Computing Generic Abstractions from Application Datasets
Nelly Barret, Ioana Manolescu, Prajna Upadhyay

To cite this version:
Nelly Barret, Ioana Manolescu, Prajna Upadhyay. Computing Generic Abstractions from Application Datasets. EDBT 2024 - 27th International Conference on Extending Database Technology, Mar 2024, Paestum, Italy. pp.94-107. hal-04131974v4

HAL Id: hal-04131974
https://hal.science/hal-04131974v4
Submitted on 28 Aug 2023

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

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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.u@hyderabad.bits-pilani.ac.in
in
BITS Pilani H, India

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 when sharing semi-structured datasets (XML, JSON, 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 focus 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 or graph-structured dataset. Our work leverages the idea that any dataset comprises some entities (data objects), often grouped in collections (sets). Such sets of entities are often connected by some relationships. Our abstractions differ from the classical Entity-Relationship schemas [41] 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

Figure 1: Abstraction pipeline.
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 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 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. 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).

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 [19] 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), or non-leaf JSON records, carry no explicit kind information. Instead, their kind is implicitly encoded in the data structure. In that case, 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∈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 attr1, attr2 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 Go=(N0, E0, λ0) where E0⊆N0×N0 is a set of directed edges, and λ0 is a function labeling each node and edge with a string label, that could in particular be e (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 Go; 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 (e). 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, 35]. For instance, from the XML fragment: ⟨paper id="p1" wB="a1 a2"/⟩⟨paper id="p2"/⟩⟨paper/⟩⟨authors/⟩(authors/)(author id="a1")Eva/author/)(author id="a2")/⟩authors/⟩the attribute wB of the first paper becomes two edges labeled wB, 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 of a matching tuple r ∈ R has an outgoing edge labeled a pointing to the S tuple node n created for the respective tuple s ∈ 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 np1 to np2, Go 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 Go an edge from np1 to ne, and one from ne to np2.

Normalized graph In Go, some edges have empty (e) labels, while others carry meaningful labels. Regardless of the input data model, for uniformity, we transform Go into a normalized graph G=(N,E,λ), copying all the nodes of Go and all its e-label edges, and replacing each Go edge of the form n1→n2 where ℓ#e by two unlabeled edges n1→X1, X1→n2 where X1 is a new intermediary node labeled ℓ. All subsequent abstraction steps apply on the normalized graph G.
We aim at a partition \( P \) of the database or CSV file. As discussed in Section 2, "kind" information is explicitly encoded in some data models, and deeply nested (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 \( P=\{C_i\}, 1 \leq i \leq N \), 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 \( R(a_1, \ldots, a_k) \) 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 \( r.a_1, r.a_2 \) 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\(^5\). 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 GO). It has been shown\(^6\) 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,\(^7\) proposes a heuristic merging the equivalence classes into at most \( r \) classes. Avoiding the need for users to specify \( r \), the Typed Strong summary\(^2, 25\) 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.

### 3.4 Property clique 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 clique\(^5\), 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\(^2\).

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

---

\(^5\)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.

\(^6\)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\(^2\).

---

**Figure 3:** Sample normalized graph.

**Figure 4:** Normalized RDF sample depicting partitioning.
document root are often of the same kind (to form the paths, we assign the special name \( \text{map} \) to all array nodes, \( \text{array} \) 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 \( \text{array.map.friends.array.map} \) is fused with the one on \( \text{array.map} \), since they have the same property clique. Thus, an edge links \( \text{array} \) to \( \text{map} \) and \( \text{map} \) has a (new) child equivalence class \( \text{mail} \).

### 3.4 The collection graph

We call \textit{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.

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 \( \text{item} \) nodes and vice versa, leading to a cycle between the list and item collections.

\[
\text{document} \langle \text{description} \rangle \langle \text{list} \rangle \langle \text{item} \rangle \langle \text{list} \rangle \langle \text{item} \rangle \langle \text{list} \rangle \langle / \text{description} \rangle
\]

### 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, \( \text{paper} \), \( \text{author} \) and/or \( \text{conf} \) 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 \textit{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 \( E_{\text{max}} \) entity collections (for a given integer \( E_{\text{max}} \)). 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 \( E_{\text{max}} \) is low.

