IRIS-DS: A New Approach for Identifiers and References Discovery in Document Stores

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

IRIS-DS: A New Approach for Identifiers and References Discovery in Document Stores

Manel Souibgui[1,2], Faten Atigui[1], Sadok Ben Yahia[2,3], Samira Si-Said Cherfi[1]
[1]Conservatoire National des Arts et Métiers, CEDRIC-CNAM, France
[2]University of Tunis El Manar, Faculty of Sciences of Tunis, LIPAH, Tunisia
[3] Department of Software Science, Tallinn University of Technology, Estonia
manel.souibgui@fst.utm.tn, faten.atigui@cnam.fr, sadok.ben@taltech.ee, samira.cherfi@cnam.fr

Abstract

NoSQL stores offer a new cost-effective and schema-free system. Although it is widely accepted today, Business Intelligence & Analytics (BI&A) remains associated with relational databases. Exploiting schema-free data for analytical purposes is issuing 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 challenge. Detecting these fields manually is time and effort consuming, and even infeasible in large-scale datasets. In this paper, we study the problem of discovering join fields automatically, and introduce an algorithm to detect both identifiers and references on several document stores. The modus operandi of our approach underscores two core stages: (i) discovery of identifier candidates; and (ii) identifying candidate pairs of identifier and reference fields. We use scoring features and pruning rules based on both syntactic and semantic aspects to efficiently discover true candidates from a huge number of initial ones. Finally, we report our experimental findings that show very promising results.

1. Introduction

Since more than a decade ago, NoSQL datastore became commonly used to store Big Data. These systems are schema-free, and built upon distributed systems, which makes it easy to scale and shard[1]. However, in the 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 performant and easy to use [1, 2]. Although the use of NoSQL stores is widely accepted today, Business Intelligence & Analytics (BI&A) remains associated with relational databases. 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. In our previous work [3], we have proposed a hybrid BI&A approach that considers both schemaless data sources and analytical needs to efficiently explore more than one document store. The ETL, i.e., extract, transform and load, is the cornerstone of our approach. Carrying out major ETL operations, particularly, the join operation still being a challenge in document stores [4, 5]. In fact, fetching relevant data that meets the decision-maker requirements, often needs to access more than one document store, thereby needs to use 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 due to the absence of integrity constraints. So, identifying the "joinable" fields to stick two document stores is a tricky challenge. Despite its importance, no previous work has paid heed to detect join keys pairs in the context of NoSQL stores, particularly document-oriented stores.

For this, we introduce IRIS-DS (**I**dentifiers and **R**eferences D**IS**covery in **D**ocument **S**tores), a new approach that aims to discover the pairs of join keys (*identifier*, *reference*) starting from more than two document stores, i.e., collections. Worthy of mention is that we focus, in this paper, on non-composite join keys. Thus, the sighting features of our approach are: *(i)* to the best of our knowledge, no former approach has been dedicated to joining keys discovery in the context of document stores; *(ii)* we adapt existing features, which identify candidates, to the context of document stores and we introduce new ones; and *(iii)* unlike existing works, we use both syntactic and semantic similarity measure for pruning pointless candidates.

The paper outline is as follows: in Section 2, we recall the basic concepts related to document stores

[1]Sharding is a method for distributing data across multiple machines

URI: https://hdl.handle.net/10125/70730
978-0-9981331-4-0

followed by a motivating example in Section 3. In Section 4, we scrutinize the related literature. In Section 5, we introduce the core stages of our approach. In Section 6, we explain the overall algorithm. In Section 7, we present a case study of our approach. Finally, we discuss the experimental results in Section 8 and we allude to takeaway messages and sketch issues of future work in Section 9.

2. Preliminaries

Document stores, *aka document-oriented databases*, are one of the families of NoSQL stores. A document, which has a schemaless nature, is the basic concept of document stores. JSON is currently the most commonly adopted format that we will use in the remainder.

Definition 1. *(***Document and Collection***)* *A document d is an object. Each object contains a set of key-value pairs; a key is a string, while a value can be either a primitive value (i.e, a number, a string, or a boolean), an object, an array of values, or null. A collection D is an array of documents.*

Before introducing the problem, we briefly present, in Table 1, the terminology related to document stores and their associated concepts in relational databases.

