especially how to correctly join two different document stores, i.e., collections, still being a challenge [7; 8].

Fetching relevant data that meet the decision-makers requirements often needs to access more than one document store using the join operation. While joining tables in relational data sources is straightforwardly owed to the availability of a precise join key, in document stores, collections are the furthest from having an exact join key because of the absence of integrity constraints. So, identifying the "joinable" fields to stick to two document stores is a tricky

*Corresponding author

**Principal corresponding author

 manel.souibgui@fst.utm.tn (M. Souibgui); faten.atigui@cnam.fr (F. Atigui); sadok.ben@taltech.ee (S. Ben Yahia); samira.cherfi@cnam.fr (S. Si-Said Cherfi)

¹<https://db-engines.com/en/ranking>

challenge. Despite its importance, no previous work has paid close heed to detect join key pairs in the context of NoSQL stores, particularly in document-oriented stores.

For this, we introduce IRIS-DS (**I**dentifiers and **R**eferences **D**IScovery in **D**ocument **S**tories), a new approach that aims to discover the pairs of join keys (*identifier*, *reference*) starting from more than two document stores.

Thus, the sighting features of our approach are:

- to the best of our knowledge, no former approach has been dedicated to joining keys discovery in the context of document stores. We consider both composite and non-composite join keys.
- we adapt existing features, which identify candidate *identifiers*, to the context of document stores and introduce new ones.
- we propose an alternative method to detect the hidden types while extracting a document schema. This method enables the avoidance of misleading results generated by a wrong data type.
- to detect the candidate pairs of *identifier* and *reference*, we practice the *node2vec* technique, which yields significant advantages.
- unlike existing works, we use both syntactic and semantic similarity measures for pruning pointless candidates.

Remark A partially outdated version of this paper has already been published in [9]. In comparison with that conference version, we propose new add-on contributions:

- In [9], we only considered non-composite join keys. Here, we focus on both composite and non-composite join keys.
- In [9], we used a naïve approach based on a set of rules. Here, we model the problem of detecting the pair of join keys as detecting similarities across nodes in a graph. Thus, we put into practice the embedding technique, particularly *node2vec*, which can be used in different machine learning tasks.
- We explore in [9] the existing approaches to extract a document schema, whereas we propose here an alternative method to identify hidden types while extracting the schema.
- In this paper, we provide a more detailed case study and a wider experimental evaluation.

The paper’s outline is as follows: in Section 2, we recall the basic concepts related to document stores followed by a motivating example. In Section 3, we scrutinize the related literature. In Section 4, we introduce the core stages of our approach. In Section 5, we explain the overall algorithm. In Section 6, we present a case study of our approach. Finally, we discuss the experimental results in Section 7, and we allude to takeaway messages and sketch issues of future work in Section 8.

2. Problem Setting

This section goes through the basic concepts related to document stores before presenting a motivating example illustrating the main challenges.

2.1. Preliminaries

Document stores, *aka document-oriented databases*, are one of the four families of NoSQL stores. A document (cf., Definition 1) is the basic concept of document stores. A document has a schemaless nature: it does not have up-front constraints or a strictly predefined schema. For instance, some attributes in documents can be missed entirely, have null values, or have different data types.

To elucidate the basic concepts, we briefly present, in Table 1, the terminology related to document stores and their associated concepts in relational databases.

JavaScript Object Notation

(JSON²) is currently the most commonly adopted format that we will use in the remainder. Syntactically, Keys and values are separated by colons, while commas separate key-value pairs. Objects and arrays can be embedded inside a document. Objects use curly braces symbols and contain an unordered set of key-value pairs, while arrays use square bracket symbols and include an ordered collection of values.

²<https://www.json.org>

Table 1

Main terminology of document stores and its equivalent in relational databases

Document stores	Relational databases
database/document stores	database
collection	table
document	row
field	column

Definition 1. (Document and Collection) A document d is an object. Each object contains an unordered set of key-value pairs (keys are also called names [10]); a key is a string, while a value can be a primitive value (i.e., Number, String, or Boolean), an object, an array of values, or null. A collection C is an array of documents.

In our work, we consider scattered data over several collections. Since in documents it is common to nest one object into another object [11], we regard the different cases of nesting objects:

- The document-related objects are represented separately from the original document (i.e., in another collection) and are referenced using identifiers.
- All the document-related objects are nested in the original document with different nesting depths.

After presenting the basic concepts related to document stores, we provide, in the remainder, a motivating example that throws light on the noteworthy challenges of detecting the join keys in the context of document stores.

2.2. Motivating Example

Here, we present a motivating example that smoothly sheds light on the significant challenges of detecting the join keys in the context of document stores. We consider n collections denoted C_1, \dots, C_n , that store two main topics, to wit orders made on marketplaces like *Amazon* and *Cdiscount*, and deliveries insured by brands like *Bosch* and *Moulinex*. For the sake of simplicity, Figure 1 shows two collections, C_1 for orders and C_2 for deliveries. Suppose that we are interested in analyzing the deliveries' delay, called DD . We need to compute the delay as the difference between the actual delivery date versus the expected one. C_1 contains all the orders made on *Amazon* marketplace and C_2 contains the deliveries done by the Bosch brand to different marketplaces.

As $DD = \text{deliveryDate} - \text{expDeliveryDate}$ where $\text{deliveryDate} \in C_1$ and $\text{expDeliveryDate} \in C_2$, it is of paramount importance to correctly join C_1 and C_2 in order to compute the DD metric. The key fields that join C_1 and C_2 are `orderID` as an *identifier* in C_1 and `orderCode` as a *reference* in C_2 . If we use existing algorithms dedicated to relational databases in order to automatically detect join keys, it would be unfitting. In fact, the `orderID` in C_1 has a null value in the third document and is absent in the fourth one. Additionally, the set of `orderCode` values: {Amazon_Bosch1, eBay_Bosch1, Cdiscount_Bosch1} is not included in the set of `orderID` values: {Amazon_Moulinex1, Amazon_Bosch1, Amazon_KenWood1}.

By and large, document stores have different aspects of heterogeneity that could be:

- Intra-document: the variety is mainly related to heterogeneous structures within a document. For instance, a JSON array can hold objects with different fields. As shown in Figure 1 (document 1 of C_1), the `product` field is a JSON array that contains different objects in terms of fields number and types. The first object comprises three fields, while the second one contains an array of simple values (i.e., [17.4, 17.4, 22.1]).
- Inter-documents: the variety is mainly related to documents with different fields. Fields can be present in some documents and absent in others. For instance, the `product` field in the first document of C_1 does not contain the same fields as in the second document. Moreover, fields do not have the same order (e.g., `orderDate`), which is an intrinsic characteristic of an object.
- Inter-collections: the variety lies in having different schemas with different knowledge domains.

In document stores, joining two collections is a thriving challenge because of their clueless schemaless nature. In fact, (i) unlike the primary key which is unique and not null, *identifier* as all the other fields, can be missing in some



Figure 1: An excerpt of two collections

documents or can, normally and not exceptionally, have null values; (ii) document stores do not have "precise" join keys beforehand due to the absence of integrity constraints; and (iii) unlike a relational database, *reference* values are not included in the *identifiers*' values, so it is impossible to use the inclusion dependencies in order to automatically detect *identifiers* and *references*.

3. Related Work

Our primary objective is to identify "joinable" key fields, i.e., *identifier* and *reference* fields, to perform a join operation between two different document stores. We survey, in this section, existing works that paid attention to this issue. We identify three major streams of approaches: (i) dealt with the ETL process over NoSQL stores, where we focus on the works addressing the join operation in the context of NoSQL stores; (ii) dealt with joinable tables discovery; and (iii) proposed contributions for the primary key and foreign key detection in the context of relational databases.

3.1. ETL over NoSQL Stores

Few researchers have addressed the problem of ETL in the context of NoSQL stores, particularly the document-oriented ones [12; 13]. For example, in [14], the authors proposed a tool called *BigDimETL*, dealing with the ETL development process in the context of NoSQL stores. Data are extracted from a document store to be converted to

a column-oriented store to apply partitioning techniques. The approach aims to minimize ETL time consuming by parallelizing the treatment of *select*, *project*, and *join* operations.

Along with these works, several approaches have focused on schema extraction [15], i.e., a list of document fields with their types, from document stores. Since it is a crucial step in an ETL process dealing with document stores, we have studied these distinct contributions in the subsection 3.1.1. On the other hand, we investigate the focuses on the join operation in the context of NoSQL stores as detailed in subsection 3.1.2.

3.1.1. Schema Extraction

The authors in [16] have proposed a method to extract the global schema of a collection of JSON documents using a graph representation. The method reveals structural data outliers using similarity measures to capture the degree of heterogeneity of JSON data. In [17], the authors have proposed a schema management framework to extract distinct schemas in a collection. To have one single view of collection data, they offer a new concept called *skeleton* used as a relaxed form of the schema. The approach supports queries by allowing developers to find a suitable collection to persist a new document.

In the same direction, the authors in [18] have proposed a tool called *JSONDiscoverer* that aims to represent implicit structures of a given set of JSON documents as a UML class diagram. This step is followed by an advanced discovery that infers the global schema of a set of JSON documents. Baazizi et al. [19] were interested in schema inference of massive JSON datasets. The distinguishing feature of their approach is that it is parametric and allows the user to specify the degree of preciseness and conciseness of the inferred schema. Besides, Gallinucci et al. [20] have extended the level of schema extraction of a collection of JSON documents, with schema profiling techniques, to capture the hidden rules explaining schema variants.

Although many authors have conducted schema extraction, this problem is still insufficiently explored. The major downside in their approaches is that they do not try to find the hidden type of each field while extracting the schema. Although most of the above approaches have only focused on the first phase of the ETL process, i.e., the extraction phase or extracting document schema, contributions in the transformation phase remain limited and require more effort. Besides, most of these contributions consider as input only a single collection of documents.