**Main entity eligibility criteria** Leveraging classical E-R design [44], we identify two criteria a collection must satisfy, to be considered as a potential main entity. (C1) 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 \textit{container} to many nodes from another, potentially interesting collection, e.g., in XMark, the (only) \textit{people} element is the parent of all person nodes. (C2) 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:

\[
\text{top} \langle \text{page} \rangle \langle \text{result} \rangle \langle / \text{result} \rangle \langle / \text{result} \rangle \langle / \text{page} \rangle \langle / \text{page} \rangle \langle / \text{top} \rangle
\]

the page collection has as single child, namely \textit{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 \textit{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 \textit{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 \textit{highest number of children} in the collection graph. For instance, paper nodes have four children: year, abstract, title, and \( \text{wh} \). This method, however, does not account for deeply nested structure. Thus, instead, we could count \( k\text{-\text{desc}} \) (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 \( k\text{-leaf} \), for some integer \( k \), consists of counting only \textit{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 (\( w_{\text{DAG}} \)).

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 \( o_w \) of the score of \( G \). Next, we define the ow of a leaf collection as the sum of the ow of its nodes, e.g. in Fig. 6, \( ow(\text{name}) = 2 \) and \( ow(\text{name#}) = 5 \). Further, for each edge \( C_i \rightarrow C_j \) in the collection graph, we attach an **edge transfer factor** \( f_{ij} \) as the fraction of nodes in \( C_j \) having a parent node in \( C_i \); \( 0 < f_{ij} \leq 1 \). Each \( f_{ij} \) 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 \( w_{\text{DAG}} \) 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 \( ow(n) \), and on others, to 0. Then, for each non-leaf collection \( C \), and non-cyclic path of the form \( C_0 \rightarrow C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_l \) where \( C_l \) is a leaf collection, we add \( cw(C) \rightarrow \sum_{j=1}^{l} f_{ij} \cdot cw(C_j) \): 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 \( \text{mail} \) and 3 from \( \text{name#} \).

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

#### 4.2.2 PageRank weight computation (\( w_{\text{PR}} \)).

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 \( w_{\text{PR}} \) starts by creating \( G_{\text{PR}} \), 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 \( \frac{1}{|\text{out-degree}(n)|} \), 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 \( w_{\text{PR}} \).

#### 4.2.3 Data-weighted PageRank computation (\( w_{\text{dPR}} \)).

A drawback of \( w_{\text{PR}} \) 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 \( w_{\text{dPR}} \) method where cardinalities are reflected in the PR edges weights. \( w_{\text{dPR}} \) builds \( G_{\text{PR}} \) and initializes node scores to \( \frac{1}{|\text{out-degree}(n)|} \) as in Sec. 4.2.2. Next, we assign to each edge in \( G_{\text{PR}} \) a weight, shown as label of each green dotted edge in Fig. 6:

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

Finally, as in PR, we iterate until convergence. We denote the node weight computed in this way \( w_{\text{dPR}} \).

**Weights from edge counts** Observe that we compute \( G_{\text{PR}} \) 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 **bound\_desc**, respectively, **bound\_leaf**.

#### 4.3.2 Boundary when using DAG weights. \( w_{\text{DAG}} \) 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** (**bound\_DAG**). 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 \( w_{\text{PR}} \) or \( w_{\text{dPR}} \), 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
The steps:

- a certain maximum number \( C \) to \( \text{cov} \) weight within (the ...
- flood boundary computation boundaries, every oriented path in the collection graph that goes
- \( E \) recomputed. all graph collections (w
- \( o w \) boundary. For instance, once ...
- chosen, or these entities together cover a sufficient fraction \( C \) nodes and edges within the boundary of \( C \) fashion, starting from ...
- excluded, which brings their own weight \( C \) nodes may thus be excluded, which brings their own weight
- \( E \) have been traversed.
- \( i \) have a parent in \( C_i \), for a fixed \( 0 < f_{\text{min}} \leq 1 \).
- The boundary flood computation method bound\( dp \) 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 bound\( ac \) proceeds like bound\( dp \), but only along edges that are not in any cycle.