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
identifier	primary key
reference	foreign key

3. Motivating Example

In the following, we present a motivating example that smoothly sheds light on the main challenges of detecting the join keys in the context of document stores. We consider n collections denoted $C_1, \ldots, C_n$, 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`. To do so, we need to compute the delay as the difference between the actual delivery date versus the expected one. C_1

Figure 1: An excerpt of two collections

contains all the orders made on *Amazon* marketplace and C_2 contains the deliveries done by the Bosch brand to different marketplaces. As $DD = deliveryDate - expDeliveryDate$ where `deliveryDate` $\in C_1$ and `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`}. In document stores, joining two collections is a thriving challenge due to their clueless schemaless nature. In fact, *(i)* unlike primary key which is unique and not null, *identifier* as all the other fields, can be missing in some documents or can, normally and not exceptionally, have null values; *(ii)* document stores doesn't 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, which means that it is impossible to use the inclusion dependencies in order to automatically detect *identifiers* and *references*.

4. Related Work

Our main 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 main streams of approaches: *(i)* dealt with the ETL process over NoSQL stores; *(ii)* addressed the join operation in the context of NoSQL stores; and *(iii)* proposed contributions for the primary key and foreign key detection in the context of relational databases.

4.1. ETL over NoSQL Stores

Few researchers have addressed the problem of ETL in the context of NoSQL stores, particularly the document-oriented ones [6, 7]. For example, in [8], 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 these works, several approaches have focused on schema extraction, i.e., a list of document fields with their types, from document stores [9]. Since it is a crucial step in an ETL process, dealing with document stores, we have studied these different contributions. Baazizi et al. [10] 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. [11] 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 most of the above-mentioned approaches have exclusively focused on the first phase of the ETL process, i.e., the extraction phase or on 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.

4.2. Join Operation in the Context of NoSQL Stores

A number of questions regarding the join operation in the context of NoSQL stores need to be addressed. In fact, the join operation is not explicitly available in NoSQL stores [4]. Few researchers have addressed this issue. For instance, in [4], the authors discussed the impact of performing the join operation in the context of 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 querying dedicated approaches. In [12], 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. During query time, the framework enables the user to declare transformations for altering join keys to make data joinable. In [13], 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 [14] inputs two sets of values from join columns and produces a predicted join relationship using a big table corpus.

The aforementioned works have all addressed the join operation in the context of NoSQL stores. However, we note that all of them rely on a strong assumption: having the join keys beforehand. It is worth mentioning that, in the context of NoSQL stores, no prior works have proposed a method to find the pair 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.

4.3. Primary Key and Foreign Key Discovery in Relational Databases

In this subsection, we present the contributions that have dealt with the detection of primary keys and foreign keys in the context of relational databases. The authors in [15, 16, 17] have paid attention to foreign keys detection assuming the presence of primary keys. Quite freshly, Jiang and Naumann [18] 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 combination and inclusion dependencies. Both types of dependencies have been used to, respectively, detect primary keys and

foreign keys in relational databases. A unique column combination is the set of attributes whose projection contains only the set of column combinations having the unique and non-null values, whereas, in document stores, fields can easily, be missing in some documents or can, normally and not exceptionally, have null values. On the other hand, their work is based on the set of inclusion dependency given as input. However, this assumption couldn't pertain in the context of document stores as described in our motivating example.

Even if this previous work [18] is the closest one to our problem, it can not be applied 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.

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

In this section, we thorroughly describe the core stages of our approach: *(i)* discovery of *identifier* candidates; and *(ii)* identifying candidate pairs of key fields.

5.1. Discovery of Identifier Candidates

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

5.1.1. Identifying the Initial List of Identifier Candidates. 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 and T_s the ones with simple types. Since an *identifier* can not, probably, be a *JSONObject* nor a *JSONArray*, then we limited the search space of *identifier* candidates 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. Thus, we classify fields in T_s as being required (F_r) or optional (F_o) (cf., Definition 2). We limited the search space of identifier candidates to the required fields within F_r. Then, within F_r, we look for those having unique values.

