Scalable Reasoning on Document Stores via Instance-Aware Query Rewriting

Olivier Rodriguez
LIRMM, Inria, Univ. Montpellier, CNRS
Montpellier, France
olivier.rodriguez@inria.fr

Federico Ulliana
LIRMM, Inria, Univ. Montpellier, CNRS
Montpellier, France
federico.ulliana@inria.fr

Marie-Laure Mugnier
LIRMM, Inria, Univ. Montpellier, CNRS
Montpellier, France
marie-laure.mugnier@inria.fr

ABSTRACT

Data trees, typically encoded in JSON, are ubiquitous in data-driven applications. This ubiquity makes urgent the development of novel techniques for querying heterogeneous JSON data in a flexible manner. We propose a rule language for JSON, called constrained tree-rules, whose purpose is to provide a high-level unified view of heterogeneous JSON data and infer implicit information. As reasoning with constrained tree-rules is undecidable, we identify a relevant subset featuring tractable query answering, for which we design an automata-based query rewriting algorithm. Our approach consists of leveraging NoSQL document stores by means of a novel instance-aware query-rewriting technique. We present an extensive experimental analysis on large collections of several million JSON records. Our results show the importance of instance-aware rewriting as well as the efficiency and scalability of our approach.

PVLDB Reference Format:
Olivier Rodriguez, Federico Ulliana, and Marie-Laure Mugnier. Scalable Reasoning on Document Stores via Instance-Aware Query Rewriting. PVLDB, 16(11): 2699 - 2713, 2023. doi:10.14778/3611479.3611481

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://gitlab.inria.fr/boreal-artifacts/pvldb2023.

1 INTRODUCTION

Rule-based languages have been studied by the database community for more than four decades because of their importance in processing enterprise data and knowledge [8]. In the recent years, companies like LogicBlox and Relational-AI commercialized full-fledged solutions for rule-based data processing and explainable AI [13, 20]. At the same time, companies like Linkedin, Google, Facebook, and Samsung, among others, developed their own rule engines for reasoning on data [43, 66, 80]. The interest in such formalisms comes from *declarativity* and *expressivity*, which make rules a universal form of knowledge suitable for many tasks such as data-integration [46], recursive queries [8], ontologies and semantic constraints [35], data quality and data preparation for feeding machine learning and analytic tools [20, 51].

Proceedings of the VLDB Endowment, Vol. 16, No. 11 ISSN 2150-8097.
doi:10.14778/3611479.3611481

The variety of data that can be handled by rule languages led to the design of solutions for reasoning on knowledge graphs and hypergraphs [20, 41, 68]. Nevertheless, in many practical cases, data exhibits a simpler structure. There is a sheer amount of semi-structured data held by transactional systems, data warehouses, and data lakes, which takes the form of *data trees*. Today, this data is typically serialized as JSON. Beside this, data trees also permeate the whole Web, where JSON is exchanged at a massive rate. As a bottom line, JSON trees are ubiquitous in data-oriented applications, hence among the most practically relevant formats today. This ubiquity makes urgent the development of novel techniques for **querying heterogeneous JSON data in a flexible manner**.

Towards this aim, we study a rule language for JSON suitable to: *(1)* act as a *mediating level* to integrate heterogeneous JSON data; *(2)* allow one to *enrich the querying vocabulary*, hence to adapt it to specific use-cases and free the end-users from mastering the complexity and irregularity of data; *(3)* be equipped with *reasoning capabilities*, i.e., be able to infer information not explicitly stored in the data thereby bringing novel and pertinent answers to the user queries. In the quest for such language, a crucial requirement is that it must enable efficient and scalable query answering techniques. Our new approach is to use *NoSQL document stores* for query-answering over data-trees together with *instance-aware query-rewriting* techniques to build **a scalable and efficient end-to-end framework for reasoning on JSON databases.**

Example 1.1. Figure 1 shows three JSON trees (pictured in black solid lines) T_1, T_2, T_3 holding GitHub data from the public archive [2]. This collection built for data analysis contains 17 types of events stemming from user actions. Data is complex and irregular. For a data-scientist willing to exploit the archive, tasks like *i*) running short explorative queries or *ii*) extracting a training-set for machine learning algorithms can both be extremely time consuming, because of the time required to formulate the "right" queries to the data. To illustrate, a simple information such as the name (login) of a user can appear within different JSON keys (e.g., `author` for commits, `actor` for push, `user` for issues, and more) and within more than 60 different paths in the trees (e.g., `actor.login`, `payload.commits`, `payload.issue`, etc). Let us consider the query q, which concerns the activity of Linus Torvalds (`torvalds`). It retrieves the `id` of the `events` where the `user` participated, through the answer variable x. This query is empty on all trees, as it has no matches. Indeed, to query the collection, *one must first learn the irregular JSON structure.* In contrast, rules $r_1 - r_6$ can be added (on benefit of all end-users) on top of data thereby providing a unified high-level vision of the JSON records. Rules $r_4 - r_6$ introduce the high-level notion of `event` from specific types of events (push, commit, issues), which

