# Computing Generic Abstractions from Application Datasets

Nelly Barret, Ioana Manolescu, Prajna Upadhyay

# Computing Generic Abstractions from Application Datasets

Nelly Barret
nelly.barret@inria.fr
Inria, IP Paris, France

Ioana Manolescu
ioana.manolescu@inria.fr
Inria, IP Paris, France

Prajna Upadhyay
prajna-devi.upadhyay@inria.fr
Inria, IP Paris, France

## ABSTRACT

Digital data plays a central role in sciences, journalism, environment, digital humanities, etc. Open Data sharing initiatives lead to many large, interesting datasets being shared online. Some of these are RDF graphs, but other formats like CSV, relational, property graphs, JSON or XML documents are also frequent.

Practitioners need to *understand* a dataset to decide whether it is suited to their needs. Datasets may come with a schema and/or may be summarized, however the first is not always provided and the latter is often too technical for non-IT users. To overcome these limitations, we present an end-to-end *dataset abstraction* approach, which (*i*) applies on any (semi)structured data model; (*ii*) computes a *description* meant for human users, in the form of an Entity-Relationship diagram; (*iii*) integrates Information Extraction and data profiling to *classify* dataset entities among a large set of intelligible categories. We implemented our approach in a system called Abstra, and detail its performance on various datasets.

## 1 INTRODUCTION

Data-driven applications are in the heart of many businesses and governmental initiatives. When possible, data is shared in **open access** which leads to circulating knowledge on various domains, ranging from journalism to education, environment or health.

With this trend of open-access data, the World Wide Web Consortium's recommends to share data as RDF graphs, and this has been widely adopted, e.g., to build the Linked Open Data Cloud. However, practitioners also use a variety of other data formats such as relational data, XML or JSON documents and property graphs. Thousands of **CSV** datasets are available on Kaggle and the French public portal data.gouv.fr. **XML** is used to share bibliographic notices on PubMed, a leading website in the medical domain. **JSON** has become the reference format for the French parliament to increase the transparency of the public life, notably on the websites NosDeputes.fr and NosSenateurs.fr. **Relational databases** are sometimes shared as dumps, including schema constraints (e.g. primary and foreign keys), or as CSV files. **Property graphs** (PGs, in short, such as pioneered by Neo4J) are used to share Offshore leaks, a journalistic database of offshore companies, or by the LDBC council.

Practitioners need to get a basic **understanding of a dataset content in order to decide whether it suits their needs**. Also, data producers need tools to **generate automatically a description of the dataset** they want to share. They both need datasets descriptions.

Datasets descriptions generally take the form of a documentation or a schema. While these can help users in their quest of the right dataset, they have limitations. Documentation is often lacking or insufficient (writing documentation is time-consuming). A *schema* may be inferred from the dataset to describe its structure. However, schemas have several limitations: (*i*) schemas are rare

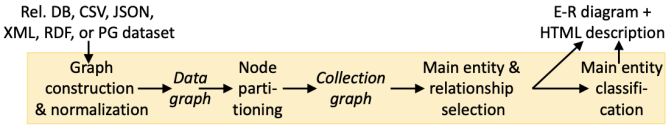


**Figure 1: Abstraction pipeline.**

when sharing semi-structured datasets (XML, JON, RDF and PG). Even when a schema is supplied with or extracted from the data: (*ii*) schema *syntactic details*, such as regular expressions, are *hard to interpret for non-IT users*; (*iii*) schema inference techniques mainly focuses on the dataset structure, not on its *content*. It does not take advantage of linguistic information available in structures and text values; (*iv*) they *do not reflect quantitatively the dataset* either, whereas showing all the structures regardless of their importance in the dataset may overwhelm the user. *Data summarization* techniques [4, 17, 19, 27, 36, 47] partially lift (*i*). In the particular case of RDF graphs, an *ontology* may accompany the graphs and give an overview of the semantic of the dataset, thus lifting (*iii*) but not the others. *Pattern mining* [31] may help users to grasp the popular patterns in their datasets, e.g. items often purchased together. This allows to bypass (*i*) and (*iv*) only.

To answer practitioners and data producers needs, we present a **novel approach for abstracting any tabular, tree-structured or graph-structured dataset.** Our work leverages the idea that any dataset comprises some *entities* (data objects), typically grouped in *collections* (sets). Such sets of entities are often connected by some *relationships*. Our abstractions differ from the classical Entity-Relationship schemas [44] in the sense that **our entities may have a deeply nested structure**. This modeling seeks to account for the recent adoption of complex, non-relational data formats. To identify entities and their relationships, we proceed as follows (Fig. 1):

**(1.)** Based on how each data model represents fragments of data, we transform any dataset into a **labeled directed graph** (Sec. 2).

**(2.)** We **group nodes into collections** by leveraging how each data model encodes similarity between data objects, e.g. data types are a way to encode such similarity (Sec. 3).

**(3.)** We elect few collections as the **main ones**, such that, they together represent a large part of the dataset. Each main collection is a set of *entities* that may have a simple or very complex structure. Also, we need to efficiently identify *main collections of entities* (referred to as *main entities*), regardless of the potentially complex and cyclic graph structure. To do so, we introduce a notion of *data weight* and dedicated algorithms (Sec. 4). Next, we identify **relationships** between main entities.

**(4.)** We attempt to **classify each main collection of entities** into a **semantic class**, such as Person, Product, etc. Our classification (Sec. 5) leverages (*i*) entities names, properties and text values on which Information Extraction is applied to find *named entities*; (*ii*) a set of *semantic properties* we built using well-known knowledge bases and existing semantic resources, while users can also specify semantic properties they have in mind; (*iii*) language models to

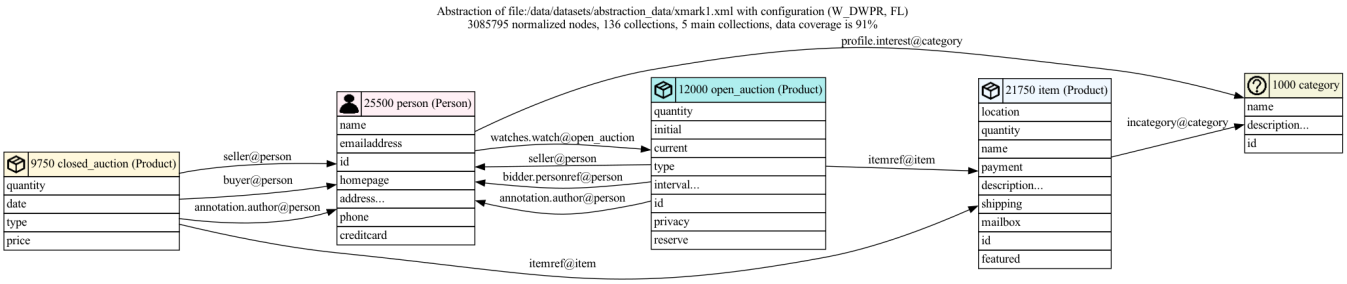


**Figure 2: E-R model computed by Abstra from an XMark [45] XML document (3M nodes).**

compute the similarity between main collections entities and the semantic classes.

Finally, we output a **compact and graphical description** comprising the main entities, together with their semantic class and their possible relationships. Note that this description is free of any syntactic details of the input format. For instance, the abstraction of an XMark [45] XML document (3M nodes, 80 different labels, 124 labeled paths) contains five main entity sets, each shown as a box in Fig. 2. Each main entity is described by its size, its name and its semantic class (within parenthesis if found). The entity properties are displayed below. Three dots indicates that the property is deeply nested, e.g. the mailbox contains mails which have four properties (date, sender, receiver, text). Users can unfold the nested structure through an HTML/JavaScript interface. Our approach is implemented in a prototype, Abstra, demonstrated recently [6].

We focus on **application datasets**, each describing a specific scenario, e.g., the Senate, companies with their stakeholders and officers, online auctions, etc. in the above examples, and do not consider "universal" datasets, such as Wikidata [48], YAGO [41], etc. No universal dataset abstraction is likely to be *both* compact and comprehensive; extracting an application dataset from a universal one, based on keywords or terms the user already knows, is an orthogonal problem, e.g., [18, 23].

In reverse w.r.t. the usual database design steps [44], our goal is to retrieve from the dataset, the conceptual model originally behind it. For this, Sec. 2 provides a unified analysis of the (semi)structured data models we consider, highlighting how they encode various aspects of an application domain. Sec. 3 shows how to represent any dataset as a directed, labeled graph, then how we partition nodes into collections, and organize these in a graph reflecting their relationships in the original dataset. Sec. 4 explains how we select main entity collections and their relationships, through novel weight-based ranking methods, leveraging graph connectivity and collections cardinalities. Sec. 5 describes how to assign a semantic class to a collection, given its properties and content. In Sec. 6, we evaluate the quality, scalability, and compactness of our abstractions, before discussing related work and concluding in Sec. 7.

Sample graphical abstractions, and our code, can be found at: https://team.inria.fr/cedar/projects/abstra/.

## 2 FROM APPLICATIONS TO DATASETS: A UNIFIED PERSPECTIVE

We consider the following data models: relational and CSV; XML and JSON (documents); finally, RDF, and property graphs (PGs, in short). Towards abstracting datasets, we first analyze *how each model describes a set of core features of any real-world application*: application objects (at various granularities), data types, relationships, and information about application objects "of the same kind". We use "kind" (not "type"), because "type" has precise (but different!) meanings in different data models. In this work, "same-kind" designates *similarity from the application perspective.*

We denote by **record** a piece of data in a dataset, describing an application domain object. Specifically, in a relational database (RDB), each tuple or attribute value is a record; in a JSON or XML document, each node is a record; in a PG or an RDF graph, each node is a record. Clearly, some records may contain other records, e.g., tuple attribute value records are contained (or part of) the respective tuple record[1]. We call **leaf record** (or **value**) a record that does not (syntactically) contain any other records: relational attribute values; leaf nodes in JSON and XML documents; literals in RDF graphs; atomic values of node or edge attributes in a PG.

Each data model supports some **value types** such as String, Float, etc., described, e.g., in the ISO/IEC 9075:2011 standard for relational databases, or the W3C's standard [50] for XML and RDF. Some value types model complex real-world concepts, e.g., the W3C's gMonthDay type denotes "a specific day of a month, recurring every year", such as "the 4th of July". We view such types as more **expressive** (closer to the application scenario) than String or Float. Even if declared of type String, values such as "Paul Jones", "paul@outdoor.com" or "https://twitter.com/pjones" denote, more expressively: a person name, respectively, an email, and an URI. Frequent expressive types are named entity types such as, Person, Location, Organization, as well as Email, URL, Date etc.

Expressive types may be *inferred from the data*, through profiling [1], pattern matching, Named Entity Recognition (NER), etc. Within a String value, one may encounter *several expressive-type values*. For instance, "Paul Jones lives in London; after ten years at Mountaineering, he has been sales director at Outdoor since Jan. 2022" contains one person name, a Place (London), two Organizations (Mountaineering and Outdoor), and a Date (Jan. 2022). Expressive types bring useful information on the dataset content. Thus, for a given expressive type set $\mathcal{T}$, we assume **any value of an expressive $\mathcal{T}$ type has been identified in the dataset**, through one of the abovementioned methods. This is done for instance in the ConnectionLens [2] system, on which this work is based. Our abstraction method leverages such expressive-type values to semantically classify the main entity collections (Sec. 5).