In general, entity boundaries may overlap, i.e., a collection \( C \) 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 ...

\[ \text{(the ... denotes nested structure in Fig. 2). This provides a quick-glance view of each main entity collection, with all its attributes.} \]

### 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 \( E_{\text{max}} \) of entities have been selected, or these entities together cover a sufficient fraction \( \text{cov}_{\text{min}} \) of the data (that is: the sum of \( o w \) over all the collections within \( C_E \)’s boundary, is at least \( \text{cov}_{\text{min}} \) 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 choosing the next main entity.

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 \( o w \) to 0, thus reducing some of \( o w \) of each leaf collections within \( C_E \)’s boundary. For instance, once author is selected with name in its boundary, the \( o w \) of name decreases to 2. Then, the scores of all graph collections (\( w_{\text{DAG}} \), respectively, \( w_{\text{PR}} \) or \( w_{\text{PDP}} \)) are recomputed.

The set of all reported main entities is denoted \( E \).

### 4.5 Selecting the relationships between the main entities

Having selected the main entities \( E = \{ C^1, \ldots, C^{E_{\text{max}}} \} \) and their boundaries, every oriented path in the (collection graph that goes

![Figure 7: Outline of the classification algorithm.](https://dbpedia.org/ontology/person)

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: \( \text{paper} \rightarrow \text{author} \), \( \text{hW} \rightarrow \text{paper} \), and \( \text{pln.conf.inv} \rightarrow \text{author} \).

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: \( \text{paper} \rightarrow \text{author} \), \( \text{conf} \rightarrow \text{paper} \), paper \( \rightarrow \text{pln.conf.inv} \), and conf \( \rightarrow \text{author} \).

The set of all reported relationships is denoted \( R \).

### 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 \( \rightarrow \text{pln.conf.inv} \rightarrow \text{author} \). In some cases, (non-main) collections are traversed by several reported relations. Such collections may help to better understand the \( R \) relations. 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 \( R \) relations traverse the (eligible) conf collection, it will be shown as an entity; \( \text{pln} \) and inv are not eligible.

### 5 MAIN ENTITY CLASSIFICATION

To each entity collection \( C \in E \), 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). Alg. 1 and Fig. 7 outline 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 \( K \), e.g., Person, and a set of properties \( P \) which may be associated to the classes, in the following sense. For each property \( p \in P \), the set \( \text{domain}(p) \subseteq K \) describes classes to which one belongs if one has property \( p \); for instance, \( \text{domain(birth\_day)} = \{ \text{Person} \} \) states that having a birth\_day means one is a Person. Similarly, \( \text{range}(p) \) states the classes to which belong the values of \( p \). For instance, \( \text{range(birth\_day)} = \{ \text{Date} \} \).

To populate \( K \) and \( P \), 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 $P$ properties from a KB Given a knowledge base $B$ and a class $k \in K$, we harvest from $B$ the properties $P_k$ that resources of class $k$ may have, as: $P_k = \{r | (a, r, b) \in B \land \langle a, \text{type}, k \rangle \in B\}$. For example, YAGO4 [4] contains the triples $\langle \text{Paris}, \text{type}, \text{Place} \rangle$, $\langle \text{Paris, country, France} \rangle$ and $\langle \text{Paris, river, Seine} \rangle$. Thus, we add country and river to the set $P_{\text{Place}}$.

5.2 Expressive-type collection-label profiles

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

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, $\text{Σ}_s$ be the $T$-vector count of expressive-type values in $s$, and $\text{Σ}_s'$ be the $T$-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, $\text{Σ}_s[\text{Location}]=2$, $\text{Σ}_s[\text{Organization}]=1$, and $\text{Σ}_s'$ is 0 for the other expressive types, whereas $\text{len}_s'=\text{Σ}_s'=13$, $\text{Σ}_s'[\text{Organization}]=4$, and $\text{Σ}_s'$ is 0 elsewhere. The expressive type profile of a string $s$ is the triple $(\text{len}_s, \text{Σ}_s, \text{Σ}_s')$.