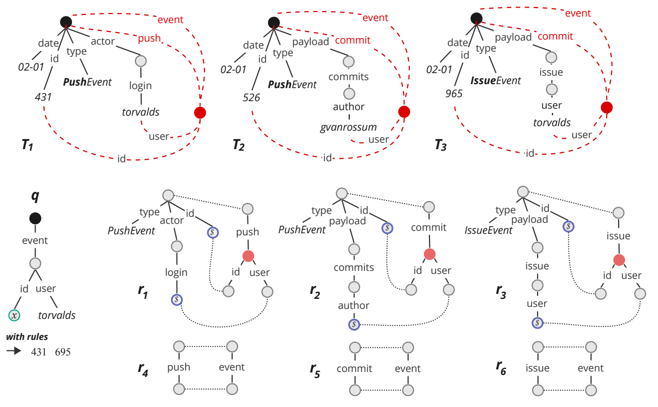

Figure 1: Rule-based Querying of GitHub Data

is handy for queries. Rules $r_1 - r_3$ abstract on the structure of data to define high-level notions of the same types of events, including the id and the user involved. Altogether, the rules enable implicit inferences (pictured as red dashed edges and nodes) which provide answers to q from T_1 and T_3 *without requiring users to master the whole variability of data.* Note that, as pictured, these are *tree rules*. The left-hand-side (the body) encodes a tree structure to match, while the right-hand-side (the head) encodes the inferences. Nodes shared between the body and the head are linked by dotted lines. Nodes that belong to the head only (in red), called *existential nodes*, extend the tree by introducing fresh nodes; they are necessary to restructure data. Further, as we shall see later, rules can be *recursive*.

As already said, querying heterogeneous JSON trees in a flexible manner is critical for many data processing and governance tasks. These include scenarios where *one-of* operations are executed, e.g., the preparation and extraction of training-sets from heterogeneous data [20, 51]. These include scenarios where *consecutive queries* are run and users expect online answers, e.g., in goal-driven explorative queries [8, 22, 78]. In both cases, the solution is to use reasoning algorithms based on *materialization* and *query-rewriting* techniques [8, 27]. These are different approaches complementing each other, for both theoretical and practical concerns [11, 16, 34, 52]. First, each can handle only certain rule fragments; this, regardless of the applicative scenarios. Indeed, both suffer from *non-termination* issues for languages such as Datalog$^{\pm}$ (a.k.a. *existential rules* or Tuple-Generating-Dependencies) [16, 19, 34]. This matters here, as tree rules like those of Example 1.1 are related to these formalisms. Second, assuming termination, every method best fits certain applicative scenarios and hardware constraints.

Materialization consists at extending the database with the inferences produced by rules, as illustrated by Figure 1. In-memory materialization is regarded as the most effective approach. Systems based on this approach exhibit extremely high-performances because of their reasoning-oriented algorithms and data-structures [20, 41, 68]. Yet, in-memory materialization is tied to the capacity at storing inferences (or even loading data) in memory. Concretely, this may confine the use of the technique to powerful hardware even for medium size datasets. Materialization can also be realized with a DBMS backend but it can be much less performant [11, 21].

Rewriting consists at propagating the rules *into the query* thereby leaving the data untouched. The aim is to produce a set of queries whose evaluation on the data yields the same answers as the evaluation of the input query on the extended database. Figure 2 illustrates the rewriting of query q from Example 1.1. In spirit, the rewriting process replaces parts of a query matching a rule head with the rule body itself. The advantages of query rewriting are multiple. First, it requires no additional space to store inferences. Second, it is insensitive to data updates. Third, it allows one to reason with read-only access rights on the database. The most important however is that rewriting can be deployed on top of *existing database technology* [32, 37, 51, 64]. This has two main advantages: it makes the technique resilient to main-memory limitations, as data is stored on disk and inferences not stored at all, and it allows one to delegate query evaluation to the DBMS. On this matter, JSON has been adopted as a data model by several NoSQL document stores, often praised for their performances (e.g., MongoDB, CouchDB, ArangoDB, to cite a few). This opens up for the opportunity of *building a novel family of reasoners relying on query rewriting and NoSQL database technology*. Indeed, these systems come with adapted data layouts, indexes, query algorithms, buffering, scheduling, and concurrent data access mechanisms among others. All these components can work together to make reasoning on JSON both *efficient and scalable*. Note that aiming at materialization on top of document stores would almost inevitably lead to data blowups and no scalability because of the tree data-model of JSON [67]. For these systems, *query rewriting is the prominent way to go.*