[1]This raises the question: where does one record end and where does another record start? This question is complex for some data models, such as XML, RDF etc. We will address it in Sec. 4.

Next, we consider where and how **relationships** between application objects are expressed in the data. A first category of relationships comprises *anonymous part-of* relations, e.g., the records of the objects within a JSON array are part of the record corresponding to the array; similarly, an XML element is part of its parent. Next, we identify *named binary relationships*: in a relation Person(id, name, address), in the tuple (1, "Alice", "Main Street"), a binary relation named "address" holds between the tuple record and the value "Main Street". Similar relations hold between: an XML element record and each of its attribute values, e.g., ⟨person id="1" name="Alice" address="Main Street"/⟩; a JSON map record and its children; a PG node and each of its attributes. Schema information may be attached to an XML document, as a DTD [49] or XML Schema [50]. If it states that an attribute, e.g., @id, is a PK (#ID) for ⟨person⟩ elements, and that, e.g., @parent is a FK (#IDREF) on person@id, then ⟨student name="Bob" parent="1"/⟩ also leads to a binary "parent" relation between the student and the person records. PG edges also are named binary relationships, and may have their own attributes; the relational model also allows this, e.g., in a relation WorksFor(personID, companyID, startDate). Last but not least, each RDF triple naturally encodes a binary relationship. Relationships of arity higher than 2 are more rare, and we do not consider them in this work.

Our last question is: how do data creators signal, in a dataset, **same-kind records**, that is, records describing real-world objects of the same (or similar) nature? In a relational database, all tuple records of the same table are of the same kind; the same holds for XML elements with the same name in a document. If an optional XML DTD or XML schema is available, each element is assigned a type, and all same-type elements are of the same kind. Moreover, leaf records (values) participating in a given named binary relation, with non-leaf records of the same kind, are of the same kind, e.g., values of "address" attribute in the Person relation $R$ above, "@email" attributes of ⟨person⟩ XML elements.

RDF and property graphs allow attaching to node records *zero, one or more RDF types (resp. PG labels)*, e.g., a node can be a "FrenchCitizen", a "Student" and also an "PhDStudent". If an (optional) ontology is attached to an RDF graph, it may lead to *infer* some node types, e.g., if $x$ is an UndergraduateStudent, then $x$ is also a Student. In this work, we assume that the set of facts inferred from the ontology is *finite, already computed* [26], and *part of the RDF graph*. A node's types can be organized in a **type DAG** (Directed Acyclic Graph) based on "subclassOf" relations. The DAG may not be connected, e.g., FrenchCitizen is unrelated to the student types.

Some records, such PG or RDF nodes w/o types (or labels), of non-leaf JSON records, carry no explicit kind information. Instead, their kind is *implicitly* encoded in the dataset structure. In these cases, and when there are several explicit types, we need a method to decide which records are of the same kind (see Sec. 3.3).

We call **kind name(s)** of a record: for a tuple $r \in R$, the relation name $R$; XML element names; RDF node types and PG node/edge labels (when present). Other records don't have kind names.

The discussion for CSV datasets is largely the same as for relational databases, except that value types may have to be inferred through profiling (as opposed to stated in a schema). Further, if a header is absent, attribute names are unknown; in this case we give them simple distinct names $attr_1, attr_2$ etc.

# 3 DATA GRAPHS AND COLLECTION GRAPHS

To abstract datasets of any model, we turn them into directed graphs (Sec. 3.1). Then, we build from this graph, a much more compact collection graph (Sec. 3.2) on which abstraction will continue.

## 3.1 Conversion to a data graph

**Target model: directed graphs** We need a common formalism to represent (all aspects of) a relational, CSV, XML, JSON, RDF, or PG dataset. Directed graphs are a natural format, since they generalize all the models. The unified graph could either have fine-granularity nodes and edges (each of which has only a label, à la RDF), or large-granularity ones (which may have their own attributes, à la PG). We prefer the former because (*i*) it is more natural to the data models of RDF, JSON and XML, and (*ii*) any straightforward conversion to PGs would force turning some dataset nodes into PG nodes, while others just become attributes. Instead, our goal is to make such judgments *at a higher (conceptual) level* (which nodes are entities, and what are their attributes, possibly nested?), based on a deeper analysis (Sec. 4).

**Conversion into data graph** We turn any dataset into a graph $G_0=(N_0, E_0, \lambda_0)$ where $E_0 \subseteq N_0 \times N_0$ is a set of directed edges, and $\lambda_0$ is a function labeling each node and edge with a string label, that could in particular be $\epsilon$ (the empty label). The conversion also prepares the ground for grouping all nodes of the same kind (recall Sec. 2), into a collection (as we will discuss in Sec. 3.2).

RDF graphs map directly into $G_0$; nodes are labeled with URIs, blank nodes, or literals, while edges are labeled with URIs.

XML documents lead to trees, where element names and text node contents become node labels, attribute names become edge labels, and all the other edges have an empty label ($\epsilon$). If ID-IDREF information is available from a schema, we leverage it to add the corresponding edges; otherwise, we find ID-IDREF pairs by profiling the data [1, 33]. For instance, from the XML fragment:
⟨paper id="p1" writtenBy="a1 a2"/⟩⟨/paper⟩⟨paper id="p2"⟩...⟨/paper⟩
⟨authors⟩⟨author id="a1"⟩Alice⟨/author⟩⟨author id="a2"⟩...⟨/authors⟩
the attribute writtenBy of the first paper becomes two edges labeled writtenBy, from the paper, to the two respective authors.

JSON documents similarly lead to trees, with a node for each map, array, or leaf, and parent-child edges connecting them.

A CSV file leads to a node for each tuple; in turn, such a node has edges (labeled with the attribute names, if the CSV file has a header) going toward each attribute value.

A relational database is similarly modeled, with a node for every tuple; in the presence of a constraint of the form "$R.a$ is a foreign key referencing $S.b$", each node $n_r$ corresponding to a tuple $r \in R$ has an outgoing edge labeled $a$ pointing to the $S$ tuple node $n_s$ created for the respective tuple $s \in S$. Relational databases have been modeled in this way for keyword search, since [9].

From a property graph, we create a node for each PG node and for each of its attributes; edges labeled with attribute names connect them. For each PG edge $e$ from $np_1$ to $np_2$, $G_0$ has a node $ne$ representing $e$, with the label(s) of $e$, and nodes representing $e$'s attributes, just like for PG node attributes. We also add to $G_0$ an edge from $np_1$ to $ne$, and one from $ne$ to $np_2$.

**Normalized graph** In $G_0$, some edges have empty ($\epsilon$) labels, while others carry meaningful labels. Regardless of the input data model,

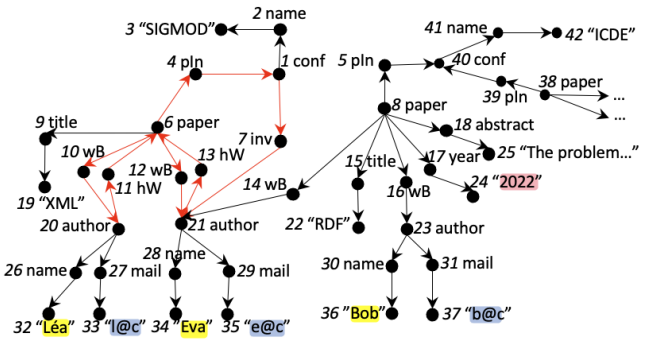


**Figure 3: Sample normalized graph.**

for uniformity, we transform $G_0$ into a **normalized graph** $G$=($N$,$E$,$\lambda$), copying all the nodes of $G_0$ and all its $\epsilon$-label edges, and replacing each $G_0$ edge of the form $n_1 \xrightarrow{l} n_2$ where $l \neq \epsilon$ by two unlabeled edges $n_1 \rightarrow x_l$, $x_l \rightarrow n_2$ where $x_l$ is a new intermediary node labeled $l$. All subsequent abstraction steps apply on the normalized graph $G$.

Fig. 3 shows a sample normalized graph $G$. It describes three papers (one is only partially shown), which are published in (pln) conferences. The papers are written by (wb) authors, described by their name and email. Note the inverse "has written" (hW) edges going from the authors to papers. Author *21* is invited (inv) by the conference organizers. As Fig. 3 shows, the graph may contain: (*i*) nodes such as papers, whose information content is *deeply nested*, and (*ii*) several *cycles* (in-cycle edges are shown in red). Note the *expressive-type values* in Fig. 3: person names are highlighted in yellow, dates in pink, and emails in light blue.

## 3.2 Building the collection graph

We now proceed to building the structure at the core of our entity and relationship detection, called *collection graph*. Sec. 3.3 explains how we partition nodes into collections; this step identifies structural similarity among nodes in the graph. Sec. 3.4 shows how we organize this information in a graph that models the links between the various collections.

## 3.3 Node partitioning into collections

We aim at a partition $P$=$\{C_i\}$, $1 \leq i$ of the graph nodes $N$, such that $\bigcup_i C_i = N$ and the $C_i$s, called *equivalence classes* or *collections*, are pairwise disjoint. Intuitively, each equivalence class should hold a set of nodes reflecting real-world objects of the same kind. As discussed in Section 2, "kind" information is *encoded explicitly, in data model-specific ways* in some data models, and *implicitly encoded* in others. Two principles guide our partitioning: (★) Whenever "kind" information is explicit, we should leverage it, as it reflects the dataset producers' application-domain knowledge; (■) The number of equivalence classes should remain "reasonable", e.g., at most in the hundreds (as opposed to thousands or more). This is because abstractions are meant for novice users, which should not be overwhelmed with detail.

Partitioning is simplest for models explicitly assigning only one "kind" to their records: relational, CSV and XML.

$G$ nodes coming from a relational database or CSV are partitioned as follows. Let $R(a_1, ..., a_n)$ be a relation (or CSV file). The nodes created from any tuple $r \in R$ are equivalent to each other. The nodes created from attribute values $r.a_i$, for $r \in R$ and an attribute $a_i$, are equivalent. All nodes introduced by normalization, labeled $a_i$, and connecting a tuple node to its $a_i$ value, are equivalent.

For $G$ nodes derived from XML, we proceed as follows.

- If a schema is present: XML elements of the same type are equivalent. Otherwise, XML element nodes with the same label are equivalent.
- The nodes created from edges connecting an element to an attribute are partitioned into equivalence classes by the element name followed by the attribute name. For example, from the XML snippet in Sec. 2, we obtain one class for paper@id, one for paper@writtenBy, and one for author@id.
- For each equivalence class $x$ whose nodes have text children, we create the equivalence class denoted $x$# which groups all such children. In the same XML example, this leads to paper@id# for the values of these attributes, author# for all the text children of author elements, etc.

For RDF graphs, many graph summarization methods could be used (see Sec. 7). Following principle (★) above, we *use types (whenever available)* in order to partition the nodes[2]. Nodes having at least one type could be partitioned according to their set of types, or just by their most general types (the set of DAG roots), or the most specific (the DAG leaves). Following principle (■), we choose to group them *by the set of their most general types*, to obtain fewer equivalence classes. Untyped nodes can be grouped based on the labels on their immediate neighbors (labels of their incoming and outgoing edges in the $G_0$). It has been shown [16, 39] that grouping RDF nodes by the set of their outgoing properties artificially creates many equivalence classes, e.g., a node describing an article, having a title, authors, year, and a note, is not equivalent to another one having the same properties except the note. To mitigate this, [39] proposes a heuristic merging the equivalence classes into at most $r$ classes. Avoiding the need for users to specify $r$, the Typed Strong summary [24, 25] groups untyped nodes based on a notion of *property clique*, as follows. Two properties $p_1, p_2$ that have a common subject are part of the same *outgoing property clique*; similarly, if $p_3, p_4$ have a common object, they are in the same *incoming property clique*. To each node is thus associated *exactly one outgoing and one incoming property clique*; one, but not both, may be empty. Two untyped nodes are equivalent when they have the same incoming and the same outgoing property clique.