Definition 2. *(***Required field***) A field is required whenever its frequency is greater than or equal to a threshold ε. The frequency is computed as $freq(field) = \frac{|\tilde{k}_c|}{|D_c|}$ [19], 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.*

5.1.2. Scoring Identifier Candidates. In the context of relational sources, several primary key features had been explored in the literature [18, 20] to distinguish true primary keys from spurious ones, i.e., position, name suffix, and value length. We reuse these features that we have adapted to the context of document stores in our proposal, and we introduce new schema-based features: **depth** and **data type**. We present these features as follows:

- **Name suffix:** *identifiers* are generally identified by their field name suffix. We consider the list of possible names' suffixes for identifiers as: "id", "key", "nr", "no", "pk", "num", and "code". The score function $suffix(f)$ is binary. It returns one if the field, denoted as f, has one of the suffixes mentioned above or zero otherwise.
- **Position:** very often, an *identifier* is ranked first in a collection. The score function is defined as $\frac{1}{|before(f)|+1}$, where $before(f)$ denotes the number of fields before a field f. Worth of mention in a document store, a field can be found in different positions in the set of documents. Hence, we consider the most frequent position of f.
- **Depth:** *identifiers* often have a shallow depth. In fact, a nested field has a lower chance to be an *identifier* for the entire collection. We define the score function as $\frac{1}{depth(f)+1}$.
- **Data type:** hands-on hints show that a field is prone to be an *identifier* whenever its data type is Integer or String. The score function $type(f)$ is binary and returns one if the field 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 $\frac{1}{max(1,|LengthMax(f)|-n)}$, where $|LengthMax(f)|$ is the length of the longest value associated to the field of f and the parameter n is used to penalize long values.

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

5.1.3. Pruning Identifier Candidates. Expectedly, the set of the initial *identifier* candidates is very large. Filtering techniques are essential to get rid of poor *identifier* candidates. For this, for each collection, we score each *identifier* candidate using the above-described features. In this paper, we use the cliff technique [18] (cf., Definition 3). As described in Example 1, the set of *identifier* candidates 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 score, we prune the candidates belonging to the *Lower* part. We note that in case of multiple instances of cliff we retain all candidates.

Definition 3. *(***Cliff [18]***) Given $S = \{S_1, S_2, \ldots, S_n\}$, the sorted score list of identifier candidates belonging to one collection, and their corresponding score difference list, $SD = \{SD_1, SD_2, \ldots SD_{n-1}\}$, 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.*

Example 1. *As depicted in Figure 2, we suppose having a list S of identifier candidates' 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 red squares, shown in Figure 2, respectively illustrate, the Upper and the Lower parts.*

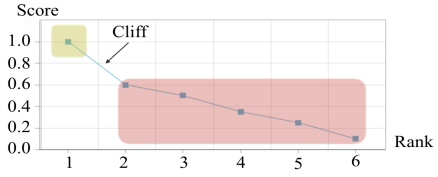

Figure 2: Example of the cliff method applied to the ranked scores of *identifier* candidates

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

5.2. 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. The stage is based on two steps detailed in the following parts.

5.2.1. Creating the Initial Set of Pairs. This step consists in identifying the initial set of pairs (*identifier*, *reference*) between every two collections. Given two collection schemas, we first perform the Cartesian product between the *identifier* candidates $IDc(C_1)$ of the first collection and the set of fields of the second collection $F(C_2) = f_1, \ldots, f_n$. Secondly, we perform the Cartesian product between $IDc(C_2)$ and $F(C_1) = f_1, \ldots, f_m$. It is worth mentioning that fields with complex types are not considered while performing the Cartesian product.

5.2.2. Filtering. For the sake of refining the initial list generated from the previous step, we propose a filtering step, which is based on three rules:

Rule 1: Compatibility of data type: we remove from the initial list, the pair of fields that do not have the same type or do not have 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 be omitted in this case. Our approach covers all possible combinations of primitive types, e.g., (*String*, *String*), (*String*, *Double*), (*Integer*, *Double*), (*Short*, *Double*). 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 [21]. 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 [18][2]. In addition, T_1 and T_2 are the

[2]In [18], 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.