3.1.2. Join Operation in the Context of NoSQL Stores

Several questions regarding the join operation in NoSQL stores need to be addressed. The join operation is not explicitly available in NoSQL stores [7]. Few researchers have addressed this issue. For instance, in [7], the authors discussed the impact of performing the join operation in document stores. They have proposed an algorithm that performs an Inner-join operation on two MongoDB collections at the application layer. The algorithm requires to be fueled with join keys. Besides, since the join is mandatory for querying tasks, we have also studied the dedicated querying approaches. In [21], the authors proposed *Squerall*, a framework that enables querying of heterogeneous data on the fly without prior data transformation. *Squerall* supports MongoDB, Cassandra, and various sources. In addition, the framework allows for the user to declare modifications for altering join keys during query time to make data joinable. In [22], Kondylakis et al. have proposed a data management solution allowing joins over NoSQL Cassandra databases where the primary keys are considered as partition keys. The approach proposed in [23] inputs two sets of values from join columns and produces a predicted join relationship using an extensive table corpus.

The previous works have all addressed the join operation in NoSQL stores. However, we note that they all rely on a strong assumption: having the join keys beforehand. It is worth mentioning that, in NoSQL stores, no prior works have proposed a method to find the pairs of join keys, i.e., both *identifiers* and their respective *references*. Hence, it would be of benefit to examine prior research carried out within a relational database context.

3.2. Joinable Table Discovery

In [24], Bogatu et al. propose an approach that aims to detect if attribute values coming from different sources belong to the same domain. Based on this, it is decided whether the sources are candidates to be joined or unioned to populate a target. In [25], given a table and one join column, authors aim to find joinable tables in data lakes by formulating the problem as an overlap set similarity search. Finding a joining table is based on a given join column regardless of whether it is a primary/foreign key or not. Additionally, their approach is based solely on values, which is unhandy in a NoSQL context. Indeed, they ignore numeric values since they create casual joins that are not meaningful. However, numeric values are exciting to discover identifiers and references in document stores in our work.

Similarly, in [26], Fernandez et al. propose an approach that aims to find objects that are semantically related. However, to identify semantic links, their approach requires domain-specific knowledge encoded in an ontology, which is not always available.

These works aim to identify relatedness between tables when explicit relationships between them are missing. The relatedness can be for joinability or unionability. Given a set of tables, they search significant datasets to populate a target table. The results generated by these approaches are in the form of similar tables or similar attributes regardless of whether among these attributes there are primary/foreign keys or not. However, our objective is to determine identifiers and references among several attributes that can be similar between two collections. On the other hand, even if these works are interesting, they are dedicated to tabular data. Hence, it is not suitable for schemaless or schema variant data stores, as in document stores, where we have different levels of objects nesting, null and missing values. Besides, by and large, these approaches are based solely on values, which is unhandy in a NoSQL context.

3.3. Primary Key and Foreign Key Discovery in Relational Databases

This subsection presents the contributions that have dealt with detecting primary keys and foreign keys in relational databases. The authors in [27; 28; 29] have paid attention to foreign keys detection, assuming the presence of primary keys. Quite freshly, Jiang and Naumann [30] have proposed an approach to discover both primary keys and foreign keys automatically for a given relational database. The approach is based on the functional dependencies that describe the characteristics of a table or relationships between tables, namely unique column combinations and inclusion dependencies. Both types of dependencies have been used to detect primary keys and foreign keys in relational databases. A unique column combination is the set of attributes whose projection contains only the column combinations having unique and non-null values. In document stores, fields can easily be missing in some documents or have null values usually and not exceptionally. Their work is based on the set of inclusion dependencies given as input. However, this assumption could not pertain to the context of document stores as described in our motivating example. Even if this previous work [30] is the closest one to our problem, we cannot apply it out of the context of relational databases. Thus, we have undergone a rethinking of the problem by using alternative methods adapted to document stores' schemaless nature.

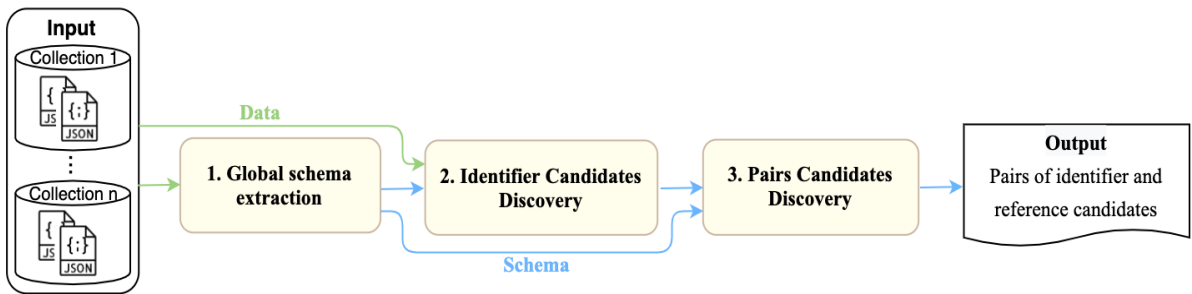


Figure 2: Approach overview

4. The IRIS-DS Approach to Automatically Detect Identifiers and References

In this section, we thoroughly describe the core stages of our approach : (i) global schema extraction; (ii) discovery of candidate *identifiers*; and (iii) identifying candidate pairs of key fields. We summarize our approach in Figure 2, where we present the sequential order of the three stages, starting from several collections. We provide a detailed description of each stage in the remainder.

4.1. Global Schema Extraction

Document stores have a dynamic schema, mainly evident through the presence or absence of specific fields with various types. Although this schemaless nature guarantees some perks, the lack of schema information significantly

negatively affects data processing tasks such as data integration and analysis. Hence, inferring a flat document schema (cf., Definition 2) and the global schema of each collection (cf., Definition 3) is of paramount importance for the data integration.

Definition 2. (Document flat schema) A list of fields with their associated types. Let $S_D = \{(p, t)_i / 1 < i < k\}$ be the schema associated to a JSON document D . It consists of k pairs (p, t) , such that:

p : the field path from the document root. It is the unique identifier of each field.

t : the field type. Since a field can have a different value types, we consider the most frequent type.

Example 1. Figure 3 depicts a JSON document on the left and shows its associated flat schema on the right, where $\$$ symbol represents the document root.

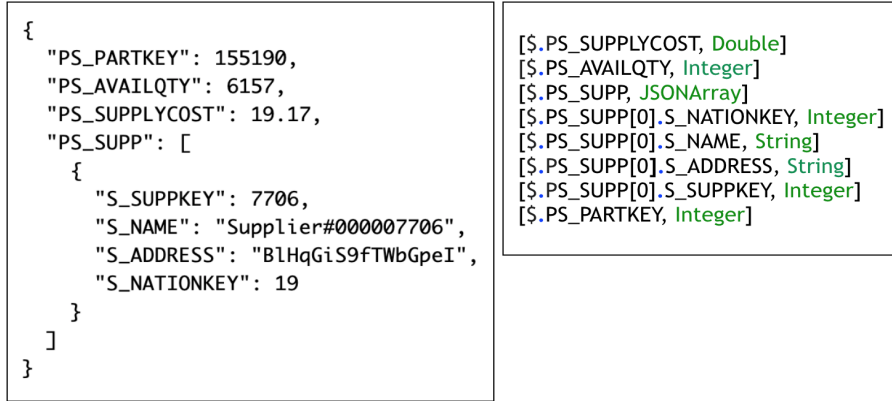


Figure 3: Example of document flat schema generation

Definition 3. (Collection global schema) the global schema of a collection C is $SG(C) = \bigcup_{j=1}^l S_{D_j}$, where S_{D_j} is the flat schema associated with a JSON document D_j that belongs to the collection C and l denoting the number of distinct documents schemas that exist in a collection C .

As mentioned in the related work, we report several methods in the literature to address this issue. The proposed solutions generate a schema as a set of document fields with their associated types. However, the types are not thoroughly described. As far as we know, no previous work has provided a schema that yields the real hidden type for documents fields.

Since we can hide a real primitive type under another primitive type, we propose an alternative method that aims to check the type of each field to detect such cases. For instance, if the values of a given field f_i are of Float type, e.g., 5.02, while they are represented between quotes: "5.02", in this case, the real type must be Float instead of String. Checking the actual type requires access to the values, making it a little awkward to use in a NoSQL context. We thus suggest using the random sampling technique. The latter is an unbiased form to collect a subset of data using randomness [31]. Hence, given a sample of values of a field f_i belonging to a collection C , we check the actual type using a set of regular expressions. For instance, in order to check if the type is really String or Float, we have used these regular expressions:

- ". * [a - zA - Z]. *": check if the type is String.
- "[-+]?[0 - 9] * \\.?[0 - 9]+": check if the type is Float.

Identifying the hidden types is extremely important in determining the candidate *identifiers* as detailed in the following stage of our approach.

Since our approach starts on several collections, it is worth noting that the number of global schemas is less than or equal to the initial number of groups because several collections can share the same schema.

4.2. Discovery of Candidate Identifiers

In this stage, we restrict our focus on the discovery of candidate *identifiers* on which depends the identification of the pairs (*identifier*, *reference*) afterwards. Hence, this stage aims to start with identifying an initial list of candidate *identifiers* for each collection and come out with a refined list after the scoring and the pruning phases.

4.2.1. Identifying the Initial List of Candidate Identifiers

In JSON format, each element in an array can be of three types: objects (set of key-value pairs), arrays, and/or simple values (Integer, String, Boolean, etc.). Since we are looking for candidate *identifiers*, we are interested in JSON elements represented as key-value pairs, whether nested in objects or arrays. Let us consider a collection C , and its global schema $SG = T_c \cup T_s$, where T_c is the set of fields with complex types (JSON object or JSON array) and T_s the ones with simple types (primitive types).