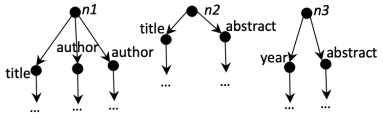


**Figure 4: Normalized RDF sample illustrating partitioning.**

For instance, consider the (normalized) RDF graph in Figure 4. Properties title and author are in the same outgoing clique because of $n_1$; title and abstract are in the same outgoing clique because of $n_2$, and similarly for abstract and year, because of $n_3$. As a result, all properties in this example are part of the same outgoing property

[2]RDF provides the very general RDF:Thing type, generalizing all other types. Since it is not informative, we ignore it, and use only the meaningful dataset types.

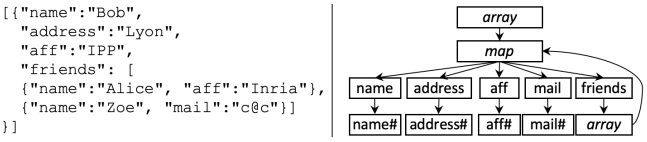


**Figure 5: Sample equivalence classes obtained from JSON.**

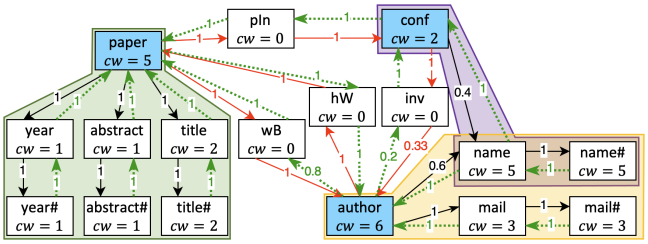


**Figure 6: Sample collection graph corresponding to the normalized graph in Fig. 3.**

clique[3], thus nodes $n_1$, $n_2$ and $n_3$, sharing also an empty incoming clique, are equivalent. We adopt this technique since it needs no parameter, and it has been shown to keep the number of equivalence classes under control, while building meaningful "same-kind" equivalence classes [25].

In graphs derived from PGs, we apply the same method as for RDF, except that node and edge labels replace RDF types. No subclass relationships hold between labels, therefore, we group labeled $G$ nodes by their complete set of labels.

In JSON, "kind" information is not explicitly present; nodes do not have types, and many have empty labels. Instead, we note that (*i*) nodes occurring on the same labeled path from the JSON document root are often of the same kind (to form the paths, we assign the special name *map* to all map nodes, *array* to all array nodes, and concatenate node and edge labels); (*ii*) map nodes having the same (or similar) attribute names typically contain similar information. Therefore, we first group nodes by their incoming path; then, we compute outgoing property cliques (just like the Typed Strong summary does with RDF properties, Fig. 4), and fuse two path-based equivalence classes whenever they have the same property cliques. For instance, in Fig. 5, the snippet on the left leads to the equivalence classes on the right. The equivalence class on the path *array.map.*friends.*array.map* is fused with the one on *array.map*, since they have the same property clique. Thus, an edge links array to *map* and *map* has a (new) child equivalence class mail.

## 3.4 The collection graph

We call *collection graph* the graph whose nodes are the collections $C_i$, and having an edge $C_i \rightarrow C_j$ if and only if for some nodes $n_i \in C_i$, $n_j \in C_j$, $n_i \rightarrow n_j \in E$.

Fig. 6 shows the collection graph for the graph in Fig. 3. Here, in each collection, all nodes have the same label, shown in the collection; this does not hold in general, e.g., in a collection of RDF nodes, each node has a different label. Shaded areas, *cw* attributes, dotted arrows and numbers on arrows will be explained in Sec. 4.

[3]Some RDF properties are *generic*, e.g., rdf:comment, and could be attached to nodes having nothing to do with each other. Generic properties are few, and fixed (application-independent); summarization disregards them when computing cliques [25].

Importantly, **even if the data is acyclic, the collection graph may have cycles** (shown in red in Fig. 6). For instance, in the XML snippet below, some list nodes are children of item nodes and viceversa, leading to a cycle between the list and item collections.

⟨description⟩ ⟨list⟩ ⟨item⟩⟨list⟩... ⟨/list⟩ ⟨/item⟩... ⟨/list⟩ ⟨/description⟩

# 4 MAIN ENTITIES AND RELATIONSHIPS

Some collections can be seen as "the main ones", while others contain "fields (or attributes)" of the main collections, potentially nested several levels deep. For instance, in Fig. 6, intuitively, paper, author and/or conf seem the most interesting entity collections; it appears natural that year, abstract and title etc. would be part of paper; similarly, mail and name are part of author.

In general, collection graphs may have hundreds of nodes, and their structure may be quite complex (involve several cycles, etc.) Towards determining main entities and relationships in such graphs, in this section, our goals are to:

(1) Identify a set of *main entity collection nodes*: a node in each of these collections is the root of a *main entity* in $G$. Sec. 4.1 discusses a set of simple baselines, while in Sec. 4.2 we introduce more elaborate methods, based on data weights and PageRank [15] scoring.
(2) For each main entity collection node, find which other collections of nodes "belong to" (should be viewed as being part of) the main collection ones. We call this task *entity boundary detection*, and provide algorithms for it in Sec. 4.3.

Sec. 4.4 shows how we report to the users the most important $E_{max}$ entity collections (for a given integer $E_{max}$). Each of these is a subgraph containing a main entity collection node, and all the other collections within its boundaries. We identify and report *all* relationships between the most important entity collections (Sec. 4.5); as in Fig. 2, their number is moderate if $E_{max}$ is low.

**Main entity eligibility criteria** Based on classical E-R design [44], we identify two criteria a collection must satisfy, to be considered as a potential main entity. ($C_1$) It should contain *more than one node*, discouraging one-node collections as degenerate "entity sets". Also, the node in such a singleton collection is typically a *container* to many nodes from another, potentially interesting collection, e.g., in XMark, the (only) people element is the parent of all person nodes. ($C_2$) It must have *at least two child collections* and/or *at least one child collection having a leaf child*. This adapts to our setting the intuition that "entities have attributes": (*i*) childless collections do not qualify; (*ii*) collections with a single child that is a leaf do not qualify, e.g., nodes of the name collection in Fig. 6 have no internal structure (just a string); (*iii*) collections with a single, non-leaf child do not qualify; they act more as "containers" for their children. For instance, in the XHTML search results grouped in pages below:

⟨top⟩ ⟨page⟩⟨result⟩...⟨/result⟩⟨result⟩... ⟨/result⟩⟨/page⟩
⟨page⟩... ⟨/page⟩... ⟨/top⟩

the page collection has as single child, namely result, with the actual data; the result collection is eligible, but page one is not.

## 4.1 Simple baselines for selecting main entity collection nodes

The intuition that "children collections belong to their parents", e.g., title belongs to paper, would suggest **the root(s) of the collection**

**graph** as main collections. However, this may be inapplicable: the collection graph may have no root, if it is cyclic as in Fig. 6.

Alternatively, we could select **the collections with the highest numbers of nodes**, based on the intuition that they cover a large part of a dataset. This may also be inappropriate in some cases, e.g., in Fig. 3, the name collection is the largest, because it contains both people names and conference names, but name says very little of what the dataset is about.

Another intuition is that important entities are likely to have many attributes. Thus, we can select the collections having **the highest number of children** in the collection graph. For instance, paper nodes have four children: year, abstract, title, and wb. This method, however, does not account for deeply nested structure. Thus, instead, we could count $k$-**desc** (descendants at distance $k$, e.g., 1 for children, 2 for children and grandchildren, etc.) etc. and select the collections with the highest values.

A variant denoted $k$-**leaf**, for some integer $k$, consists of counting only **leaf descendants**, since data content in leaf (value) nodes (paper titles, paper author names etc.) is more important than the structural nodes which only serve to "organize" the values.

These simple baselines have limitations: they depend on a fixed distance $k$, and do not reflect the distribution of data nodes in the graph, or the global graph structure.

## 4.2 Weight-based selection of the main entity collections nodes

We now describe more elaborate ways to select main entities, based on the cardinalities and structure of a data graph.

*4.2.1 DAG weight computation ($w_{DAG}$) .* We start by defining a notion of data weight as follows. We assign to each leaf node $n$ an **own weight** $ow(n)$ equal to the number of $G$ edges incoming $n$. In general, in tree data, $ow = 1$ for all leaf nodes, e.g. the $ow$ of the value node "Léa" is 1. However, this does not hold for graphs, such as in RDF where a literal may be the value of many triples, thus $ow>1$. Next, we define the $ow$ **of a leaf collection** as the sum of the $ow$ of its nodes, e.g., in Fig. 6, $ow$(title#)=2 and $ow$(name#)=5. Further, for each edge $C_i \rightarrow C_j$ in the collection graph, we attach an **edge transfer factor** $f_{i,j}$ as the fraction of nodes in $C_j$ having a parent node in $C_i$; $0<f_{i,j}\leq 1$. Each $f_{i,j}$ labels one solid arrow in Fig. 6; it quantifies how much $C_j$ is part of its parent $C_i$. For instance, there are 5 name nodes, but two belong to conferences, thus the transfer factor from name to conf is $2/5$.

Now, we describe how $w_{DAG}$ works. The intuition is that "children weight propagates to parents", that is, the ancestors of leaf nodes should reflect the weight of their descendants. Such ancestor-descendant relationships are well defined for directed acyclic graphs (DAGs) only. Therefore, we (*i*) compute all the cycles that the collection graph may have, (*ii*) label every edge as being part of some cycle(s) or not, and (*iii*) propagate weights along each path made exclusively of edges that do not participate in any cycle (we call these *non-cyclic paths*). In-cycle edges appear in red in Fig. 6,

In details, we assign to each collection a **collection weight** $cw$, which on leaf collection is initialized to $ow$, and on others, to 0. Then, for each non-leaf collection $C_0$, and non-cyclic path of the form $C_0 \rightarrow C_1 \rightarrow C_2 \rightarrow \ldots C_l$ where $C_l$ is a leaf collection, we add $ow(C_l) \cdot \Pi_{0\leq j\leq l-1}(f_{j,j+1})$ to $cw(C_0)$: observe the multiplication of transfer factors along the edges. For instance, the collection author in Fig. 6 obtains $cw = 6$, i.e. 3 spread from mail# and 3 from name#.

While $w_{DAG}$ reflects collection hierarchies, its drawback is to ignore the collection graph structure encoded by cycles.

*4.2.2 PageRank weight computation ($w_{PR}$).* PageRank (PR, in short) [15] is a well-known rank computation method for graphs, including cyclic ones. Starting from an initial node weight distribution, PR iteratively reallocates weights among nodes, each node distributing its weight among its children. Our method $w_{PR}$ starts by creating $\mathcal{G}_{PR}$, the collection graph with *inverted* collection edges (dotted green edges in Fig. 6), to reflect that leaf collections transmit their weight towards their ancestors. Next, as in the classical PR, each of the $|P|$ collection nodes is assigned a score of $\frac{1}{|P|}$, and each edge outgoing a node $n$ is given the weight $\frac{1}{|\text{out\_degree}(n)|}$. Finally, as in PR, we iterate until convergence. We denote the node weight computed in this way $\mathbf{w_{PR}}$.

