ArchiTXT: A Grammar-Based Metamodel Approach for Structuring Textual Data Toward Database Models

Jacques Chabin*
Université d'Orléans, INSA CVL, LIFO UR 4022
Orléans, France
jacques.chabin@univ-orleans.fr

Mirian Halfeld-Ferrari
Université d'Orléans, INSA CVL, LIFO UR 4022
Orléans, France
mirian@univ-orleans.fr

Nicolas Hiot†
Université d'Orléans, INSA CVL, LIFO UR 4022
Orléans, France
nicolas.hiot@univ-orleans.fr

ABSTRACT

We propose a method for structuring textual data, seen as syntax trees enriched with semantic information, guided by a meta-model $\mathbb{G}$, defined as an attribute grammar. The approach follows an evolution process in which instances and their grammars evolve through rewriting rules, eventually generating a target grammar G_T that satisfies $\mathbb{G}$. Applied to the construction of a database from text, this method generates both a schema and its instance, independent of specific database models. We show its feasibility with clinical medical cases as a proof of concept.

1 INTRODUCTION

This work concerns structuring textual data into a format that conforms to a predefined framework, specified by a meta-model $\mathbb{G}$. The textual data is represented as a rooted forest (i.e., syntax trees combined into a single structure with a common root) based on an initial grammar, G_0. The process involves transforming this initial structure to meet the constraints of a new format defined by a target grammar, G_T. The meta-model $\mathbb{G}$ guides the process by outlining the desired structure, with transformations applied to achieve it. These transformations occur incrementally, evolving both the data and the grammar from G_0 to G_T. The approach, based on structural transformation, can be enhanced by enriching the syntax trees with pre-processed knowledge, such as detected named entities, or by incorporating semantic constraints within $\mathbb{G}$.

Based on this general idea, this paper presents ArchiTXT. It deals with the structuring of textual data to populate a database. The process involves the generation of both the database schema and its corresponding instance. Rather than targeting a specific database model, we adopt a general abstraction that captures common concepts and relationships found in database design, independent of the implementation. Our motivation is twofold. First, textual data continues to present significant challenges for AI and data analytics. Structuring the information contained in texts enables more efficient analysis and database storage. Second, modern applications increasingly need to interface with multiple data models simultaneously. In these cases, a generalized structure offers a practical solution for facilitating data exchange and integration across diverse database models.

We developed our approach using clinical medical cases as the application domain. Starting from textual descriptions, we generate a generic hierarchical structure representing a database schema and its instance. This instance consists of key text fragments suitable for database storage. Our approach is hybrid: it relies on syntax to transform the initial tree by pruning or aggregating sub-trees, but begins with an enriched tree containing semantic information extracted during pre-processing, taking into account the application domain. These transformations are thus indirectly guided by this enrichment step. Thus, in our approach, trees represent the database instance, and a grammar capable of generating such trees serves as the database schema.

In this context, the main contributions of this paper are:

- The proposal of an original way of looking at the structuring of textual data.
- The proposal of an attribute grammar $\mathbb{G}$ that represents a generic database structure. A grammar respecting $\mathbb{G}$ is seen as a database schema which can be then translated to any data model such as relational or graph models.
- The formalization of textual data structuring through the evolution process of a grammar G_0 (associated to the syntactic trees of the sentences) to a target grammar G_T which respects the meta-grammar $\mathbb{G}$.
- The formalization of an evolution process through transformations on the trees (instances) guided by tree rewriting rules and a similarity measure.
- A proof-of-concept implementation that uses clinical cases as input and $\mathbb{G}$ as a generic database structure.

Paper Organization. Section 2 overviews our approach. In Section 3 we position our work with respect to some related work. Section 4 gives some background concepts. Section 5 presents the details of our structuring method, including the definition of our meta-grammar $\mathbb{G}$ (Section 5.1). Section 6 carefully analyses a proof of concept and Section 7 offers some final comments.

2 OVERVIEW

The ultimate goal of our approach is to organize textual information by extracting a database instance from the text, as demonstrated in Figure 1. This structuring method involves altering the level of abstraction, allowing for the generalization of information when feasible.

Our approach is grounded from a grammatical perspective: each sentence in the text is associated with its syntactic tree, and we consider the initial grammar G_0 as the one that accepts these syntactic trees. Figure 3a presents an example of a syntactic tree.

Our proposal consists in an iterative process which is visually summarized in Figure 2. We work progressively by transforming the data instance (trees) and the data schema (a grammar). Our goal is to obtain an instance respecting a target grammar G_T, which in turn respects the meta-grammar $\mathbb{G}$.

*Authors in alphabetical order.
†Main contact author. The foundation of this work stems from Dr. Nicolas Hiot's doctoral research [14], supported financially by EnnovLabs—Ennov.

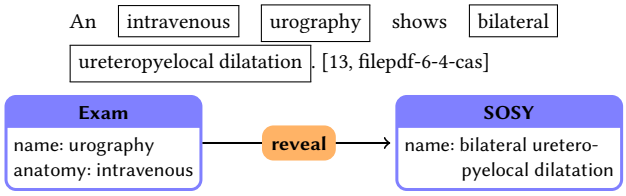


Figure 1: An example of a graph database instance generated by structuring a text describing a clinical case.

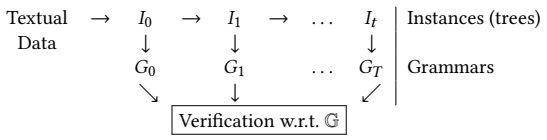
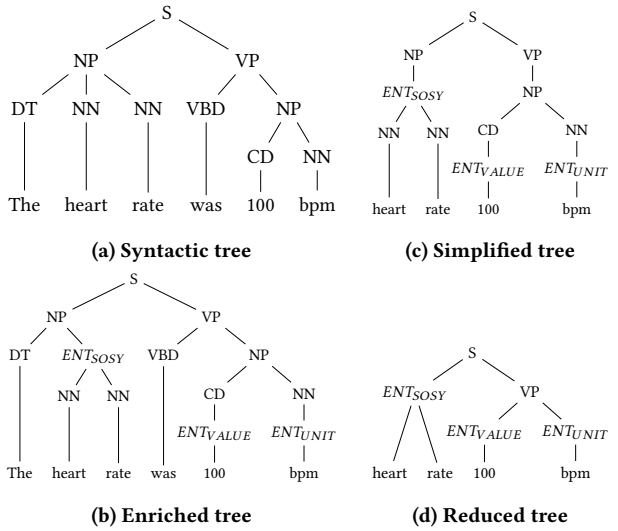


Figure 2: Iterative process for automatic structuring

In Figure 2, the vertical axis represents the extraction of a grammar from an instance I_i in the form of a rooted forest of enriched syntactic trees, while the horizontal axis represents the progression of the process from step i to step $i+1$ based on transformations on the instance. These transformations follow the following reasoning:

(1) **Initialisation.** The process starts with the transformation of sentences into syntactic trees, which gives rise to an instance denoted by I_0.
(2) **Enrichment.** The second step is to enrich the trees of the I_0 instance by incorporating information previously extracted from the text analysis. This may involve inserting named entities, relationship or other relevant semantic information into the trees.
(3) **Successive Evolutions:**
  (a) **Evolution of the instance**. To evolve from instance I_i to instance I_{i+1}, the branches of the tree are grouped or transformed based on similarity measures. This may involve reorganising the structure of the sub-trees to improve their consistency or to better align them with the desired representation.
  (b) **Evolution of the grammar**. The evolution of G_i towards G_{i+1} is triggered by checking whether the grammar G_i conforms to the meta-grammar $\mathbb{G}$. At step i, if G_i does not conform to $\mathbb{G}$, the process continues by transforming the tree structures of I_i, giving rise to the instance I_{i+1}, which generates a new grammar G_{i+1}. The process ends when we find a grammar G_T which satisfies $\mathbb{G}$.

Example 2.1. Figure 3 illustrates an example of transformation on a tree instance. In Figure 3a, the syntactic tree corresponding to the sentence "The heart rate was 100 bpm" is shown. Figure 3b displays the same tree enriched with the named entities ENT_{SOSY}, ENT_{VALUE} and ENT_{UNIT}, identified during a preprocessing step. Figures 3c et 3d present trees where unnecessary terms and nodes have been removed. For instance, in Figure 3c, only the branches corresponding to terms relevant to the database are retained (e.g., the branch corresponding to the article "The" in Figure 3a is deleted). In Figure 3d, nodes indicating grammatical functions such as VP, NN, NP, CD, etc., are deemed irrelevant for the database context and are therefore removed. The objective



(a) Syntactic tree **(c) Simplified tree**

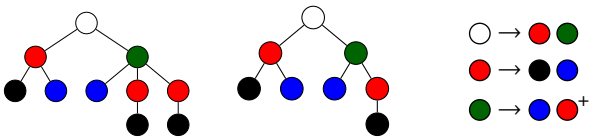
(b) Enriched tree **(d) Reduced tree**

Figure 3: Example of entity integration in a syntax tree and simplifications

here is to maintain the overall structure with the minimum number of nodes while preserving those that convey the necessary semantic information for database reasoning. □

Each iteration introduces modifications to the tree instance, guided by tree rewriting rules and a similarity measure that helps define equivalence classes. The rewriting rules aims to prune subtrees or reorganize them differently. Section 5.5 discusses this aspect in details.

With reference to the vertical axis of Figure 2, at each step i, a grammar G_i is derived from instance I_i through the computation of a *quotient tree*. The idea behind this process is illustrated in Figure 4 (see Section 5.3 for formal definitions).



(a) Instance tree I **(b) Quotient tree Q** **(c) Grammar G**

Figure 4: Example of quotient tree

The instance tree in Figure 4 is summarized into a quotient tree based on equivalence classes, highlighting the main patterns in the tree. This quotient tree (Figure 4b) shows that the tree in Figure 4a has green nodes with blue and red children. Once the quotient tree is created, extracting the corresponding grammar becomes straightforward, as shown in Figure 4c. In this example, each production rule in grammar G associates a node's color with the color of its children. For instance, a red node generates black and blue nodes, but in Figure 4a, some red nodes lack their blue child, indicating missing information. Our approach accounts for such cases, enabling the extraction of grammars like G in Figure 4c. Additionally, our grammars are expressed as extended context-free grammars, using symbols like + to indicate repetition, as seen in the production rule for the green node.

The target grammar G_T is the one that respects the meta-grammar $\mathbb{G}$. The purpose of $\mathbb{G}$ is to offer a general description

of the main database abstractions, independent of the specific database model. $\mathbb{G}$ defines four key concepts that guide the generalizations and transformations applied to trees:

Attribute A name associated with a data value.
Group A named set of attributes.
Relation A relationship between distinct groups.
Collection A set of equivalent groups and relations.

In Figure 5, these concepts are illustrated within a tree instance. The figure also displays the grammar extracted from this tree.