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 $(\text{Σ}_{x \in N_{C,x}} \text{len}_x, \text{Σ}_{x \in N_{C,x}} (\text{Σ}_x', \text{Σ}_{x \in N_{C,x}} (\text{len}_x'))$ where the last two are $T$-vector sums. We denote the first two and third components of the collection-label profile by $\text{len}_{C,x}$, $\text{Σ}_{C,x}$, $\text{Σ}_{C,x}'$, respectively. Continuing with the collection $C$ of author nodes, $\text{len}_{C,\text{Name}}$ is 9 (length of “Léa”, “Eva” and “Bob” combined), $\text{Σ}_{C,\text{Name}}[\text{Person}]=3$ and $\text{Σ}_{C,\text{Name}}'[\text{Person}]=0$ in all other positions, while $\text{len}_{C,\text{Name}}'[\text{Person}]=9$ and $\sigma_{C,\text{Name}}'$ is 0 in all the other positions.

5.3 Classification algorithm

Alg. 1 details our classification algorithm, also illustrated in Fig. 7. We attempt to classify $C$ in one or a few classes $k \in K$, leveraging three different signals: (1) the kind names of $C$ nodes (if available), compared with those of $K$ classes; (2) the labels of $C$’s properties, compared with those of $P$ properties; (3) the expressive-type profiles of $C$, combined with $P$ range information.

For (1), if $C$ nodes have one or few kind names, we compute their Word2Vec [38] similarity $\text{sim}^W(C, k)$ with each class $k$ using Eq. 1 (in blue in Fig. 7, lines 1 to 3 in Alg. 1): we sum the pairwise cosine similarities of each keyword $w$ in $s_1$ and $s_2$, embedded in a multidimensional Word2Vec-based vector $e(w)$. For instance, medical and service are the keywords in the class medical service and that class and hospital have a high $\text{sim}^W$.

Next, each data property $dp_i$ outgoing $C$ may “vote” to classify $C$ in one or several classes (violet in the figure), as follows. For (2), we compute $\text{sim}^T(dp_i, p)$ between the name of $dp_i$ and that of each $p \in P$ ($\text{sim}^W$ at line 7). For (3), we check if expressive values of a class $k \in \text{range}(p)$ make up a large fraction of $dp_i$ values of $C$ nodes, to reflect the similarity between those named entities types and classes in $\text{range}(p)$. This is quantified by Eq. 2:

$$\text{sim}^T(C, dp_i, p) = \frac{\sum_{k \in \text{range}(p)} \text{Σ}_{C,dp_i} [k] > 0 \text{len}_{C,dp_i}[k]}{|\text{range}(p)| \cdot \text{Σ}_{C,dp_i}'}$$

As classification result (at right in black in Fig. 7, lines 15 to 26), we pick the class(es) $k^n$ with the highest modulated score if this value is above a threshold $\theta$, otherwise, the most general class in $K$, e.g., Thing. If there are ties, we may use: (i) the class hierarchy of $K$; (ii) $C$ properties having names typically associated to identifiers (e.g., name or label). For (i), $C$ is classified in the most specific class, e.g., human wins against being. For (ii), if $C$ has such a property $p$ and if the expressive-type profile of $C$ and $p$ contains only named entities of a single $K$ class, we classify $C$ in that class. In Fig. 3, the author collection may be classified in several classes, but the property name has led to Person entities only, thus it will be classified as Person. If there is no sufficient insight to classify $C$, we assign it the most general class in $K$.

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

Astra is implemented in Java, leveraging the graph creation (including entity extraction) and Postgres-based store of ConnectiLens [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 abduction 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).
Algorithm 1: Classifying a collection C.