Since an *identifier* can not, probably, be a *JSON object* nor a *JSON array*, then we limited the search space of candidate *identifiers* to the ones having simple types (T_s). Moreover, due to schema flexibility, documents within the same collection may present some structural variety. Some fields are not present in all documents or may have null values. Thus, we classify fields in T_s as being required (F_r) or optional (F_o) (cf., Definition 4). We limited the search space of candidate *identifiers* to the required fields within F_r . Then, within F_r , we look for single fields and combinations of fields having unique values. Throughout this paper, we use the acronym *IDc* to refer to a candidate *identifier* (cf., Definition 5) which can be constituted of one or more fields. We note that we regard only minimal unique fields' combinations. To put it another way, let $\text{Comb} = \{f_1 \dots f_n\}$ be a combination of fields. Comb is considered as a minimal unique combination if $\forall f_i \in \text{Comb}$ is not unique. The generation of these combinations is done steadily. Firstly, we look for the *IDc* made up of single fields. Secondly, we generate combinations of two fields from the remaining list of non-unique fields that are frequent and of simple types. We repeat the same step for the triad combinations.

Checking unique values brings us back to evoke the problems related to duplicate detection in case of missing values. To better understand this point, consider for example two instances' values of a composite identifier: ("1", "2") and ("1", "null"). This case calls into question some past assumptions: (i) assume that the two instances' values are identical; or (ii) assume that the two instances' values are different. Multiple strategies have also been proposed to deal with missing values. By and large, an identifier must be unique and not null [32]. However, owing to schema flexibility, any field in each collection can be missing in some documents or have null values. Thus, we apply the uniqueness checking only to required fields (cf., Definition 4). If the field is required (with a high frequency of appearance with values different from null and missing), we verify the uniqueness constraint. The latter is verified based on non-null values [33]. If so, the field is retained as a candidate *identifier*, and its score is computed in the following stage.

Definition 4. (Required Field) A field f_i is required whenever its frequency is greater than or equal to a threshold ϵ . The frequency is computed as $\text{freq}(f_i) = \frac{|\tilde{k}_c|}{|D_c|}$ [10], where $|\tilde{k}_c|$ is the number of documents in which the key in the given field is not missing and has a not null value, and $|D_c|$ stands for the total number of documents within the collection C .

Definition 5. (Candidate Identifier) Given a collection C , a candidate identifier (*IDc*) is one or more fields that are of simple types, required, and form a minimal combination of unique values.

4.2.2. Scoring Candidate Identifiers

In the context of relational sources, we had explored several primary key features in the literature [30; 34] to distinguish valid primary keys from spurious ones. We reuse some of these features that we have adapted to the context of document stores in our proposal, and we introduce extra features: **depth**, **data type**, and **name prefix**. We describe these features in the following.

- **Cardinality:** in practice, schema designers show a tendency to use fewer fields for the identifier definition: fewer fields enable better understandability and maintainability. The score function is defined as $\frac{1}{|IDc|}$
- **Name prefix/suffix:** *identifiers* are generally identified by their field name prefix/suffix. We consider the list of possible names' prefixes/suffixes for identifiers as: "id", "key", "nr", "no", "pk", "num", and "code". We define the score function as $\frac{\text{prefixSuffix}(IDc)}{|IDc|}$, where $\text{prefixSuffix}(IDc)$ counts the number of fields in the *IDc* whose name contains one of the prefixes/suffixes mentioned above.
- **Depth:** *identifiers* often have a shallow depth. In fact, nested fields has a lower chance to be an *identifier* for the entire collection. We define the score function as $\frac{1}{|IDc|} \left(\sum_{i=1}^{|IDc|} \frac{1}{\text{depth}(f_i)+1} \right)$, where $f_i \in IDc$.

- **Data type:** hands-on hints show that a field is prone to be an *identifier* whenever its data type is Integer or String.

We define the data type score as $\frac{1}{|IDc|} \left(\sum_{i=1}^{|IDc|} type(f_i) \right)$ where $type(f_i)$ is a binary function that returns one if the field f_i has a *String* or an *Integer* type or zero otherwise.

- **Value length:** fields that are used as *identifiers* are supposed to have a short value length, as they are typically non-semantic *identifiers*. The score function is defined as $\frac{1}{\overline{LengthMax(f_i) - n}}$, where
 - $\overline{LengthMax(f_i)}$ is the average length of the longest values associated to the *IDc* fields. This function is defined as $\overline{LengthMax(f_i)} = \frac{1}{|IDc|} \sum_{i=1}^{|IDc|} LengthMax(f_i)$.
 - n is a parameter used to penalize long values.

Value length is a value-based feature, which makes it cumbersome to be used in a NoSQL context. To take advantage of this critical feature, we use the random sampling technique as reported earlier (cf., Subsection 4.1).

We use these features to score each candidate *identifier* related to each collection. For the total score, we use the overall average of the computed scores.

4.2.3. Pruning Candidate Identifiers

Expectedly, the set of the initial candidate *identifiers* is very large. Filtering techniques are essential to get rid of irrelevant candidate *identifiers*. For this, for each collection, we score each candidate *identifier* using the above-described features. In this paper, we use the cliff technique [30] (cf., Definition 6). As described in Example 2, the set of candidate *identifiers* is split into two parts: (i) *Upper*: it contains the candidates before the cliff; and (ii) *Lower*: it contains the remaining candidates. Since the candidates that appear in the *Upper* part do have the highest scores, we prune the candidates belonging to the *Lower* part. We note that in case of multiple instances of cliff we retain all candidates before the last cliff (cf., Definition 7).

Definition 6. (Cliff [30]) Given $S = \{S_1, S_2, \dots, S_n\}$, the sorted score list of candidate identifiers belonging to one collection, and their corresponding score difference list, $SD = \{SD_1, SD_2, \dots, SD_{n-1}\}$, where a score difference is defined as $SD_i = S_i - S_{i+1}$ of each pair of adjacent candidates, the cliff is the pair of adjacent candidates S_i and S_{i+1} having the largest *SD* score.

Definition 7. (Multiple instances of the cliff) Given $SD = \{SD_1, SD_2, \dots, SD_{n-1}\}$ the set of score differences where SD_i is the largest score. $\forall SD_j \in SD$, if $\exists SD_j \mid SD_j = SD_i$ such that $j \neq i$ then we prune the lower part of SD_k , where

$$k = \begin{cases} i, & S_i < S_j \\ j, & \text{otherwise} \end{cases} \quad (1)$$

Example 2. As depicted in Figure 4, we suppose having a list S of candidate identifiers' scores, which are decreasingly sorted as follows: $S = \{1.0, 0.6, 0.58, 0.39, 0.23, 0.1\}$. We generate the score difference list $SD = \{0.4, 0.02, 0.19, 0.16, 0.13\}$. The cliff is the largest score difference value in SD , i.e., 0.4. The green and the pink squares, shown in Figure 4, respectively illustrate, the *Upper* and the *Lower* parts.

The pruning phase is dedicated to refining the initial list of candidate *identifiers* for each collection. Furthermore, the refined list is used to identify candidate pairs of key fields as detailed in the remainder.

4.3. Identifying Candidate Pairs of Key Fields: Identifiers and References

This stage aims to constitute the pairs of *identifier* and *reference* fields related to every two document stores. Given two collections, we search the candidate pairs in both directions. Roughly speaking, we firstly find for each candidate *identifier* $IDc(C_1)$ of the first collection C_1 the most similar candidate *reference*, if it exists, from the set of fields $F(C_2) = f_1, \dots, f_n$ of the second collection C_2 . Secondly, we find for each $IDc(C_2)$ the most similar candidate *reference*, if it exists, from $F(C_1) = f_1, \dots, f_n$. We filter the obtained candidate pairs afterward.

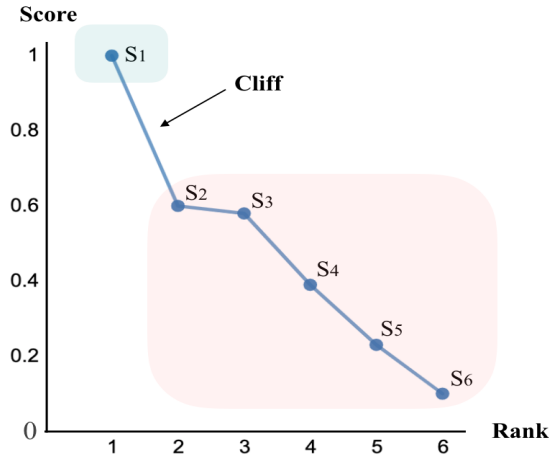


Figure 4: Example of the cliff method applied to the ranked scores of candidate *identifiers*

The degree of similarity depends on the shared properties' values between an *identifier* and its *reference*. In order to inspect these similarities, we model the relation between $IDc(C_i)$ and $F(C_j) = f_1, \dots, f_n$ as a graph.

Graph-based techniques are becoming ubiquitous since they are essential to yield new insights into data. However, graphs with their traditional representation do not allow to entirely take benefit from the existing machine learning approaches and techniques [35].

In recent years, there has been considerable interest in graph embedding that aims to convert graph nodes into a lower-dimensional space in which the neighborhood similarity between nodes is preserved in the embedded space [36]. To this end, using embedding and vector spaces offers a richer toolset and machine learning approaches. Our objective is in line with the graph embedding goal, particularly the node embedding one.

Hence, we uptake a graph embedding technique, called *node2vec* [37]. The latter learns feature representations for the nodes across a graph to be used for different machine learning tasks. *Node2vec* explores network neighborhood. It designs a flexible neighborhood sampling, called a random walk, by interpolating between Breadth-first and Depth-first sampling strategies.

The graph we defined is simple³, heterogeneous⁴, undirected and unweighted.