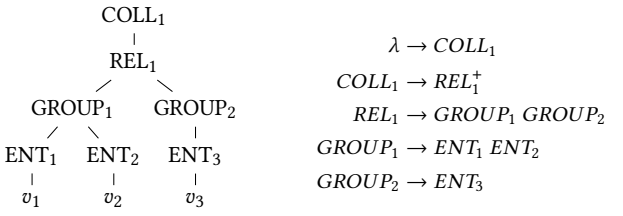


Figure 5: An example of a tree instance after our iterative process, where internal nodes represent concepts from $\mathbb{G}$, along with its corresponding grammar G_T.

3 RELATED WORK

Text structuring can be considered through top-down and bottom-up perspectives [1]. In the top-down approach, a schema is provided, and the problem is seen as a query on the text to extract or identify *relevant* information. In [9, 16, 25, 30, 32] we find examples of traditional approaches, while recent trends are shifting towards machine learning (ML) and large language models (LLMs) for the extraction of entities and relationships [10, 17, 20]. ML methods often depend on large annotated corpora for training, which can be both expensive and time-consuming. While they perform well, they may lack the flexibility to handle novel or unexpected data types that deviate from the established schema.

Bottom-up approaches are typically referred to as *open information extraction* (OpenIE). They are used, for example, in *ontology learning* and involve the extraction of terms, entities, and relationships from text, followed by their classification and grouping, often based on similarity or syntactic rules [2, 22]. These methods operate without predefined schemas, offering greater flexibility across diverse domains. Early techniques, such as those described in [12, 26], employed syntactic patterns to extract ⟨subject, predicate, object⟩ triples, linking them to knowledge bases. Recent advancements also shifted towards ML techniques [11, 31], particularly leveraging neural networks to enhance extraction accuracy [19, 27, 29, 37, 38]. Even with these developments, ontology learning remains a complex challenge [4] that often requires human supervision, although LLMs tend to avoid it. Current OpenIE models still struggle to extract meaningful relationships and lack a standardized output format [23].

As mentioned in Section 1 our approach is hybrid, combining syntax tree transformation with semantic enrichment of the tree beforehand. Hybrid methods have demonstrated potential for improved efficiency. For instance, [35] introduces Semi-Open Information Extraction (SOIE) to discover domain-independent facts, and [34] proposes a tool to merge a bottom-up graph from unstructured text with a top-down graph from structured data.

As data volume and variety grow, information system administrators must find effective solutions for storing, managing, and integrating data from multiple sources while addressing user needs. The concept of using a common meta-model that can be mapped to different database models is increasingly being proposed as a solution [3, 24]. In [3], the authors propose schema extraction for structured and semi-structured data coming in different formats, with the aim of presenting datasets uniformly to help users understand and make choices. They use an intermediate structure based on concepts (sub-records, records, collections) similar to those in our generic schema (entities, groups, relations, collections). For this reason, we consider it the work most closely related to ours. Unlike our method, which transforms syntax trees (text) to generate new instances, their approach does not handle unstructured data. Instead, they map (semi-)structured data from various sources to a graph format, detecting nodes that correspond to the key concepts they define. By interpreting these nodes, data from different sources can be better understood and integrated. Their focus is on helping users understand the data; once nodes have been classified based on some user input, they are given semantic meaning, often using an ontology. This last step is beyond the scope of our work, as our grammar is not designed for non-specialist users.

In [24], the authors propose a framework that, starting from a conceptual model designed according to a meta-model, uses transformation rules to determine the most appropriate NoSQL model for implementation. Similar to our approach, both data and schema are transformed. However, unlike our method, their goal is to adapt a general model to a specific NoSQL or relational model. It would be worth exploring these ideas as a post-processing to our approach: from our target grammar G_T and its associated instance I_T, find the most appropriate database model (e.g. many relationships might suggest a graph model).

To take our method a step further and also produce a user-friendly final schema, we might consider how to (1) assign semantic names to our structures for better clarity (as in [3]) and (2) explore (as in [24]) which database model best suits our schema. But this is out of the scope of this paper.

4 BACKGROUND

This section reviews ordered trees and formal grammars, highlights the role of trees in representing linguistic structures, and considers tree rewriting rules for transformation and editing.

Definition 4.1 (Ordered tree). A tree $T = (D, l)$ consists of a domain D and a labelling function l. The domain D is a subset of $(\mathbb{N})^*$ (i.e. a set of integer sequences of the form $x.y.z$). The labelling function $l : D \to \Sigma \cup \{\lambda\}$ maps elements of D to labels from a set Σ or a special symbol λ. The domain D satisfies the following properties: (1) D is closed under prefixes, i.e. for $u, u' \in (\mathbb{N})^*$ if u is a prefix of u' and $u' \in D$, then $u \in D$, and (2) For all $u \in \mathbb{N}^*$ and $j \in \mathbb{N}$ if $u.j \in D$ then for all $i \in \mathbb{N}$ such that $0 \leq i < j$ we have $u.i \in D$.

Each element of D is called *position*. For a node n at position p, $|p|$ defines the length of the sequence, also called the *depth* of n. The root of a tree is at position ϵ and is labeled with the special symbol λ, i.e. $l(\epsilon) = \lambda$. An empty tree is therefore defined by $T = (\{\epsilon\}, \langle \epsilon \mapsto \lambda \rangle)$. We write $v \prec u$ if $u = v.i$ for some $i \in \mathbb{N}$ where the node at position v is the *parent* of the node at u, and u is the *child* of v. We write $v \prec^* u$ if $\exists v'$ such that $u = v.v'$, meaning that v is a *prefix* of u. A node n at position u is a *descendant* of a node m at position v if and only if v is a direct prefix of u (denoted $v \preceq u$) or indirect (denoted $v \preceq^* u$), conversely m is an *ancestor* of n. A *leaf* is a node at a position u such that $u.0 \notin D$, i.e. a node with no children. We note $\mathbb{T}$ the set of all trees. □

Definition 4.2 (Sub-tree). Given a tree $T = (D, l)$, a sub-tree of T at position $u \in D$ is denoted by $T|_u = (D', l')$ and has the following properties: (1) $D' \subseteq D$ such that $\forall v \in D'\ u \preceq^* v$ and (2) $l' = \langle v \mapsto l(v) \mid v \in D' \rangle$. Moreover, if $t = T|_u$ is a sub-tree, we denote by $t' = P_i^t$ the ith tree-ancestor of t when $t' = T|_v$, $u = vw$ and $|w| = i$. We note $\mathbb{ST}$ the set of all sub-trees. □

Example 4.3. Let $T = (D, l)$ be a tree with $D = \{\epsilon, 0, 1, 1.0, 1.1\}$. The node labels are defined as follows: $l(\epsilon) = root$, $l(0) = child1$ (the left child of the root); $l(1) = child2$ (the right child of the root), and for the children of node at position 1, we have $l(1.0) = grandchildren1$ and $l(1.1) = grandchildren$. $T|_1 = (D', l')$ is a sub-tree of T. Note that $T|_1$ is not a tree because $D' = \{1, 1.0, 1.1\}$ does not respect the conditions of Definition 4.1. Here, $P_1^{T|_1}$ coincides with T. □

Definition 4.4. A context-free grammar (CFG) $G = (N, T, R, S)$ is a quadruplet where N is a finite set of non-terminal symbols; T is a finite set of terminal symbols; R is a finite set of production rules, and $S \in N$ is the initial symbol. A production rule is defined by the form $X \to \alpha$, where $X \in N$ and α is a string of terminal and non-terminal symbols. As syntactic sugar, a condensed CFG, allows production rules of the form $X \to \alpha^+$, where α^+ is a regular expression that repeats the string α one or more times. This is equivalent to the rules $X \to \alpha$ and $X \to \alpha\ X$. □

A parse tree, also known as a derivation tree or syntax tree, describes how the starting symbol of a grammar G derives a word in the language. When a non-terminal U is associated with a production rule $U \to X\ Y\ Z$, the derivation tree will contain an internal node labelled U with three children, X, Y and Z, arranged from left to right.Each internal node represents a non-terminal symbol of the G grammar, while the leaves represent the terminal symbols of G. The links between the nodes illustrate how the symbols are derived from each other. The derivation tree is built recursively, following the production rules of the grammar. It starts with a root node corresponding to the initial symbol of the grammar, and at each level of the tree the nodes are replaced by symbols according to the production rules.

Definition 4.5 (Parse Tree). Given a grammar $G = (N, T, R, S)$, a parse (or derivation) tree of G is a tree which satisfies the following properties:

(1) The root is designated by the start symbol S, i.e. $l(\epsilon) = S$;
(2) Each leaf f is denoted by a terminal symbol, i.e. $l(f) \in T$;
(3) Each internal node x is denoted by a non-terminal symbol, i.e. $l(x) \in N$;
(4) If U is a non-terminal used as a label of an internal node x and $X_1, \ldots, X_n$ are the labels of the children of n from left to right, then there exists a production rule $U \to X_1\ \ldots\ X_n$ in R. The labels $X_1, \ldots, X_n$ represent a sequence of terminal and non-terminal symbols. □

Example 4.6. Let $G = (\{P\}, \{0, 1\}, R, P)$ where R is the set of production rules $\{P \to 0 \mid 1 \mid P\ 0 \mid P\ 1\}$. The grammar produces binary numbers. For instance, the parse tree for 0011 is

$$\begin{array}{ccccccc} & & & & & & (root) \\ P & - & P & - & P & - & P \\ | & & | & & | & & | \\ 0 & & 0 & & 1 & & 1 \end{array}$$

□

An attribute grammar extends a CFG by adding semantic information, stored in attributes tied to the grammar's terminal and non-terminal symbols. Attribute values are computed through rules linked to the grammar's productions. For each non-terminal in a CFG G, there are two sets of attributes: (1) *synthesized attributes*, which pass information from the leaves to the root of a derivation tree and (2) *inherited attributes*, which pass information from the root to the leaves. Each production is paired with semantic rules that define how to calculate the output attribute – synthesized attributes of the left-hand non-terminal and inherited attributes of the right-hand non-terminals – based on the input attributes. The following definition formalizes this concept.