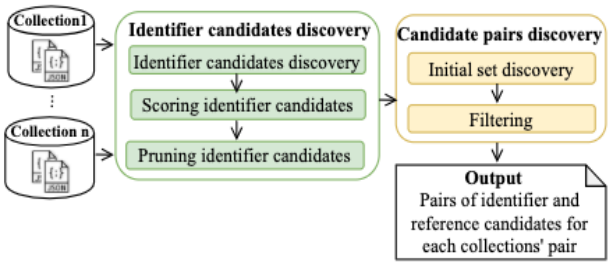

Figure 3: IRIS-DS general's architecture at a glance

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.

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 leads certainly 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, which is based on the lexical database Wordnet[3]. Similarly to the syntactic measure, we use tokenization to divide the attached words into meaningful separated words.

6. The IRIS-DS Algorithm

Figure 3 shows the overall process of our proposed algorithm IRIS-DS. Starting from several collections, it firstly discovers the initial set of identifier candidates that will be refined after the scoring and the pruning steps. Secondly, it refines the initial set of candidate pairs discovered by performing the filtering step. The pseudo-code is sketched in Algorithm 1, which in turn invokes various methods that are detailed separately.

In line A1.L2, i.e., Algorithm 1, Line 2, we start with the extraction of collections' global schemas. In line A1.L3, we search *identifier* candidates from the set of fields presented in collections' global schemas. This step is detailed separately in Algorithm 2. In line A1.L4, 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 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.

[3]https://wordnet.princeton.edu/

Algorithm 1: IRIS-DS

Input: Collections C
Output: Pairs of identifier and reference candidates for each collections' pair $IRcand$

```
1  FLCP_1 = ∅, FLCP_2 = ∅;
2  SG_C = GenerateCollectionsSchemas(C);
3  SearchCandidateIDs(C, SG_C);
4  L = GetListOfCollectionsPairs();
5  foreach CP=(C_i, C_j) in L do
       ▷ |L| = |C|(|C|−1)/2
6      if GetID(C_i) ≠ ∅ then
7          foreach IDc in GetID(C_i) do
8              L_1 = {IDc} × GetFields(SG(C_j))
                   ▷ GetFields(SG(C_j)): the set
                   of fields with simple types
9              FLCP_1 = FLCP_1 ∪ Filter(L_1) ;
                   ▷ FLCP: Filtered list of
                   candidate pairs
10         end
11     end
12     if GetID(C_j) ≠ ∅ then
13         foreach IDc in GetID(C_j) do
14             L_2 = {IDc} × GetFields(SG(C_i));
15             FLCP_2 = FLCP_2 ∪ Filter(L_2);
16         end
17     end
18     Store(C_i, C_j, FLCP_1, IRcand) ;
           ▷ Store: store the IRcand for each
           collection pair
19     Store(C_j, C_i, FLCP_2, IRcand);
20 end
21 return IRcand
```

The idea is to iterate over L to find the set of pairs of candidate join keys (*identifier*, *reference*) between every two collections (lines A1.L5-20). We consider both directions: a field in C_i can refer to another field in C_j (lines A1.L6-11) or vice versa (lines A1.L12-17). The loop from line A1.L7 to line A1.L10 iterates over the list of *identifiers* of the current collection C_i generated with *GetID*(C_i). To constitute the pairs of candidate keys, we use the Cartesian product between the *identifier* candidate of the first collection and the filtered fields from the second one. Once the list of candidates pairs of key fields is generated, we propose a filtering step (line A1.L9), which is detailed separately in Algorithm 3.

Algorithm 2 shows the procedure of identifying the initial list of *identifiers* candidates.