We denoted it as $G(V, E)$ where

- $V = \{IDc(C_i), F(C_j), PV\}$: the set of vertices (aka nodes) with three types, where
 - $IDc(C_i)$: a candidate *identifier* of the collection C_i .
 - $F(C_j)$: the set of fields of the collection C_j .
 - PV: the set of properties' values related to both identifiers and references, e.g., String is a value of the data type property.
- E : the set of edges that link and define the present relationships between nodes. An edge can be established between:
 - a node of an identifier $IDc(C_i)$ and a node of a candidate *reference* from $F(C_j)$ if this pair has a syntactic or a semantic similarity greater than a threshold.
 - a field in $\{IDc(C_i), F(C_j)\}$ and a property value.

Example 3. Figure 5 shows an example of the input graph where the blue nodes represent the fields belonging to both collections *CountryRegion* and *Orders_customer* and the green nodes represent their properties' values.

³do not allow multiple edges

⁴with nodes of different types

The candidate identifier of the collection `CountryRegion` is `CountryKey`. The colored edge linking the two nodes `CountryKey` and `NationKey` shows the existence of a syntactic or semantic relationship between the two nodes. We note that for the sake of simplicity, we use only a simple label for each node, whereas in the schema, each field is identified by its full path from the document root.

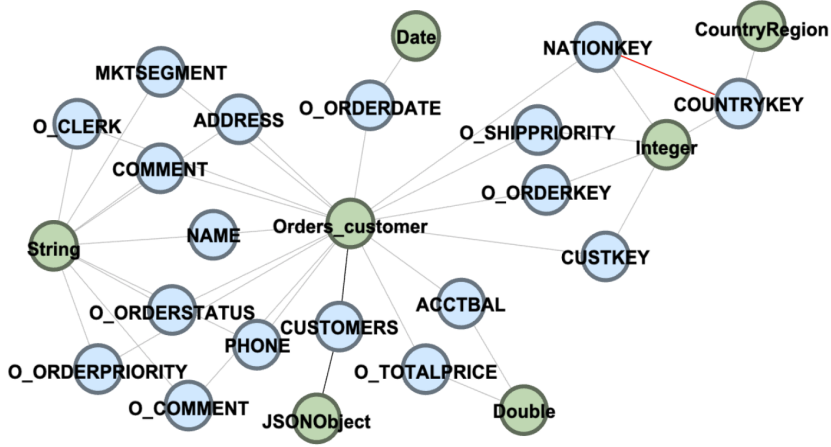


Figure 5: Example of the input graph for `node2vec`

We propose a set of rules to identify the properties related to identifiers and references:

- Rule 1: Compatibility of data type:** the identifier, and its reference must have the same type or compatible types. For example, if we have an *identifier* with a *String* type and a *reference* with an *Integer* type, and they are not convertible, this pair will not be considered in this case. Our approach covers all possible combinations of primitive types, e.g., $(String, String)$, $(String, Double)$, $(Integer, Double)$, $(Short, Double)$. As mentioned earlier in the global schema extraction stage, since a real primitive type can be hidden under another primitive type, we check the type of each field pair to detect such cases.
- Rule 2: Syntactic similarity based pruning:** in many instances, fields' names are not randomly assigned for the sake of better understanding. Hence, taking into account the similarity between the fields' names of each pair could be a kick-off beacon. To this end, we use a syntactic similarity measure, and we opt for the *Fuzzy-Token* similarity since it is the most suitable for our case [38]. Thus, the similarity combines both token-based similarity and string similarity. To use this similarity function, the input strings s_1 and s_2 are tokenized. We consider both cases for the tokenization: (i) having a delimiter, e.g., "_" and/or uppercase letter; (ii) strings are attached without a delimiter, e.g., "LINESTATUS".

The function is defined as $syntac(s_1, s_2) = \frac{|T_1 \tilde{\cap}_\sigma T_2|}{|T_1| + |T_2| - |T_1 \tilde{\cap}_\sigma T_2|}$, where s_1 is the *reference* name and s_2 is either the *identifier* name or the collection name of that identifier. Then, we retain the maximum value obtained between the two similarity measures. We note that s_1 and s_2 are amended comparing to [30]⁵. In addition, T_1 and T_2 are the tokens' sets related to s_1 and s_2 respectively and σ is the edit distance threshold used to penalize lower similarities. To compute this similarity, a weighted bigraph should be constructed using T_1 and T_2 . The weight, i.e., edit distance measure, is assigned to each edge. Then, we keep only the edges with a weight larger than σ . The fuzzy overlap, denoted by $|T_1 \tilde{\cap}_\sigma T_2|$, is used to define the maximum weight matching of the constructed graph. Note that if $|T_1|$ or $|T_2|$ are more significant than one, we filter them from the set of possible suffixes or prefixes such as "key" and "id." For instance, by considering the two fields "CountryKey" and "CustomerKey," the syntactic similarity measure may rise due to the common suffix "Key," which becomes misleading to get the

⁵In [30], the authors have concatenated the table name for both primary key and foreign key presented in an inclusion dependency. However, it remains unclear to concatenate table name for both of them because, generally the foreign key is likely to be similar to the name of the referenced table, but the inverse rarely happens.

right result and enhance the probability of getting false-positive results. On the other hand, the case of fields "Id" and "UserId" reveals the necessity to keep T_1 and T_2 without the filtering step.

- Rule 3: Semantic similarity based pruning:** using only the syntactic similarity between two fields is not sufficient to cover all cases, e.g., `customer` and `client`. It indeed leads to generate some false-positive and false-negative results. To this end, we propose a filtering step based on semantic similarity. To do so, we use the *Wup* semantic similarity measure (cf., Definition 8), which is based on the lexical database Wordnet⁶. Similar to the syntactic measure, we use tokenization to divide the attached words into meaningful separated words. Given two fields f_1 and f_2 , we split each of them into a set of tokens, T_1 and T_2 respectively. We consider f_1 and f_2 semantically similar if exists at least a semantic similarity between elements of a pair (t_1, t_2) , where $t_1 \in \{T_1 \setminus SP\}$ and $t_2 \in \{T_2 \setminus SP\}$. We denote with *SP* the set of possible suffixes or prefixes such as "key" and "id". Likewise the syntactic similarity, we deal differently with a unary set of tokens (T_1 or T_2) by considering all of the tokens without a filtering step.

Definition 8. (Wup similarity [39; 40]) *Wup* is a path-based semantic similarity. Given two concepts, it finds the path length to root from the Least Common Subsumer (LCS), the most specific concept they share as an ancestor. The *Wup* similarity is computed as follows: $sim_{Wup} = \frac{2 \times depth(LCS(C_1, C_2))}{depth(C_1) + depth(C_2)}$ where $depth(C)$ is the depth of the concept in the Wordnet hierarchy.

Based on the rules defined above, we define the properties that concern both identifiers and references, namely: collection name, data type, IsSyntacticallySimilar and IsSemanticallySimilar.

5. The IRIS-DS Algorithm

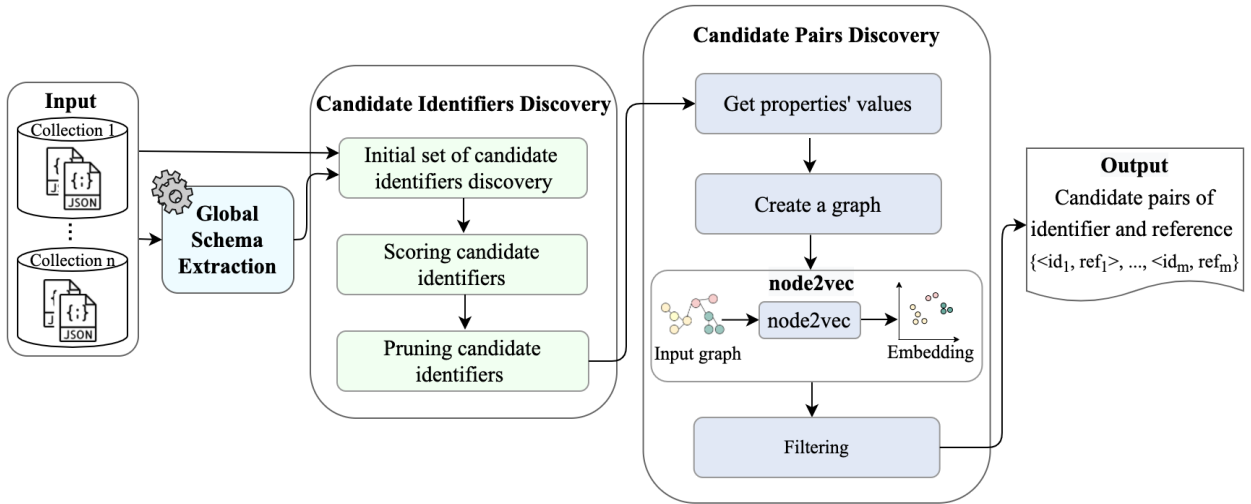


Figure 6: IRIS-DS general's architecture at a glance

Figure 6 shows the overall process of our proposed algorithm IRIS-DS. Starting from several collections, it firstly extracts the collections' global schemas. Secondly, it discovers the initial set of candidate *identifiers* that will be refined after the scoring and the pruning steps.

Then, it performs a graph embedding technique to track down the set of candidate pairs of *identifier* and *reference*.

The pseudo-code is sketched in Algorithm 1, which in turn invokes various methods that are detailed separately.

In line $A_1.L_2$, i.e., Algorithm 1, Line 2, we start with the extraction of collections' global schemas. In line $A_1.L_3$,

⁶<https://wordnet.princeton.edu/>

we search candidate *identifiers* from the set of fields presented in collections' global schemas. This step is explained separately in Algorithm 2.

In line $A_1.L_4$, the list of collections' pairs, denoted as L , is generated using the Cartesian product while keeping only pairs with different elements, i.e., each collection pair (CP) is defined as $CP = (C_i, C_j)$ where $C_i \neq C_j$. The cardinality of L is defined as $|L| = \frac{|C|(|C|-1)}{2}$, where $|C|$ is the number of distinct collections.