Definition 4.7 (Attribute Grammar). An attribute grammar [18] is a CFG $G = (N, T, R, S)$ with a set of semantic rules Φ_r added to each production rule $r \in R$. Each symbol $X \in (N \cup T)$ is associated to a finite set of attributes $A(X)$ consisting of two disjoint subsets of attributes: (i) $A_\uparrow(X)$ for synthesized attributes where $\forall X \in T,\ A_\uparrow(X) = \emptyset$, and (ii) $A_\downarrow(X)$ for inherited attributes where, for the initial symbol S, $A_\downarrow(S) = \emptyset$

Each attribute $a \in A(X)$ has a (potentially infinite) set of possible values V_a from which a value is selected for each occurrence of X in the derivation tree. The production rule r has the form $X_0 \to X_1, \ldots, X_n$ where $n \geq 1$, $X_0 \in N$ and $X_i \in (N \cup T)$ for $1 \leq i \leq n$. A semantic rule $\varphi \in \Phi_r$ associated with r is a function such that $a = \varphi(b_1, \ldots, b_k)$ for each output attribute a of r where b_i $(1 \leq k)$ are input attributes of r. This rule computes the value of an attribute of X_j from the attributes of the symbols $X_0, \ldots, X_n$. If a is an attribute of X_0, it is a synthesized attribute. If a is an attribute of X_j $(1 \leq j \leq n)$, it is an inherited attribute. □

Example 4.8. Let G' be an attribute grammar built from G of Example 4.6 by associating a synthesized attribute val with the non-terminal P and providing rules to compute the value of val relative to the value of the previously computed attribute val' associated with the right side of the production rule.

$$\begin{array}{llll|llll} P_{val} \to 0 & [val \leftarrow 0] & & & P_{val} \to P_{val'}\ 0 & [val \leftarrow 2 * val'] \\ P_{val} \to 1 & [val \leftarrow 1] & & & P_{val} \to P_{val'}\ 1 & [val \leftarrow 2 * val' + 1] \end{array}$$

In the production rules, the attributes are indicated as subscripts and the semantic rules are presented in square brackets to the right of the production rule. The grammar G' associates the decimal value with a binary number. The parse tree for 0011

$$\begin{array}{ccccccc} P(val = 0) & - & P(val = 0) & - & P(val = 1) & - & \overset{(root)}{P(val = 3)} \\ | & & | & & | & & | \\ 0 & & 0 & & 1 & & 1 \end{array}$$

shows, for each level, the computed value of the attribute val. □

S-attribute grammars are attribute grammars containing only synthesized attributes, making them simpler and easier to verify through bottom-up propagation.

Definition 4.9 (Meta-grammar). A meta-grammar $\mathbb{G} = (N, T, R, S)$ is an S-attribute grammar where N is the set of meta-non-terminals, T is the set of meta-terminals, R is the set of production rules, and $S \in N$ is the start symbol. $\mathbb{G}$ specifies the syntax for production rules of condensed CFGs. Words in the language recognized by $\mathbb{G}$ are lists of production rules for condensed CFGs.

A synthesized attribute γ is used to verify that each derivation produces a valid condensed CFG. If $S_\gamma = \top$, the derivation is valid; if $S_\gamma = \bot$, it is invalid.

Semantic rules in production rules $r \in R$ fall into two types:

- $a \leftarrow \alpha$, where a is an attribute and α is a formula on attributes in r;
- $\gamma \leftarrow \beta$, where β is a logical formula on attributes in r.

For clarity, we will omit the part $\gamma \leftarrow$ and rules of the form $a \leftarrow a$ in this paper. □

Syntax trees (as the on in Figure 3a) are often used to represent texts, serving as derivation trees for the grammar of the natural language. In linguistics, they depict the syntactic structure of a sentence, highlighting the hierarchical relationships between words or word groups and their roles within parts of speech (PoS) such as nouns (NN), verbs (VBD), adjectives (ADJ), and determiners (DT). The leaves of the tree represent the lexical units (words), while the intermediate nodes correspond to abstract structures like verb phrases (VP) or noun phrases (NP).

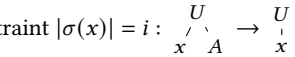
Tree rewriting, or term rewriting, involves transforming trees into other trees using specific rewriting rules. In our context, these rules allow the formalization of tree transformations, with the goal of organizing information in a more structured manner while disregarding unnecessary elements. In the following we recall the basis of tree rewriting.

Definition 4.10 (Hedge). A hedge is a possibly empty sequence of trees, represented as $h = [t_0, \ldots, t_n]$, with $|h|$ indicating the number of trees (i.e. $|h| = n + 1$). A substitution, denoted σ, is a bijective mapping from a set of variables V to a set of hedges and from a set of labels to a set of sub-trees, homomorphically extended to trees. □

Definition 4.11 (Rewriting rule). A rewriting rule on a tree specifies how a tree t can be rewritten as t' at a given position u. It consists of a left-hand side (LHS), representing a pattern, and a right-hand side (RHS), representing the transformation, written as $LHS \to RHS$. The pattern is a subtree formed from the set $\Sigma \cup V \cup \{\lambda\}$, where Σ is the set of labels of t, V is a set of variables, and λ is the root symbol. A morphism maps variables from LHS to RHS, enabling the transformation.

The rule applies by substituting σ, a subtree of t at position u, into the LHS pattern. This creates a correspondence between elements of the LHS and a subtree of t. The application of the rule is denoted as $t \mapsto_{[u,LHS \to RHS,\sigma]} t'$, where $t|_u = \sigma(LHS)$ and $t'|_u = \sigma(RHS)$.

A rewriting rule may also include application conditions that specify when the rule can be applied, such as constraints on node or edge attributes or topological requirements. □

Example 4.12. Let $\{X, Y, A, B, C, D\} \subseteq \Sigma$ be a set of labels (non-terminals) in trees. Consider the rewriting rule $rule(u.i)$ with the constraint $|\sigma(x)| = i$: 

Here, U is the node at position u in the target tree, and $x \in V$ is a variable. The rule applies to a tree T if there is a sub-tree $t = T|_u$ with a substitution σ such that $\sigma(x) = [t|_0, \ldots, t|_{i-1}]$ and $l(u.i) = A$. Applying the rule deletes the sub-tree labeled A from T. Figure 6 show the application of $rule(0.2)$ with $u = 0$ and $i = 2$ where $\sigma(x) = [C, B]$. □

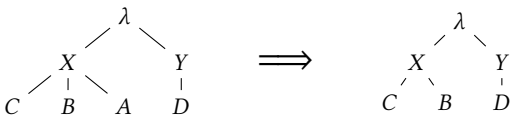


Figure 6: Example of the application of $rule(0.2)$ on a tree

5 AUTOMATIC STRUCTURING

Algorithm 1 summarizes our process for structuring textual data. Textual data is represented by trees, from which grammars are extracted to serve as a general representation. The structure that emerges from this iterative process is a grammar that plays the role of a database schema, together with its corresponding parse tree, which corresponds to a database instance. The resulting grammar can be mapped to various database models.

As input, Algorithm 1 receives: (*i*) an instance I_0 corresponding to a forest of syntax trees generated from text sources, which have been merged into a single tree with a common root; (*ii*) the meta-grammar $\mathbb{G}$ and (*iii*) a set of entities **E**. We define a *named entity* as a tuple $E = (entityName, startToken, endToken)$ where $entityName$ is the name or type of the named entity, $startToken$ is the index of the token that marks the beginning of the entity, and $endToken$ is the index of the token that marks the end of the entity. $L_{tokens}(E)$ corresponds to the sequence of tokens forming part of the entity, defined as $L_{tokens}(E) = [startToken, \ldots, endToken]$. Named entities represent real-world objects and are instances of a class, with "Paris" as an example of a "City" entity. Algorithm 1 outputs a target grammar G valid with respect to $\mathbb{G}$.

The iterative process can be summarized as follows. It begins with an enrichment step (line 1) where entities and relationships are added as internal nodes to the syntax trees, followed by the removal of redundancies to simplify the structure. Next, the grammar is extracted (line 2) and checked against a pre-established meta-grammar (line 3). If the resulting grammar is not valid, tree transformations are applied. This involves computing equivalence classes for non-terminals (line 4) and then unifying and structuring equivalent sub-trees according to the meta-grammar $\mathbb{G}$ (line 5). Then, a new grammar is extracted from the new instance (line 6) and the while loop proceeds with verification.

Algorithm 1: AlgoStructMain(I_0 , $\mathbb{G}$, **E**)

```
1 I ← EnrichSimplify(I_0, E)
2 G ← ExtractGrammar(I)
3 while G not valid wrt 𝔾 do
4 |   ComputeEqClasses(I)
5 |   I ← Rewrite(I)
6 |   G ← ExtractGrammar(I)
7 end
8 return G
```

In the following sections, we outline each step of our approach. First, we define the meta-grammar $\mathbb{G}$, followed by an explanation of each step of Algorithm 1.

5.1 Meta-Grammar: generic database schema definition

The purpose of the meta-grammar is to define the core concepts of a database model, in a generic manner (Section 2). These concepts – attribute (or entities), group, relation, and collection – must now be expressed by trees to enable their identification within the tree structure representing our data instance I.

Table 1 presents our meta-grammar $\mathbb{G}$, an attribute grammar that defines valid data structures. Meta-non-terminals are indicated by angle brackets $\langle\cdot\rangle$, while the semantic rules are shown on the right side of the table within square brackets $[\cdot]$.

The first production meta-rule of $\mathbb{G}$ (1) indicates that the target grammar G is defined by an initial rule, followed by a possibly empty list of rules. The initial rule generated by $\mathbb{G}$ (meta-rule 2) contains the symbol λ (initial non-terminal of G) on its left-hand side. Its right-hand side is defined by the meta-rules 3-8 which

$$\epsilon ::= \langle root_{eL',gL',cgL',rL',crL'} \rangle \text{ EOL } \langle ruleList_{eL,gL,cgL,rL,crL} \rangle \quad [eL' \subseteq eL; gL' \subseteq gL; cgL' \subseteq cgL; rL' \subseteq rL; crL' \subseteq crL] \quad (1)$$

$$\langle root_{eL,gL,cgL,rL,crL} \rangle ::= \lambda \rightarrow \langle rootList_{eL,gL,cgL,rL,crL} \rangle \quad (2)$$

$$\langle rootList_{eL,gL,cgL,rL,crL} \rangle ::= \epsilon \quad [eL \leftarrow \emptyset; gL \leftarrow \emptyset; cgL \leftarrow \emptyset; rL \leftarrow \emptyset; crL \leftarrow \emptyset] \quad (3)$$

$$| \; ENT_{name} \langle rootList_{eL',gL,cgL,rL,crL} \rangle \quad [name \notin eL'; eL \leftarrow \{name\} \cup eL'] \quad (4)$$

$$| \; GROUP_{name} \langle rootList_{eL,gL',cgL,rL,crL} \rangle \quad [name \notin gL'; gL \leftarrow \{name\} \cup gL'] \quad (5)$$

$$| \; REL_{name} \langle rootList_{eL,gL,cgL,rL',crL} \rangle \quad [name \notin rL'; rL \leftarrow \{name\} \cup rL'] \quad (6)$$

$$| \; COLL_{name} \langle rootList_{eL,gL,cgL',rL,crL} \rangle \quad [name \notin cgL'; cgL \leftarrow \{name\} \cup cgL'] \quad (7)$$

$$| \; COLL_{name} \langle rootList_{eL,gL,cgL,rL,crL'} \rangle \quad [name \notin crL'; crL \leftarrow \{name\} \cup crL'] \quad (8)$$

$$\langle ruleList_{eL,gL,cgL,rL,crL} \rangle ::= \epsilon \quad [eL \leftarrow \emptyset; gL \leftarrow \emptyset; cgL \leftarrow \emptyset; rL \leftarrow \emptyset; crL \leftarrow \emptyset] \quad (9)$$

$$| \; \langle entity_{name} \rangle \text{ EOL } \langle ruleList_{eL',gL,cgL,rL,crL} \rangle \quad [name \notin eL'; eL \leftarrow \{name\} \cup eL'] \quad (10)$$

$$| \; \langle group_{name,eL'} \rangle \text{ EOL } \langle ruleList_{eL,gL',cgL,rL,crL} \rangle \quad [name \notin gL' \wedge eL' \subseteq eL; gL \leftarrow \{name\} \cup gL'] \quad (11)$$