Contribution. Designing a rewriting-based reasoner on NoSQL document stores has been very little investigated so far [23, 24, 67]. This task poses significant technical challenges, namely identifying *i*) adequate languages for the task, together with *ii*) algorithms and optimizations to make query answering efficient and scalable.

Concerning the first point, to start with, one needs a query and rule language that is *closed*, i.e., *it generates only rewritings that can be handled by the underlying NoSQL system.* This is subtle, as NoSQL systems often ensure efficient access to data by reducing the expressivity of queries to rooted path- or tree-shaped queries or forbidding joins. Here, we shall present a language of *constrained* tree queries and rules enjoying this property. Then, *the rewriting process must be finite*, which can limit the use recursion in rules. Rewriting recursive rules (e.g., Datalog [8]) is generally disregarded, and the typical solution is to restrict the language to ensure that the set of rewritings of any query is finite [34, 36, 38]. Here, we take another approach and rather try to cope with infinite rewriting sets, while *encoding them finitely.* Finally, query answering with the target rule language must have *tractable data complexity* (i.e., polynomial complexity in the size of the data) which is the minimum requirement for efficiency over large data [20].

Concerning the second point, to design algorithms and optimizations for efficient and scalable reasoning, we believe that query rewriting process *cannot be agnostic to the underlying database.* Therefore, our goal is to couple query rewriting with an innovative *instance-aware* evaluation strategy. This leverages on a combination of data-summaries, partitioning, and parallelization that allow us to achieve efficiency by *distributing the evaluation of rewritings where the matching data is* and at the same time achieving scalability on *very large collections of JSON data* thanks to the NoSQL facilities.

Summing up, our contributions are the following.

1. We introduce a novel language of *constrained rules and queries* for reasoning on JSON and present a (general) sound and complete *query rewriting algorithm* for this setting.

2. As reasoning with constrained rules is undecidable, we identify a relevant subset, made of *relabeling* and *frontier-constrained* rules, featuring decidable query answering with PTime data-complexity. We design an automata-based rewriting algorithm for such rules.

3. We propose a novel set of *instance-aware* evaluation techniques that - for a fixed database - allow one to efficiently evaluate large rewritings sets and also scale on large collections of data.

4. We present an extensive *experimental analysis* on large collections of several million JSON records showing the efficiency of our approach, and its scalability beyond state-of-the-art reasoners.

Paper Organization. We introduce our framework in Section 2 then give a high-level presentation of our query answering techniques (Sections 3 and 4). Sections 5 and 6 are devoted to formal developments on query rewriting algorithms. The experimental analysis is presented in Section 7. Section 8 discusses related work. Proofs and further details can be found in the technical report [69].

2 FRAMEWORK

We begin by presenting an abstract setting for reasoning on JSON. We start by posing the notion of tree which will be used throughout the paper. Then, we introduce a new language made of *constrained tree rules and queries* and define their semantics.

Briefly, a JSON record is a set of key-value pairs, where a value is recursively defined as a terminal value (constant or null), a sequence of values, or a record. We see a record as a *rooted labeled unordered tree*, in which edges are labeled by keys, leaves are labeled by constant or null values, and all the internal nodes are unlabeled. Note that a key-value pair (k, v) where v is a sequence is represented by edges labeled by k leading to the nodes that represent the elements of v.[1] Example 1.1 illustrates the tree vision of records.