Algorithm 1: IRIS-DS

Input: Collections \mathbb{C}

Output: Pairs of candidate *identifier* and *reference* for each collections' pair $IRcand$

```

1  $LCP_1 \leftarrow \emptyset, LCP_2 \leftarrow \emptyset$ 
2  $SG_{\mathbb{C}} \leftarrow \text{GenerateCollectionsSchemas}(\mathbb{C})$ 
3  $\text{SearchCandidateIDs}(\mathbb{C}, SG_{\mathbb{C}})$ 
4  $L \leftarrow \text{GetListOfCollectionsPairs}()$ 
5 foreach  $CP=(C_i, C_j)$  in  $L$  do                                     ▷  $|L| = \frac{|C|(|C|-1)}{2}$ 
6   if  $\text{GetID}(C_i) \neq \emptyset$  then
7     |  $LCP_1 \leftarrow \text{discoverPairs}(C_i, C_j)$ 
8   end
9   if  $\text{GetID}(C_j) \neq \emptyset$  then
10    |  $LCP_2 \leftarrow \text{discoverPairs}(C_j, C_i)$ 
11  end
12   $IRcand \leftarrow \text{filter}(LCP_1, LCP_2)$ 
13   $\text{Store}(C_i, C_j, IRcand)$                                        ▷ Store: store the  $IRcand$  for each collection pair
14 end
15 return  $IRcand$ 

```

Algorithm 2: SearchCandidateIDs()

Input: Collections \mathbb{C} , Collections schemas $SG_{\mathbb{C}}$

Output: Initial list of candidate *identifiers* IL

```

1  $I \leftarrow \emptyset, R \leftarrow \emptyset, U \leftarrow \emptyset, IDs \leftarrow \emptyset, L \leftarrow \emptyset$ 
2 foreach  $C$  in  $\mathbb{C}$  do
3   |  $I \leftarrow \text{GetFieldsWithSimpleTypes}(SG_{\mathbb{C}})$ 
4   |  $R \leftarrow \text{GetRequiredFields}(I)$ 
5   |  $U \leftarrow \text{GetUnique}(R)$ 
6   |  $S \leftarrow \text{Score}(U)$ 
7   |  $IDs \leftarrow \text{Cliff}(S)$ 
8   |  $L \leftarrow L \cup IDs$ 
9 end
10 return  $L$ 

```

The idea is to iterate over L to find the set of pairs of candidate join keys (*identifier*, *reference*), if they exist, between every two collections (lines $A_1.L_{5-14}$). We consider both directions: a field in C_i can refer to another field in C_j (lines $A_1.L_{5-8}$) or vice versa (lines $A_1.L_{9-11}$). The pairs discovery assured in line $A_1.L_7$ and line $A_1.L_{10}$ are detailed in Algorithm 3 where the loop from line $A_3.L_1$ to line $A_3.L_{14}$ iterates over the list of *identifiers* of the current collection C_i , which are generated with $\text{GetID}(C_i)$.

In line $A_3.L_3$, we start by creating the input graph G for the embedding. Building the graph has a linear time complexity $O(n)$ as execution time depends on the size of the collection schema. The properties values are among the constituting nodes of the input graph. We detail the extraction of the properties values in Algorithm 4. In lines $A_3.L_{4-5}$, we apply the *node2vec* algorithm, and we generate the model after learning feature representations for the

nodes across the graph. The loop in lines $A_3.L_{6-10}$, iterates over the field(s) of an *identifier* and search the most similar field(s) that will be considered as a candidate *reference*.

Algorithm 3: discoverPairs()

Input: Collection C_i , collection C_j
Output: list of candidate pairs where the *identifiers* in C_i and reference in C_j LCP

```

1 foreach  $IDc$  in  $GetID(C_i)$  do
2    $PV \leftarrow GetPV(IDc) \cup GetPV(GetFields(SG(C_j)))$  ▷ PV: properties values
3    $G \leftarrow CreateGraphForEmbedding(IDc, GetFields(SG(C_j)), PV)$  ▷ G: input graph for node2vec
4    $n \leftarrow node2vec(G)$ 
5    $model \leftarrow LearnEmbedding(n)$  ▷ Learning node representations  $R \leftarrow \emptyset$ 
6   foreach  $part$  in  $IDc$  do
7     if  $model.GetSimilar(part) \neq \emptyset$  then
8        $R \leftarrow R \cup model.GetSimilar(part)$  ▷ list of the most similar fields to  $IDc$ 
9     end
10  end
11  if  $|IDc| = |R|$  then
12     $LCP \leftarrow LCP \cup (IDc, R)$  ▷ LCP: list of candidate pairs
13  end
14 end
15 return  $LCP$ 
    
```

In Algorithm 4, based on the rules mentioned above, we search the properties' values related to an *identifier* of the first collection and the set of fields of the second collection. In line $A_4.L_3$, we find the data type of the field f . In line $A_4.L_5$, we check if the current field and the *identifier* have a syntactic similarity measure equal to or greater than a given threshold. If they are syntactically similar, we assign to f the name of the identifier as a property value. In this way, an edge will be established between the *identifier*'s node and the field's node. Similarly, we verify the semantic similarity between the *identifier* and the current field.

Algorithm 4: GetPV()

Input: List of fields L , candidate *identifier* IDc , collections' names $CN(L)$ and $CN(IDc)$
Output: properties values PV

```

1  $PV \leftarrow \{CN(L), CN(IDc)\}$  ▷ add collections' names as properties values
2 foreach  $f$  in  $L$  do
3    $t \leftarrow dataType(f)$  ▷ get the data type of the field  $f$  according to Rule 1
4    $PV_f \leftarrow PV_f \cup t$ 
5   if  $CheckSyntacticSimilarity() = true$  then  $PV_f \leftarrow PV_f \cup IDc$  ▷ check if the field  $f$  is syntactically similar to  $IDc$ 
6   else  $PV_f \leftarrow PV_f \cup null$ 
7   if  $CheckSemanticSimilarity() = true$  then  $PV_f \leftarrow PV_f \cup IDc$ 
8   else  $PV_f \leftarrow PV_f \cup null$ 
9    $PV \leftarrow PV \cup PV_f$ 
10 end
11 return  $PV$ 
    
```

6. Case Study

Figure 7 shows two JSON collections, i.e., `Orders_customer` and `CountryRegion`, that are based on the TPC-H⁷ benchmark. This benchmark comprises relational sources that we have transformed into JSON collections⁸. For the sake of readability, we only present an excerpt of one document from each collection. Based on this benchmark, our basic scenario is to perform a join operation between `Orders_customer` and `CountryRegion`. In doing so, we should perform *identifiers* and *references* discovery between the two aforementioned collections.

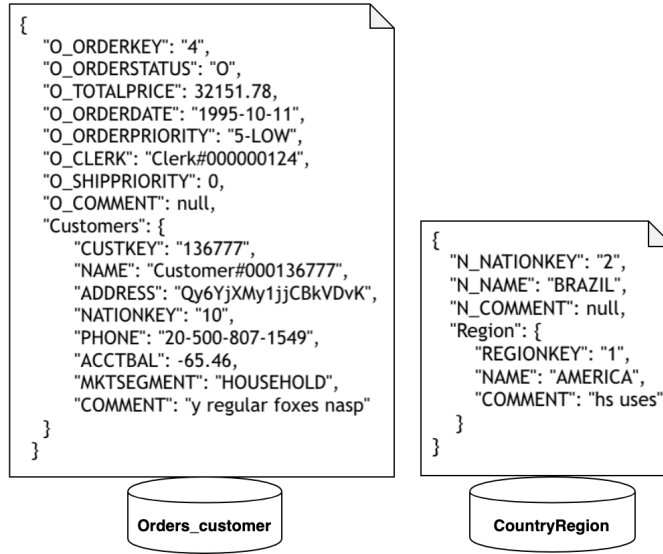


Figure 7: Sample JSON documents of the TPC-H benchmark

We start by identifying an initial list of candidate *identifiers* for each collection as described in subsection 4.2.1. Then, we present the obtained candidate *identifiers* for each collection in the second column of Table 2. We note that we present a field using its path from the document root, where $\$$ symbol represents the document root.

For each candidate *identifier*, we compute its score based on the features described in subsection 4.2.2. To prune irrelevant candidates for each collection, we search the cliff value after ranking the scores. We split the candidate *identifiers* into *Upper* and *Lower* parts. The *Upper* part of the collection `Nation` contains $[\$.COUNTRYKEY]$, and the *Upper* part of the collection `Orders_customer` contains $[\$.O_ORDERKEY, \$.customer.CUSTKEY]$, which is a composite candidate *identifier*. The remaining candidate identifiers related to each collection are pruned since they are in the *Lower* part.

We start by preparing the graph embedding input. As shown in Figure 8, we create three graphs:

- G_1 : presents the relationships between the candidate *identifier* $[\$.COUNTRYKEY]$ with the different fields of the collection `Orders_customer`.
- G_2 : presents the relationships between the first part of the composite candidate *identifier* $[\$.O_ORDERKEY, \$.customer.CUSTKEY]$ with the different fields of the collection `CountryRegion`.
- G_3 : presents the relationships between the second part of the composite candidate *identifier* $[\$.O_ORDERKEY, \$.customer.CUSTKEY]$ with the different fields of the collection .