$$| \; \langle relation_{name,gL'} \rangle \text{ EOL } \langle ruleList_{eL,gL,cgL,rL',crL} \rangle \quad [name \notin rL' \wedge gL' \subseteq gL; rL \leftarrow \{name\} \cup rL'] \quad (12)$$

$$| \; \langle collGrp_{name,grpName} \rangle \text{ EOL } \langle ruleList_{eL,gL,cgL',rL,crL} \rangle \quad [name \notin cgL' \wedge grpName \in gL; cgL \leftarrow \{name\} \cup cgL'] \quad (13)$$

$$| \; \langle collRel_{name,relName} \rangle \text{ EOL } \langle ruleList_{eL,gL,cgL,rL,crL'} \rangle \quad [name \notin crL' \wedge relName \in rL; crL \leftarrow \{name\} \cup crL'] \quad (14)$$

$$\langle group_{name,eL} \rangle ::= GROUP_{name} \rightarrow \langle entList_{eL} \rangle \quad (15)$$

$$\langle collGrp_{name,grpName} \rangle ::= COLL_{name} \rightarrow GROUP^{+}_{grpName} \quad (16)$$

$$\langle relation_{name,gL} \rangle ::= REL_{name} \rightarrow GROUP_{name1} \; GROUP_{name2} \quad [name1 \neq name2; gL \leftarrow \{name1, name2\}] \quad (17)$$

$$\langle collRel_{name,relName} \rangle ::= COLL_{name} \rightarrow REL^{+}_{relName} \quad (18)$$

$$\langle entList_{eL} \rangle ::= ENT_{name} \quad [eL \leftarrow \{name\}] \quad (19)$$

$$| \; ENT_{name} \langle entList_{eL'} \rangle \quad [name \notin eL'; eL \leftarrow \{name\} \cup eL'] \quad (20)$$

$$\langle entity_{name} \rangle ::= ENT_{name} \rightarrow \langle data \rangle \quad (21)$$

Table 1: Meta-grammar $\mathbb{G}$ using BNF format

specify the construction of a series of G non-terminals. These non-terminals are: *ENT*, *GROUP*, *REL* and *COLL*, representing, respectively, entities, groups of entities, relations between groups and collections of groups or relations. To distinguish each structure specific to a G grammar, we associate a *name* attribute with each non-terminal. The attributes of $\mathbb{G}$ are synthesized and represent lists of names (*name*) used as identifiers:

- eL (or eL'): list of entity names;
- gL (or gL') : list of group names;
- rL (or rL') : list of relation names;
- cgL (or cgL') : list of group collection names;
- crL (or crL') : list of relation collection names.

They are initialized in a bottom-up fashion. Syntax rules are used to check that a name is unique. For example, if the meta rules 19-20 are applied, the list eL gets entity names that are unique. This is also the case for all other lists: the uniqueness of the name of a new non-terminal is guaranteed by the semantic rules. It is also important to note that these semantic rules ensure that any non-terminal appearing to the right of a production rule in the G grammar must have a rule defining it. For example, if the meta rule 1 is applied, all elements in the list gL' must be present in gL.

Example 5.1. Consider the grammar G from figure 5 and the derivation of $\mathbb{G}$ that leads to the rule $\lambda \rightarrow COLL_1$. By applying meta-rule 2, we derive the rule $\lambda \rightarrow \langle rootList \rangle$, where the right-hand side contains a meta-non-terminal. Next, applying meta-rule 8 results in the intermediate rule $\lambda \rightarrow COLL_1 \; \langle rootList \rangle$, and finally, meta-rule 3 produce $\lambda \rightarrow COLL_1$. The set of production rules for the grammar G is defined by meta-rules 9-14, where each rule introduces a non-terminal for G.

Figure 7 presents a partial derivation of G from the meta-grammar $\mathbb{G}$. Attributes are displayed in blue, excluding γ and empty-set attributes for clarity. Note that the semantic rules of $\mathbb{G}$ impose constraints to ensure that every non-terminal in the target grammar G is properly defined, a requirement for constructing a valid grammar. In this context, Figure 7 shows, on the the left-hand side of the root, that $crL' = \{1\}$, signifying that $COLL_1$ is referenced in the root rule. On the right-hand side, $\langle ruleList \rangle$ holds $crL = \{1\}$. Since $crL' \subseteq crL$, this confirms that every non-terminal appearing in the root rule of G has a corresponding defined production rule, ensuring a valid derivation. □

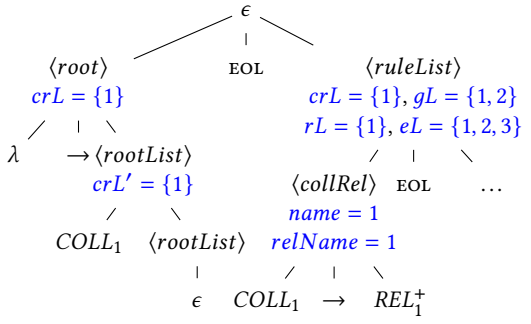


Figure 7: Extract of a derivation of $\mathbb{G}$

5.2 Tree Enrichment and Simplifications

The enrichment step involves simplifying the syntax trees and adding semantic information, mainly formalised by tree rewriting rules. This section gives an overview of these operations.

5.2.1 Simplifying Conjunction Sub-Trees. Syntactic trees are generated using parsers, typically trained on treebanks. However, due to linguistic variations, syntactic annotations across treebanks are not standardized. In both French and English, coordinating conjunctions can be represented using either recursive or flat structures. In this paper, we utilize the English and French parsers provided by CoreNLP, where the English parser tends to favor a flatter structure, placing coordinated elements at the same hierarchical level. Since sub-trees for coordinating conjunctions are suitable candidates for *collection* structures (mentioned in Section 5.1), we rewrite the conjunctions sub-trees to this flatter representation to better align with our meta-model.

5.2.2 Enriching Trees. To incorporate an entity in a syntax tree T means to create an ENT-labeled sub-tree, specifying the entity type (e.g., Person, Country, Disease) and containing leaves for the entity's tokens.

Definition 5.2 (Ordered entity subtree). Let $T = (D, l)$ be a syntax tree, $E = (entityName, startToken, endToken)$ be an entity and $L_{tokens}(E)$ be the list of indices of E's tokens. An ordered entity sub-tree is a tuple $E_T = (entityName, L_{tree}(T, E))$ where $L_{tree}(T, E)$ is the sequence of positions of the tokens of E in the tree T such that $L_{tree}(T, E) = [u.b, \ldots, u.e] = [treePos(T, startToken), \ldots, treePos(T, endToken)]$ with u being the position common to all the tokens of E and $treePos : \mathbb{T} \times \mathbb{N} \to D$ being a function which associates for each index of a token its position in the tree. $|E|$ is the size of the entity (or number of tokens) such that : $|E| = |L_{tokens}(E)| = |L_{tree}(T, E)| = (endToken - startToken) + 1$. □

Entities correspond to the concept of attributes (Section 2) in our generic database model, defined by the meta-grammar $\mathbb{G}$. We maintain the classical notion of attributes; therefore, nested entities are considered invalid under $\mathbb{G}$. To comply, nesting is represented as a relationship between the encompassing entity and the contained entities, forming a tree with two children. This transformation is performed by the application of the tree rewriting rule unnest_ent depicted in Figure 8a.

5.2.3 Simplifications. The trees are simplified in two steps:

(1) Sub-trees without entities are deleted. They are identified by checking the parents of all leaves in T. If they aren't labelled with an entity name, they are removed (e.g., sub-trees $T|_{0.0}$ and $T|_{1.0}$ in Figure 3b).
(2) Nodes not labelled as entities and with only one child are deleted (e.g., nodes at position 0, 1.0.0 and 1.0.1 in Figure 3c). This is done using the rewriting rule reduce from Figure 8b.

Figure 3d shows an enriched tree after simplifications.

5.3 Grammar Extraction

In lines 2 and 6 of Algorithm 1, we extract the grammar G from a given instance I and then verify its correctness against the meta-grammar $\mathbb{G}$ (line 3). The grammar G is derived by calculating the quotient tree S, a hierarchical representation of I, used to obtain G.

We recall that a partition of a set X is a division of its elements into non-empty, disjoint subsets. An *equivalence relation* on a set defines a partition, and every partition corresponds to an equivalence relation. A set family F is a partition of a set X if and only if all the following conditions are satisfied: (i) $\emptyset \notin F$; (ii) $\bigcup_{A \in F} A = X$ and (iii) $\forall A, B \in F\ (A \neq B) \Rightarrow (A \cap B = \emptyset)$. The sets of F are called *blocks*. In graph theory, a quotient graph Q of a graph G is a graph whose vertices are blocks of a partition of the vertices of G and where a block A is adjacent to a block B if at least one vertex of A is adjacent to a vertex of B with respect to the set of edges of G. To construct the grammar from a tree, we introduce the definition of *quotient tree*, which corresponds to a quotient graph with no cycles and no vertices with multiple parents.

To extract the grammars G_i we use the equivalence relation R_l between the labels of a tree T, defined by $(\forall x, y \in D)\ x\ R_l\ y \iff l(x) = l(y)$. Intuitively, we obtain as equivalence classes a set of positions for each label present in the T tree. For example, in Figure 9a, $C_\lambda = \{\epsilon\}$ and $C_X = \{0, 1\}$ are the equivalence classes for the labels λ and X respectively.

To construct a quotient tree, we follow two steps:

- *Compute the hierarchy of equivalence classes.* We define the function Succ, which, for a given class C, returns the set of equivalence classes containing at least one element that is a child of an element in C.

Definition 5.3 (Function Succ). Let $T = (D, l)$ be a tree and R an equivalence relation. Let $D/R = \{C_0, \ldots, C_n\}$ be the set of equivalence classes of T. Define the function Succ as $\text{Succ}(C) = \{C' \mid \exists u \in C, v \in C' \text{ such that } u \prec v\}$ □

Example 5.4. Let $T = (D, l)$ be the tree Figure 9a. The first step in constructing the quotient tree Q_T is to retrieve the equivalence classes of D given by the relation R_l. We then obtain the following classes: $C_\lambda = \{\epsilon\}, C_X = \{0, 1\}, C_Y = \{2\}, C_a = \{00, 20\}$, $C_b = \{01, 10\}, C_c = \{11\}$. The construction of Q_T involves recovering the hierarchy of equivalent sets. The algorithm starts with the class C_λ and recursively traverses its successors. For each class, we have the following successors: $\text{Succ}(C_\lambda) = \{C_X, C_Y\}$, $\text{Succ}(C_X) = \{C_a, C_b, C_c\}$, $\text{Succ}(C_Y) = \{C_a\}$, $\text{Succ}(C_a) = \emptyset$, $\text{Succ}(C_b) = \emptyset$, $\text{Succ}(C_c) = \emptyset$. □

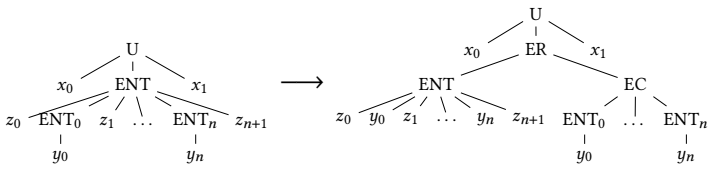
- *Assign each equivalence class a position in the ordered tree.* The function QDom computes the domain of the quotient tree using the Succ function. QDom assigns each successor class of C a relative position, $u.p$, where u is the position of class C. In other words,

$$\text{QDom}(C, u) = \{(C_j, u.p) \mid C_j \in \text{Succ}(C) \text{ and } 0 \leq j \leq n - 1\} \quad (22)$$

where n is the number of elements in $\text{Succ}(C)$. It is worth noting that it is possible for an equivalence class to appear as the successor of more than one class. In Example 5.4, C_a is a successor of both C_Y and C_X, resulting in different positions for C_a.