Input: Collection C, properties \( P \), classes \( \mathcal{K} \)

1. foreach \( k \in \mathcal{K} \) do
   2. \( \mathcal{K}[k] \leftarrow \text{sim}^w(l,k) \)

3. foreach \( dp \), property of some nodes in C do
   4. foreach \( p \in P \) do
      5. if \( \text{sim}^w(C, dp, p) > 0 \) then
         6. \( s \leftarrow \text{sim}^w(dp, p) \)
      else
         7. \( s \leftarrow \text{sim}^w(dp, p) \)
      8. if \( s > 0 \) then
         9. \( \text{score}(dp, p) \leftarrow \text{score}(dp, p) + s \)

10. foreach \( p \in P \) do
    11. foreach \( k \in \text{domain}(p) \) do
        12. \( \mathcal{K}[k] \leftarrow \mathcal{K}[k] + \text{supp}(dp, p) \cdot \text{score}(dp, p) \)/\( \text{domain}(p) \)

13. if \( \max(\mathcal{K}) > 0 \) then
    14. Classify \( C \) as \( k^* \)
    15. else if \( \text{arg.max}(\mathcal{K}) \) has just one element \( k' \) then
      16. Classify \( C \) as \( k' \)
    17. else if \( 3 \leq k' \in \text{arg.max}(\mathcal{K}) \) such that \( k' \) is a specialization of all the other \( \text{arg.max}(\mathcal{K}) \) classes then
      18. Classify \( C \) as \( k' \)
    19. else if \( \text{C has an identifier-related property and all their entity profiles lead to a single \( K_g \) entity type} \) then
      20. Classify \( C \) as that type
    21. else
      22. Classify \( C \) as the most general class in \( \mathcal{K} \)
    23. else
      24. Classify \( C \) as the most general class in \( \mathcal{K} \)

6.1 Datasets, semantic resources, and settings

We evaluated ABSTRACT on JSON, PG, RDF and XML synthetic and real-life, open datasets. 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), NYT 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.

| Dataset name | \( |V| \) | \( |E| \) | \( |N| \) | \( |E| \) |
|--------------|--------|--------|--------|--------|
| JSON         | CoreResearch | 2,112,023 | 2,112,022 | 2,112,025 | 2,112,024 |
|              | GitHub   | 5,094   | 5,093   | 5,096   | 5,095   |
|              | NYTTimes | 1,053,979 | 1,053,978 | 1,053,981 | 1,053,980 |
|              | Prescriptions | 14,214,033 | 14,214,032 | 14,214,033 | 14,214,034 |
|              |Researchers | 4,505,657 | 4,505,656 | 4,505,659 | 4,505,658 |
|              | TelRoguisiness | 8,111,982 | 8,111,981 | 8,111,984 | 8,111,983 |
|              | YelpCheckIn | 659,093 | 659,632 | 659,635 | 659,654 |
|              | YelpTips   | 9,998,068 | 9,998,067 | 9,998,070 | 9,998,069 |
|              | PG        | 1,053,980 | 1,053,979 | 1,053,981 | 1,053,983 |
|              | LDBCsmall | 9,851   | 43,011  | 48,621  | 77,546  |
|              | LDBC0.3   | 2,366,540 | 10,907,534 | 11,890,494 | 19,167,734 |
|              | Movies250K | 12,886,250 | 14,817,871 | 23,140,245 | 21,507,996 |

Table 1: Datasets used in the evaluation.

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 \( P \) properties (on top of the 101 we had manually identified). From GitTables [32], we derived 4,187 \( P \) 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 \( E_{\text{max}} = 5 \) (select at most 5 main entities), \( \text{cos} _{\text{min}} = 1.0 \) (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.

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 institution. 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 depicts items to be sold in auctions, people with nested addresses, some of which are interested in open auctions; there are also closed auctions. Items for sale have nested descriptions and categories they belong to. There is also a category hierarchy. As in larger datasets, items are in several continents (e.g. (frica) and (asia) tags).

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 together? Are there any attributes 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, bound(desc)) 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 desc, bound(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 (w(desc, bound(desc))) 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 (w(dwPR, bound(desc))) leverages both complex graph structure and data cardinalities, from now on, we focus on the (w(dwPR, bound(desc))) abstraction method, which performed best, as it fulfills our main objectives (Sec. 1).

Table 2: Datasets used in the user relevance feedback.

