Toward Generic Abstractions for Data of Any Model

Nelly Barret, Ioana Manolescu, Prajna Upadhyay

Toward Generic Abstractions for Data of Any Model

Nelly Barret, Ioana Manolescu, Prajna Upadhyay
Inria & Institut Polytechnique de Paris
nelly.barret@inria.fr,ioana.manolescu@inria.fr,prajna-devi.upadhyay@inria.fr

ABSTRACT

Digital data sharing leads to unprecedented opportunities to develop data-driven systems for supporting economic activities, the social and political life, and science. Many open-access datasets are RDF graphs, but others are CSV files, Neo4J property graphs, JSON or XML documents, etc.

Potential users need to *understand* a dataset in order to decide if it is useful for their goal. While some datasets come with a schema and/or documentation, this is not always the case. Data summarization or schema inference tools have been proposed, specializing in XML, or JSON, or the RDF data models. In this work, we present a *dataset abstraction approach*, which (*i*) applies on relational, CSV, XML, JSON, RDF or Property Graph data; (*ii*) computes an *abstraction meant for humans* (as opposed to a *schema meant for a parser*); (*iii*) integrates Information Extraction data profiling, to also *classify* dataset content among a set of categories of interest to the user. Our abstractions are conceptually close to an Entity-Relationship diagram, if one allows nested and possibly heterogeneous structure within entities.

1 INTRODUCTION

Open-access data is multiplying over the Internet. This leads on one hand, to the development of new businesses, economic opportunities and applications, and on the other hand, to circulating knowledge on a variety of topics, from health to education, environment, the arts, or leisure activities.

Many of the openly available datasets follow the **RDF** standard, the W3C's recommendation for sharing data. The Linked Open Data Cloud portal lists thousands of such datasets containing, e.g., national or worldwide statistics, music, scientific bibliographies, and many other interesting, open RDF graphs are not listed there. However, Open Data sets are not (or not only) RDF, as demonstrated by the following examples of very popular open datasets. (*i*) **CSV** files (each of which can be seen as a table) are shared on machine learning portals such as Kaggle or the French public portal data.gouv.fr; The French national transparency database HATVP (Haute Autorité pour la Transparence de la Vie Publique) also publishes CSV files; (*ii*) **relational** databases, comprising several interrelated tables, are used e.g. to disseminate the DBLP bibliographic data; (*iii*) **XML** is the format used in hundreds of million of bibliographic notices, e.g., on PubMed; DBLP and HATVP data has also been shared as XML; (*iv*) **JSON** has become more recently the format of choice, used e.g. to describe the complete activity of the French parliament on the websites NosDeputes.fr and NosSenateurs.fr; (*v*) **property graphs**, such as pioneered by Neo4J, are oriented graphs whose nodes and edges may have labels and properties; this is the format used by the International Consortium of Investigative Journalists to share their investigative data.

Decades of data management research have shown that no new data model completely replaces the previous ones. Some models thought obsolete re-surface under new incarnations (think of nested relations vs. document stores, or object databases vs. property graphs). The model in which a dataset is produced or exported is decided by the producers, depending on what they understand/are familiar with, the system at their disposal for storing the data, and the needs of foreseeable data users. The Open Data exchange scenarios, where data users often lack any institutional connection to the producers, also forces users to *cope with the data as it is*, since the producers have no incentive (and, often, lack the resources) to restructure the data in a different format. Thus, we believe *the data model variety in Open Data is here to stay.*

Users who must decide whether to use a dataset in an application need to have a basic **understanding of its content and the suitability to their need**. Towards this goal, *schemas* may be available to describe the data structure, and/or *documentation (text)* may describe its content in natural language. As help to the users, schemas and documentations have some limitations: (*a*) *schemas are often unavailable*, especially for semistructured data formats such as JSON, RDF or XML and *documentation is also often unavailable or too terse* to inform the users; (*b*) even when they are published, or built by automated tools, e.g., [3–5, 7–9, 15], schemas are *not helpful (or too complex) for casual users*, who ignore what an "XML element", "JSON array" or "RDF property node" is; (*c*) schemas as well as documentation describe the data *according to the data producer's terminology*, not according to the consumer's. For instance, in the XML dataset at the left in Figure 1, a public library labels its data entries `item`, with the implicit knowledge that these are documents (books, magazines, etc.), whereas from the users' perspective, this dataset describes *books*; (*d*) by design, *schemas do not quantitatively reflect the data*, whereas such information could be very useful as a first insight on the data.