Algorithm 2: SearchCandidateIDs()

Input: Collections C, Collections schemas SG_C
Output: Initial list of identifier candidates IL

```
1  I = ∅, R = ∅, L = ∅, IL = ∅;
2  foreach C in C do
3      I= GetFieldsWithSimpleTypes(SG_C);
4      R= GetRequiredFields(I);
5      L= GetUnique(R);
6      IL = IL ∪L
7  end
8  return IL
```

Algorithm 3: Filter()

Input: List of candidate pairs of key fields L, thresholds ϵ and γ
Output: Filtered List FL

```
1  foreach p in L do
2      if CheckTypeCompatibility()== true and
         SyntacticSimilarityMeasure(p) ≥ ϵ then
         Add_p(T1);
3      else Add_p(T2);
4  end
5  if T2 ≠ ∅ then
6      foreach p in T2 do
7          if SemanticSimilarityMeasure(p) ≤ γ
             then
8              discard(p);
9          end
10     end
11 end
12 FL = T1 ∪ T2;
13 return FL
```

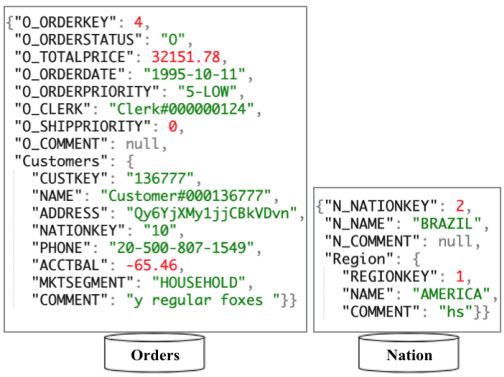

Figure 4: Sample JSON documents of the TPC-H benchmark

The basic steps are described in subsection 5.1.1. Algorithm 3 describes the filtering step, which is based on both syntactic and semantic similarity measures. Given a list of candidate pairs of key fields, we iterate over it to check the data type compatibility and to compute the syntactic similarity measure using the method *SyntacticSimilarityMeasure()*. All the pairs are split into two groups. The first one holds pairs that have a syntactic similarity measure equal to or greater than a given threshold, i.e., both fields are syntactically similar. The remaining pairs are held into the second group T_2 (lines A3.L2-5) to be semantically checked. In our experiments, we varied the thresholds and we found that $\epsilon = 0.5$ and $\gamma = 0.7$ are the most suitable values for penalization.

7. Case Study

Figure 4 shows two JSON collections, i.e., `Orders` and `Nation`, that are based on the TPC-H[4] benchmark. This benchmark consists of relational sources that we have transformed into JSON collections[5] For the sake of legibility, we have presented only an excerpt of one document from each collection. Based on this benchmark, our basic scenario is to perform a join operation between `Orders` and `Nation`. In doing so, we should perform *identifiers* and *references* discovery between the two aforementioned collections. We start by identifying an initial list of identifier candidates for each collection as described in subsection 5.1.1. 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

Table 2: Initial list of candidate identifiers with their scores

Collection	Candidate identifiers	Score
Nation	$/N_NATIONKEY	1.00
	$/N_COMMENT	0.40
	$/N_NAME	0.63
Orders	$/Customers/PHONE	0.34
	$/O_ORDERDATE	0.35
	$/Customers/ADDRESS	0.32
	$/Customers/NAME	0.33
	$/O_CLERK	0.46
	$/Customers/CUSTKEY	0.71
	$/Customers/COMMENT	0.31
	$/Customers/ACCTBAL	0.31
	$/O_ORDERKEY	1.00
	$/O_TOTALPRICE	0.46

[4]Decision support benchmark: http://www.tpc.org/tpch/
[5] https://github.com/souibguimanel/TPCHjson

document root. For each identifier candidate, we compute its score based on the features described in subsection 5.1.2. To prune poor quality 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 `$/N_NATIONKEY`, and the *Upper* part of the collection `Orders` contains `$/O_ORDERKEY`. The remaining candidate identifiers related to each collection are pruned since they are in the *Lower* part. We use