To construct the quotient tree $Q_T = (Q_D, Q_l)$ from a tree T, we start by assigning the equivalence class C_λ to the root of Q_T, i.e., $Q_D \leftarrow \{\epsilon\}$ and $Q_l \leftarrow \langle \epsilon \mapsto \lambda \rangle$. We also initialize a set *classes* with pairs in the format *(equivalence class, position in Q_T)*, denoted (CL, u). While *classes* is not empty, the algorithm iterates, selecting a pair (CL, u), updating *classes* with its successors computed using the QDom function, and updating the domain Q_D with the new position u. The algorithm checks if the equivalence class CL includes multiple positions with the same parent. Indeed, to construct a condensed CFG, sibling nodes with the same non-terminal in a parse tree are compactly represented in the quotient tree with the $^+$ symbol. This involves checking for repeated non-terminals with the same parent and merging their derivations, allowing for trees with incomplete information, as shown in the following example.

Example 5.5. Let us consider the result obtained in Example 5.4. Function QDom is used to associate positions to equivalent classes. We start with $\text{QDom}(C_\lambda, \epsilon) = \{(C_X, 0), (C_Y, 1)\}$ and



(a) unnest_ent$(T, u.i)$ **where** $|\sigma(x_0)| = i$

(b) reduce$(T, u.i, S_{labels})$ **where** $|\sigma(x)| = i$ **and** $|\sigma(y_1)| = 1$ **with** $|\sigma(y_2)| = 1$ **if** $S_{label} = \emptyset$ **;** $t(u.i) \notin S_{labels}$ **if** $S_{label} \neq \emptyset$

Figure 8: Basic operations

(a) Instance tree T **(b) Quotient tree** Q_T

Figure 9: Example of quotient tree computation

after successive applications of QDom (see expression (22)) we obtain:

$$\text{QDom}(C_X, 0) = \{(C_a, 00), (C_b, 01), (C_c, 02)\}$$
$$\text{QDom}(C_Y, 1) = \{(C_a, 10)\}$$
$$\text{QDom}(C_a, 00) = \emptyset \qquad \text{QDom}(C_a, 10) = \emptyset$$
$$\text{QDom}(C_b, 01) = \emptyset \qquad \text{QDom}(C_c, 02) = \emptyset$$

Figure 9b illustrates the obtained quotient tree. The node corresponding to class C_a is duplicated in Q_T, because it is linked to two positions: 00 and 10. The node corresponding to class C_X in Q_T is marked with '+', indicating that X can be repeated as a child of λ. This conclusion comes from Example 5.4: $C_X = \{0, 1\}$ contains two positions with the parent at ϵ. In the construction of Q_T, after a first iteration, $classes = \{(C_X, 0), (C_Y, 1)\}$. For the pair $(C_X, 0)$, we have $successors = \{(C_a, 00), (C_b, 01), (C_c, 02)\}$. The class C_X is not a singleton and its positions have the same parent. □

A tree T may represent an incomplete derivation of the condensed CFG G_T obtained from a quotient tree Q_T. For instance, if a production rule of G_T is $X \to a\ b\ c$, the tree T of Figure 9a is accepted as a valid derivation only if c or a are considered as missing values, which reflect omissions or errors in the syntactic analysis of natural language texts. Incomplete information has been a challenge for database research (see, e.g, [7, 8, 15, 21]). The transformation of a quotient tree into a grammar is formally defined below.

Definition 5.6 (Construction of a grammar from a quotient tree). The condensed CFG G_T, obtained from the quotient tree $Q_T = (Q_D, Q_l)$ of T, is defined by the quadruplet (N, T, P, λ) where: (i) the set of non-terminals N, possibly decorated by $^+$, is the set of labels $Q_l(u)$ for any position $u \in Q_D$ which is not a leaf; (ii) the set of terminals T, is the set of labels $Q_l(u)$ for any position $u \in Q_D$ which is a leaf; (iii) the set P of production rules contains, for any position $u \in Q_D$ which is not a leaf, rules of the form $Q_l(u) \to Q_l(u.0), \ldots Q_l(u.i)$ and (iv) λ is the starting symbol. □

Example 5.7. From Q_T in Example 5.5, Figure 9b, we obtain the grammar G_T with the following rules:

$$\lambda \to X^+\ Y \qquad X \to a\ b\ c \qquad Y \to a$$

5.4 Computing Equivalence Classes

Algorithm 1, on line 4, modifies instance I by identifying equivalence classes of sub-trees (non-terminals of the target grammar). At this stage, trees have been enriched, simplified, or rewritten. In the initial iteration, the tree contains named entity information, as shown in Figure 3d.

In our context, identifying equivalent sub-trees is essential for aggregating information. The textual representation of a real-world object can take different forms, reflected in different entity sub-trees. Besides, natural language often omits or implies information. For example, the parse trees of "The patient takes 500 mg of Paracetamol" and "The patient takes Paracetamol every day" are different, but both represent a *treatment*.

Determining sub-tree equivalence requires more than comparing entity labels, because natural language is ambiguous, and the same entity tree may represent different objects. Context must also be taken into account. We use the concept of regular equivalence from [33], where two vertices in a graph are equivalent if their neighbourhoods are equivalent. For example, two people can be considered *equivalent* (e.g. both representing a patient) if they are connected to *equivalent* vertices, such as a disease or a treatment, even if those vertices are different.

To define the equivalence relation, we introduce a similarity measure between sub-trees. A similarity measure is a symmetric function $f : \mathbb{ST} \times \mathbb{ST} \to [0, 1]$ with $f(x, x) = 1$ for all $x \in \mathbb{ST}$. Various measures f like Jaccard, Levenshtein, Jaro, or tree edit distance [36] can be used. The contextual similarity between two enriched sub-trees $x = T|_u$ and $y = T|_v$, denoted $sim_f(x, y)$, is computed as a weighted average of the recursive similarities provided by the function f for each tree-ancestor. The weights decrease as the distance from the tree-ancestor increases. The formula for $sim_f(x, y)$ is given by the following equation, where $depth_{min}$ is the minimum depth of the sub-trees x and y (i.e. $depth_{min} = \min\{|u|, |v|\}$), and P_i^x (or P_i^y) is the i-th tree-ancestor of x (or y).

$$sim_f(x, y) = \frac{\sum_{i=0}^{depth_{min}} \frac{1}{i+1} \cdot f(P_i^x, P_i^y)}{\sum_{j=0}^{depth_{min}} \frac{1}{j+1}} \tag{23}$$

Axiom 1. *The function sim_f is a weighted average of f. Therefore, sim_f is symmetric, bounded by the interval $[0, 1]$ and for all $x \in \mathbb{ST}$, $sim_f(x, x) = 1$.* □

Example 5.8. Let T be the tree of Figure 10. To define f, we use the Jaccard index ($J(X, Y) = |X \cap Y|/|X \cup Y|$) on the entity names present in the sub-tree. Although $f(\text{X}_1, \text{X}_2) = 1$, their contexts — one related to a drug (paracetamol) and the other to a frequency (every day) — suggest a similarity of less than 1. The function sim_f (equation 23) accounts for this difference. We find $f(\text{NP}_1, \text{NP}_2) = 0.75$ and continue recursively to the root, where

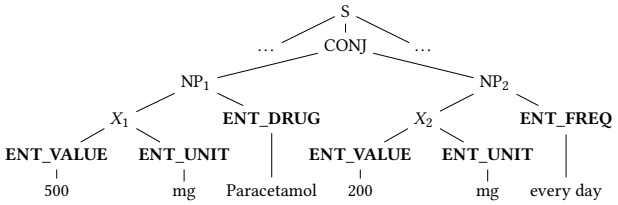


Figure 10: Extract of an enriched tree

the sub-trees are identical with a similarity of 1. This results in :

$$sim_f(\mathrm{X}_1, \mathrm{X}_2) = \frac{\overbrace{1\times 1}^{\mathrm{X}} + \overbrace{0.5\times 0.75}^{\mathrm{NP}} + \overbrace{0.33\times 1}^{\mathrm{CONJ}} + \overbrace{0.25\times 1}^{\mathrm{S}}}{1+0.5+0.33+0.25} \simeq 0.94$$

□

The example shows that although the two sub-trees are similar, they are not equivalent because they do not refer to the same objects. Our similarity measure defines when two sub-trees are similar, based on a threshold set for each dataset.

Definition 5.9 (Sub-tree similarity). Given an enriched tree T, let $st_1 = T|_u$ and $st_2 = T|_v$ be two sub-trees. Let $\tau \in [0,1]$ be a threshold. We say that st_1 and st_2 are τ-similar, denoted $st_1 \sim_\tau st_2$, if and only if $sim_f(st_1, st_2) \geq \tau$. □

Proposition 5.10. *τ-similarity is a* reflexive *and* symmetric *similarity relation.* □

Definition 5.11 (Sub-tree equivalence). Let T be an enriched tree. Given a τ-similarity relation (Definition 5.9), we define an equivalence relation between the sub-trees $x = T|_u$ and $y = T|_v$ (denoted $x \equiv_\tau y$) by the following equation :

$$(\forall x, y \in \mathbb{ST})\ x \equiv_\tau y \iff x \sim_\tau y \vee (\exists z \in \mathbb{ST})\ x \equiv_\tau z \wedge y \equiv_\tau z \quad (24)$$

Proposition 5.12. *The τ-equivalence is an equivalence relation, that is,* reflexive, symmetric *and* transitive.

Definition 5.13 (Equivalence classes). Let $[x]_\tau$ denote the τ-equivalence class of x, where $y \in [x]_\tau$ if and only if $y \equiv_\tau x$. For a tree $T = (D, l)$, $D/_{\equiv_\tau} = \{[x]_\tau \mid x \in D\}$ represents the quotient set (or partition) of D by $\equiv_\tau$, i.e., it is the set of all τ-equivalence classes of D. □

Partitioning a set based on distance can be done using single-link hierarchical clustering with the similarity measure sim_f. According to [5], single-link hierarchical clustering aligns with partitioning by an equivalence relation. This method constructs a hierarchy by initially treating each element as a separate class and merging the closest classes step by step based on similarity. A single-link hierarchy evaluates the similarity between classes as the maximum similarity (or minimum dissimilarity) between pairs of elements in the classes. A classification at similarity threshold τ merges classes if their similarity exceeds τ. For example, at $\tau = 0.3$, we have two classes: $\{a, b\}$ and $\{c, d, e\}$. When the threshold increases, e.g. $\tau = 0.5$, three classes emerge: $\{a\}$, $\{b\}$, and $\{c, d, e\}$. Further raising the threshold to $\tau = 0.7$ results in four classes: $\{a\}$, $\{b\}$, $\{c, d\}$, and $\{e\}$.