Trees Let $\mathcal{L}$ and $\mathcal{V}$ be (infinite) sets of edge labels and (terminal) node values, respectively. A *tree* is a tuple $T = (\mathcal{N}, \mathcal{E}, t, \lambda_{\mathcal{E}}, \lambda_{\mathcal{N}})$ where $\mathcal{N}$ is a node set and $\mathcal{E} \subseteq \mathcal{N}^2$ is a non-empty edge set such that $(\mathcal{N}, \mathcal{E})$ is a directed tree with root $t \in \mathcal{N}$. Then, $\lambda_{\mathcal{E}} : \mathcal{E} \rightarrow \mathcal{L}$ is a (total) edge labeling function (for JSON keys) and $\lambda_{\mathcal{N}} : \mathit{leaves}(T) \rightarrow \mathcal{V} \cup \{\epsilon\}$ is a (total) leaf labeling function (for JSON terminal values), where $\mathit{leaves}(T) \subseteq \mathcal{N}$ is the set of leaves of the tree. We distinguish between *valued* leaves, which are those labeled by a value in $\mathcal{V}$, and *unvalued* leaves labeled by ϵ. The corresponding subsets are denoted by $\mathit{leaves}_{\mathcal{V}}(T)$ and $\mathit{leaves}_{\epsilon}(T)$, respectively. Note that all edges are labeled, while only leaf nodes may be valued. Standard definitions about trees are extended in the obvious way. In particular, a *subtree* of a tree T is a tree included in T; it is a *roooted subtree* of T if it has the same root as T.

Processing trees A *leaf assignment* of a tree T is a function $\nu : \mathit{leaves}_{\epsilon}(T) \rightarrow \mathcal{V}$ whose application is defined as $\nu(T) = (\mathcal{N}, \mathcal{E}, t, \lambda_{\mathcal{E}}, \nu \uplus \lambda_{\mathcal{N}})$. Next, given a function f, a set $A = \{a_1, \dots, a_l\}$ and a sequence $\mathbf{b} = (b_1, \dots, b_k)$, we note $f(A) = \{f(a_1), \dots, f(a_l)\}$ and $f(\mathbf{b}) = (f(b_1), \dots, f(b_k))$. Let T_1 and T_2 be trees with $T_i = (\mathcal{N}_i, \mathcal{E}_i, t_i, \lambda_{\mathcal{E}i}, \lambda_{\mathcal{N}i})$. A *homomorphism* from T_1 to T_2 is a total function $h : \mathcal{N}_1 \rightarrow \mathcal{N}_2$ such that: *(i)* for all $e \in \mathcal{E}_1$, $h(e) \in \mathcal{E}_2$ and $\lambda_{\mathcal{E}1}(e) = \lambda_{\mathcal{E}2}(h(e))$; and *(ii)* for all $n \in \mathit{leaves}_{\mathcal{V}}(T_1)$, $\lambda_{\mathcal{N}1}(n) = \lambda_{\mathcal{N}2}(h(n))$. A homomorphism h is *rooted* if $h(t_1) = t_2$. A homomorphism h from T_1 to T_2 is an *isomorphism* if h^{-1} is also a homomorphism (from T_2 to T_1).

Instances and rules An *instance* is simply a tree (like T_1, T_2, T_3 in Figure 1). To distinguish, we will call *collection* a set of trees. In pictures, edges are always considered as oriented from the root (black node) to the leaves. The body and the head of a *rule* are both trees, which share their root and some leaves; moreover, leaves in the body may be *constrained*, which means that they must be mapped to *valued* nodes in the data. Formally, a *constrained tree-rule* (or simply rule) is a triple $r = (B, H, C)$ where B and H are trees representing the *body* and *head* of r, also denoted by Body(r) and Head(r), and $C \subseteq \mathit{leaves}(B)$ is a (possibly empty) subset of leaves, said *constrained*; C is also denoted Constrained(r); B and H share the same root, and, beside the root, only leaves can be shared between B and H. The *frontier* of r, denoted by $\mathit{frontier}(r)$, is the set of nodes shared between B and H. Furthermore, $\mathit{frontier}_{\mathit{leaves}}(r)$ denotes the set of leaves of r shared by B and H, i.e., $\mathit{frontier}_{\mathit{leaves}}(r) = \mathit{leaves}(B) \cap \mathit{leaves}(H) \subset \mathit{frontier}(r)$. In the formal development, w.l.o.g, we assume no constants in the rules (in contrast with data and queries), that is, $\mathit{leaves}_{\mathcal{V}}(B) = \emptyset = \mathit{leaves}_{\mathcal{V}}(H)$.