Table 3: Fields' pairs with compatible data types

Field 1		Field 2	
Path	**Type**	**Path**	**Type**
`$/O_ORDERKEY`	Integer	`$/N_NATIONKEY`	Integer
`$/O_TOTALPRICE`	Double	`$/N_NATIONKEY`	Integer
`$/O_SHIPPRIORITY`	Integer	`$/N_NATIONKEY`	Integer
`$/customer/CUSTKEY`	String	`$/N_NATIONKEY`	Integer
`$/customer/ACCTBAL`	Double	`$/N_NATIONKEY`	Integer
`$/customer/NATIONKEY`	String	`$/N_NATIONKEY`	Integer
`$/Region/REGIONKEY`	String	`$/O_ORDERKEY`	Integer
`$/N_COMMENT`	Null	`$/O_ORDERKEY`	Integer

the candidate identifiers belonging to the *Upper* part of each collection to identify candidate pairs of key fields, i.e., identifiers and references. For this, we start by creating the initial set of pairs using the Cartesian product. Due to space limitation, we host the Cartesian product result in a GitHub repository[6]. Since the initial list of candidate pairs is large, we refine it by applying the set of filtering rules. As presented in Table 3, we first check for pairs, i.e. each pair consists of a reference candidate (Field 1) and an identifier candidate (Field 2), containing fields with compatible types. Then, we compute the syntactic similarity measure for each pair as shown in Table 4. The penalization threshold is fixed to 0.5. Thus, we held the pair having a syntactic similarity equal to or greater than the threshold value, i.e., {`$/customer/NATIONKEY`, `$/N_NATIONKEY`}. The remaining fields, which have a syntactic similarity measure less than the threshold, are filtered based on the semantic similarity measure. In this example, no semantic similarity was detected among the list of pairs. Hence, the discovered pair of key fields is {`$/customer/NATIONKEY`, `$/N_NATIONKEY`}, where `$/N_NATIONKEY` is the identifier related to the `Nation` collection and `$/customer/NATIONKEY` is the reference that is related to the `Orders` collection.

[6]https://github.com/souibguimanel/TPCHjson/blob/master/CaseStudy.txt

Table 4: Syntactic similarity measures (SSM) of the candidate pairs of key fields

Candidate pair	SSM
{`$/O_ORDERKEY, $/N_NATIONKEY`}	0.20
{`$/O_TOTALPRICE, $/N_NATIONKEY`}	0.00
{`$/O_SHIPPRIORITY, $/N_NATIONKEY`}	0.00
{`$/customer/CUSTKEY, $/N_NATIONKEY`}	0.20
{`$/customer/ACCTBAL, $/N_NATIONKEY`}	0.00
{`$/customer/NATIONKEY, $/N_NATIONKEY`}	0.66
{`$/Region/REGIONKEY, $/O_ORDERKEY`}	0.25
{`$/N_COMMENT, $/O_ORDERKEY`}	0.00

Table 5: IRIS-DS results for the *identifier* candidates discovery in the TPC-H and the TPC-E collections

	Collection	# d[1]	P[2]	R[3]	A[4]	Pd[5] %
TPC-H	`Orders`	1000	1	1	1	94.11
	`Nation`	24	1	1	1	87.50
	`Supplier`	1000	1	1	1	85.71
TPC-E	`Trade_TT`	1000	1	1	1	94.73
	`CustAcc`	1000	1	1	1	97.14
	`Holding`	1000	1	1	1	83.33

[1]# documents, [2]Precision, [3] Recall, [4]Accuracy,[5]Percentage decrease

8. Experimental Study

In this section, we report our experimental findings after describing the considered data collections.

8.1. Data Collection

Our experimental study is based on both the TPC-H and the TPC-E benchmarks. Since our approach deals with document stores, we have implemented a transformation phase to convert TPC-H and TPC-E generated flat files to JSON ones. Each record in the flat file is considered as a document in the JSON collection.

We perform a data preparation stage with respect to the document-oriented model characteristics,

Table 6: IRIS-DS results for candidate pairs discovery in the TPC-H and the TPC-E collections

	C1[1]	C2[2]	P[3]	R[4]	A[5]	Pd[6] %
TPC-H	`Orders`	`Nation`	1	1	1	95.40
	`Supplier`	`Order`	N/A	N/A	1	100
	`Nation`	`Supplier`	1	1	1	91.60
TPC-E	`Trade_TT`	`CustAcc`	1	1	1	98.80
	`Holding`	`Trade_TT`	0.50	1	0.96	93.10
	`CustAcc`	`Holding`	1	1	1	97.77

[1]Collection 1, [2]Collection 2, [3] Precision, [4]Recall,[5]Accuracy,[6]Percentage decrease

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 `CustomerAcc` 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[7,8] to make them openly available.