*4.2.3 Data-weighted PageRank computation ($w_{d_w PR}$).* A drawback of $w_{PR}$ is that it does not take into account the number of nodes in each collection. This contradicts our goal (Sec. 1) of selecting entities which cover a large part of the dataset. One could think of initializing the PageRank nodes scores with $cw$ as defined in Sec. 4.2.1; however, PageRank final node scores are independent of the initial ones. Instead, we devise the $w_{d_w PR}$ method where cardinalities are reflected *in the PR edges weights*. $w_{d_w PR}$ builds $\mathcal{G}_{PR}$ and initializes node scores to $\frac{1}{|P|}$, as in Sec. 4.2.2. Next, we assign to each edge in $\mathcal{G}_{PR}$ a weight, shown as label of each green dotted edge in Fig. 6:

(1) For each collection $C_i \in \mathcal{G}_{PR}$ *with* some outgoing edges, the weight of each outgoing edge $C_i \rightarrow C_j \in \mathcal{G}_{PR}$ is the number of *data* edges (in $\mathcal{G}$) corresponding to that edge, divided by the total number the data edges outgoing $C_i$ nodes. Note that these weights sum up to 1 for each $C_i \in \mathcal{G}_{PR}$, following the PR convention.
(2) For each collection $C_i \in \mathcal{G}_{PR}$ *without* outgoing edges, we create an edge from $C_i$ to every node $C_j \in \mathcal{G}_{PR}$, with a weight equal to the number of data nodes in $C_j$, divided by the total number of nodes in the graph $\mathcal{G}$. Such weights would also sum up to 1 for each $C_i$.

Finally, as in PR, we iterate until convergence. We denote the node weight computed in this way $\mathbf{w}_{d_w \mathbf{PR}}$.

**Weights from edge counts** Observe that we compute $\mathcal{G}_{PR}$ edge weights from *edge* (not node) counts. To see why, consider an address shared by two companies (thus, two incoming edges). Intuitively, each company has this address, not just half of it. More generally, *shared nodes should weigh as many times as they are shared*; this is reflected by counting incoming edges.

## 4.3 Determining entity boundaries

We now discuss methods for deciding which nodes of the collection graph should be part of (contribute to) which entity collection node. Different methods are more naturally suited to different ways of selecting the main collections, as follows.

*4.3.1 Boundaries for baseline methods.* Our baseline methods for selecting main collections, $k$-desc and $k$-leaf (Sec. 4.1), naturally lead to defining the boundary of a collection $C^*$ as: all the nodes

(respectively, all the leaves) reachable from $C^*$ at a distance of at most $k$. We call these methods $\mathbf{bound_{desc}}$, respectively, $\mathbf{bound_{leaf}}$.

*4.3.2 Boundary when using DAG weights.* $w_{DAG}$ (Sec. 4.2.1) leads to a different notion of boundary: any collection that *transfers weight* to $C$, is within the boundary of $C$. We call this the **DAG propagation boundary ($\mathbf{bound_{DAG}}$)** of $C$. In Fig. 6, name is within the boundary of conf as well as within the boundary of author, which further includes the name and email collections.

*4.3.3 Boundary when using PR weights.* When using $w_{PR}$ or $w_{dwPR}$, weight transfers are global and iterative, potentially among all the collections, thus, they do not lead to a clear way of deciding which nodes should be part of a boundary. Further, the $bound_{DAG}$ boundary method does not reflect the weight transfers across cyclic paths, thus, it is not consistent with the PR propagation principle.

Instead, we devise flooding-style boundary computation methods, which exploit the edges along paths rooted in $C$, a collection just selected as a "main" one. (*i*) We say an edge $C_i \rightarrow C_j$ is *at-most-one*, if each node from $C_i$ has at most one child in $C_j$. As in conceptual data modeling, this is an indication that $C_j$ may hold attributes of records in $C_i$. (*ii*) We say $C_i \rightarrow C_j$ is *strong*, if at least $f_{min}$ of the nodes in $C_j$ have a parent in $C_i$, for a fixed $0 < f_{min} \leq 1$. The **flood boundary computation** method $\mathbf{bound_{fl}}$ includes in the boundary of $C$ any node reachable from $C$ along a path consisting of edges that are at-most-one and/or strong. Finally, the **acyclic flood boundary computation** method $\mathbf{bound_{fl\text{-}ac}}$ proceeds like $bound_{fl}$, but only along edges that are not in any cycle.

In general, *entity boundaries may overlap*, i.e., a collection $C_s$ may end up within the boundaries of two distinct collections $C'$ and $C''$, e.g., in Fig. 6, name may be within the boundaries of author as well as conference. A non-leaf collection may also be shared together with all its descendants. A shared collection is reported in each of the main entities it belongs to, e.g., description... (the ... denotes nested structure) in Fig. 2. This provides a quick-glance view of each main entity collection, with all its attributes.

## 4.4 Selecting the entity collections to report

We have described alternative ways of determining a *best* collection, methods for assigning a *boundary* to a collection, and suitable (collection score, collection boundary) method pairs.

Now, we show how to put such a combination at work to identify main entities. In a *greedy* fashion, we repeat the steps:

(1) Elect the *best* eligible (recall the criteria early in Sec. 4) node $C_E$ (Sec. 4.1 or 4.2, in blue in Fig. 6) as a *root of a main entity*;
(2) Compute *the boundary* of $C_E$ (Sec. 4.3, shaded areas in Fig. 6);
(3) *Update the collection graph* to reflect the selection of $C_E$ and its boundaries;

until a certain maximum number $E_{max}$ of entities have been selected, or these entities together cover a sufficient fraction $cov_{min}$ of the data (that is: the sum of $ow$ over all the collections within $C_E$'s boundary, is at least $cov_{min}$ times the number of leaves in $G$).

We still need to explain step (3) above: how to reflect, on the collection graph, the selection of a collection $C_E$ as main collection, and the other collections within its boundary, before chosing the next main entity.

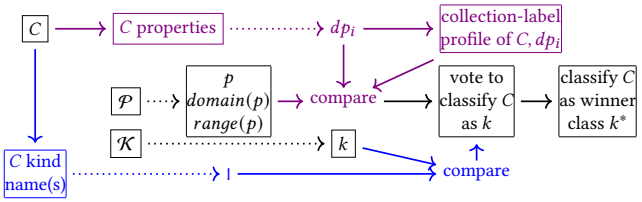


**Figure 7: Outline of the classification algorithm.**

When using a simple baseline (Sec. 4.1), any collection within the boundary of $C_E$, is excluded from all future main entity selections.

When using weights (Sec. 4.2), in contrast, we need to update the graph so as to identify the next main entity with the largest weight *if one excludes all nodes and edges from the previously chosen main collection $C_E$ and its boundary*. In details, we start by marking as *excluded* all nodes in $C_E$; then, in a breadth-first fashion, starting from $C_E$, edges outgoing $C_E$ nodes are disabled, then their target nodes, then their outgoing edges etc., until all nodes and edges within the boundary of $C_E$ have been traversed. Leaf nodes may thus be excluded, which brings their own weight $ow$ to 0, thus reducing $ow$ of some leaf collections within $C_E$'s boundary. For instance, once author is selected with name in its boundary, the $ow$ of name decreases to 2. Then, the scores of all graph collections ($w_{DAG}$, respectively, $w_{PR}$ or $w_{dwPR}$) are recomputed.

The set of all reported main entities is denoted $\mathcal{E}$.

## 4.5 Selecting the relationships between the main entities

Having selected the main entities $\mathcal{E}=\{C^1, \ldots, C^{E_{max}}\}$ and their boundaries, every oriented path in the collection graph that goes from a given $C^i$ to another $C^j$ is reported as a relationship. For instance, in Fig. 6, if the main entities are author (with mail and name in its boundary) and paper (with year, title and abstract in its boundary), the relationships reported are: paper $\xrightarrow{wB}$ author, author $\xrightarrow{hW}$ paper, and paper $\xrightarrow{pIn.conf.inv}$ author.

If the scores lead to reporting three main entities, the two above entities and also conf (with name and inv in its boundary), the relationships are: paper $\xrightarrow{wB}$ author, author $\xrightarrow{hW}$ paper, paper $\xrightarrow{pIn}$ conf, and conf $\xrightarrow{inv}$ author.

The set of all reported relationships is denoted $\mathcal{R}$.

## 4.6 Multi-traversed non-main entities

As shown in Sec. 4.5, a reported relationship may traverse several non-reported collections, e.g., paper $\xrightarrow{pIn.conf.inv}$ author. In some cases, *(non-main) collections are traversed by several reported relations*. Such collections may help to better understand the $\mathcal{R}$ relationships. As a post-processing heuristic, the eligible non-main, multi-traversed collections are promoted to be shown as entities in the abstraction (not just on relationship labels). For example, if two $\mathcal{R}$ relations traverse the (eligible) conf collection, it will be shown as an entity; pIn and inv are not eligible.

# 5 MAIN ENTITY CLASSIFICATION

To each entity collection $C \in \mathcal{E}$, we seek to assign one or a few classes with intuitive names. For that, we analyze: the *kind names* of $C$ nodes; their *data property names*; the *expressive-type values found*

*for these properties* (recall Sec. 2). Fig. 7 outlines our algorithm: solid arrows connect associated data items and trace the classification process, while dotted arrows go from a set to one of its elements.

## 5.1 Semantic resources

We rely on a set of *classes* $\mathcal{K}$, e.g., `Person`, and a set of *properties* $\mathcal{P}$ which may be associated to the classes, in the following sense. For each property $p \in \mathcal{P}$, the set $domain(p) \subseteq \mathcal{K}$ describes classes to which one belongs if one has property $p$; for instance, $domain$(`birthDate`)={`Person`} states that having a `birthdate` means one is a `Person`. Similarly, $range(p)$ states the classes to which belong the values of $p$. For instance, $range$(`birthDate`)={`Date`}.

To populate $\mathcal{K}$ and $\mathcal{P}$, users may specify a few terms of interest to them; we started with the terms `Person`, `Place`, `Organization`, `Creative Work`, `Event` and `Product`. Users may also specify properties for their classes, e.g., they may expect a `Product` to have `price` or `quantity`. We manually identified 101 properties for our six concepts. More significantly, to scale up, we leverage a **knowledge base** (**KB**, in short), a large RDF graph of general-purpose information.

**Anchoring concepts to KB classes** We mapped each of the initial 6 concepts to a class from $B$, the given KB, e.g., `Person` is mapped into `https://dbpedia.org/ontology/person`.

**Acquiring $\mathcal{P}$ properties from a KB** Given a knowledge base $B$ and a class $k \in \mathcal{K}$, we harvest from $B$ the properties $P_k$ that resources of class $k$ may have, as: $P_k = \{r \mid \langle \mathtt{a,r,b} \rangle \in B \wedge \langle \mathtt{a,type,k} \rangle \in B\}$. For example, YAGO4 [41] contains the triples ⟨`Paris, type, Place`⟩, ⟨`Paris, country, France`⟩ and ⟨`Paris, zipcode, 75000`⟩. Thus, we add `country` and `zipcode` to the set $P_{Place}$.