Consider again the rules in Figure 1. Rule r_1 defines a high-level notion of a `push` event. First, it verifies if structural conditions defined by the body are met in the data. That is, that a key `type` with value `PushEvent` is present, as well as an event `id` and the `login` of the user who made the event. Then, it checks for constrained nodes (marked with \$ in the picture). That is, it verifies that `id` and `login` are associated with terminal values of $\mathcal{V}$. Finally, it computes the result by copying these values within a new structure associated with the key `push`. Rules r_2 and r_3 are similar. Rule r_4 states that the key `push` is a particular case of `event`. This rule is called a *relabeling* rule. Rules r_5 and r_6 are similar. Note that all frontier leaves are constrained in r_1-r_3, and none in r_4-r_6. Finally, a key feature of rules is that they may have non-frontier nodes in the head, called *existential nodes*. This is the case for r_1-r_3 in Figure 1, where existential nodes are marked in red. These nodes allow to reorganize extracted values into new structures. In this, our framework is leaning towards Datalog±, which extends Datalog with *existentially quantified variables* [16, 35].

Rule Semantics As Figure 1 illustrates, the application of rules leads to an *extended instance* that we compactly see as a (rooted) acyclic graph. This is to simplify the formal development, in that acyclic graphs can always be unfolded into trees. Regardless, since we will focus on query rewriting, these extended instances never have to be computed, as they remain *virtual*. A *trigger* for a rule $r = (B, H, C)$ on an (extended) instance I is a pair (r, h) where h is a (not necessarily rooted) homomorphism from B to I respecting the constrained nodes, i.e., such that $h(C) \subseteq \mathit{leaves}_{\mathcal{V}}(I)$. The application of (r, h) to I results in $I \cup h^+(H)$, where $h^+ \supseteq h$ is an extension of h mapping every *non-frontier node of H* to a fresh node. Given a set of rules Π, we denote by $\alpha(I, \Pi)$ the instance obtained from I by applying all triggers on I in parallel, i.e., $\alpha(I, \Pi) = I \cup \bigcup_{(r,h)} h^+(H)$ where $r \in \Pi$ and (r, h) is a trigger on I. Given an instance tree T,

[1] We do not represent the ordering on the elements of a sequence, since the considered queries do not exploit this order. Moreover, a nested sequence is seen as a constant.

we define $Sat_0(T, \Pi) = T$ and $Sat_{i+1}(T, \Pi) = \alpha(Sat_i(T, \Pi))$. Finally, the *saturation* of T is $Sat(T, \Pi) = \bigcup_{i=0}^{\infty} Sat_i(T, \Pi)$. This notion of saturation is also known as *chase* [8] and is well-defined since the order in which the rule applications are performed has no incidence on the result (up to isomorphism).

Queries and Certain Answers A *constrained tree-query* is a triple $q = (T, C, \mathbf{x})$ where T is a tree, $C \subseteq leaves(T)$ is a set of *constrained leaves* that must be mapped to *valued* nodes in the data, and $\mathbf{x} \in C^{|\mathbf{x}|}$ is a sequence of *answer nodes*. We also denote C by Constrained(q) and $\mathbf{x}$ by AnswerSeq(q). We assume valued leaves are always constrained, i.e., $leaves_{\mathcal{V}}(T) \subseteq C$. To illustrate, consider the query q in Figure 1. The only constrained node is the answer node (marked by x). Hence, this node must be mapped to constant values (here, "431" and "695"). A query q is called *Boolean* if $\mathbf{x} = ()$. A tuple $\mathbf{a} \in \mathcal{V}^{|\mathbf{x}|}$ is an answer to a query q over an extended instance I if there is a *rooted* homomorphism h from T to I such that $\lambda_{\mathcal{N}} \circ h(\mathbf{x}) = \mathbf{a}$ and $h(C) \subseteq leaves_{\mathcal{V}}(I)$, i.e., the leaf constraint is fulfilled. The set of answers to q on I is denoted by Ans(q, I). A tuple $\mathbf{a}$ is a (certain) *answer* to q on I and Π if it is an answer to q on $Sat(T, \Pi)$. The set of all answers to q on (I, Π) is denoted by Ans(q, I, Π). A query q is *more general* than a query q', denoted by $q \geq q'$, if there is a rooted homomorphism h from T_q to T'_q such that $h(C_q) \subseteq C_{q'}$ and $h(\mathbf{x}_q) = \mathbf{x}_{q'}$. A classical query containment property [8] holds here: $q \geq q'$ iff Ans$(q', I) \subseteq$ Ans(q, I), for all I.