Towards a data model-independent dataset abstraction To facilitate the understanding of a dataset by a user, we compute a *compact description of the data, focusing on its most frequent content, free of data model-specific syntactic details, and formulated in terms that interest the user.* To that end, we develop a *single, integrated method, applicable to any of the data models mentioned above.* For example, given any of the four bibliographic datasets in Figure 1 and a user interested in "books", our approach would correctly identify the dataset as pertinent for the user's question. As we will explain, users can specify their categories of interest through a few *hints*; with the help of popular knowledge bases, our system makes suggestions to enlarge the set of hints and improve the chances of users to have an accurate description of their dataset. We proceed in several steps, which also organize the remainder of the paper.

(1.) We view each dataset as holding *records*, that is: objects with some internal structure, simple or complex, representing a concept or an object. For instance, the XML data in Figure 1 contains two `item` records. Further, we identify *collections* grouping similar records (some records may belong to no collection). For instance, the

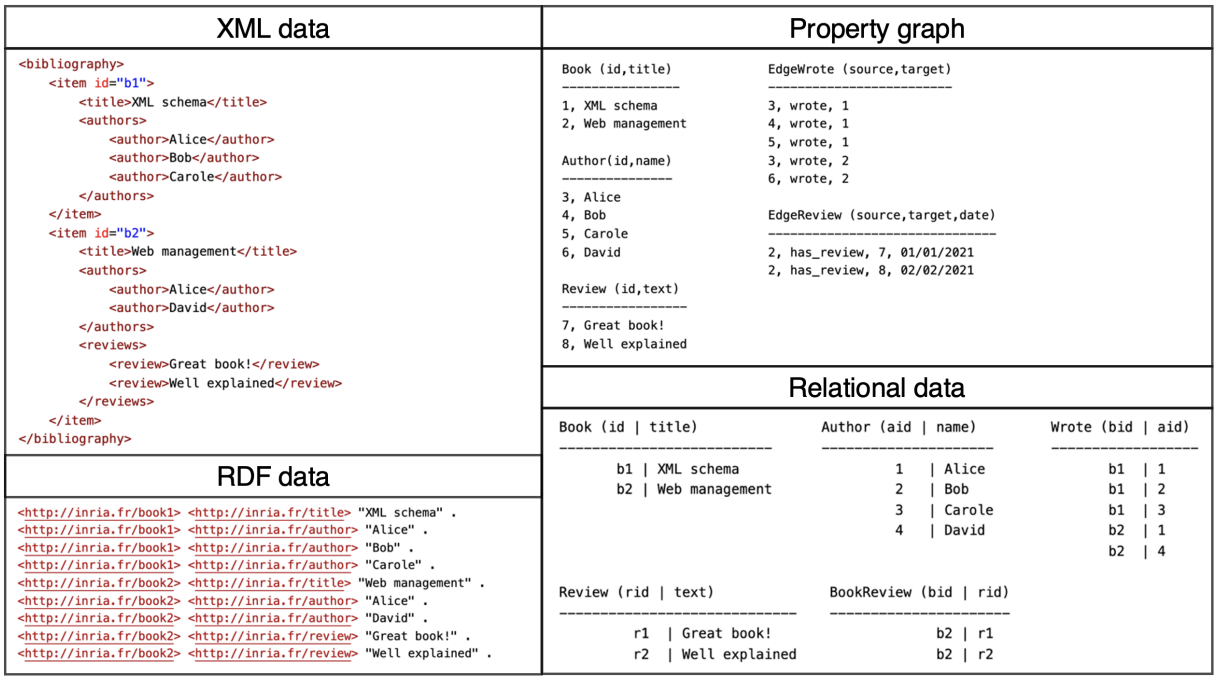

Figure 1: Motivating example: four bibliographic datasets, which we view as *Collections of CreativeWork records*.

`bibliography` XML element in Figure 1 can be seen as a collection which holds the two records. We identify a set of requirements for our record and collection detection method, and formalize the problem of deriving them automatically from a dataset (Section 2).

(2.) We propose an algorithm for *automatically identifying records and collections* in a dataset of any of the supported data models (Section 3). For instance, given the XML dataset in Figure 1 (*or* the relational *or* the RDF dataset from the same figure), our algorithm understands it as a *collection of records*, each record corresponding to a book, and having: a title, a collection of authors, and possibly a collection of reviews.

(3.) We separate *collections of Entities* from *collections of Relationships*, in order to report to the user a structured, meaningful dataset abstraction (Section 4).

(4.) To help users understand the data *in their terms*, we attempt to assign each collection to a *category* from a predefined category set, derived from a general-purpose ontology and/or based on the concepts of interest to users, e.g., books in the above example. We provide a method for our tool to enrich its knowledge of user-specified categories by selectively gathering information from large online knowledge bases. For instance, our algorithm classifies the `bibliography` in Figure 1 as a *Creative Work* (a Schema.org standard type including books, paintings, songs, etc.)