8.2. Evaluation Protocol

The experiments we conducted aim to validate our approach, in terms of result relevance. The approach validation consists of both levels: *(i) identifier* candidate's 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 *(i)* **precision**: the fraction of the predicted true *identifier*/pairs among the predicted *identifiers*/pairs; *(ii)* **recall**: the fraction of predicted true *identifier*/pairs among *identifiers*/pairs of the gold standard; *(iii)* **accuracy**: the number of correct results returned by our algorithm; and *(iv)* **percentage decrease**: rate the reduce of the number of candidates that will be proposed to the end-user, this metric is computed as $[((OriginalNumber - NewNumber)/OriginalNumber) * 100]$.

Table 7: Comparison of IRIS-DS with HoPF algorithm [18] applied on the TPC-H and the TPC-E benchmarks

		Id[1]		**(Ref**[2]**, Id**[1]**)**	
	Algorithm	**P**[3]	**R**[4]	**P**[3]	**R**[4]
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	1	1	0.83	1

[1]Identifier, [2]Reference, [3]Precision,[4]Recall

8.3. 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. The used collections of JSON documents are stored on MongoDB as a document-oriented DBMS. We also used the python Wordninja library[9] to split the attached words into tokens.

As shown in Tables 5 and 6, we compare the output sets of both *identifier* candidates and candidate pairs (*reference*, *identifier*) with the gold standard of the TPC-H and the TPC-E benchmarks, 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 good results by reducing the number of candidates that will be proposed to the end-user.

Since our approach considers several collections, we apply key pairs discovery on every two collections. Certainly, there is at least one pair of collections that they 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 6, the pairs' discovery performed between `Supplier` and `Order` collections returns no pairs. This implies that the precision and recall are N/A, i.e. Not Applicable because the number of true-positive values is null. This might occur in cases in which the gold standard does not contain key join fields, and our algorithm returns correctly no pairs. We note that in Table 6, our algorithm shows a sharp penultimate row drop-in precision to 0.5 on detecting the pairs of join keys between `Holding` and `Trade_TT` collections. This drop is because the syntactic similarity rule generates one false-positive result. However, this error accounts for only a small portion of the used collections, and fortunately, the percentage decrease is still high. 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 [18] proposed in the context of relational databases as shown in Table 7. Outcomes from our experiments give insight into the feasibility of detecting join key fields in document stores containing scattered out data over several collections.

9. Conclusion and Future Work

Document stores have a variable schema, where fields can be missing in some documents or can have null values. Due to the absence 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.

[7] https://github.com/souibguimanel/TPCHjson
[8] https://github.com/souibguimanel/TPCEjson
[9]https://pypi.org/project/wordninja/

For NoSQL data stores, existing contributions rely on a strong assumption: having the join keys pairs beforehand. The pair of identifiers and references are seldom known in document stores.

To this end, we have proposed, in this paper, a new approach for *identifiers* and *references'* discovery based on several document stores. We have introduced the IRIS-DS algorithm that discovers *identifier* candidates for each collection, and then identifies the candidate pairs of *identifier* and *reference* for every two collections. We use scoring features and pruning rules based on both syntactic and semantic levels to efficiently discover true candidates from a huge number of initial ones. The carried out experiments, on the TPC-H and the TPC-E benchmarks, underscore that our approach fulfills the accuracy of the generated results.

As part of our future work, we started already performing larger experimentations over real-life datasets to better study the efficiency and the boundaries of our approach. Secondly, we intend to integrate the discovery of composite *identifiers*, i.e., an identifier that is composed of two or more fields and their equivalent join reference fields. In the long run, as our ultimate objective is to provide a NoSQL-dedicated BI&A approach, we plan to complete our proposal through a formalization of all ETL operations.