**Acquiring $\mathcal{P}$ properties from GitTables [32]** As a second source of semantic knowledge, we use GitTables [32], a repository of 1.5M tables extracted from GitHub. It aligns each attribute name to properties from DBPedia [3] and/or schema.org, and provides domain and range information for these properties. For instance, for `gender`, GitTables proposes schema:gender, with domain:["schema:Person", "schema:SportsTeam"] and range: ["schema:GenderType", "schema:Text"]. For each GitTables property $p$, we add domain($p$) and range($p$) to $\mathcal{K}$.

## 5.2 Expressive-type collection-label profiles

Next, we *summarize* the expressive-type values of $C$ nodes' properties, e.g., the `author` nodes in Fig. 3 have values of the Person, Location, and Email expressive types (on colored background). Recall (from Sec. 2) $\mathcal{T}$, the set of such expressive types. We call **$\mathcal{T}$-vector** an array of natural numbers indexed by the types in $\mathcal{T}$.

Each leaf collection in our collection graph contains a set of values, each of which can be represented (serialized) in a string. For a string $s$, let $\text{len}_s$ denote its length, $\#_s^{\mathcal{T}}$ be the $\mathcal{T}$-vector count of expressive-type values in $s$, and $\text{len}_s^{\mathcal{T}}$ be the $\mathcal{T}$-vector of the total length of these values. For instance, if $s$ is "France and Germany are part of NATO", $\text{len}_s$ is 34, $\#_s^{\mathcal{T}}$[Location]=2, $\#_s^{\mathcal{T}}$[Organization]=1, and $\#_s^{\mathcal{T}}$ is 0 for the other expressive types, whereas $\text{len}_s^{\mathcal{T}}$[Location]=13, $\text{len}_s^{\mathcal{T}}$[Organization]=4, and $\text{len}_s^{\mathcal{T}}$ is 0 elsewhere. The **expressive types profile** of a string $s$ is the triple $(\text{len}_s, \#_s^{\mathcal{T}}, \text{len}_s^{\mathcal{T}})$.

For a collection $C$ and a label $x$, we denote by $N_{C,x}$ the set of text values of $x$-labeled children of $C$ nodes. For example, if $C$ is the `author` collection in Fig. 6, $N_{C,\,\text{name}}$ contains the nodes {32, 34, 36} (recall the graph in Fig. 3). The **collection-label profile of $C$ and $l$** is the triple $(\Sigma_{s \in N_{C,x}} \text{len}_s,\ \Sigma_{s \in N_{C,x}} (\#_s^{\mathcal{T}}),\ \Sigma_{s \in N_{C,x}} (\text{len}_s^{\mathcal{T}}))$ where the last two are $\mathcal{T}$-vector sums. We denote the first, second and third components of the collection-label profile by $\text{len}_{C,x}$, $\#_{C,x}^{\mathcal{T}}$, $\text{len}_{C,x}^{\mathcal{T}}$, respectively. Continuing with the collection $C$ of `author` nodes, $\text{len}_{C,\text{name}}$ is 9 (length of "Léa", "Eva" and "Bob" combined), $\#_{C,\text{name}}^{\mathcal{T}}$[Person]=3 and $\#_{C,\text{name}}^{\mathcal{T}}$ is 0 in all other positions, while $\text{len}_{C,\text{name}}^{\mathcal{T}}$[Person] is 9 and $\sigma_{C,\text{name}}^{\mathcal{T}}$ is 0 in all the other positions.

## 5.3 Classification algorithm

Recall the algorithm outline in Fig. 7. We attempt to classify $C$ in one or a few classes $k \in \mathcal{K}$, leveraging three different signals: ① the kind names of $C$ nodes (if available), compared with those of $\mathcal{K}$ classes; ② the labels of $C$'s properties, compared with those of $\mathcal{P}$ properties; ③ the expressive values of $C$'s properties, combined with $\mathcal{P}$ range information. When ② and possibly ③ give a strong hint, we combine it with $\mathcal{P}$ domain information.

In details, for ①, if $C$ nodes have one or a few kind names, we compute the Word2Vec [38] similarity $sim(C, k)$ between each of these, and the name of each class $k$ (in blue in Fig. 7).

Next, each data property $dp_i$ outgoing $C$ may "vote" to classify $C$ in one or several classes (violet in the figure), as follows. ② We compute the Word2Vec similarity $sim(dp_i, p)$ between the name of $dp_i$ and that of each $p \in \mathcal{P}$ (e.g. $dp_i$="salary" will match $p$="wage"). ③ We check if *expressive values of a class $k \in range(p)$ make up a large fraction of $dp_i$ values of $C$ nodes*. This is quantified by:

$$sim^{\mathcal{T}}(C, dp_i, p) = \frac{\Sigma_{k \in range(p), \#_{C,dp_i}^{\mathcal{T}}[k]>0} (\text{len}_{C,dp_i}^{\mathcal{T}}[k])}{|range(p)| \cdot \text{len}_{C,dp_i}}$$

If $sim(dp_i, p)$ (averaged with $sim^{\mathcal{T}}(C, dp_i, p)$ when this is not 0) is higher than a threshold $\alpha$, $dp_i$ casts a "vote" to classify $C$ into *each class $k \in domain(p)$*. The weight of this vote is: the percentage of $C$ nodes having property $dp_i$, times the average of $sim(dp_i, p)$ and $sim^{\mathcal{T}}(C, dp_i, p)$, divided by $|domain(p)|$ (to reward properties with few domain that type $C$ most precisely).

As classification result (at right in black in Fig. 7), we pick the class(es) $k^*$ with the highest value for $sim(C, k)$ or the supports in ② or in ③, if this value is above a threshold $\theta$, otherwise, as *Thing*. Regardless of the classification success, as an extra help to users, our E-R diagram also shows the *kind name* (Sec. 2) most popular among the nodes in $C$, if these nodes do have kind names. For instance, in Fig. 2, XML element names are shown.

# 6 EVALUATION

Abstra is implemented in Java, leveraging the graph creation (including entity extraction) and Postgres-based store of ConnectionLens [2]. We experimented on a Linux server with an Intel Xeon Gold 5218 CPU @ 2.30GHz and 196GB of RAM, and relied on PostgreSQL v9.6 for storing the data.

We focus our evaluation on **JSON**, **PG**, **RDF** and **XML** datasets, given their popularity, and because their complexity make their abstraction more challenging (Sec. 6.1). The questions we study are: (*i*) *what is the best-performing main entity selection method?* (Sec. 6.2), (*ii*) *does this method succeed in identifying the main entities, their boundaries, and the main relationships in a dataset?* (Sec. 6.3), (*iii*) *how well does the classification algorithm perform?* (Sec. 6.4), (*iv*) *how efficiently can abstractions be computed?* (Sec. 6.5) and (*v*) *how do they compare to inferred schemas?* (Sec. 6.6).

## 6.1 Datasets, semantic resources, and settings

We evaluate Abstra on JSON, PG, RDF and XML synthetic and real-life, open datasets, as follows. Tab. 1 shows, for each dataset, the number of nodes and edges in $G_0$ (the graph obtained by loading the dataset) and the normalized graph $G$; $\diamond$ indicates synthetic datasets (as opposed to real-world). Since each RDF edge is labeled, normalization (Sec. 3.1) adds many extra nodes and edges.

For **JSON**, we wrote the Researchers dataset generator. Each researcher has a first and last name, id, H-index, status, gender, age, date of birth, and continent. It also includes: the titles of their three best papers, and a list of five co-authors (each with a first name and a last name). Real-life JSON datasets comprise a dataset of commits and push events in GitHub repositories [54], Prescriptions [57] (medical prescriptions), NYTimes articles [5], three datasets from Yelp [60], a crowd-sourced app to review businesses, and bibliographic notices from CoreResearch [51].

For **PG** datasets, we used two datasets generated based on the LDBC benchmark [21]: LDBCsmall, used in [12], and LDBC0.3 obtained using the generator. We also wrote a custom generator to obtain Movies250K. Each movie has a title, a synopsis, a duration, a year and a number of entries. Some movies are also typed as sequel. Movie directors have a first and a last name while actors also have a nationality. Actors and movie directors are also both typed as Person in addition to their own type. Cinemas have a name and a city. Awards have a label, a year and a boolean indicating whether it is for international nominations. Each movie is projected in some cinemas at a specific date, and some of them received awards. Each actor has played in some movie(s); for each acting, a role is associated. Finally, movie directors have supervised the production of one or several movies.

For **RDF**, we used BSBM [10] and LUBM [29] benchmarks, as well as our generator of graphs about scientific papers (having a title, a DOI and a year) written by authors (with first name, last name, affiliation university, birthdate, gender, mail and honorific prefix) and published in conferences (with a place, a year, an organizer, and a duration). Authors are also invited in conferences, thus leading to cycles. We used real-life datasets about: gas and electricity in Italy (EnelShops [52]), food recipes (Foodista [53]) and NASA flights [56].

For **XML**, we used XMark [45] documents, as well as the real-life PubMed [58], Mondial [55] and WikiMedia [59] datasets.

As **knowledge bases** (Sec. 5), we used YAGO4 [41] and WikiData, which lead to 565 $\mathcal{P}$ properties (on top of the 101 we had manually identified). From GitTables [32], we derived 4.187 $\mathcal{P}$ properties; 3.687 (respectively, 3.898) among them have domain (respectively, range) statements, involving a total of 810 classes.

**Settings** Unless otherwise specified, we used $E_{max}$=5 (select at most 5 main entities), $cov_{min}$=1.0 (Sec. 4.4), $\alpha$=0.8, $\theta$=0.3 (Sec. 5), which worked best in our experiments.

## 6.2 Quality of the main entity selection methods

We assessed through a user study, the quality of the main entities $\mathcal{E}$ and relationships $\mathcal{R}$ selected as described in Sec. 4. For this task, we built four **dataset samples**, one for each data model, as subsets of datasets in Tab. 1. The samples are small enough (Tab. 2) for users to inspect them to decide if our abstractions are relevant, yet sufficiently complex to test our algorithms.

| Dataset name | $\lvert N_0\rvert$ | $\lvert E_0\rvert$ | $\lvert N\rvert$ | $\lvert E\rvert$ |
|---|---|---|---|---|
| **JSON** | | | | |
| CoreResearch | 2,112,023 | 2,112,022 | 2,112,025 | 2,112,024 |
| GitHub | 5,094 | 5,093 | 5,096 | 5,095 |
| NYTimes | 1,053,979 | 1,053,978 | 1,053,981 | 1,053,980 |
| Prescriptions | 14,214,033 | 14,214,032 | 14,214,035 | 14,214,034 |
| Researchers $\diamond$ | 4,505,657 | 4,505,656 | 4,505,659 | 4,505,658 |
| YelpBusiness | 8,111,982 | 8,111,981 | 8,111,984 | 8,111,983 |
| YelpCheckIn | 659,653 | 659,652 | 659,655 | 659,654 |
| YelpTips | 9,998,068 | 9,998,067 | 9,998,070 | 9,998,069 |
| **PG** | | | | |
| LDBCsmall $\diamond$ | 9,851 | 43,011 | 48,621 | 77,546 |
| LDBC0.3 $\diamond$ | 2,306,540 | 10,907,534 | 11,890,404 | 19,167,734 |
| Movies250K $\diamond$ | 12,386,250 | 14,817,871 | 23,140,245 | 21,507,996 |
| **RDF** | | | | |
| BSBM4M $\diamond$ | 1,134,834 | 4,000,740 | 4,755,392 | 7,241,116 |
| BSBM16M $\diamond$ | 5,344,748 | 16,038,464 | 19,870,744 | 29,051,992 |
| Conferences $\diamond$ | 121 | 184 | 273 | 304 |
| EnelShops | 14,694 | 51,639 | 56,426 | 83,464 |
| Foodista | 190,271 | 1,019,801 | 1,162,086 | 1,943,630 |
| LUBM1M $\diamond$ | 248,261 | 1,035,610 | 1,076,519 | 1,656,516 |
| NASA | 53,528 | 99,423 | 140,540 | 174,024 |
| **XML** | | | | |
| Mondial | 129,567 | 135,588 | 130,756 | 136,777 |
| PubMed | 49,035 | 49,034 | 49,037 | 49,036 |
| XMark1 $\diamond$ | 3,392,392 | 3,392,391 | 3,392,394 | 3,392,393 |
| XMark4 $\diamond$ | 13,615,551 | 13,615,550 | 13,615,553 | 13,615,552 |
| Wikimedia | 1,824,185 | 1,768,803 | 1,872,877 | 1,817,495 |