Deployment scenarios On one hand, data producers can use our tool to automatically derive data abstractions, to be shared next to the data; on the other hand, users can generate the abstractions after downloading a candidate dataset, in order to assess its interest.

2 PROBLEM STATEMENT

We start by analyzing a set of requirements of our problem, then formally state it.

2.1 Requirements

R1: data model-independent abstractions Our method needs to go beyond the syntactic details to extract semantically meaningful *records*, and understand if they are organized in *collections*.

R2: structurally rich abstractions The above problem can be seen as reverse-engineering a dataset to identify its conceptual model, which generalizes the classic Entity-Relationship model behind relational databases [13] by allowing records to have *multi-valued attributes*, and to *contain other records and/or collections*. For instance, the second `item` record in Figure 1 contains a collection of `review` records. A collection uniformly represents a *list*, *set* or *bag* of records: data order and possible duplicates are not reflected in our abstractions.

R3: see beyond the data structures In some cases, data syntax features are insufficient to distinguish records from collections. On one hand, in some data models, such as XML or RDF, the syntax does not distinguish them, e.g., in XMark [14] benchmark documents, an ⟨open_auctions⟩ element is clearly a collection, while a ⟨user⟩ element describes a record. On the other hand, even when the data model distinguishes nodes that should naturally be records (e.g., JSON maps), from others that should be collections (e.g., JSON arrays), we should not make this decision purely based on syntax. Indeed, as also noted in [15], a short array could in fact designate an object (e.g., three coordinates describe a geographical point), while a map may be used to encode a list, e.g., with attributes named "1", "2", "3" etc., as in Le Monde's Decodex dataset. It is important that such variations in data design do not confuse our abstraction.

R4: implicit or explicit collections In our motivating example, each of the four datasets holds a collection of books. In XML, the collection is *explicit* (materialized by the ⟨bibliography⟩ node), and the table node labeled `Book` plays the same role in the relational dataset. In contrast, in the RDF dataset, there is no common parent

of the books; we say that the collection here is *implicit*. Note that whether collections are explicit or implicit does not depend on the original data model: in the relational dataset, the book collection is explicit but the review collection is not; similarly, the RDF dataset *could have* included a common parent to all the book nodes, which would have made that collection explicit.

Availability and role of schemas and types What schema information can we expect to have, and how should we treat it?

Relational databases always have a schema, describing elementary data types, the attributes of each table, and possible integrity constraints. In a *CSV file*, the number of attributes can be identified easily; their names may or may not be present, and data types are not explicitly declared; data profiling [1] is needed to infer their domains. *XML documents* may or may not have a schema, expressed as a Document Type Description (DTD) or XML Schema Description (XSD); these specify the allowed children for each type of element. *JSON documents* usually come without a schema, but recent methods [3, 4, 15] derive schemas from the documents. *RDF graphs* are often schemaless, but they *may* be endowed with: (*i*) an ontology, expressed in RDF Schema or OWL, describing relationships between the types and properties present in the graph, e.g., *Any Student is a Person*, or *Anyone taking a Course is a Student*; (*ii*) a SHACL (Shapes Constraint Language) schema specification, against which a graph may be validated or not. Schemas for *property graphs* are being investigated actively [10], although no standard has emerged yet.

Data types are basic components of schemas. When present, types encapsulate valuable insights into the data organization and semantics. Thus, we formulate the following requirement:

R5: explicit types When available, types should *guide* our identification of records and collections, even though our approach should not depend on them.

2.2 Abstraction approach

To satisfy requirement **R1**, we leverage the ConnectionLens system [2, 6] which models the information from any relational database, CSV, XML, JSON, RDF document, or property graph, as a **graph** $G = (N, E)$ where $E \subseteq N \times N$ is a set of directed edges, and λ_N, λ_E are two functions labeling each node (respectively, edge) with a label (a string), that could in particular be ϵ (the empty label). Figure 2 illustrates this; for now, just focus on the nodes and edges (their colors and the shaded areas will be explained later).

Relational data A relational table or a CSV dataset can be seen as a set of tuples, each with the same attributes. This can be turned into a graph by modeling the dataset as a *table node*, having one child *tuple node* for each tuple (or line in the CSV file); this child has one child *attribute node* for each attribute. Figure 2 illustrates this for the sample relational dataset in Figure 1. Following **R5**, we leverage foreign key constraints expressed in a relational database schema as follows. Whenever relation S includes a foreign key to relation R, an edge is created leading from each tuple in S, to the respective R tuple node.

XML and JSON data are naturally converted into trees.

RDF data Each triple (s, p, o) from an RDF graph is converted into an edge between the (single) node labeled s to the (single) node labeled o; the edge is labeled p. We denote by τ the special *RDF type property* used to explicitly connect an URI to its type (which is also a node in the graph).