References

[1] J. Mali, F. Atigui, A. Azough, and N. Travers, "ModelDrivenGuide: An Approach for Implementing NoSQL Schemas," in *Database and Expert Systems Applications - 31st International Conference, DEXA, Bratislava, Slovakia, Proceedings,*, pp. 141–151, 2020.

[2] F. Abdelhédi, A. A. Brahim, F. Atigui, and G. Zurfluh, "Mda-based approach for nosql databases modelling," in *Big Data Analytics and Knowledge Discovery - 19th International Conference, DaWaK 2017, Lyon, France, Proceedings*, pp. 88–102, 2017.

[3] M. Souibgui, F. Atigui, S. B. Yahia, and 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*, pp. 556–561, 2020.

[4] A. Celesti, M. Fazio, and M. Villari, "A study on join operations in mongodb preserving collections data models for future internet applications," *Future Internet*, vol. 11, no. 4, p. 83, 2019.

[5] M. Souibgui, F. Atigui, S. Zammali, S. S. Cherfi, and 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*, pp. 676–687, 2019.

[6] P. D. Asanka, "ETL framework design for NoSQL Databases in Dataware housing," *IJRCAR*, vol. 3, pp. 67–75, 2015.

[7] R. Yangui, A. Nabli, and F. Gargouri, "ETL based framework for NoSQL warehousing," in *Information Systems - 14th European, Mediterranean, and Middle Eastern Conference, EMCIS 2017, Coimbra, Portugal, Proceedings*, pp. 40–53, 2017.

[8] H. Mallek, F. Ghozzi, O. Teste, and F. Gargouri, "BigDimETL with NoSQL database," in *Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 22nd International Conference KES, Belgrade, Serbia*, pp. 798–807, 2018.

[9] M. Klettke, U. Störl, and 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*, pp. 425–444, 2015.

[10] M. A. Baazizi, D. Colazzo, G. Ghelli, and C. Sartiani, "Parametric schema inference for massive JSON datasets," *VLDB J.*, vol. 28, no. 4, pp. 497–521, 2019.

[11] E. Gallinucci, M. Golfarelli, and S. Rizzi, "Schema profiling of document-oriented databases," *Inf. Syst.*, vol. 75, pp. 13–25, 2018.

[12] M. N. Mami, D. Graux, S. Scerri, H. Jabeen, S. Auer, and 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*, pp. 229–245, 2019.

[13] H. Kondylakis, A. Fountouris, A. Planas, G. Troullinou, and D. Plexousakis, "Enabling joins over cassandra NoSQL databases," in *Big Data Innovations and Applications - 5th International Conference, Innovate-Data 2019, Istanbul, Turkey, Proceedings*, pp. 3–17, 2019.

[14] Y. He, K. Ganjam, and X. Chu, "SEMA-JOIN: joining semantically-related tables using big table corpora," *PVLDB*, vol. 8, no. 12, pp. 1358–1369, 2015.

[15] M. Memari, S. Link, and G. Dobbie, "SQL data profiling of foreign keys," in *Conceptual Modeling - 34th International Conference, ER 2015, Stockholm, Sweden, Proceedings*, pp. 229–243, 2015.

[16] X. Wu, N. Wang, and H. Liu, "Discovering foreign keys on web tables with the crowd," *Comput. Informatics*, vol. 38, no. 3, pp. 621–646, 2019.

[17] M. Zhang, M. Hadjieleftheriou, B. C. Ooi, C. M. Procopiuc, and D. Srivastava, "On multi-column foreign key discovery," *Proc. VLDB Endow.*, vol. 3, no. 1, pp. 805–814, 2010.

[18] L. Jiang and F. Naumann, "Holistic primary key and foreign key detection," *Journal of Intelligent Information Systems*, pp. 1–23, 2019.

[19] M. L. Chouder, S. Rizzi, and R. Chalal, "Exodus: Exploratory OLAP over document stores," *Inf. Syst.*, vol. 79, pp. 44–57, 2019.

[20] T. Papenbrock and F. Naumann, "Data-driven schema normalization," in *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, 2017*, pp. 342–353, 2017.

[21] J. Wang, G. Li, and 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*, pp. 458–469, 2011.