G_2 and G_3 are related to the same candidate *identifier* $[\$.O_ORDERKEY, \$.customer.CUSTKEY]$, which is composed of two fields. The green nodes denote the candidate *identifier* or a part of the candidate *identifier* of the first collection. The blue nodes indicate the fields of the second collection that can contain the candidate *reference*. Finally, the nodes

⁷Decision support benchmark: <http://www.tpc.org/tpch/>

⁸ <https://github.com/souibguimanel/TPCHjson>

Table 2
Initial list of candidate *identifiers* with their scores

Collection	Candidate <i>identifiers</i>	Score
CountryRegion	[\$.COUNTRYKEY]	1.00
	[\$.N_NAME]	0.63
Orders_customer	[\$.O_ORDERKEY]	1.00
	[\$.Customers.CUSTKEY]	0.90
	[\$.O_CLERK, \$.customer.NATIONKEY]	0.75
	[\$.customer.MKTSEGMENT, \$.customer.NATIONKEY]	0.70
	[\$.O_ORDERSTATUS, \$.customer.NATIONKEY, \$.O_ORDERPRIORITY]	0.70
	[\$.O_ORDERSTATUS, \$.O_CLERK]	0.70
	[\$.O_TOTALPRICE]	0.60
	[\$.O_COMMENT]	0.60
	[\$.Customers.PHONE]	0.52
	[\$.customer.NAME]	0.52
	[\$.O_CLERK, \$.O_ORDERPRIORITY]	0.52
	[\$.customer.ACCTBAL]	0.50
	[\$.O_ORDERDATE]	0.50
	[\$.customer.COMMENT]	0.50
	[\$.customer.ADDRESS]	0.50
	[\$.customer.MKTSEGMENT, \$.O_CLERK]	0.49

of properties' values are presented with the pink color. The red edge in G_1 marks the existence of a semantic similarity between the two fields `[$.customer.NATIONKEY]` and `[$.COUNTRYKEY]`.

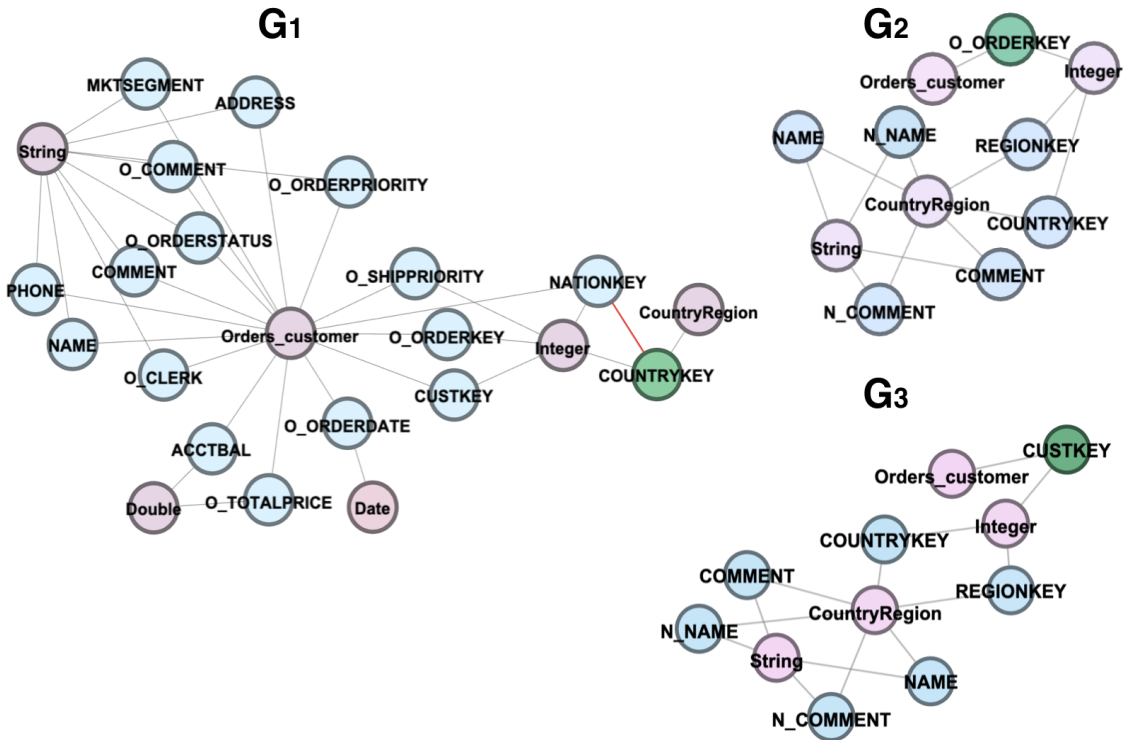


Figure 8: Node2vec input

To establish the relationships between nodes in each graph, we identify the values of the properties: collection

name, data type, syntactic similarity measure, and semantic similarity measure. Since the two last properties require pairs of fields, we create pairs using the Cartesian product between a candidate *identifier* of the first collection and the group of fields of the second collection. Then, we compute the syntactic and semantic similarity measures for each pair.

The pair with a similarity measure greater than a threshold will have an edge between their representing nodes in the graph.

Each graph obtained in the previous step is taken as an input of the `node2vec` algorithm⁹ to seek the key pairs based on the candidate *identifiers*. By performing the `node2vec` algorithm, we get the most similar fields for each identifier with the corresponding probabilities, which approximate the similarities between nodes. Regarding probabilities values, we keep the highest ones to identify the candidate pairs of key fields. We hosted the python code samples that are used in this step in a GitHub repository¹⁰. As a result, we obtain `[$.customer.NATIONKEY]` as the most similar field to the candidate *identifier* `[$.COUNTRYKEY]` with a probability measure in the region of 0.994.

7. Experimental Study

In this section, we report our experimental findings after describing the considered data collections. We base our experimental study on two benchmarks and two real-world datasets:

- **TPC-H:** a benchmark for decision support systems. It represents the activity of any industry that must manage, sell or distribute a product worldwide.
- **TPC-E:** a benchmark was offering a set of flat files that models a brokerage firm with customers who start transactions concerning trades, account inquiries, and market research.
- **Twitter:** these datasets, which comprise users and their related tweets, are scraped from Twitter’s API. We consider two collections: `Tweets` and `Users`. Each document of the `Tweets` collection contains nested objects such as the coordinates object that gives the geographic location. These datasets are initially in JSON format. They are heterogeneous in terms of schema variety, different nesting levels (i.e., field’s depth), missing values, different types, etc.
- **Musicians:** are datasets extracted from Wikidata¹¹, a real-world data source. These datasets are implemented in Valentine [41] where the authors proposed a well-thought-out dataset creation process that is tailored to the scope of matching techniques. The datasets represent data about American singers. It contains around 11k tabular data that we converted into JSON documents. To resemble a real-life scenario, authors in [41] have varied the names of the attributes in the source and the target sources included in Valentine. In addition, they provide with the source and the target dataset a JSON file containing the possible matching that we used as ground truth to check the accuracy of our results as shown in Figure 9.

```
{
  "matches": [
    {
      "source": "MusiciansSource",
      "source_attribute": "musician",
      "target": "MusiciansTarget",
      "target_attribute": "musicianID"
    }
  ]
}
```

Figure 9: Example of a ground truth file for the Musicians datasets

⁹<https://snap.stanford.edu/node2vec/>

¹⁰https://github.com/souibguimanel/Get_similarities_node2vec

¹¹https://www.wikidata.org/wiki/Wikidata:Main_Page

7.1. Data Collection

Since our approach deals with document stores, we carried out a data preparation phase to use the two benchmarks in our experimental process. We have implemented a transformation phase to convert TPC-H and TPC-E generated flat files to JSON ones. We consider each record in the flat file as a document in the JSON collection.

We perform the data preparation stage regarding the document-oriented model characteristics, e.g., randomly assigning null or missing values. Furthermore, in order to have different storage models, we have denormalised data in both benchmarks: (i) **TPC-H**, we have denormalised the `Orders` collection by embedding documents from `Customer`. Similarly, we have denormalised the `Nation` collection by embedding documents from `Region`; and (ii) **TPC-E**, we have denormalised the `Trade` collection by embedding documents from `TradeType`. Similarly, we have denormalised `CustomerAccounts` by embedding documents from both `Address` and `Customer` collections. This is done by replacing each foreign key with its full object. We hosted the generated data in a GitHub repositories^{12,13} to make them openly available.

7.2. Evaluation Protocol

The experiments we conducted aim to validate our approach, in terms of result relevance. The approach validation comprises both levels: (i) candidate *identifiers* discovery for each collection; and (ii) identification of candidate pairs of key fields (*identifier* and *reference*) for every two collections. To this end, for each level, we use four metrics

- **precision**: the fraction of the predicted true *identifier*/pairs among the predicted *identifiers*/pairs.
- **recall**: the fraction of the predicted true *identifier*/pairs among *identifiers*/pairs of the gold standard.
- **accuracy**: the number of correct results returned by our algorithm.
- **percentage decrease**: rate the reduction of the number of candidates that will be proposed to the end-user, this metric is computed as $\frac{(Original\ Number - New\ Number)}{Original\ Number} * 100$.

We distinguish two cases to compute the percentage decrease:

- Discovery of candidate *identifiers* for each collection:
 - * original number: the schema size of the given collection.
 - * new number: the number of the detected candidate *identifiers*.
- Identification of candidate pairs of key fields (identifier and reference) for every two collections:
 - * original number: given two collection schemas, the original number is the sum of the cardinality of the Cartesian product between the candidate *identifiers* $IDc(C_1)$ of the first collection and the set of fields of the second collection $F(C_2) = f_1, \dots, f_n$ and the cardinality of the Cartesian product between $IDc(C_2)$ and $F(C_1) = f_1, \dots, f_m$.
 - * new number: the number of the discovered pairs of identifier and reference.

7.3. Experimental Setup and Results

As proof of the concept of our approach, we have developed java prototypes to support the main phases and tested them under macOS High Sierra machine, Processor Intel Core i5, 2.7 GHz, and 8 GB of DDR3 RAM. We store the used collections of JSON documents on MongoDB as a document-oriented DBMS. We used the python Wordninja library¹⁴ to split the attached words into tokens. We also used the Python 3 implementation of the node2vec algorithm. As shown in Tables 3 and 5, we compare the output sets of both candidate *identifiers* and candidate pairs (*reference*, *identifier*) with the gold standard of the Twitter collections, Musicians collections, TPC-H benchmark, and TPC-E benchmark, and we report the precision, recall, accuracy, and the percentage decrease. The results show that our approach reaches a high precision and accuracy without diminishing the recall. Furthermore, the percentage decrease metric yields increasingly excellent results by reducing the number of candidates proposed to the end-user.