Query Rewritings. A *sound* and *complete* set of *rewritings* of q wrt Π is a set of queries Q such that, for every instance I, it holds that Ans$(q, I, \Pi) = \bigcup_{q' \in Q}$ Ans(q', I); in this equality, $\supseteq$ expresses soundness, and $\subseteq$ completeness. In words, Q must embarck all possible ways in which a query q can be satisfied via the rules of Π. For example, Figure 2 illustrates a sound and complete rewriting set for q of Example 1.1, made by the queries $q, q_1, q_2, q_3, q'_1, q'_2, q'_3$. In Section 3 we will show how our query rewriting algorithms exploit two features of rules. First, rules are tree-shaped, and this makes rewriting within the language of tree queries possible, thereby fitting the requirements of NoSQL APIs (a property one can loose beyond tree rules). Second, the applicability of a rule can be limited via constrained leaves, which, as shown later, will pave the way to decidability. As a last remark, note that, as the definition states, a sound and complete set of rewritings Q *is independent of any input instance.* This may lead to useless rewritings not matching the data, like q, q_1, q_2, q_3 over the trees in Figure 1. In Section 4 we will present an instance-aware mechanism to prune these efficiently.

3 NEW QUERY REWRITING ALGORITHMS

As a first step towards effective query answering on NoSQL stores, we give a *query rewriting algorithm for (general) constrained tree-rules.* We revisit techniques developed for existential rules, based on so-called *piece-unifiers* [16, 61], also reminiscent of view-based query rewriting [71]. Roughly speaking, a *unifier* is a node substitution (i.e., a function replacing nodes with nodes) that, as the name suggests, makes part of a query *equal* to part of a rule head. A *piece*-unifier checks further structural constraints that are necessary for the *soundness* of rewriting when the rule head features *existential nodes*, as the rules r_1-r_3 from Figure 1. Given a piece-unifier of a query q with a rule r, a *rewriting step* produces a new query by substituting the unified part of q with the body of r. Then, given a set of rules Π, a Π-*rewriting* of q is a query obtained by a sequence of rewriting steps with rules from Π. E.g., each query in Figure 2 is a Π-rewriting of q, with Π the rule set of Figure 1.

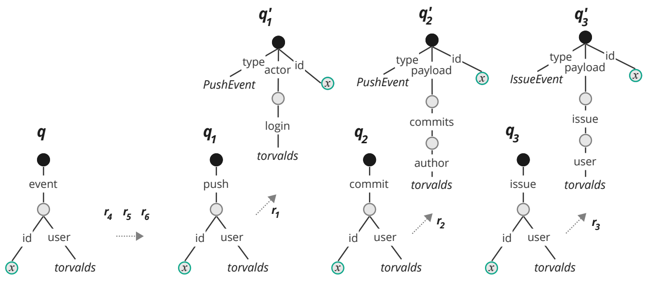

Figure 2: Query Rewriting Associated with Example 1.1

The contribution of this work is to identify a restricted class of piece-unifiers, called *(semi-)twig-unifiers*, which have the nice property of producing *tree-shaped rewritings* as we sought.[2] The formal development of semi-twig query rewriting is presented in Section 5. This tool allows us to compute a set of rewritings of a query, whose soundness and completeness is stated as follows.

Theorem 3.1 (Soundness and Completeness of Rewriting). *For any query q, instance T, and set of constrained tree-rules* Π, $\mathbf{a} \in$ Ans(q, T, Π) *iff there is a* Π*-rewriting q' of q such that* $\mathbf{a} \in$ Ans(q', T).

Hence, semi-twig query rewriting allows one to compute a *finite* sound and complete set of rewritings *when such set exists.* And indeed, there are rule sets and queries that only admit *infinite* sound and complete sets of rewritings (even modulo query containment). This can happen already with simple rules, as shown by Figure 3. Query q searches for projects `forkedFrom` a repository owned by the `keras-team`. By rewriting q with the rules r_1 and r_2, one builds paths of arbitrary length along the key `from`. All of the obtained queries are incomparable to each other w.r.t. query containment, hence the only sound and complete rewriting set here is infinite. This raises the issue of algorithmic feasibility for query answering.

3.1 Taming Infinite Rewritings

In spite of an apparent simplicity, query answering with constrained tree-rules is undecidable. This has been shown already for the specific case of (unconstrained) path rules in various settings [9, 28, 40]. This negative result opens the quest for decidable rule languages for reasoning on JSON data. Strategies for recovering decidability include the use of acyclicity notions or syntactic conditions ensuring the termination of (forward or) backward chaining [16, 35, 46]. Here, we take a different approach. We consider a language with *non-terminating* (both forward and) backward chaining, but ensuring that the rewritings of a query can always be finitely captured. We draw inspiration from the language of so-called suffix path-rules proposed in [23] and lift it to the case of trees, which yields a specific constrained tree-rule fragment we call *frontier-constrained rules.* Crucially, this fragment makes the rewritings of a query forming a *regular tree-language* one can capture with automata techniques.