<table>
<thead>
<tr>
<th>Dataset name</th>
<th>Nc</th>
<th>Rf</th>
<th>N</th>
<th>E</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>Researchers (JSON)</td>
<td>569</td>
<td>665</td>
<td>684</td>
<td>780</td>
<td>26</td>
</tr>
<tr>
<td>LDBC (PG)</td>
<td>445</td>
<td>791</td>
<td>1082</td>
<td>1280</td>
<td>78</td>
</tr>
<tr>
<td>Conferences (RDF)</td>
<td>145</td>
<td>256</td>
<td>369</td>
<td>448</td>
<td>32</td>
</tr>
<tr>
<td>XMark (XML)</td>
<td>847</td>
<td>865</td>
<td>904</td>
<td>920</td>
<td>126</td>
</tr>
</tbody>
</table>

Table 3: Users' ranking of dataset sample abstractions.

<table>
<thead>
<tr>
<th>Dataset name</th>
<th>JSON</th>
<th>PG</th>
<th>RDF</th>
<th>XML</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Researchers (JSON)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CoreResearch</td>
<td>48</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>GitHub</td>
<td>43</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>NYTimes</td>
<td>116</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Prescriptions</td>
<td>851</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Researchers</td>
<td>25</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>YelpBusiness</td>
<td>22</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>YelpCheckIn</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>YelpTips</td>
<td>22</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LDBC small</td>
<td>61</td>
<td>4</td>
<td>9</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>LDBC.0.3</td>
<td>62</td>
<td>6</td>
<td>16</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Movies250K</td>
<td>38</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RSIM14M</td>
<td>64</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>UMB1M</td>
<td>108</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Nasa</td>
<td>61</td>
<td>5</td>
<td>13</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mondial</td>
<td>168</td>
<td>5</td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PubMed</td>
<td>126</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>XMark1</td>
<td>386</td>
<td>5</td>
<td>10</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>XMark2</td>
<td>386</td>
<td>5</td>
<td>10</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Wikipedia</td>
<td>22</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4: Main entities found in the application datasets.

6.3 Main entity collections in all datasets

We now study the main entities E and relationships R identified by the (wdwPR, bound(ac)) abstraction method, in all our datasets (Tabs. 4). A ○ indicates that the dataset's collection graph has cycles (14 datasets out of 25). For each main entity, we also report a human-readable label (not yet our classification - see Sec. 6.4) as follows. When the nodes in the collection share an explicit kind 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-REF. In BSBM4M, the collection Product is multi-traversed (recall Sec. 4.6) by the relationships Re- 