We note that in Table 3, our algorithm shows a decrease in the precision to 0.25 when detecting the identifier in the `Financial` collection. This decrease is because of the existence of several non-composite unique fields, which generate

¹² <https://github.com/souibguimanel/TPCHjson>

¹³ <https://github.com/souibguimanel/TPCEjson>

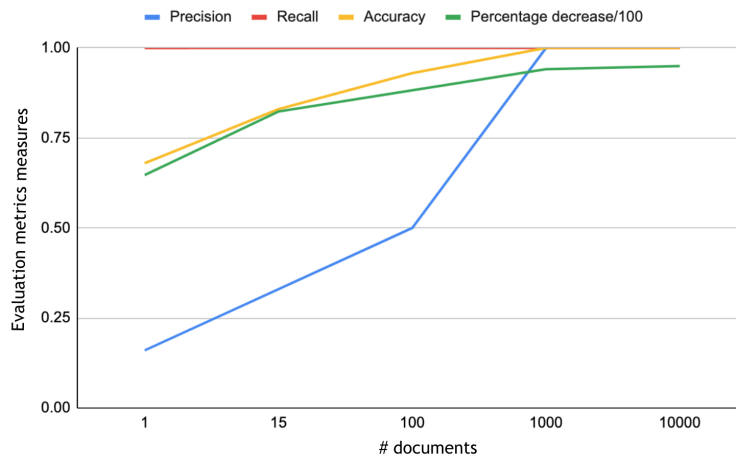
¹⁴ <https://pypi.org/project/wordninja/>

Table 3IRIS-DS results for the candidate *identifiers* discovery in TPC-H, TPC-E and Twitter collections

	Collection	# documents	Precision	Recall	Accuracy	Percentage decrease %
TPC-H	Orders	1k	1	1	1	94.11
	NationRegion	24	1	1	1	85.71
	Supplier	1k	1	1	1	94.73
TPC-E	Trade_TT	10k	1	1	1	94.73
	Financial	20k	0.25	1	0.73	71.42
	CustomersAccounts	10k	1	1	1	97.14
	Holding	10k	1	1	1	83.33
	AccountPermission	10k	1	1	1	80
Twitter	Tweets	1k	1	1	1	96.42
	TwitterUsers	10k	1	1	1	96.66
	Tweets	2M	1	1	1	97.61
	TwitterUsers	2M	1	1	1	97.29
Musicians	MusiciansSource	11k	1	1	1	92.30
	MusiciansTarget	11k	1	1	1	92.30

false-positive results. However, this error accounts for only a tiny portion of the used collections, and fortunately, the percentage decrease is still high.

Although our approach is dedicated to heterogeneous document stores (consisting of diverse data contents), we have also used as a part of our experimental study the two Musicians datasets, which encompass homogeneous data devoted to evaluating matching techniques.

**Figure 10:** Impact of the dataset size: Orders collection

We carried out further tests that corroborated the correlation between the collection size and the values of the evaluation metrics. To gauge this effect, we have varied the Orders collection size, as shown in Figure 10. The result demonstrates that the more we increase the collection size, the better precision, recall, accuracy, and percentage decrease we get. The number of fields with unique values increases if we reduce the number of documents into the same collection. Although the precision often decreases when using a collection with few documents, the percentage decrease remains significantly high, reflecting the reduction of the final number of candidates proposed to the end-user.

On the other hand, since we introduce new features to discover candidate *identifiers*, we performed additional tests to emphasize the importance of these features. For instance, the depth feature is intrinsic to document stores since collections' fields have different depths due to the embedding of objects inside documents. Thus, as shown in Table 4, by omitting this feature while computing the score for the CustomersAccounts collection, we realize the remarkable

Table 4

Importance of the depth feature: CustomersAccounts collection

	Precision	Recall	Accuracy	Percentage decrease
Without depth	0.05	1	0.51	48.57
With depth	1	1	1	97.14

Table 5

IRIS-DS results for candidate pairs discovery in TPC-H, TPC-E and Twitter collections

	Collection 1	Collection 2	Precision	Recall	Accuracy	Percentage decrease %
TPC-H	Orders	Nation	1	1	1	95.83
	Supplier	Orders	N/A	N/A	1	100
	CountryRegion	Supplier	1	1	1	92.85
TPC-E	Trade_TT	CustomersAccounts	1	1	1	98.14
	Holding	Trade_TT	1	1	1	96
	Financial	CustomersAccounts	N/A	N/A	1	100
	CustomersAccounts	Holding	1	1	1	97.56
	AccountPermission	CustomersAccounts	1	1	1	97.50
	Holding	AccountPermission	N/A	N/A	1	100
Twitter	Tweets	Users	1	1	1	98.50
Musicians	MusiciansSource	MusiciansTarget	1	1	1	96.15

Table 6

Comparison of IRIS-DS with HoPF algorithm [30] applied on the TPC-H and the TPC-E benchmarks

	Algorithm	Identifier		(Reference, Identifier)	
		Precision	Recall	Precision	Recall
TPC-H	HoPF	1	1	0.88	0.88
	IRIS-DS	1	1	1	1
TPC-E	HoPF	0.80	0.80	0.72	0.91
	IRIS-DS	0.85	1	1	1

decrease in the precision, accuracy, and percentage decrease. Similarly, we remark a significant difference between their absence and presence for the data type and field name prefix features while computing scores and discovering candidate *identifiers*.

Since our approach considers several collections, we apply key pair discovery on every two collections. Indeed, there is at least one pair of collections that are not joinable so that they did not have a relationship (*reference, identifier*). Our approach can handle such cases. In fact, as depicted in Table 5, the pairs' discovery performed between the collections' pairs (i) Supplier and Order; (ii) AccountPermission and Holding; and (iii) Financial and CustomersAccounts returns no join key pairs. This implies that the precision and recall are N/A, i.e., Not Applicable, because the number of true-positive values is null. For example, this might occur where the gold standard does not contain join key fields, and our algorithm returns no pairs correctly. We note that the parameter settings (*dimensions* and *walk-length*) of the *node2vec* algorithm that we used in our tests gives a greater result when they are set to respectively 10 and 30.

Since we are the first to propose an approach for *identifier* and *reference* discovery in document stores, we compare our algorithm against the most recent work [30] proposed in the context of relational databases as shown in Table 6. Our and HoPF approaches have the same objective: joining multiple sources while not having the join keys beforehand. However, our approach concerns document stores, while the HoPF approach [30] applies to relational databases. Relational databases have functional dependencies: (i) the primary key values are unique and not null; (ii) the foreign key values are included in the primary key values. However, document stores miss all these features. Roughly speaking, unlike the HoPF approach, we can not apply functional dependencies in the context of document stores. Hence, we have undergone a rethinking of the problem by using alternative methods adapted to document stores' schemaless nature, namely, adapting existing features and providing new features and pruning rules. Outcomes from our experiments give insight into the feasibility of detecting join key fields in document stores containing scattered data over

several collections.

8. Conclusion and Future Work

Document stores have a variable schema, where fields can be missing in some documents or have null values. Because of integrity constraints and inclusion dependencies, a document store is the furthest from having an exact join key. For this, detecting join key pairs between two document stores is a tricky task. In the literature, existing works have provided dedicated solutions to relational databases. For NoSQL data stores, current contributions rely on a strong assumption: having the join keys pairs beforehand. We seldom know the pair of identifier and reference in document stores.

To this end, we have proposed, in this paper, an alternative approach for *identifiers* and *references*' discovery based on several document stores. We have introduced the IRIS-DS algorithm that discovers candidate *identifiers* for each collection, and then identifies the candidate pairs of *identifier* and *reference* for every two collections. We use scoring features and pruning rules to discover actual candidate *identifiers* from many initial ones efficiently. To find candidate pairs between several document stores, we put into practice *node2vec* as a graph embedding technique, which yields significant advantages while using syntactic and semantic similarity measures for pruning pointless candidates. The carried out experiments on the TPC-H and the TPC-E benchmarks, and the Twitter dataset underscore that our approach fulfills the accuracy of the generated results. We claim that our findings might be useful for

- Business Intelligence & Analytics systems, particularly ETL processes dealing with document stores: today, exploiting NoSQL data for analytical purposes lacks maturity, traceability, and metadata management. In the context of BI&A, data are often scattered over distinct sources and several collections of documents. Hence, fetching relevant data that meets the decision-maker requirements often needs to access more than one document store, thence needs to use the join operation.

Moreover, open-source and commercial ETL tools offer join components. However, it is up to the user to choose the join keys before applying the join operation. In a NoSQL context, a document may have hundreds of fields, making it tricky to fetch the identifier of a given collection manually.

- Querying tasks: the join is mandatory for querying tasks. Hence, it is compulsory to have the join keys beforehand. For instance, MongoDB uses the MQL (MongoDB Query Language) to query stored data in document databases using different operators. Furthermore, since developers know well that join operations among collections are essential [7], MongoDB, the open-source community, has recently introduced the *\$lookup* operator in version 3.2. This operator performs a left join between two collections using a *localField* and a *foreignField* that the user specifies.
- Database design: identifying the interrelationships in legacy databases certainly requires discovering join keys. Even though the lack of a rigid schema characterizes NoSQL frameworks, developers need to overview the data and take appropriate steps and decisions during the application design process. These decisions can have a significant effect on application efficiency and readability of the code [42].

As part of our future work, we intend to study the impact of the data incremental refresh [43] and how to schedule the process of identifiers and references discovery in document stores accordingly. In the long run, we plan to complete our proposal by formalizing all ETL operations since our ultimate aim is to provide a NoSQL-dedicated BI&A approach.