**Table 1: Datasets used in the evaluation.**

| Dataset name | $\lvert N_0\rvert$ | $\lvert E_0\rvert$ | $\lvert N\rvert$ | $\lvert E\rvert$ | $\lvert C\rvert$ |
|---|---|---|---|---|---|
| Researchers (JSON) | 569 | 665 | 684 | 780 | 26 |
| LDBC (PG) | 445 | 791 | 1082 | 1280 | 78 |
| Conferences (RDF) | 145 | 256 | 369 | 448 | 32 |
| XMark (XML) | 847 | 863 | 904 | 920 | 126 |

**Table 2: Datasets used in the user relevance feedback.**

The **JSON sample** is extracted from the Researchers dataset. The **PG sample** is extracted from LDBCsmall. It contains forums, in which posts and comments are posted by people who study or work at an organization. Each forum, post or comment may have some tags. The **RDF sample** is a subset of Conferences. The **XML sample** is extracted from XMark1. It contains items to be sold in open auctions, people with nested addresses, some of which are interested in the open auctions; there are also closed auctions. Items for sale have nested descriptions and the categories they belong to. There is also a category hierarchy. As in larger datasets, items are in different continents, materialized by ⟨africa⟩ and ⟨asia⟩ elements.

We abstracted each sample using **11 methods** (first column of Tab. 3), each of which pairs a way to select the next main collection (Sec. 4.1, 4.2) with a method for determining its boundary (Sec. 4.3). We showed the abstractions as E-R diagrams, where they could click to unfold nested attributes, to a group of **16 evaluators** (graduate and post-graduate researchers) and asked them to rank them, for each sample, based on **what a good abstraction is**: *"Does it capture all important entities? Are there spurious entities?"; "Are the relationships informative? Do they help understand the entities?"; "Do each entity's attributes logically belong there? Are any attributes*

| Abstraction method | JSON | | PG | | RDF | | XML | | Total | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1st | top3 | 1st | top3 | 1st | top3 | 1st | top3 | 1st | top3 |
| (1-desc, $\text{bound}_{\text{desc}}$) | 2 | 8 | **8** | 14 | **16** | **16** | 2 | 5 | 28 | 43 |
| (2-desc, $\text{bound}_{\text{desc}}$) | 1 | **15** | 0 | 8 | 0 | 8 | 2 | 10 | 3 | 41 |
| (3-desc, $\text{bound}_{\text{desc}}$) | **11** | **15** | 1 | 13 | 0 | 15 | 0 | 5 | 12 | 48 |
| (2-leaf, $\text{bound}_{\text{leaf}}$) | 1 | 9 | **8** | 14 | **16** | **16** | 5 | 7 | **30** | 46 |
| (3-leaf, $\text{bound}_{\text{leaf}}$) | 1 | 3 | **8** | 14 | **16** | **16** | 0 | 8 | 25 | 41 |
| ($\text{w}_{\text{DAG}}$, $\text{bound}_{\text{DAG}}$) | **11** | **15** | **8** | **16** | 0 | 12 | 4 | 7 | 23 | 50 |
| ($\text{w}_{\text{PR}}$, $\text{bound}_{\text{fl}}$) | **11** | **15** | **8** | **16** | 0 | 1 | 6 | 11 | 25 | 43 |
| ($\text{w}_{\text{PR}}$, $\text{bound}_{\text{fl-ac}}$) | **11** | **15** | **8** | **16** | 0 | 12 | **7** | **13** | 26 | **56** |
| ($\text{w}_{d\,w\text{PR}}$, $\text{bound}_{\text{fl}}$) | **11** | **15** | **8** | **16** | 0 | 1 | 6 | 11 | 25 | 43 |
| ($\text{w}_{d\,w\text{PR}}$, $\text{bound}_{\text{fl-ac}}$) | **11** | **15** | **8** | **16** | 0 | 12 | **7** | **13** | 26 | **56** |

**Table 3: Users' ranking of dataset sample abstractions. Results are at https://team.inria.fr/cedar/projects/abstra/.**

*missing?"*. For each abstraction method and sample dataset, Tab. 3 reports the number of times that the method was ranked first, respectively in the top-3 methods. The (leaf-1, $\text{bound}_{\text{leaf}}$) is omitted, since it always leads to an empty abstraction: the only collections with a non-zero score (Sec. 4.1) are not eligible (Sec. 4). We show in bold the best-performing method(s) for each dataset.

On JSON, our weight-based methods perform (much) better than the baselines, except (3-desc, $\text{bound}_{\text{desc}}$) which coincides with the weighted methods. All weight-based abstractions of this dataset led to the same result. In the JSON sample abstraction, we fused the "co-authors" collection with the one of researchers because they both have similar properties (recall Sec. 3.3). On the PG sample, again all weight-based methods lead to the same result, and are ranked best; some simple baselines come close. They compute similar results as the weighted ones, but they either miss the comment entity (even though important) or report the organization entity, which can be nested into the person entity. On the RDF sample, users prefer some simple baselines to the weight-based methods. This is because all weight-based methods included university within the person entity, given that university only appears there; in contrast, users preferred to see university as a separate entity. The XML sample, with the most complex structure, lead to more diverse evaluations. ($\text{w}_{\text{PR}}$, $\text{bound}_{\text{fl-ac}}$) and ($\text{w}_{d\,w\text{PR}}$, $\text{bound}_{\text{fl-ac}}$) win, followed by ($\text{w}_{\text{PR}}$, $\text{bound}_{\text{fl}}$) and ($\text{w}_{d\,w\text{PR}}$, $\text{bound}_{\text{fl}}$). Users downgraded baseline methods, which fail to reflect the complex XMark structure.

In the Totals, (2-leaf, $\text{bound}_{\text{leaf}}$) is ranked 1st most often: this method only chooses tuple-like entities, with named properties having atomic values. The samples used here were simple, to ease user evaluation. In real-life datasets with dozens or hundreds of collections (see Tab. 4), (2-leaf, $\text{bound}_{\text{leaf}}$) can only return $E_{max}$ such entities, which degrades its coverage. The methods ($\text{w}_{\text{PR}}$, $\text{bound}_{\text{fl-ac}}$) and ($\text{w}_{d\,w\text{PR}}$, $\text{bound}_{\text{fl-ac}}$) are close behind in the 1st place ranking, and they both score best in the top-3 ranking. Based on this, and the remark that only ($\text{w}_{d\,w\text{PR}}$, $\text{bound}_{\text{fl-ac}}$) leverages both complex graph structure and data cardinalities, from now on, we focus on the **($\text{w}_{d\,w\text{PR}}$, $\text{bound}_{\text{fl-ac}}$) abstraction method, which performed best**, as it fulfils our main objectives (Sec. 1).

## 6.3 Main entity collections in all datasets

We now study the main entities $\mathcal{E}$ and relationships $\mathcal{R}$ identified by the ($\text{w}_{d\,w\text{PR}}$, $\text{bound}_{\text{fl-ac}}$) abstraction method, in all our datasets (Tab. 4). A ⟳ indicates that the dataset's collection graph has cycles (14 datasets out of 23). For each main entity, we also report a human-readable label (not yet our classification - see Sec. 6.4) obtained as follows. When the nodes in the collection share an explicit kind

| Dataset name | $\|C\|,\|\mathcal{E}\|,\|\mathcal{R}\|$ | $cov$ | $\mathcal{E}$ entities (max depth / number of nodes) |
|---|---|---|---|
| JSON | | | |
| CoreResearch ⟳ | 48, 1, 0 | 1.0 | *Notice* (5 / 39,767) |
| GitHub ⟳ | 343, 1, 0 | 1.0 | *Repository* (8 / 30) |
| NYTimes | 116, 1, 0 | 1.0 | *Document* (9 / 4,482) |
| Prescriptions | 4814, 1, 0 | 1.0 | *Prescription* (3 / 239,930) |
| Researchers ⟳ | 25, 1, 1 | 1.0 | *Researcher* (5 / 38,090) |
| YelpBusiness | 122, 1, 0 | 1.0 | *Business* (4 / 150,346) |
| YelpCheckIn | 6, 1, 0 | 1.0 | *Check-in* (3 / 131,930) |
| YelpTips | 12, 1, 0 | 1.0 | *Tip* (3 / 908,915) |
| PG | | | |
| LDBCsmall ⟳ | 61, 4, 9 | 1.0 | Post (6 / 3,189); Comment (6 / 471); Forum (9 / 381); Person (6 / 50) |
| LDBC0.3 ⟳ | 82, 6, 16 | 1.0 | Comment (9 / 523,222); Post (6 / 324,825); Forum (16 / 31,097); Tag (4 / 16,080); Organization (6 / 7,955); Person (13 / 3,514) |
| Movies250K ⟳ | 38, 4, 3 | 1.0 | playedIn (9 / 1,758,142); Actor (12 / 1,099,489); Movie (8 / 250,000); Director (10 / 124,818) |
| RDF | | | |
| BSBM4M ⟳ | 340, 6, 7 | 1.0 | Offer (3 / 226,800); Review (5 / 113,400); *Product* (5 / 11,340); ProductFeature (3 / 10,519); Producer (3 / 232) |
| BSBM16M ⟳ | 724, 6, 7 | 1.0 | Review (3 / 1,324,146); Offer (3 / 914,000); Person (3 / 134,570); *Product* (5 / 45,700); Producer (3 / 921); Vendor (3 / 464) |
| Conferences ⟳ | 29, 2, 2 | 0.83 | Author (5 / 20); Paper (3 / 10) |
| EnelShops | 46, 1, 0 | 1.0 | *Shop* (6 / 1,136) |
| Foodista ⟳ | 49, 4, 20 | 0.47 | Recipe (5 / 32,782); Food (3 / 7,651); Tool (3 / 150); PreparationMethod (3 / 149) |
| LUBM1M | 108, 5, 3 | 1.0 | Publication (11 / 60,342); UndergraduateStudent (5 / 59,437); GraduateStudent (6 / 9,259); ResearchAssistant (6 / 5,454); TeachingAssistant (6 / 4,156) |
| Nasa ⟳ | 61, 5, 13 | 0.95 | Spacecraft (5 / 6,692); Launch (5 / 5,090); Image (3 / 303); MissionRole (3 / 142); Person (3 / 59) |
| XML | | | |
| Mondial ⟳ | 168, 5, 8 | 0.85 | City (3 / 3,152); Province (3 / 1,455); Country (4 / 231); Organization (4 / 168); River (4 / 135) |
| PubMed | 26, 1, 0 | 1.0 | PubMedArticle (5 / 957) |
| XMark1 ⟳ | 136, 5, 10 | 0.91 | Person (4 / 25,500); Item (7 / 21,750); Open_Auction (8 / 12,000); Closed_Auction (8 / 9,750); Category (2 / 1,000) |
| XMark4 ⟳ | 136, 5, 10 | 0.90 | Person (4 / 102,000); Item (7 / 87,000); Open_Auction (8 / 48,000); Closed_Auction (8 / 39,000); Category (2 / 4,000) |
| Wikimedia | 59, 2, 0 | 1.0 | Page (4 / 54,750); Namespace (3 / 32) |

**Table 4: Main entities found in the application datasets.**

name, e.g., Post in LDBC data, we show it in normal font. Otherwise, we provide ourselves a label, shown in *italic*, e.g., *Notice* in CoreResearch. After each collection label, we show (within parenthesis) *the maximal depth, in the data*, of an entity from that collection and the number of nodes in the collection. Our JSON datasets lead to one entity and no relationships. This is because (*i*) unlike the sample used in Sec. 3, they feature no shared entities; (*ii*) they don't use references in the style of XML ID-IDREF. In BSBM4M, the collection Product is multi-traversed (recall Sec. 4.6) by the relationships Review.reviewFor.Product.productFeature.ProductFeature and Offer.product.Product.productFeature.ProductFeature.