5.5 Rewriting Trees

The aim of the structuring step on line 5 of Algorithm 1 is to rewrite the instance tree to conform to a valid schema based on the $\mathbb{G}$ meta-grammar. Nodes representing *groups*, *relations*, or *collections* are identified, and equivalent sub-trees are rewritten to align with their respective categories. This iterative process checks schema validity after each modification. The rewriting approach may vary depending on the objectives. In this paper, the purpose of the rewriting function is to unify sub-trees by eliminating structural variations, maximizing their frequency, and minimizing the number of grammar production rules.

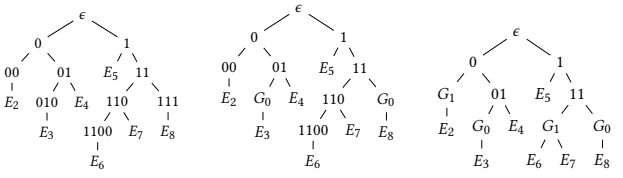
In Algorithm 1, the evolution from instance I_i to I_{i+1} follows a transformation process, where sub-trees are grouped or modified based on a similarity measure. The tree rewriting function on line 5 takes as input an instance I_i, i.e, a single tree representing a rooted forest of instance trees. Called within the while loop (line 3), this function iteratively transforms the instance into a condensed tree that represents the grammar. Each iteration focuses on invalid parts of I_i, and when no further transformations are possible, the resulting grammar G_T is valid under $\mathbb{G}$ (end condition of the while loop). In practice, to ensure termination, a maximum cycle limit K is set, which may leave some tree parts unresolved.

The whole process is governed by four parameters: f (similarity function), τ (similarity threshold), *minSup* (minimum element frequency), and K (maximum cycles).

Our rewriting function consists of five main transformation operations, tracked by a variable indicating whether a transformation has occurred. If the tree is modified, further operations are skipped, and the cycle restarts to update equivalence classes. The five *operations*, executed in logical sequence, are: detecting groups, unifying them, grouping into collections, identifying relationships between groups, and relationships between collections. Operations proceed sequentially; if no changes occur in the earlier steps at iteration i, the subsequent operations are applied until a modification occurs, advancing to I_{i+1}. If none of the five operations modify the tree, more drastic transformations are applied. These final steps work bottom-up, removing intermediate levels above uncategorized entities (operation reduce(bottom)). Once all nodes are categorized, the function deletes any remaining upper levels that aren't classified as an *entity*, *group*, *relation*, or *collection* (operation reduce(top)).

The reminder of this section provides an overview of the five key operations employed in the rewriting function (line 5 of Algorithm 1).

• The *findGroups* operation identifies frequent groupings of entities in the tree by partitioning the sub-trees above the entity sub-trees. It filters out these partitions with support below *minSupport*, resulting in a set, denoted *equivalent_st*, containing subtree equivalence classes with sufficient support, as illustrated in the following example. In general, the *findGroups* operation defines new sub-trees T_{GROUP} with roots labeled $GROUP_k$, where $GROUP$ is a non-terminal symbol and k is an attribute identifying equivalent groupings. The selection order of equivalence classes in *equivalent_st* is determined by the tree depths within each class.



(a) Initial tree T **(b) Tree T'** **(c) Tree T''**

Figure 11: Example of the findGroups operation

Example 5.14. Let T be the tree in Figure 11a where entity sub-trees are those with roots labeled E_i. Let $D/\equiv_\tau = \{\{0\}, \{1\}, \{00, 110\}, \{01\}, \{010, 111\}, \{11\}, \{1100\}\}$ and $minSupport = 2$. Consequently, we have $equivalent\_st = \{\{00, 110\}, \{010, 111\}\}$.

The nodes 010 and 111 are relabeled as $GROUP_0$, while the nodes 00 and 110 are assigned the label $GROUP_1$. Node 1100 is deleted to ensure only entities remain as children of group-nodes, resulting in the updated tree T'' (Figure 11c). □.

• Unifying groups includes to operations: finding sub-groups and merging them. The *findSubGroups* operation aims at minimizing distinct groups and maximizes their frequencies. It ensures no group contains a more frequent subgroup. This operation is also applied to sub-trees whose root does not yet have a label corresponding to a non-terminal of the target grammar. If a more frequent sub-tree st_i can be constructed from a subset of the entity trees descending from a given sub-tree st, st is replaced by a new unlabeled sub-tree. The children of this new sub-tree are st_i and the sub-trees in st that are not part of st_i, as illustrated in the following example.

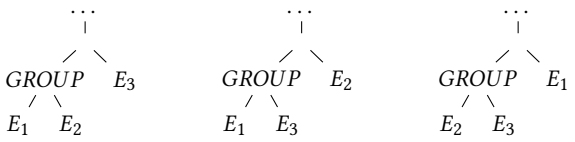
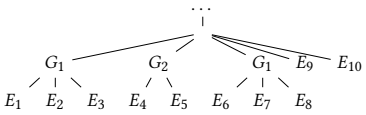


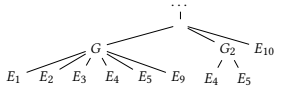
Figure 12: Example of the findSubgroups operation

Example 5.15. Given the tree T in Figure 13a, let st represent the sub-tree for the first group G_1. To check if a subgroup is more frequent than G_1, we test all possible subsets of entities. With 3 entity sub-trees in st, we test groupings of 2. The results for these groupings are shown in Figure 12. □

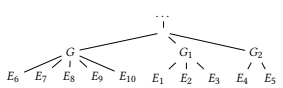
The goal of the *mergeGroups* operation is to increase group size by merging sibling groups or adding sibling entity sub-trees. This operation only applies to groups not already in a relation or collection. For each sub-tree st, two sets are considered: S_{GROUP} (children labeled *GROUP*) and S_{ENT} (children labeled *ENT*). The operation explores all combinations between these sets, starting with the largest. Transformations are applied only if they produce more frequent sub-trees than the original. The following example illustrates this process.



(a) Sub-tree st



(b) Sub-tree st_0 with the combination $\{u0, u1, u3\}$

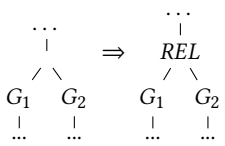


(c) Sub-tree st_1 with the combination $\{u2, u3, u4\}$

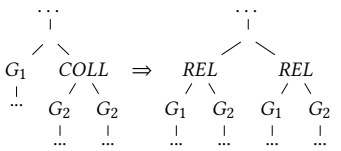
Figure 13: Example of the mergeGroups operation

Example 5.16. Let T be the tree in Figure 13a. The root of sub-tree st is at position u and is not yet labelled with a target grammar symbol. The operation attempts to merge children of st. We have $S_G = \{u0, u1, u2\}$ and $S_{ENT} = \{u3, u4\}$. The number of testable combinations depends on $|S_G/\equiv_\tau|$ (Definition 5.13), which is the number of elements in S_G, excluding those in the same equivalence class. Since $u0$ and $u2$ (both labeled G_1) are in the same equivalence class, the *mergeGroups* operation will attempt to build sub-trees by grouping 4 (= $|S_G/\equiv_\tau| + |S_{ENT}|$) positions as siblings, then 3 positions, and so on. All possible combinations are tested by creating a new tree with the new group and calculating its support. With 3 positions the combinations to be tested are: $\{\{u0, u1, u3\}, \{u0, u1, u4\}, \{u0, u3, u4\}, \{u1, u2, u3\}, \{u1, u2, u4\}, \{u1, u3, u4\}, \{u2, u3, u4\}\}$. Two of these combinations are shown in Figure 13. □

• The operation *findRelations* creates relationships on unlabelled sub-trees in two ways: (1) If the sub-tree has two children labelled *GROUP*, the sub-tree's root is labelled *REL* (Figure 14a). (2) If the sub-tree has one child labelled *GROUP* and another labelled *COLL*, the sub-tree is restructured. It creates a new sub-tree with root *REL* having two children: the child labelled *GROUP* and another sub-tree derived from the *COLL* child. This transformation is distributive for each child of *COLL* (Figure 14b).



(a) Relationship between 2 groups



(b) Relationships between 1 group and a collection

Figure 14: Example of the findRelationship operation

• The *findCollections* operation groups sibling sub-trees (either groups or relations) into a collection if they belong to the same equivalence class. Here we explain how group collections are formed. Relations are handled similarly. For each *unlabelled* sub-tree st, the operation works in three steps (Figure 15): (1) It creates a subtree labeled *COLL* with all *GROUP* subtrees from the same equivalence class as its children; (2) It groups collections containing children from the same equivalence class; (3) It adds child groups of st to the collection containing equivalent groups.

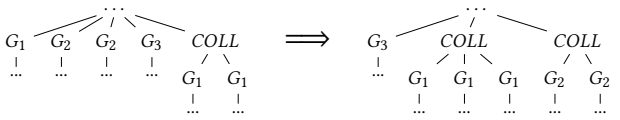


Figure 15: Example of the findCollections operation

6 PROOF OF CONCEPT

Since approaches in the literature are not directly comparable to ours, we evaluate our prototype based on how well it meets our objectives. We tested our structuring method with a proof-of-concept use case using the CAS corpus [13], which contains real and fictitious clinical cases describing patients' medical histories, symptoms, diagnoses, and treatments. We chose a small example for this initial experiment – a corpus of 100 texts and 8098 manually annotated named entities across 10 categories, with some entities potentially nested – to carefully track each step of our method.

Evaluation Methodology. We track the instance's evolution by comparing each iteration of Algorithm 1. The rewriting function (line 5, Section 5.5) alternates between specializing (making details explicit) and generalizing (aggregating). Due to its behavior, only one operation is triggered per iteration, which can delay annotating nodes as *relation* or *collection*, even when conditions are met. To evaluate progress, we count valid structures at the end of the while loop, even if they aren't fully labelled yet.

The following aspects are guidelines for the analysis of the behavior of our approach:

(1) *The impact of our approach on the reduction of production rules.* A comprehensible grammar G_T should be more concise than G_0, which directly represents the original instance. Reducing the number of production rules reflects aggregation.
(2) *The average number of instances for core structures defined by* $\mathbb{G}$. Our aggregations use *groups*, *relations* and *collections* as the core structures of our schema, and we expect their instances to increase with each iteration. For example, we track how often a non-terminal name such as $GROUP_9$ appears in the tree. Ideally, you would expect to see a significant number of occurrences.
(3) *The hierarchy evolution.* Our structures follow a defined hierarchy, with, for instance, *collections* aggregating *groups* for greater generalization. During the iterations of Algorithm 1 we monitor the number of structures in each category: *groups*, *relations*, and *collections*.