Property graphs (PG) In this rich, directed graph data model: (*i*) each node may have a set of attributes with a name and a value; (*ii*) each edge can similarly have attributes; (*iii*) zero or more labels may be attached to each node and/or edge, playing roughly the role of a type. In our graphs, each node/edge has at most one label. Therefore, we transform a property graph as follows. Each PG node becomes a node $n \in N$, labeled ϵ. Each label l of a PG node n becomes an edge $n \xrightarrow{\tau} n_l \in E$, where n_l is a leaf node labeled l and τ is the RDF type property mentioned above. Each attribute of n, named a and whose value is b, becomes an edge $n \xrightarrow{a} n_b \in E$ where n_b is a leaf node labeled b. Each PG edge e is turned into a node n_e, labeled ϵ, plus two edges, connecting it to its source and target nodes in the original PG; n_e also has outgoing edges modeling the attributes of e, similarly to PG nodes.

Extracted entities The graph built by ConnectionLens out of any dataset is enriched through *entity extraction*, applied on each value (string, leaf) node present in the dataset [2]. In our example, Alice, Bob, Carole and David are recognized as Person entities. The set T_E of entity types also includes: Location, Organization, Date, URIs, emails, hashtags, etc. Entities will be used to categorize collections (Section 5).

Problem statement Given the graph $G = (N, E)$ obtained as above, our goal is to:

(1) *Identify records and collections*, that is: find (*i*) a set $\mathcal{R} \subseteq N$ of nodes which we call *records*; (*ii*) a set of sets $\mathcal{C} = \{C_1, C_2, \ldots\}$, where each C_i is a set of elements from $\mathcal{R}$, the C_i's are pairwise disjoint, and for each C_i, there *may* exist a node n_{C_i} such that $(n_{C_i}, r_i^j) \in E$ for every $r_i^j \in C_i$, in other words: n_{C_i} is a parent (in G) of all the nodes from C_i; (*iii*) for each $r \in \mathcal{R}$, a set of nodes and edges of G which we view as *part of* the record r.
(2) *Separate the collections* in $\mathcal{C}$ into $\mathcal{C}_E$ and $\mathcal{C}_R$, respectively collections of entities and collections of relationships.
(3) *Classify the collections of entities*: given a set of categories, and a set of *hints* (see Section 5), assign to each collection the category it is closest to (or none if does not fit the given categories).

The nodes $N \setminus \mathcal{R} \setminus \{n_{C_i} \mid C_i \in \mathcal{C}\}$ which are neither records nor collection nodes are called *sub-records* and their set is denoted $\mathcal{SR}$. Together with edges connecting them to each other and to a record r, sub-records form the record content sought in (3b) below.

As shown above, for now, only collections of *entities* are classified (not those of relationships). The reason not to classify relationships is that entities ("things") seem even easier to understand for non-IT users, and it is easier for them to provide hints (see later) about entities, than about relationships.

3 IDENTIFYING RECORDS AND COLLECTIONS

We start by a convenient transformation on the graph G. Some of its edges have labels (e.g., RDF triples, edges in JSON maps) while others carry the empty label ϵ. For uniformity, we transform G into an unlabeled graph G', replacing each labeled edge $n_1 \xrightarrow{l} n_2$ with