Tab. 4 shows that **from each dataset, the selected entities are frequent, coherent, and semantically central**: this confirms that our approach, based on the collection graph and the main entity selection method ($\text{w}_{d\,w\text{PR}}$, $\text{bound}_{\text{fl-ac}}$), attain their goal. *The main entity depth varies from 2 (XMark) to 11 (LUBM1M)*, with 3 and 5 being frequent values. This shows that ($\text{w}_{d\,w\text{PR}}$, $\text{bound}_{\text{fl-ac}}$)

| Dataset | Collection (Class, relevance) |
|---|---|
| JSON | |
| CoreResearch | *Notice* (CreativeWork, **H**) |
| GitHub | *Repository* (Thing, L) |
| NYTimes | *Document* (CreativeWork, **H**) |
| Prescriptions | *Prescription* (CreativeWork, M) |
| Researchers | *Researcher* (Person, **H**) |
| YelpBusiness | *Business* (Thing, L) |
| YelpCheckIn | *Check-in* (Thing, L) |
| YelpTips | *Tip* (Thing, L) |
| PG | |
| LDBCsmall | Post (Thing, L); Comment (Comment, **H**); Forum (Blog, **H**); Person (Person, **H**) |
| LDBC0.3 | Comment (Comment, **H**); Post (Thing, L); Forum (Blog, **H**); Tag (Thing, L); Organisation (Organisation, **H**); Person (Person, **H**) |
| Movies250K | playedIn (Play, M); Actor (Actor, **H**); Movie (Movie, **H**); Director (Person, **H**) |
| RDF | |
| BSBM4M | Offer (Offer, **H**); Review (Review, **H**); *Product* (Product, **H**); ProductFeature (Product, **H**); Producer (Thing, L) |
| BSBM16M | Review (Review, **H**); Offer (Offer, **H**); Person (Person, **H**); *Product* (Product, **H**); Producer (Thing, L); Vendor (Thing, L) |
| Conferences | Author (Person, **H**); Paper (CreativeWork, **H**) |
| EnelShops | *Shop* (Restaurant, M) |
| Foodista | Recipe (Recipe, **H**); Food (Food, **H**); Tool (Thing, L); PreparationMethod (Thing, L) |
| LUBM1M | Publication (CreativeWork, **H**); UndergraduateStudent (Person, **H**); GraduateStudent (Person, **H**); ResearchAssistant (Person, **H**); TeachingAssistant (Person, **H**) |
| Nasa | Spacecraft (Spacecraft, **H**); Launch (LaunchPad, **H**); Image (ImageObject, **H**); MissionRole (SpaceMission, **H**); Person (Person, **H**) |
| XML | |
| Mondial | City (City, **H**); Province (Province, **H**), Country (Country,**H**); Organization (Organization,**H**); River (River, **H**) |
| PubMed | PubMedArticle (CreativeWork, **H**) |
| XMark1 | Person (Person, **H**); Item (Product, **H**); Open Auction (Product, M); Closed Auction (Product, M); Category (Thing, L) |
| XMark4 | Person (Person, **H**); Item (Product, **H**); Open Auction (Product, M); Closed Auction (Product, M); Category (Thing, L) |
| Wikimedia | Page (Thing, L); Namespace (Thing, L) |

**Table 5: Quality of $\mathcal{E}$ collection classification.**

**identifies nested entities of various depths**, which fixed-depth baseline methods (Sec. 4.1) cannot do. Finally, the node coverage (*cov* in Tab. 4) shows that, in general, **our abstractions represent most of the dataset**, thus fulfilling our objective (recall Sec. 1).

## 6.4 Quality of main entity classification

We now analyze the quality of the semantic classes assigned to main entities through classification (Sec. 5). We inspected sample entity instances, and ranked the assigned class' relevance **high (H)**, **medium (M)** or **low (L)**. We graded H if the class describes the entities well, e.g., a `Person` or `Author` class for authors, `CreativeWork` or `Publication` for articles, etc. We graded M acceptable (if suboptimal) classes, and L any clear misclassification, e.g., `Place` instead of `Person`, as well as `Thing` (not enough insight to classify). Tab. 5 shows for each dataset and collection, the assigned class, and the relevance values. As in Tab. 4, names in *italics* are those we manually chose for collections that lack a kind name.

Out of 68 collections, 76.4% (52) obtained an informative class through classification: 86.5% (45) are rated H, and 13.5% (7) are rated M. In contrast, 23.6% (16) collections were classified as Thing, among which: 12 have informative kind names, and 4 (the JSON ones) do not. In all L-rated results, the linguistic and semantic signal from entities is below the algorithm's thresholds (Sec. 5). This happens for two reasons: an entity has very few properties with not-so-informative names, e.g., Yelp check-ins have only `user_id` and `date`; most (or all) of the entity's properties have no equivalent in $\mathcal{P}$, e.g., GitHub properties such as `allow_forking` and `ssh_url`, etc.

A classification examples illustrates the combined effect of property matching (step ② in Sec. 5.3) and expressive types (step ③). The *Notice* entity has 20 properties, such as publisher, downloadUrl and doi. Values of the publisher property include the Organization and Person expressive types. Classification identifies three $\mathcal{P}$ properties matching the data property publisher: editor, illustrator and publisher, which vote towards the following classes: Creative work, Written work, Work and Book. However, publisher adds more support for Creative work, since the expressive-type collection-label profile for *Notice* and publisher (Sec. 5.2) matches the publisher data property range, raising the $sim^{\mathcal{T}}(C, dp_i, p)$ value in favor of Creative work, which is best in this case. In other cases, the classification is due to $sim(dp_i, p)$ and $\mathcal{P}$ domain typing, e.g., for PubMedArticle, the properties PubMedLink, DOI and KeywordList all voted towards Creative work; for XMark items, seller voted towards the Offer, Order, BuyAction, Demand, Flight and Product classes, however, quantity only voted for Product and helped it be selected.

In conclusion, **classification was overall successful**, leveraging the property names, expressive types encountered in their values, and the background information ($\mathcal{K}$ and $\mathcal{P}$). In general, classification quality depends on the overlap between the dataset vocabulary, and the background information: class types may not be found for very specific entities, e.g., GitHub. Even when the classification is not very precise, e.g., Creative work for (medical) Prescriptions, it is still useful (in particular in this example, where the collection has no kind name to guide the users).

## 6.5 Scalability of the abstraction computation

In this section, we study the performance of graph abstraction as a function of the input data size.

We measure the execution time of the all steps involved in the abstraction computation method using ($\mathrm{w}_{dwPR}$, $\mathrm{bound}_{\text{fl-ac}}$). To study this, we use four of our synthetic, controlled-size datasets: Researchers (JSON), Movies (PG), BSBM (RDF), and XMark (XML), and show the results in Fig. 8; all axes are logarithmic. Identifying the main collections takes most time; this is due to the cost of updating the graph (excluding nodes as described in Sec. 4.4) before recomputing weights. Collection identification is faster on the JSON dataset as there are few entities selected, thus few graph updates. The time taken to identify relationships (Sec. 4.5) is negligible (thus not shown). This is because this task only manipulates the collection graph, orders of magnitude smaller than the graph.

**All our abstraction methods scale up linearly in the data size.** The methods using $\mathrm{bound}_{\text{desc}}$ and $\mathrm{bound}_{\text{leaf}}$ (not plotted to

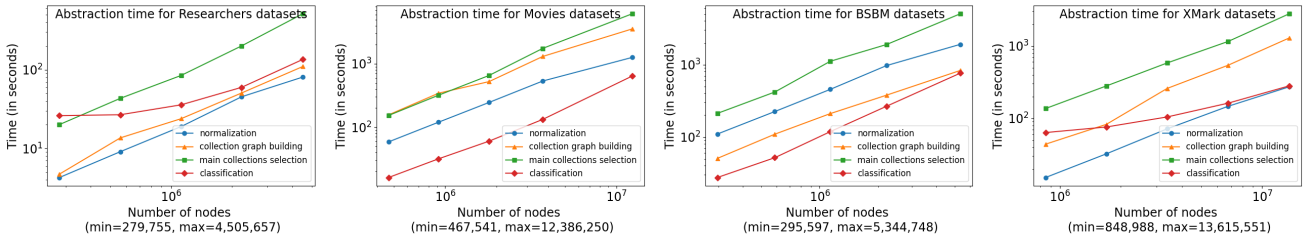


**Figure 8: Abstraction computation times on synthetic JSON, PG, RDF and XML datasets, using ($w_{dwPR}$, $bound_{fl\text{-}ac}$).**

avoid clutter) are faster, since they do not recompute weights. However, as shown in Sec. 6.2, their results are of lower quality.

## 6.6 Inferred schemas vs. abstractions

Abstractions have different goals and uses than dataset schemas. Schemas define a notion of data *validity*, and are *used by code* that processes the data; abstractions aim to give *human users a simple, first glance* at the data. Schemas have *technical features* such as inheritance, property cardinalities, etc. that abstraction voluntarily leave out. They contain interconnected types, while abstraction *wrap some collections into the main collection boundaries*, to facilitate understanding. At the same time, abstractions share with schemas, in particular those *inferred from the data*, e.g., [4, 5, 12, 13, 43, 47], the goal to compactly describe a dataset.

Keeping this in mind, we tested recent schema extractors for: JSON [4, 47], PGs [12, 13], and RDF graphs, respectively [43]. We report results on datasets from Tab. 1. For JSON, on the CoreResearch, Prescriptions, YelpTips, YelpBusiness and YelpCheckIn datasets, respectively, [4] produces schemas of 128, 7,221, 191, 18 and 12 nodes, while [47] leads to schemas of 146, 4,218, 144, 21 and 12 nodes. The schema itself is a JSON document, and we count its nodes as a measure of its complexity. Clearly, schemas of more than hundreds of nodes are unsuited as abstractions. For PGs, on the LDBC dataset, [12] creates a schema of 10 types connected by 22 edges. While compact, it contains several nodes for the same concept, e.g., three nodes labeled "Post", but with slightly different properties. Our abstractions are more compact (Tab. 4), and prefer node kinds over structural precision. For RDF, the LUBM schema [43] features 23 node shapes (types), one for each RDF type, and 187 property shapes (an RDF property such as worksFor leads to 9 property shapes: worksForProfessor, worksForChair, etc.). Such precision leads to many syntactic details, going against our need for clarity.