References

- [1] J. Mali, F. Atigui, A. Azough, N. Travers, ModelDrivenGuide: An Approach for Implementing NoSQL Schemas, in: Database and Expert Systems Applications - 31st International Conference, DEXA, Bratislava, Slovakia, Proceedings., 2020, pp. 141–151.
- [2] F. Abdelhédi, A. A. Brahim, F. Atigui, G. Zurfluh, Mda-based approach for nosql databases modelling, in: Big Data Analytics and Knowledge Discovery - 19th International Conference, DaWaK 2017, Lyon, France, Proceedings, 2017, pp. 88–102.
- [3] F. Abdelhédi, R. Jemmali, G. Zurfluh, Ingestion of a data lake into a nosql data warehouse: The case of relational databases, in: Proceedings of the 13th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management, IC3K 2021, Volume 3: KMIS, Online Streaming, October 25-27, 2021, 2021, pp. 64–72.
- [4] J. Ram, C. Zhang, A. Koronios, The implications of big data analytics on business intelligence: A qualitative study in china, Procedia Computer Science 87 (2016) 221–226.
- [5] R. Hecht, S. Jablonski, Nosql evaluation: A use case oriented survey, in: 2011 International Conference on Cloud and Service Computing, CSC 2011, Hong Kong, December 12-14, 2011, 2011, pp. 336–341.

- [6] M. Souibgui, F. Atigui, S. B. Yahia, S. S. Cherfi, Business intelligence and analytics: On-demand ETL over document stores, in: *Research Challenges in Information Science - 14th International Conference, RCIS 2020, Limassol, Cyprus, September 23-25, 2020, Proceedings, 2020*, pp. 556–561.
- [7] A. Celesti, M. Fazio, M. Villari, A study on join operations in mongodb preserving collections data models for future internet applications, *Future Internet* 11 (4) (2019) 83.
- [8] M. Souibgui, F. Atigui, S. Zammali, S. S. Cherfi, S. Ben Yahia, Data quality in ETL process: A preliminary study, in: *Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 23rd International Conference KES-2019, Budapest, Hungary, 2019*, pp. 676–687.
- [9] M. Souibgui, F. Atigui, S. B. Yahia, S. S.-S. Cherfi, Iris-ds: A new approach for identifiers and references discovery in document stores, in: *Proceedings of the 54th Hawaii International Conference on System Sciences, 2021*, pp. 970–979.
- [10] M. L. Chouder, S. Rizzi, R. Chalal, Exodus: Exploratory OLAP over document stores, *Inf. Syst.* 79 (2019) 44–57.
- [11] J. Pokorný, JSON functionally, in: *Advances in Databases and Information Systems - 24th European Conference, ADBIS 2020, Lyon, France, August 25-27, 2020, Proceedings, 2020*, pp. 139–153.
- [12] P. D. Asanka, ETL framework design for NoSQL Databases in Dataware housing, *IJRCAR* 3 (2015) 67–75.
- [13] R. Yangui, A. Nabli, F. Gargouri, ETL based framework for NoSQL warehousing, in: *Information Systems - 14th European, Mediterranean, and Middle Eastern Conference, EMCIS 2017, Coimbra, Portugal, Proceedings, 2017*, pp. 40–53.
- [14] H. Mallek, F. Ghozzi, F. Gargouri, Towards extract-transform-load operations in a big data context, *Int. J. Sociotechnology Knowl. Dev.* 12 (2020) 77–95.
- [15] P. Gómez, R. Casallas, C. Roncancio, Automatic schema generation for document-oriented systems, in: *Database and Expert Systems Applications - 31st International Conference, DEXA 2020, Bratislava, Slovakia, September 14-17, 2020, Proceedings, Part I, 2020*, pp. 152–163.
- [16] M. Klettke, U. Störl, S. Scherzinger, Schema extraction and structural outlier detection for json-based NoSQL data stores, in: *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), Hamburg, Germany. Proceedings, 2015*, pp. 425–444.
- [17] L. Wang, O. Hassanzadeh, S. Zhang, J. Shi, L. Jiao, J. Zou, C. Wang, Schema management for document stores, *PVLDB* 8 (9) (2015) 922–933.
- [18] J. L. C. Izquierdo, J. Cabot, Jsondiscoverer: Visualizing the schema lurking behind JSON documents, *Knowl.-Based Syst.* 103 (2016) 52–55.
- [19] M. A. Baazizi, D. Colazzo, G. Ghelli, C. Sartiani, Parametric schema inference for massive JSON datasets, *VLDB J.* 28 (4) (2019) 497–521.
- [20] E. Gallinucci, M. Golfarelli, S. Rizzi, Schema profiling of document-oriented databases, *Inf. Syst.* 75 (2018) 13–25.
- [21] M. N. Mami, D. Graux, S. Scerri, H. Jabeen, S. Auer, J. Lehmann, Squerall: Virtual ontology-based access to heterogeneous and large data sources, in: *The Semantic Web - ISWC 2019 - 18th International Semantic Web Conference, Auckland, New Zealand, Proceedings, Part II, 2019*, pp. 229–245.
- [22] H. Kondylakis, A. Fountouris, A. Planas, G. Troullinou, D. Plexousakis, Enabling joins over cassandra NoSQL databases, in: *Big Data Innovations and Applications - 5th International Conference, Innovate-Data 2019, Istanbul, Turkey, Proceedings, 2019*, pp. 3–17.
- [23] Y. He, K. Ganjam, X. Chu, SEMA-JOIN: joining semantically-related tables using big table corpora, *PVLDB* 8 (12) (2015) 1358–1369.
- [24] A. Bogatu, A. A. A. Fernandes, N. W. Paton, N. Konstantinou, Dataset discovery in data lakes, in: *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020, 2020*, pp. 709–720.
- [25] E. Zhu, D. Deng, F. Nargesian, R. J. Miller, JOSIE: overlap set similarity search for finding joinable tables in data lakes, in: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019, 2019*, pp. 847–864.
- [26] R. C. Fernandez, E. Mansour, A. A. Qahtan, A. K. Elmagarmid, I. F. Ilyas, S. Madden, M. Ouzzani, M. Stonebraker, N. Tang, Seeping semantics: Linking datasets using word embeddings for data discovery, in: *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018, 2018*, pp. 989–1000.
- [27] M. Memari, S. Link, G. Dobbie, SQL data profiling of foreign keys, in: *Conceptual Modeling - 34th International Conference, ER 2015, Stockholm, Sweden, Proceedings, 2015*, pp. 229–243.
- [28] X. Wu, N. Wang, H. Liu, Discovering foreign keys on web tables with the crowd, *Comput. Informatics* 38 (3) (2019) 621–646.
- [29] M. Zhang, M. Hadjieleftheriou, B. C. Ooi, C. M. Procopiuc, D. Srivastava, On multi-column foreign key discovery, *Proc. VLDB Endow.* 3 (1) (2010) 805–814.
- [30] L. Jiang, F. Naumann, Holistic primary key and foreign key detection, *J. Intell. Inf. Syst.* 54 (3) (2020) 439–461.
- [31] T. D. Nguyen, M. Shih, S. S. Parvathaneni, B. Xu, D. Srivastava, S. Tirthapura, Random sampling for group-by queries, in: *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020, 2020*, pp. 541–552.
- [32] D. Pejcoch, Critical evaluation of validation rules automated extraction from data, *Journal of Systems Integration* 5 (2014) 32–46.
- [33] L. Berti-Équille, H. Harmouch, F. Naumann, N. Novelli, S. Thirumuruganathan, Discovery of genuine functional dependencies from relational data with missing values, in: *Actes du XXXVIIème Congrès INFORSID, Paris, France, June 11-14, 2019, 2019*, pp. 287–288.
- [34] T. Papenbrock, F. Naumann, Data-driven schema normalization, in: *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, 2017, 2017*, pp. 342–353.
- [35] P. Cui, X. Wang, J. Pei, W. Zhu, A survey on network embedding, *IEEE Trans. Knowl. Data Eng.* 31 (5) (2019) 833–852.
- [36] H. Cai, V. W. Zheng, K. C. Chang, A comprehensive survey of graph embedding: Problems, techniques, and applications, *IEEE Trans. Knowl. Data Eng.* 30 (9) (2018) 1616–1637.
- [37] A. Grover, J. Leskovec, node2vec: Scalable feature learning for networks, in: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016, 2016*, pp. 855–864.
- [38] J. Wang, G. Li, J. Feng, Fast-join: An efficient method for fuzzy token matching based string similarity join, in: *Proceedings of the 27th International Conference on Data Engineering, ICDE, Hannover, Germany, 2011*, pp. 458–469.
- [39] Z. Wu, M. Palmer, Verbs semantics and lexical selection, in: *Proceedings of the 32nd Annual Meeting on Association for Computational Linguistics, ACL '94, Association for Computational Linguistics, USA, 1994*, p. 133–138.

- [40] T. Pedersen, S. Patwardhan, J. Michelizzi, Wordnet: : Similarity - measuring the relatedness of concepts, in: Demonstration Papers at HLT-NAACL 2004, Boston, Massachusetts, USA, May 2-7, 2004, 2004.
- [41] C. Koutras, G. Siachamis, A. Ionescu, K. Psarakis, J. Brons, M. Fragkoulis, C. Lofi, A. Bonifati, A. Katsifodimos, Valentine: Evaluating matching techniques for dataset discovery, in: 37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021, 2021, pp. 468–479.
- [42] M. J. Mior, K. Salem, Renormalization of nosql database schemas, in: Conceptual Modeling - 37th International Conference, ER 2018, Xi'an, China, October 22-25, 2018, Proceedings, 2018, pp. 479–487.
- [43] W. Qu, S. Deßloch, Incremental ETL pipeline scheduling for near real-time data warehouses, in: Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings, 2017, pp. 299–308.