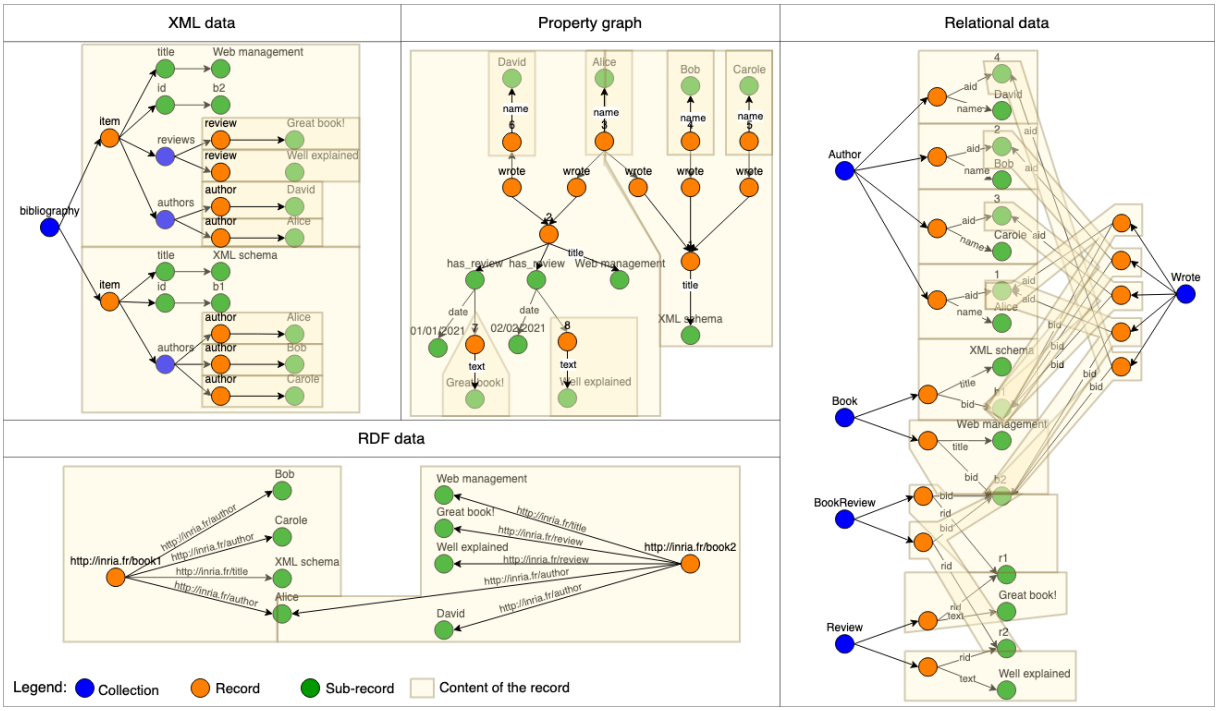

Figure 2: Graph representations of the three bibliographic datasets shown in Figure 1.

an intermediary node labeled l, and connected to n_1, n_2 as follows: $n_1 \xrightarrow{\epsilon} n_l \xrightarrow{\epsilon} n_2$. From now on, we will work on G', whose nodes and edges will be simply denoted as (N', E'). Then:

(1) First, we compute a *quotient summary* of G', that is: we identify a partition $\mathcal{P} = \{N_i\}_i$ of its nodes N', such that $\bigcup_i N_i = N'$ and the N_i are pairwise disjont. We say the nodes from a given set N_i are *equivalent*, and call N_i an *equivalence class*. Then, the quotient summary of G' is a graph whose nodes are the equivalence classes, and such that whenever G' contains an edge $n_1 \xrightarrow{n} {}_2$, its summary contains the edge $EC(n_1) \xrightarrow{E} C(n_2)$, where $EC(n_i)$ denotes the equivalence class of n_i for $i \in \{1, 2\}$. Many quotient summarization techniques have been proposed [5]; we discuss some of them below. Note that while the quotient summary *guides* the abstraction, it may still *differ* from it quite substantially: a set N_i may contain records from *multiple* collections; it may contain an explicit collection node; finally, it may contain nodes which turn out to be in $\mathcal{SR}$.

(2) Next, we compute the *signature* of each equivalence class, i.e. an object reflecting the entities extracted out of the nodes of that equivalence class. This signature holds such statistics for every entity type in T_E (Section 2.2). For instance, given the XML document presented in Figure 2, the signature of the equivalence class representing the nodes ⟨author⟩ is: `{"total_length":14, "PERSON": {"occurrences":3, "extracted_length":14}}` because the three authors have been recognised as Person entities.

(3) We consider that each N_i is *a union of one or more collections, plus possibly (more rarely) a few records*. To separate these, we proceed as follows:

(a) We *cluster the nodes in each* N_i according to their structure.

(i) We transform N_i into a set of transactions $\mathcal{D}$, by turning each node into a transaction, whose item set is the set of labels of the node's non-leaf children. Thus, the first `book` from the XML bibliography in Figure 2 has the items `title`, `id`, `reviews` and `authors` while the second has `title`, `id` and `authors`.

(ii) Given an item set $X \in \mathcal{D}$, we denote by $S(X)$ the *support* of X, defined by $S(X) = |\{Y \in \mathcal{D} \mid X \subseteq Y\}|$. We also define the *transferred support* of an item set X, denoted $T(X)$, based on the set of itemsets $\mathcal{Y}$ obtained as follows. $\mathcal{Y}$ is initialized with all the proper subsets of X (not X and not the empty set) that appear in $\mathcal{D}$. Then, we traverse $\mathcal{Y}$ in the decreasing order of the itemset size, and remove all subsets of Y from $\mathcal{Y}$. For a given X and $\mathcal{Y}$, the transferred support $T(X)$ is defined as: $T(X) = S(X) + \sum_{Y|Y \in \mathcal{Y}} T(Y)$.