These results confirm that **compact abstractions are needed to give human users a first idea of a dataset**. They helpfully **complement schemas**, whose objectives are different.

## 6.7 Experiment conclusion

The abstraction method ($w_{dwPR}$, $bound_{fl\text{-}ac}$) attains the best results overall, even on complex, cyclic collection graphs. This method successfully identifies the central dataset entities, when their structure ranges from simple (depth 2 in Tab. 4) to very complex (depth 11). Classification, with record kind names, property names, expressive types in their values, and semantic resources, is overall successful identifying what each main entity is about. Abstra sets parameters' default values to $w_{dwPR}$, $bound_{fl\text{-}ac}$ and thresholds in Sec. 6.1. Our abstractions are computed in linear time in the size of the input. Intuitive, and more compact than (inferred) schemas due to their focus on structured entities (as opposed to node types), they provide useful first-glance dataset summaries.

# 7 RELATED WORK AND CONCLUSION

**Data summarization** builds concise, structured summaries from much larger (semi)structured datasets. Dozens of summarization methods exist; recent surveys include [17, 37]. As explained in Sec. 3, we picked the Typed Strong summary [25] for RDF since in accordance with our principle (★) it leverages RDF types (kinds) when available, and relies on property cliques which tolerate some heterogeneity among properties of equivalent nodes, keeping summaries compact. Beyond [25], we abstract many non-RDF data formats, and propose novel methods for detecting complex entities.

Related to summarization is the problem of **schema inference from the data** [34], most recently studied for JSON [4, 5, 36, 47] and for graphs [12, 13, 28, 43]. As discussed in Sec. 6.6, schemas and abstractions have different goals; schemas are more detailed and technical, thus, less suited for first-time dataset users.

Several works **recommend a relational schema for a semi-structured dataset**, e.g. [11, 14, 20, 46]. Each of the resulting relations can be seen as a (flat) entity. However, they do not meet our goals: (*i*) multiple-valued attributes are always separated from their parents, leading to many relations; (*ii*) to improve performance for a given workload, records of the same kind may be stored separately, making data harder to understand by users.

Entity classification (Sec. 5) is related to works seeking to infer column and table names, as well as column types, to **find which tables can be unioned or joined**, e.g., [1, 22, 32, 35]. In contrast, we also abstract **semistructured** data, to help understand it; we do not aim for schema compatibility between datasets.

Works such as [8, 30, 40, 42] aim to help users understand heterogeneous datasets stored in **data lakes**. [8] only handles relational and NoSQL (JSON-style) databases. [30] builds and annotates schemas with semantic information for tree-structured data, leveraging [42]. They only return tree models for semi-structured data sources, while our abstractions/E-R diagrams may be (cyclic) graphs; also, we identify entity boundaries. [40] blends keyword search and navigation to help explore a relational datalake; abstraction can be seen as a complementary task.

In follow-up work, inspired by data journalism applications, we study dataset exploration along paths connecting expressive-type value collections, in the collection graph [7].

**Acknowledgments.** This work is funded by DIM RFSI PHD 2020-01 and AI Chair SourcesSay project (ANR-20-CHIA-0015-01) grants.

# 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] S. Auer et al. DBpedia: A nucleus for a web of open data. In *The Semantic Web*, pages 722–735, Berlin, Heidelberg, 2007.

[4] 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.

[5] M. A. Baazizi, D. Colazzo, G. Ghelli, and C. Sartiani. Parametric schema inference for massive JSON datasets. *VLDB J.*, 28(4), 2019.

[6] N. Barret, I. Manolescu, and P. Upadhyay. Abstra: toward generic abstractions for data of any model (demonstration). In *CIKM*, 2022.

[7] N. Barrret, A. Gauquier, J. J. Law, and I. Manolescu. PathWays: entity-focused exploration of heterogeneous data graphs (demonstration). In *ESWC*, 2023.

[8] A. Beheshti, B. Benatallah, R. Nouri, and A. Tabebordbar. CoreKG: A knowledge lake service. *Proc. VLDB Endow.*, 11(12):1942–1945, aug 2018.

[9] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, 2002.

[10] C. Bizer and A. Schultz. The Berlin SPARQL benchmark. *IJSWIS*, 5(2):1–24, 2009.

[11] P. Bohannon, J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Siméon. Bridging the XML relational divide with LegoDB. In *ICDE*, pages 759–761, 2003.

[12] A. Bonifati, S. Dumbrava, E. Martinez, F. Ghasemi, M. Jaffré, P. Luton, and T. Pickles. DiscoPG: Property graph schema discovery and exploration. *PVLDB*, 15(12), 2022.

[13] A. Bonifati, S. Dumbrava, and N. Mir. Hierarchical clustering for property graph schema discovery. In *EDBT*, 2022.

[14] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee. Building an efficient RDF store over a relational database. In *SIGMOD*, pages 121–132, 2013.

[15] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Networks*, 30(1-7):107–117, 1998.

[16] S. Campinas, R. Delbru, and G. Tummarello. Efficiency and precision trade-offs in graph summary algorithms. In B. C. Desai, J. L. Larriba-Pey, and J. Bernardino, editors, *IDEAS*. ACM, 2013.

[17] 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.

[18] W. Chen, F. Guo, D. Han, J. Pan, X. Nie, J. Xia, and X. Zhang. Structure-based suggestive exploration: A new approach for effective exploration of large networks. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):555–565, 2019.

[19] D. Colazzo, G. Ghelli, and C. Sartiani. Schemas for safe and efficient XML processing. In *ICDE*. IEEE Computer Society, 2011.

[20] A. Deutsch, M. F. Fernández, and D. Suciu. Storing semistructured data with STORED. In *SIGMOD*, pages 431–442, 1999.

[21] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz. The LDBC social network benchmark: Interactive workload. In *SIGMOD*, 2015.

[22] G. Fan, J. Wang, Y. Li, D. Zhang, and R. Miller. Semantics-aware dataset discovery from data lakes with contextualized column-based representation learning. https://arxiv.org/abs/2210.01922, 2022.

[23] K. Fu, T. Mao, Y. Wang, D. Lin, Y. Zhang, J. Zhan, X. Sun, and F. Li. TS-Extractor: large graph exploration via subgraph extraction based on topological and semantic information. *Journal of Visualization*, 24, 2021.

[24] F. Goasdoué, P. Guzewicz, and I. Manolescu. Incremental structural summarization of RDF graphs. In *EDBT*, Lisbon, Portugal, Mar. 2019.

[25] F. Goasdoué, P. Guzewicz, and I. Manolescu. RDF graph summarization for first-sight structure discovery. *The VLDB Journal*, 29(5), Apr. 2020.

[26] F. Goasdoué, I. Manolescu, and A. Roatis. Efficient query answering against dynamic RDF databases. In G. Guerrini and N. W. Paton, editors, *EDBT*, pages 299–310. ACM, 2013.

[27] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, 1997.

[28] B. Groz, A. Lemay, S. Staworko, and P. Wieczorek. Inference of shape graphs for graph databases. In D. Olteanu and N. Vortmeier, editors, *ICDT*, volume 220, pages 14:1–14:20, 2022.

[29] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 3(2-3):158–182, 2005.

[30] R. Hai, S. Geisler, and C. Quix. Constance: An intelligent data lake system. In *SIGMOD*, SIGMOD '16, page 2097–2100, New York, NY, USA, 2016.

[31] J. Han, M. Kamber, and J. Pei. *Data mining concepts and techniques, third edition*. 2012.

[32] M. Hulsebos, Ç. Demiralp, and P. Groth. Gittables: A large-scale corpus of relational tables. *CoRR*, abs/2106.07258, 2021.

[33] L. Jiang and F. Naumann. Holistic primary key and foreign key detection. *J. Intell. Inf. Syst.*, 54(3):439–461, 2020.

[34] K. Kellou-Menouer, N. Kardoulakis, G. Troullinou, Z. Kedad, D. Plexousakis, and H. Kondylakis. A survey on semantic schema discovery. *The VLDB Journal*, 2021.

[35] A. Khatiwada, G. Fan, R. Shraga, Z. Chen, W. Gatterbauer, R. J. Miller, and M. Riedewald. SANTOS: Relationship-based semantic table union search. https://arxiv.org/abs/2209.13589, 2022.

[36] H. Lbath, A. Bonifati, and R. Harmer. Schema inference for property graphs. In *EDBT*, 2021.

[37] Y. Liu, T. Safavi, A. Dighe, and D. Koutra. Graph summarization methods and applications: A survey. *ACM Comput. Surv.*, 51(3), June 2018.

[38] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, 2013.

[39] T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In S. Abiteboul, K. Böhm, C. Koch, and K. Tan, editors, *ICDE*. IEEE Computer Society, 2011.

[40] P. Ouellette, A. Sciortino, F. Nargesian, B. G. Bashardoost, E. Zhu, K. Pu, and R. J. Miller. RONIN: data lake exploration. *Proc. VLDB Endow.*, 14(12):2863–2866, 2021.

[41] T. Pellissier Tanon, G. Weikum, and F. Suchanek. Yago 4: A reason-able knowledge base. In *ESWC*, 2020.

[42] C. Quix, R. Hai, and I. Vatov. Metadata extraction and management in data lakes with GEMMS. *Complex Syst. Informatics Model. Q.*, 9:67–83, 2016.

[43] K. Rabbani, M. Lissandrini, and K. Hose. Extraction of validating shapes from very large knowledge graphs. *PVLDB*, 2033.

[44] R. Ramakhrishnan and J. Gehrke. *Database Management Systems (3rd edition)*. McGraw-Hill, 2003.

[45] 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.

[46] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB*, pages 302–314, 1999.

[47] W. Spoth, O. A. Kennedy, Y. Lu, B. Hammerschmidt, and Z. H. Liu. Reducing ambiguity in JSON schema discovery. In *SIGMOD*, 2021.

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

[49] W3C XML Document Type Specification. https://www.w3.org/TR/REC-xml/#dt-doctype, 2008.

[50] W3C XML Schema Definition Language (XSD). https://www.w3.org/TR/xmlschema11-1/, 2012.

[51] CoreResearch JSON dataset. Available online at https://core.ac.uk/services/dataset, 2022.

[52] ENELShops RDF dataset. Available online at https://old.datahub.io/dataset/enel-shops, 2022.

[53] Foodista RDF dataset. Available online at https://old.datahub.io/dataset/foodistal, 2022.

[54] GitHub JSON dataset. Available online at https://api.github.com/events, 2022.

[55] Mondial XML dataset. Available online at http://aiweb.cs.washington.edu/research/projects/xmltk/xmldata/www/repository.html#mondial, 2022.

[56] NASA flights RDF dataset. Available online at https://old.datahub.io/dataset/data-incubator-nasa, 2022.

[57] Prescription JSON dataset. Available online at https://www.kaggle.com/datasets/roamresearch/prescriptionbasedprediction, 2022.

[58] PubMed biomedical database (XML). Available online at https://www.ncbi.nlm.nih.gov/books/NBK25501/, 2022.

[59] WikiMedia XML dump. Available online at https://dumps.wikimedia.org/frwikinews/20221001/, 2022.

[60] Yelp open dataset: An all-purpose dataset for learning. https://www.yelp.com/dataset, 2018.