Experiments. Figure 16 illustrates the behavior of Algorithm 1 in reducing the number of production rules (aspect 1). It shows the number of rules in grammar G_i (——) along with its trend line (- - -), and the number of uncategorized nodes (——) with its trend line (- - -). The lower part of the graph shows the transformation applied at each step i ( ). Operations number correspond to their order : (1) findSubgroups; (2) mergeGroups; (3) findCollectionsOfGroups; (4) findRelationships; (5) findCollectionsOfRelationships; (6) reduce(bottom); (7) reduce(top).

The algorithm effectively minimizes the grammar, reducing production rules from 178 to 68, and uncategorized nodes from 155 to 34 after 50 iterations.

A detailed analysis of Figure 16 reveals moments where the number of production rules remains relatively constant, such as in iterations 21 and 25. During these phases, Algorithm 1 applies *generalistion* operations, leading to significant changes in the number of rules. However, these operations do not produce valid or frequent trees, requiring subsequent restructuring. This is where the previously mentioned back-and-forth behavior occurs. In steps 24, 30, and 31, the number of production rules rises sharply as the procedure introduces more specific structures to later recognize more generic ones, like collections.

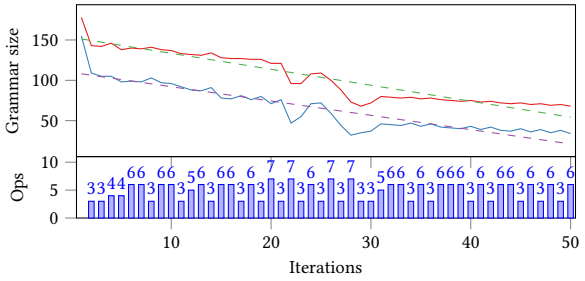


Figure 16: Evolution of the grammar during our approach.

Figure 17 shows the behavior of Algorithm 1 regarding the number of instances for each structure (aspect 2). The curve —— represents the number of equivalence classes at each step, which decreases from 25 to 17. This suggests that certain structures unify over time, with the transformations successfully remodeling different groups that eventually fall into the same equivalence class. The curves ——, ——, and —— show the average number of instances for each *group*, *relation*, and *collection*, respectively. Despite the decrease in unlabeled nodes, these numbers remain relatively stable. Initially, the average number of instances is 34.2 for groups and 2.2 for relations. By the end, groups have an average of 29.8 instances and relations 3. The drop in group instances is due to their promotion to relations, increasing the average for relations. The decrease in collection instances is explained by successive merging, resulting in 1.4 instances per collection after 50 iterations. Ideally, there should be 1 collection per *group* and per *relation*, meaning all collections have been successfully merged.

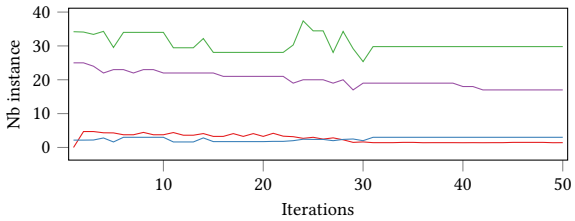


Figure 17: Mean number of instance for each structure

Figure 18 illustrates the behavior of Algorithm 1 regarding the number of structures per category (aspect 3). Specifically, —— shows the number of groups, curve —— the relations, and —— the collections. In instance I_0 (iteration 1), 10 groups, 6 relations, and 0 collections are formed. By iteration 28, the procedure creates 11 groups, 8 relations, and 15 collections, with the fewest production rules. Iterations 29 and 30 merge collections after the tree's top reduction in iteration 28(see Section 5.5). This reduces relations to 2, with a third added at iteration 31, due to the reduction in groups from 15 to 10. Fewer groups lead to fewer relations, but more instances per relation as equivalence classes merge. A similar unification occurs at iteration 21. The procedure concludes with 10 groups, 3 relations, and 12 collections (nearly one collection per group and relation). After iteration 31, the number of groups and collections stabilizes, indicating the structure effectively represents the data. Subsequent steps only modify or refine the instance without altering the schema significantly.

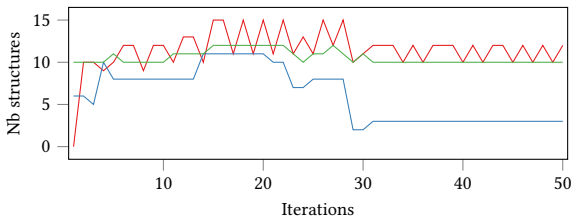


Figure 18: Number of each kind of structures

Reproducibility. The procedures are described in detail in [14]. The implementation is written in Python and utilizes CoreNLP for parsing. The source code, required to reproduce the experiments, can be found on GitHub [6]. For the corpus, [13] provides the data access modalities.

Evaluating the rewriting policy. The policy of Algorithm 1 for identifying frequent elements converges to a satisfactory solution but is highly sensitive to parameters like *minSup* or τ. For example, consider changing τ from 0.5 to 0.7 on a tree where a node has X- rooted and Y-rooted sub-trees as children, both containing $GROUP_7$ and *Anatomy* as their children. We might expect the *mergeGroups* operation, mentioned in Section 5.5, to merge $GROUP_7$ with *Anatomy*. However, this merges fails. Adding *Anatomy* to $GROUP_7$ places the group too far from other instances, creating a set with insufficient support. This happens because transformations are made individually, not collectively; simultaneous modifications could have enabled the merge.

Evaluating the resulting grammar. The target grammar G_T, obtained from our experiments on the CAS corpus, is valid with respect to $\mathbb{G}$ and consists of 26 production rules (12 collections, 3 relations, 10 groups, and 9 entities). Analyzing the groups and their semantics reveals expected associations, such as: $GROUP_0 \rightarrow ENT_{Dose}\ ENT_{Frequence}\ ENT_{Mode}\ ENT_{Substance}\ ENT_{Sosy}\ ENT_{Treatment}$ indicating a treatment; $GROUP_1 \rightarrow ENT_{Exam}\ ENT_{Value}$; and $GROUP_3 \rightarrow ENT_{Dose}\ ENT_{Exam}\ ENT_{Sosy}\ ENT_{Substance}$ indicating different formats for exam results.

The results are very promising: the obtained generic schema is coherent and reliable. While differences with human analysis are noted and will be explored in the following discussion, it's important to understand that our method relies primarily on the frequency of syntactic structures, with semantic information mainly derived from the enrichment step. Consequently, although the results are coherent and logical, they do not always fit perfectly with an intuitive database model. Here are some examples. (1) G_T includes the production rules: $GROUP_4 \rightarrow ENT_{Anatomy}$, ENT_{Exam}, ENT_{Sosy} and $GROUP_8 \rightarrow ENT_{Exam}$, ENT_{Sosy}. While domain knowledge may suggest splitting them into two groups — one for exams (*Exam*, *Anatomy*) and one for symptoms (*Sosy*, *Anatomy*) — connected by a relation, the corpus shows these entities are often interdependent with few external links. Grouping them together is therefore consistent, reducing grammar complexity by avoiding extra groups and relations. (2) A prescription typically includes elements like *Treatment*, *Substance*, *Dose*, *Mode*, and *Frequency*, related to a symptom (*Sosy*). In the corpus, treatments are almost always tied to symptoms, naturally forming $GROUP_0$ with both. While a database model might separate the symptom into its own group to connect it to both exams and treatments, our approach doesn't account for such domain-specific structures. Instead, it combines them (e.g., production rules for $GROUP_0$, $GROUP_4$ and $GROUP_8$ above), reducing grammar size by avoiding extra relations. If the corpus had more isolated instances of *Sosy*, our method might have created a distinct *Sosy* group, leading to more relations. (3) G_T provides various production rules to represent examination results, such as $GROUP_3 \rightarrow ENT_{Dose}\ ENT_{Examination}\ ENT_{Sosy}\ ENT_{Substance}$ and those for groups 1, 4 and 8 (mentioned above). Although this can be criticized, each group reflects different types of examination: some are only mentioned (group 4), some result in a value (group 1) and others measure a dose (group 3). In particular, $GROUP_3$ is ambiguous and may also represent post-test treatments. As in the previous point, the presence of *Sosy* could also be handled by a relation. (4) For our experiments, the resulting G_T offers few production rules for relations. In some cases we might question whether a relation should be classified as a group or vice versa. Indeed, our approach may decide to define a relation between groups representing entities with many missing instances, rather than grouping these entities together. This highlights that, in this case, while the semantics may be similar, the similarity of the corresponding sub-trees is quite low.

The resulting G_T defines a general database structure, which can be used to implement various database models. Our approach (and this paper) does not cover the next step - selecting a specific database model based on G_T. This step should account for semantic and performance factors, as discussed in Section 3. In our example, a database analyst could convert G_T into a relational schema with 5 tables: $Prescription[treatment, mode, substance, dose, frequency, sosy]$ (describes the prescription of a substance or treatment for a symptom); $Examination[examId, exam, value]$ and $Measure[examId, exam, dose, substance]$ (represent two types of exams: basic one the those measuring substance levels e.g., in blood); $forAnat[examId, anatomy]$ (linking exams to part of the body) and $forSosy[examId, sosy]$ (linking exams to signs or symptoms). No table links examinations/symptoms to treatments - they typically appear in different sentences and the system is limited to sentence-level associations.

7 CONCLUDING REMARKS

ArchiTXT aims to organize unstructured data into a flexible structure that abstracts different database models. Currently, the resulting grammar G_T can be used as input for other methods that focus on semantic and performance aspects to propose a specific database model. The weakness of ArchiTXT, noted in Section 3, stems from the use of limited semantic information. To address this, enhancements to ArchiTXT could include incorporating semantics through tree rewriting strategies or within the meta-grammar $\mathbb{G}$, taking into account business rules and functional dependencies. Additionally, integrating functional dependency discovery, as explored in [28], could further refine the structuring process through semantic analysis Future work also includes extending the proof of concept, improving performance, and exploring incremental structuring methods.

The strength of our approach lies in discovering a schema while structuring the data, making it easier to create database instances from textual data. Avoiding targeting one precise database model ArchiTXT incorporates a multi-model philosophy. ArchiTXT stands apart from both traditional and machine learning information extraction methods by offering a hybrid strategy that provides some flexibility across application domains without the need for training data. Its transparent process allows users to track and validate each step, ensuring confidence in both the system and the quality of the data.

Finally, our paper presents an innovative generic approach to structuring textual data using tree rewritings and grammar extractions guided by a meta-model defined by an attribute grammar $\mathbb{G}$. Since $\mathbb{G}$ can specify different target structures, ArchiTXT serves as an instantiation of this methodology.

8 ACKNOWLEDGMENTS

We acknowledge the support of the APR-IA DOING project and the DATA project from ARD-JUNON.

REFERENCES

[1] Mohd Hafizul Afifi Abdullah, Norshakirah Aziz, Said Jadid Abdulkadir, Hitham Seddig Alhassan Alhussian, and Noureen Talpur. 2023. Systematic literature review of information extraction from textual data: recent methods, applications, trends, and challenges. *IEEE Access* 11 (2023), 10535–10562.