Our clustering algorithm starts by computing $S(X)$ and $T(X)$ for each X in $\mathcal{D}$. Then, it proceeds in a greedy manner, choosing the $X \in \mathcal{D}$ with the highest value of $T(X)$, and creating a collection c containing X and all the $\mathcal{D}$ transactions whose items are all included in those of X. We then remove c and the previously selected transactions from $\mathcal{D}$ and repeat the procedure. Each c of more than t elements (where t is a threshold,

e.g., $t = 2$) is considered a **collection**; if the c elements have a common parent, that becomes the collection node, otherwise, the collection is implicit. For each c, every child $r \in c$ is considered a **record**, part of c. The elements of c that are not considered collections nor records are considered **sub-records**. For instance, in Figure 2, the XML sample describes 4 collections (blue nodes): a ⟨bibliography⟩ containing ⟨item⟩ records (orange nodes), two sets of ⟨authors⟩ containing ⟨author⟩ records and a set of ⟨reviews⟩ containing ⟨review⟩ records. The sub-records are the green nodes.

(b) For each record $r \in \mathcal{R}$, we build its content, i.e. a directed acyclic graph (DAG) d_r by following edges outgoing from r, until we reach leaf nodes, or another record node r', or a collection node. The content of each record is represented by a light yellow box in Figure 2. For instance, in the XML bibliography, the second ⟨book⟩ record includes the ⟨title⟩, the two ⟨author⟩, and the two ⟨review⟩. Similarly, for the RDF example, the second ⟨book⟩ record consists of its ⟨title⟩, plus few ⟨authors⟩ and ⟨reviews⟩. For the relational database, the ⟨book⟩ record consists of the ⟨title⟩ and the ⟨id⟩ of the book. Note that the content of a record is extensive using transitivity (e.g. the ⟨author⟩ collection is part of the ⟨book⟩).

In (1), knowledge about possible node types should be injected in $\mathcal{P}$, thus satisfying **R5**. **R2** is met by including in a record many nodes reachable from it, in step (3b). Finally, step (3a) satisfies **R3** by operating on the graph content (not on the original syntax), and **R4** by detecting both implicit and explicit collections.

We now discuss possible choices for the quotient summary technique. The most general method, applicable to *arbitrary* graphs, consists of building a quotient graph summary [5], such as those described in [8] which can be built in linear time in the input size. In particular, a *type-first quotient summary* [8] uses type information *when available* to group nodes by their set of most general types (an RDF node may have several types, and a PG node may have several labels), while partitioning untyped nodes according to their incoming and outgoing nodes. Alternatively, in the particular case of *tree* data models, such as XML or JSON, $\mathcal{P}$ may be a Dataguide [9], which can also be constructed in linear time in the size of the input.

4 DISTINGUISHING ENTITIES FROM RELATIONSHIPS

We now analyse the collections to separate C into C_E, the set of entity collections, from C_R, the set of relationships collection. For instance, the ⟨authors⟩ node in the XML data describes entities while the ⟨wrote⟩ nodes in the property graph describe relationships.

We say a collection c is in C_R iff there exist two collections a and b such that: $\forall r_c \in c, \exists! r_a \in a, \exists! r_b \in b$ such that r_a, r_c are connected by an edge in E' and similarly r_b, r_c are connected by an E' edge. If this is not the case, we consider that c is a collection of entities. For instance, in the property graph in Figure 2, each record in the implicit collection ⟨wrote⟩ links one ⟨author⟩ and one ⟨book⟩, therefore the collection ⟨wrote⟩ contains relationships; the ⟨author⟩ and ⟨book⟩ collections are collections of entities. This check can be sped up by exploiting connection statistics that can be gathered while computing the quotient summary of G'.

5 CLASSIFYING COLLECTIONS

Our next step is to classify each collection in C_E into a given set $\mathcal{K}$ of **categories of interest** to the user, or `Other` if no category is pertinent. In our example, we consider the categories `Person`, `Organization`, `Location`, `Event` and `Creative Work`. As input to the classification process, we are also given a set of **hints** $\mathcal{H}$. A hint $h \in \mathcal{H}$ is a tuple (A, l, B) where $A \subseteq \mathcal{K}$, l is a label and B is a *signature pattern*, which is matched (satisfied) by an *individual signature*, or not. Such a hint states that a node which has a child labeled l and its signature matches B, should be classified as one of the types in A. For instance, the hint $(\{Organization\}, hasCEO, \{Person\})$ states that a record having a property *hasCEO*, whose signature matches *Person*, should be assigned to the category *Organization*.

5.1 Classification algorithm

Algorithm 1 details our classification.