[2]In passing, note we cannot use the query rewriting algorithm based on piece-unifiers as-it-is, because 1) it may yield (more specific) rewritings that are not trees and 2) it may not terminate when it should, as it does not take into account the shape of data.

A Tractable Fragment. We consider combinations of 1) *frontier-constrained* rules and 2) *relabeling* rules, which are defined next.

Definition 3.2. Let $r = (B, H, C)$ be a constrained tree-rule. Then:

- r is *frontier-constrained* if $frontier_{leaves}(r) \subseteq C$
- r is *relabeling* if B and H are edges and $frontier_{leaves}(r) \neq \emptyset$

In Figure 1, r_3-r_6 are frontier-constrained rules while r_1-r_4 are relabeling rules. Frontier-constrained rules impose that a trigger maps *all the frontier leaves of the rule to data values*, but *without any further condition on the use of recursion.* From a practical viewpoint, they allow one to select values in the data and to reorganize them into structures adapted to the targeted application. Relabeling rules are among the most useful rules for reasoning on trees, as they allow one to define hierarchies of keys. In contrast with frontier-constrained rules, they apply anywhere on a tree instance.

The decidability of query answering for this fragment follows from a natural translation into first-order logic. Constrained queries and rules are translated into tree-shaped conjunctive queries and existential rules, respectively. *Frontier-constrained* and *relabeling* rules are more specifically translated into a decidable fragment of existential rules called *body-acyclic frontier-guarded* [17]. For this, we obtain that query answering is in ExpTime for combined complexity [36]. Our framework can also be translated into a specific description logic, namely $\mathcal{ELHV}$ [63], which furthermore allows one to derive a PTime upper bound for data complexity. Note however that these complexity results do not make use of query rewriting. By relying on the tree-automata rewriting described next, we will design a query answering technique that effectively runs in polynomial time w.r.t. the size of the data.

Capturing Infinite Rewritings With Automata A key feature of our (general) query rewriting algorithm is that, when rules are *frontier-constrained*, it is ensured in any direct rewriting of q that a *single* node is shared between the remaining part of q and the subtree coming from the rule body. Hence, we fall into an even more specific case of unifiers we call *twig*-unifiers. As a consequence, infinite sets of rewritings such as those illustrated in Figure 3 can be captured by a tree automaton [44]. In a nutshell, the constructed automaton is made of sub-automata that encode the initial query q as well as all the specializations of rule bodies that can be involved in a rewriting step. This set of sub-automata is finite as there is a finite number of (non-equivalent) specializations for each rule body. Transitions between states of different sub-automata allow to encode rewritings. The detailed construction is provided in Section 6 and its adequacy now stated (follows by Theorem 6.2).

Theorem 3.3. *Let* Π *be a set of* frontier-constrained *and* relabeling *rules and* q *be a query. Then, there exists a finite tree automaton* A *recognizing a sound and complete set of rewritings of* q *w.r.t.* Π.

4 INSTANCE-AWARE EVALUATION

The automata approach gives us a ground for query answering but still does not suffice, *in practice*, to exploit NoSQL databases. Evidently, no database API will take an automaton as a query. Tree queries are instead accepted. Concretely, this means that the automaton language has to be *enumerated*, and that every single query in a rewriting set has to be evaluated. We are thus facing two issues.

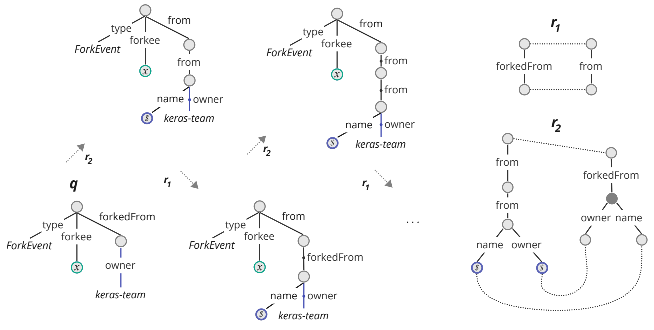

Figure 3: Rules and Infinite Rewriting Sequences

First, we need to deal again with the case where rewriting sets are infinite. Second, we need to cope with rewriting sets that are finite but possibly large. To tackle these problems, we extend our query rewriting approach to make it *instance-aware*, and more specifically able to leverage on structural information of data for the sake of query answering. Our approach consists in using a novel combination of *i*) summarization techniques to make rewritings finite and *ii*) partitioning and parallelization techniques for efficiency.