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Collection (Class, relevance)</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSON</td>
<td></td>
</tr>
<tr>
<td>GitHub</td>
<td>Repository (Thing, L)</td>
</tr>
<tr>
<td>NvTimes</td>
<td>Document (CreativeWork, H)</td>
</tr>
<tr>
<td>Prescriptions</td>
<td>Prescription(CreativeWork, M)</td>
</tr>
<tr>
<td>Researchers</td>
<td>Researcher(Person, H)</td>
</tr>
<tr>
<td>YelpBusiness</td>
<td>Business(Thing, L)</td>
</tr>
<tr>
<td>YelpCheckin</td>
<td>Check-in(Thing, L)</td>
</tr>
<tr>
<td>YelpTips</td>
<td>Tip(Thing, L)</td>
</tr>
<tr>
<td>YelpTipsPG</td>
<td>Tip(Thing, L)</td>
</tr>
<tr>
<td>LDBCsmall</td>
<td>Post(Thing, L); Comment(Comment, H); Forum(Blog, H); Person(Person, H)</td>
</tr>
<tr>
<td>LDBC0.3</td>
<td>Comment(Comment, H); Post(Thing, L); Forum(Blog, H); Tag(Thing, L); Organisation(Organisation, H); Person(Person, H)</td>
</tr>
<tr>
<td>Movies250k</td>
<td>playedin(Play, M); Actor(Actor, H); Movie(Movie, H); Director(Person, H)</td>
</tr>
<tr>
<td>BSBM4M</td>
<td>Offer(Offer, H); Review(Review, H); Product(Product, H); ProductFeature(Product, H); Producer(Thing, L)</td>
</tr>
<tr>
<td>BSBM16M</td>
<td>Review(Review, H); Offer(Offer, H); Person(Person, H); Product(Product, H); Producer(Thing, L); Vendor(Thing, L)</td>
</tr>
<tr>
<td>Conferences</td>
<td>Author(Person, H); Paper(CreativeWork, H)</td>
</tr>
<tr>
<td>EnelShops</td>
<td>Shop(Restaurant, M)</td>
</tr>
<tr>
<td>Foodista</td>
<td>Recipe(Recipe, H); Food(Food, H); Tool(Thing, L); PreparationMethod(Thing, L)</td>
</tr>
<tr>
<td>LUBM1M</td>
<td>Publication(CreativeWork, H); GraduateStudent(Person, H); ResearchAssistant(Person, H); TeachingAssistant(Person, H)</td>
</tr>
<tr>
<td>Nasa</td>
<td>Spacecraft(Spacecraft, H); Launch(Launchpad, H); Image(ImageObject, H); MissionRole(SpaceMission, H); Person(Person, H)</td>
</tr>
<tr>
<td>XML</td>
<td></td>
</tr>
<tr>
<td>Mondial</td>
<td>City(City, H); Province(Province, H); Country(Country, H); Organization(Organization, H); River(River, H)</td>
</tr>
<tr>
<td>PubMed</td>
<td>PubMedArticle(CreativeWork, H)</td>
</tr>
<tr>
<td>XMark1</td>
<td>Person(Person, H); Item(Product, H); Open Auction(Produot, M); Closed Auction(Produot, M); Category(Thing, L)</td>
</tr>
<tr>
<td>XMark4</td>
<td>Person(Person, H); Item(Product, H); Open Auction(Produot, M); Closed Auction(Produot, M); Category(Thing, L)</td>
</tr>
<tr>
<td>Wikimedia</td>
<td>Faq(Taing, L); Namespace(Thing, L)</td>
</tr>
</tbody>
</table>

Table 5: Quality of \( \mathcal{E} \) collection classification.


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 \((w_{dwp, pr}, bound_{d-ae})\), attain their goal. The main entity depth varies from 2 (XMark) to 11 (LUBM1M), with 3 and 5 being frequent values. This shows that \((w_{dwp, pr}, bound_{d-ae})\) 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,

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 \( \text{sim}^X(C, dp, P) \) value in favor of Creative work, which is best in this case. In other cases, the classification is due to \( \text{sim}(dp, P) \) 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 \((w_{dwp, pr}, bound_{d-ae})\). 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)
Abstractions have different goals and uses than dataset schemas. The abstraction method (without voluntarily leave out. They contain interconnected types, while to give human users a first idea of a dataset abstraction wrap some collections into the main collection bound-
ples (depth 11). Classification, with record kind names, property their structure ranges from simple (depth 2 in Tab. 4) to very com-
method successfully identifies the central dataset entities, when results overall, even on complex, cyclic collection graphs. This
All our abstractions scale up linearly in the data size. The methods using bound\(d_{ac}\) and bound\(d_{ac}\) (not plotted to 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 \(w_{d,w_{PR}},\) bound\(d_{ac}\) 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. Anstra sets parameters’ default values to \(w_{d,w_{PR}},\) bound\(d_{ac}\) and thresholds in Sec. 6.1. Our abstractions are computed in linear time in the input size. 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 large (semi)structured datasets. A lot of summarization methods exist; recent surveys include [17, 37]. As explained in Sec. 3, we picked theTyped Strong summary [25] for RDF to comply with our principle (★): it leverages RDF types (kinds) when present, 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 graphs [12, 13, 28, 45]. 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 semi-structured 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 partially funded by DIM RFSI PhD 2020-01 and AI Chair SourcesSay project (ANR-20-CHIA-0015-01) grants.

Figure 8: Abstraction computation times on synthetic JSON, PG, RDF and XML datasets, using \(w_{d,w_{PR}},\) bound\(d_{ac}\).