(1) For each record $r \in c$, we initialize $\mathcal{K}_r$, a multiset of candidate categories for r, and a vector of scores of r for each hint. $\mathcal{K}_r$ is a multiset to accomodate the possibility that a given category may be suggested by several hints.
(2) If the record r has a label semantically close to one of the categories in $\mathcal{K}$, this category is stored as a candidate category in $\mathcal{K}_r$, and the similarity score recorded in scores.
(3) For each child *nc* of the record r, we create a pair π containing the label of *nc* and the signature of *nc*.
(4) Next, we compute the similarity of π with each hint h in $\mathcal{H}$ according to Equation 1. This equation gives the similarity between a node and a hint, based on the label and the signature of both elements:

$$sim(\pi, h) = sig\_sim(\pi.S, h.B) + \sum_{k_i \in K_1, k_j \in K_2} cosine\_sim(V(k_i), V(k_j)) \quad (1)$$

where K_1 is the set of keywords present in $\pi.label$, K_2 is the set of keywords present in $h.l$, and $sig\_sim(\pi.S, h.B)$ is the similarity between the individual signature $\pi.S$ and the signature pattern $h.B$, $V(k_i)$ is a vector representation (embedding) of keyword k_i in a multidimensional space. Such embeddings enable detecting that a node labeled `writer` is close to a hint labeled `author`, even if they are different words. We currently use the Word2Vec model [11] for this task.
(5) For each π, we choose the hint h leading to the highest similarity score for π. Each category indicated by the domain of h is added to $\mathcal{K}_r$.
(6) We classify the record r in the category that is the most frequent in $\mathcal{K}_r$, if one category is more frequent than 50%; otherwise, we classify it as `Other`.
(7) Finally, we classify the collection c in the most popular category among its records.

5.2 Constructing hints

The classification of the collections depends on the quality of hints provided as input. Users looking for a concept, e.g., creative work, may have in mind a few properties that creative works have, such as `title` or `author`, and may provide them as hints. Our approach works better if there are *many* hints, in order to obtain a decisive category vote. To overcome burdening human experts, we use

Algorithm 1: Classifying a collection c

Input: a collection c, hints $\mathcal{H}$, categories $\mathcal{K}$

foreach $r \in C$ **do**
  $\mathcal{K}_r \leftarrow \emptyset$
  scores $\leftarrow \emptyset$
  foreach $k \in \mathcal{K}$ **do**
    if *the similarity between k and the label of r is higher than a threshold* **then**
      $\mathcal{K}_r \leftarrow \mathcal{K}_r \cup \{k\}$
  foreach $nc \in r.children$ **do**
    $\pi \leftarrow (nc.label, nc.signature)$
    foreach $h \in \mathcal{H}$ **do**
      scores $\leftarrow$ scores $\bigcup(h, sim(\pi, h))$
    bestHint $\leftarrow$ argmax(scores)
    $\mathcal{K}_r \leftarrow \mathcal{K}_r \cup$ bestHint.domain
  Classify r in the most frequent $k \in \mathcal{K}_r$, or `Other`
Classify c with the most frequent category of its records

knowledge bases like Wikidata [16] and Yago [12] to enhance an existing set of hints, as follows.

A knowledge base KB consists of triples of the form $\langle$`a,r,b`$\rangle$, where `r` is the relationship between the entities `a` and `b`. For example, the triple $\langle$`Albert Wessels, spouse, Elisabeth Eybers`$\rangle$ states that *Albert Wessels' spouse is Elisabeth Eybers*. It also makes statements about the entity type, such as $\langle$`Albert Wessels, type, Person`$\rangle$, from which we can obtain the category an entity belongs to.

Let $k \in \mathcal{K}$ be the category we want to enhance hints for and KB be the knowledge base. The set of properties P_k that are likely to be associated with k can be acquired using the following equation:

$$P_k = \{r \mid \langle \texttt{a,r,b} \rangle \in KB \wedge \langle \texttt{a,type,k} \rangle \in KB\} \quad (2)$$

For example, the triple $\langle$`Albert Wessels, type, Person`$\rangle$ exists in YAGO, and `Albert Wessels` participates in triples such as $\langle$`Albert Wessels, nationality, South Africa`$\rangle$ and $\langle$`Albert Wessels, spouse, Elisabeth Eybers`$\rangle$ among many others. So, `nationality` and `spouse` would be added to the set P_{Person}.

The acquired set of properties may contain some inaccurate information, e.g., for the category `Organization`, some properties such as `date of birth` were retrieved. This happens because knowledge bases such as Wikidata are collaboratively created, which can lead to errors, as evident from the triples $\langle$`Steven Shankman, type, Organization`$\rangle$ and $\langle$`Steven Shankman, date of birth, 1947`$\rangle$. We avoid such errors by *scoring* each property in the set P_k, i.e. errors such as the one reported above will lead to a low score. Formally:

Category score. This score is set proportional to the number of instances of the category the property was participating with. $c\_score(p, k)$ for each $p \in P_k$ is computed as follows:

$$c\_score(p, k) = |\{a \mid \langle \texttt{a,p,b} \rangle \in KB \wedge \langle \texttt{a,type,k} \rangle \in KB\}| \quad (3)$$