[2] Fatima N. Al-Aswadi, Huah Yong Chan, and Keng Hoon Gan. 2020. Automatic Ontology Construction from Text: A Review from Shallow to Deep Learning Trend. *Artificial Intelligence Review* 53, 6 (Aug. 2020), 3901–3928. https://doi.org/10.1007/s10462-019-09782-9

[3] Nelly Barret, Ioana Manolescu, and Prajna Upadhyay. 2022. Abstra: Toward Generic Abstractions for Data of Any Model. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management (CIKM '22)*. Association for Computing Machinery, Atlanta GA USA, 4803–4807. https://doi.org/10.1145/3511808.3557179

[4] Abel Browarnik and Oded Maimon. 2015. Ontology Learning from Text: Why the Ontology Learning Layer Cake Is Not Viable. *International Journal of Signs and Semiotic Systems (IJSSS)* 4, 2 (2015), 1–14. https://doi.org/10.4018/IJSSS.2015070101

[5] Gunnar Carlsson and Facundo Mémoli. 2010. Characterization, Stability and Convergence of Hierarchical Clustering Methods. *Journal of Machine Learning Research* 11, Apr (Aug. 2010), 1425–1470. https://doi.org/10.5555/1756006.1859898

[6] Jacques Chabin, Mirian Halfeld-Ferrari, and Nicolas Hiot. 2024. ArchiTXT. https://github.com/Neplex/ArchiTXT

[7] Jacques Chabin, Mirian Halfeld-Ferrari, Nicolas Hiot, and Dominique Laurent. 2023. Managing Linked Nulls in Property Graphs: Tools to Ensure Consistency and Reduce Redundancy. In *Advances in Databases and Information Systems (Lecture Notes in Computer Science, Vol. 13985)*. Springer Nature Switzerland, Cham, 180–194. https://doi.org/10.1007/978-3-031-42914-9_13

[8] Jacques Chabin, Mirian Halfeld-Ferrari, and Dominique Laurent. 2020. Consistent Updating of Databases with Marked Nulls. *Knowledge and Information Systems (KAIS)* 62, 4 (April 2020), 1571–1609. https://doi.org/10.1007/s10115-019-01402-w

[9] Mengmeng Cui, Ruibin Huang, Zhichen Hu, Fan Xia, Xiaolong Xu, and Lianyong Qi. 2024. Semantic rule-based information extraction for meteorological reports. *International Journal of Machine Learning and Cybernetics* 15, 1 (2024), 177–188.

[10] John Dagdelen, Alexander Dunn, Sanghoon Lee, Nicholas Walker, Andrew S Rosen, Gerbrand Ceder, Kristin A Persson, and Anubhav Jain. 2024. Structured information extraction from scientific text with large language models. *Nature Communications* 15, 1 (2024), 1418.

[11] Wenfei Fan, Ping Lu, Kehan Pang, Ruochun Jin, and Wenyuan Yu. 2024. Linking Entities across Relations and Graphs. In *ACM Transactions on Database Systems*, Vol. 49. 1–50. https://doi.org/10.1145/3639363

[12] Pablo Gamallo, Marco González, Alexandre Agustini, Gabriel Lopes, and Vera Lúcia Strube de Lima. 2002. Mapping Syntactic Dependencies onto Semantic Relations. In *Proceedings of the ECAI Workshop on Machine Learning and Natural Language Processing for Ontology Engineering*. 15–22. https://www.semanticscholar.org/paper/Mapping-Syntactic-Dependencies-onto-Semantic-Gamallo-Gonz%C3%A1lez/45ee40319a8b57f2fe7ca3ecc33ee78b97697c92

[13] Natalia Grabar, Vincent Claveau, and Clément Dalloux. 2018. CAS: French Corpus with Clinical Cases. In *Proceedings of the Ninth International Workshop on Health Text Mining and Information Analysis*. Association for Computational Linguistics, Brussels, Belgium, 122–128. https://doi.org/10.18653/v1/w18-5614

[14] Nicolas Hiot. 2024. *Construction automatique de bases de données pour le domaine médical : Intégration de texte et maintien de la cohérence.* Ph. D. Dissertation. Orléans. https://theses.fr/s265149

[15] Tomasz Imielinski and Witold Lipski Jr. 1984. Incomplete Information in Relational Databases. *Journal of the ACM (JACM)* 31, 4 (Sept. 1984), 761–791. https://doi.org/10.1145/1634.1886

[16] Dan Jurafsky and James H. Martin. 2008. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition* (2nd edition ed.). Vol. 4. Prentice Hall, Upper Saddle River, N.J. https://books.google.fr/books?id=fZmj5UNK8AQC

[17] Imed Keraghel, Stanislas Morbieu, and Mohamed Nadif. 2024. A survey on recent advances in named entity recognition. *arXiv preprint arXiv:2401.10825* (2024).

[18] Donald E. Knuth. 1968. Semantics of Context-Free Languages. *Mathematical Systems Theory* 2, 2 (June 1968), 127–145. https://doi.org/10.1007/BF01692511

[19] Belinda Z. Li, Sewon Min, Srinivasan Iyer, Yashar Mehdad, and Wen-tau Yih. 2020. Efficient One-Pass End-to-End Entity Linking for Questions. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Online, 6433–6441. https://doi.org/10.18653/v1/2020.emnlp-main.522 arXiv:2010.02413 [cs]

[20] Jing Li, Aixin Sun, Jianglei Han, and Chenliang Li. 2020. A survey on deep learning for named entity recognition. *IEEE transactions on knowledge and data engineering* 34, 1 (2020), 50–70.

[21] Leonid Libkin. 2006. Data exchange and incomplete information. In *Proceedings of the Twenty-Fifth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 26-28, 2006, Chicago, Illinois, USA*. 60–69.

[22] Ming Liu, Lei Chen, Bingquan Liu, Guidong Zheng, and Xiaoming Zhang. 2018. DBpedia-Based Entity Linking via Greedy Search and Adjusted Monte Carlo Random Walk. *ACM Transactions on Information Systems* 36, 2 (April 2018), 1–34. https://doi.org/10.1145/3086703

[23] Pai Liu, Wenyang Gao, Wenjie Dong, Songfang Huang, and Yue Zhang. 2022. Open information extraction from 2007 to 2022–a survey. *arXiv preprint arXiv:2208.08690* (2022).

[24] Jihane Mali, Shohreh Ahvar, Faten Atigui, Ahmed Azough, and Nicolas Travers. 2024. FACT-DM: A Framework for Automated Cost-Based Data Model Transformation. In *Proceedings 27th International Conference on Extending Database Technology, EDBT 2024, Paestum, Italy, March 25 - March 28*. OpenProceedings.org, 822–825. https://doi.org/10.48786/EDBT.2024.79

[25] Anne-Lyse Minard, Andréane Roques, Nicolas Hiot, Mirian Halfeld-Ferrari, and Agata Savary. 2020. DOING@ DEFT: cascade de CRF pour l'annotation d'entités cliniques imbriquées. In *6e conférence conjointe Journées d'Études sur la Parole (JEP, 33e édition), Traitement Automatique des Langues Naturelles (TALN, 27e édition), Rencontre des Étudiants Chercheurs en Informatique pour le Traitement Automatique des Langues (RÉCITAL, 22e édition). Atelier DÉfi Fouille de Textes*. ATALA et AFCP, Nancy, France, 66–78. https://hal.science/hal-02784743

[26] Emmanuel Morin. 1999. Automatic Acquisition of Semantic Relations between Terms from Technical Corpora. In *Proc. of the Fifth International Congress on Terminology and Knowledge Engineering-TKE'99*.

[27] Roberto Navigli. 2024. Word Sense Disambiguation: A Survey. *ACM computing surveys (CSUR)* 41, 2 (2024), 1–69. https://doi.org/10/dhsjt2

[28] Thorsten Papenbrock, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener, and Felix Naumann. 2015. Functional Dependency Discovery: An Experimental Evaluation of Seven Algorithms. *Proceedings of the VLDB Endowment* 8, 10 (June 2015), 1082–1093. https://doi.org/10.14778/2794367.2794377

[29] Ahmad Sakor, Kuldeep Singh, Anery Patel, and Maria-Esther Vidal. 2020. Falcon 2.0: An Entity and Relation Linking Tool over Wikidata. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management (CIKM '20)*. Association for Computing Machinery, New York, NY, USA, 3141–3148. https://doi.org/10.1145/3340531.3412777 arXiv:1912.11270 [cs]

[30] Agata Savary, Alena Silvanovich, Anne-Lyse Minard, Nicolas Hiot, and Mirian Halfeld-Ferrari. 2022. Relation Extraction from Clinical Cases for a Knowledge Graph. In *New Trends in Database and Information Systems (Communications in Computer and Information Science, Vol. 1652)*. Springer International Publishing, Turin, Italy, 353–365. https://doi.org/10.1007/978-3-031-15743-1_33

[31] Wei Shen, Jiawei Han, and Jianyong Wang. 2014. A Probabilistic Model for Linking Named Entities in Web Text with Heterogeneous Information Networks. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 1199–1210. https://doi.org/10.1145/2588555.2593676

[32] Marco A Valenzuela-Escárcega, Gus Hahn-Powell, and Dane Bell. 2020. Odinson: A fast rule-based information extraction framework. In *Proceedings of the Twelfth Language Resources and Evaluation Conference*. 2183–2191.

[33] Douglas R. White and Karl P. Reitz. 1983. Graph and Semigroup Homomorphisms on Networks of Relations. *Social Networks* 5, 2 (June 1983), 193–234. https://doi.org/10.1016/0378-8733(83)90025-4

[34] Gwendolin Wilke, Sandro Emmenegger, Jonas Lutz, and Michael Kaufmann. 2016. Merging bottom-up and top-down knowledge graphs for intuitive knowledge browsing. In *Flexible Query Answering Systems 2015: Proceedings of the 11th International Conference FQAS 2015, Cracow, Poland, October 26-28, 2015*. Springer, 445–459.

[35] Bowen Yu, Zhenyu Zhang, Jiawei Sheng, Tingwen Liu, Yubin Wang, Yucheng Wang, and Bin Wang. 2021. Semi-open information extraction. In *Proceedings of the Web Conference 2021*. 1661–1672.

[36] Kaizhong Zhang and Dennis Shasha. 1989. Simple Fast Algorithms for the Editing Distance between Trees and Related Problems. *SIAM J. Comput.* 18, 6 (Dec. 1989), 1245–1262. https://doi.org/10.1137/0218082

[37] Shaowen Zhou, Bowen Yu, Aixin Sun, Cheng Long, Jingyang Li, Haiyang Yu, Jian Sun, and Yongbin Li. 2022. A survey on neural open information extraction: Current status and future directions. *arXiv preprint arXiv:2205.11725* (2022).

[38] Yutao Zhu, Huaying Yuan, Shuting Wang, Jiongnan Liu, Wenhan Liu, Chenlong Deng, Haonan Chen, Zhicheng Dou, and Ji-Rong Wen. 2023. Large language models for information retrieval: A survey. *arXiv preprint arXiv:2308.07107* (2023).