While this prunes away errors, we might still get some properties which are common for all the categories in $\mathcal{K}$, thus are not very useful in distinguishing between these categories. For example, the property `Google Knowledge Graph ID` appears for all the different categories. We introduce another score to penalize such properties:

Inverse category score. This score quantifies how unique a property is for distinguishing between categories. Let $P_{all} = \{P_{k_1}, P_{k_2}, \ldots, P_{k_{|\mathcal{K}|}}\}$ be the set of sets of properties retrieved for each category $k_i \in \mathcal{K}$ according to Equation 2. The inverse category score of a property p is computed as follows:

$$ivc\_score(p) = log\left(\frac{1 + |\mathcal{K}|}{1 + |\{P \mid p \in P, P \in P_{all}\}|}\right) \quad (4)$$

A property p that appears in all sets $P_{k_i}, \forall i \in 1, 2, ..., |\mathcal{K}|$ will get $ivc\_score(p) = 0$ since $log(1) = 0$. A property p which appears in only one of the sets $P_{k_1}, P_{k_2}, ..., P_{k_{|\mathcal{K}|}}$ will get the highest score.

Total score. The total score of a property is given by a product of category score and inverse category score, as shown by Equation 5.

$$tot\_score(p) = c\_score(p) * ivc\_score(p) + 1 \quad (5)$$

We score the properties in decreasing order of tot_score and retain the top-z P_{k_z} properties for each category k. For each $k \in \mathcal{K}$ and $p \in P_{k_z}$, we create a hint $(\{A\}, p, range(p))$, where $A = \{k : p \in P_{k_z}\}$ and $range(p)$ is the range of the property p if available from KB, and $\emptyset$ if not present. We are currently working to make the signature patterns $range(p)$ more specific, by exploiting the knowledge the KB may have about the domain of the property p.

6 CONCLUSION AND PERSPECTIVES

The approach described above aims at producing expressive abstraction of datasets organized in a variety of data models. This goes through their transformation in graphs, identifying records and collections, analyzing and classifying collections. The implementation and fine-tuning of our system is ongoing.

Acknowledgments. This work is funded by the DIM RFSI PHD 2020-01 grant and by the AI Chair project SourcesSay Grant no ANR-20-CHIA-0015-01.

REFERENCES

[1] Z. Abedjan, L. Golab, F. Naumann, and T. Papenbrock. *Data Profiling*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2018.
[2] A. C. Anadiotis, O. Balalau, C. Conceicao, H. Galhardas, M. Y. Haddad, I. Manolescu, T. Merabti, and J. You. Graph integration of structured, semistructured and unstructured data for data journalism. *Information Systems*, July 2021.
[3] M. A. Baazizi, C. Berti, D. Colazzo, G. Ghelli, and C. Sartiani. Human-in-the-loop schema inference for massive JSON datasets. In *EDBT*, 2020.
[4] M. A. Baazizi, D. Colazzo, G. Ghelli, and C. Sartiani. Parametric schema inference for massive JSON datasets. *VLDB J.*, 28(4), 2019.
[5] S. Cebiric, F. Goasdoué, H. Kondylakis, D. Kotzinos, I. Manolescu, G. Troullinou, and M. Zneika. Summarizing Semantic Graphs: A Survey. *The VLDB Journal*, 28(3), June 2019.
[6] C. Chanial, R. Dziri, H. Galhardas, J. Leblay, M. L. Nguyen, and I. Manolescu. ConnectionLens: Finding connections across heterogeneous data sources (demonstration). *PVLDB*, 11(12), 2018.
[7] D. Colazzo, G. Ghelli, and C. Sartiani. Schemas for safe and efficient XML processing. In *ICDE*. IEEE Computer Society, 2011.
[8] F. Goasdoué, P. Guzewicz, and I. Manolescu. RDF graph summarization for first-sight structure discovery. *The VLDB Journal*, 29(5), Apr. 2020.
[9] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, 1997.
[10] H. Lbath, A. Bonifati, and R. Harmer. Schema inference for property graphs. In *EDBT*. OpenProceedings.org, 2021.
[11] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, 2013.
[12] T. Pellissier Tanon, G. Weikum, and F. Suchanek. Yago 4: A reason-able knowledge base. In *ESWC*, 2020.
[13] R. Ramakhrishnan and J. Gehrke. *Database Management Systems (3rd edition)*. McGraw-Hill, 2003.
[14] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. Xmark: A benchmark for XML data management. In *PVLDB*, 2002.
[15] W. Spoth, O. A. Kennedy, Y. Lu, B. Hammerschmidt, and Z. H. Liu. Reducing ambiguity in JSON schema discovery. In *SIGMOD*, 2021.

[16] D. Vrandečić and M. Krötzsch. Wikidata: A free collaborative knowledgebase. *Commun. ACM*, 57(10), Sept. 2014.