Summarization. To always reduce rewritings to *finite* sound and complete sets - for a given database instance - we use summaries of data. In theory, it is sufficient to know the *size* of the data, as the automaton allows one to enumerate queries by their size. So, once exceeded the size of data, enumeration can be safely stopped. This rough bound can be improved by considering only the *depth* of data as nodes in a rewriting have a bounded outdegree (Theorem 6.2).

In practice, however, as rewriting is independent of the instance, this can generate many queries that are not matching any data. So we consider a data summary built on the set of *maximal paths* (root to leaf), i.e., a dataguide [53]. When we enumerate the automaton language we can use this information to discard some empty queries by checking whether the tree query we are generating contains an invalid path according to the summary. For instance, the set of maximal paths of the trees in Figure 1 is

```
(date)    (id)    (type)    (actor · login)
(payload · commits · author)    (payload · issue · user)
```

Obviously, any rewriting featuring a path that is not in this list can be discarded, as it will have no answer on the collection. Summaries can sometimes be simplified when there is little ambiguity in the data, by considering only the depth and the set of edge *labels*. For the collection of Figure 1 these labels are {`date`, `id`, `type`, `actor`, `login`, `payload`, `commits`, `issue`, `author`, `user`}. The label summary is less precise but more compact. Paths can also be enriched with k-length prefixes of the *values* found at their end. For instance, with $k = 5$, we can associate the value "gvanr" to path `payload.commits.author` instead of `gvanrossum`. This can help in filtering queries which use values not found in the data. Working with k-prefixes instead of the values themselves enables more concise summaries while retaining the precision of filtering. So, the *path* (dataguide), *label*, and *path+prefix* will be the three main summaries we consider. All can be used to filter empty queries in the automaton language, and hence to reduce the (possibly infinite) set of rewritings of a query to a finite and still complete set (for the given instance). Furthermore, they can all be computed with a linear traversal of the data, and are typically of limited size. We will use the *depth* summary as a baseline. We did not consider bisimilarity-based summaries [55] as they tend to be quite large and more complex to compute; the label and path summaries also proved to be quite effective.

Partitioning and Parallelization Even if finite, rewriting sets can be hard to evaluate for any database because of the number of queries they may contain. To improve the situation, our intuition is that we must provide a mean to *lead the single queries (within a rewriting set) where the matching data is*. With this aim, we define an instance-aware reasoning strategy based on *partitioning*. In doing so, we therefore introduce a pre-processing of a *collection* providing means for the efficient evaluation of large rewriting sets.

Let $J = \{T_1, \ldots, T_n\}$ be a (single) collection of tree instances. Our goal is to define a set of collections $J_1, \ldots, J_k$ making for a partitioning of J. We adopt a simple partitioning function oriented towards rooted queries, defined as follows: $\Lambda_d(T) = \Pi_{i=1}^{d}\, labels_i(T)$. The function $\Lambda_d(T)$ concatenates the labels of edges at each level (we denote by $labels_i(T)$ the labels of edges going from level $i-1$ to level i in T) up to the depth set by the parameter d; $\Lambda_d(T)$ is then mapped to an integer denoting the partition number of T. So, two trees T_1, T_2 will be in the same partition J_i if $\Lambda_d(T_1) = \Lambda_d(T_2)$, meaning that they just agree on the set of labels they use level-wise, up to depth d. Note that for Λ_d the number of partitions is not fixed in advance, and rather depends on *i*) the data and *ii*) the parameter d. The function Λ_d could also be replaced by more complex clustering functions for trees also taking care of load balancing [10]; exploring this range of possibilities is however beyond the scope of this work. Note that $\Lambda_d(T)$ can be computed in linear time at the moment when data is loaded into the database.

The first net advantage of partitioning is that, instead of a rough summarization for the whole collection, we can deploy a set of narrower data-summaries, *one for each partition* J_i. This obviously leads to a greater filtering power. With this scheme, the queries within a rewriting are more likely to be evaluated on the partitions where they can have a match, according to the summary, instead of being evaluated against the whole database. The use of rewriting, summarization, and partitioning is shown in Figure 4: *A rewriting automaton* $\mathsf{A}_{(q,\Pi)}$ *is built from a query* q *and a set of rules* Π. *A collection* J *is partitioned into the collections* $J_1, \ldots, J_k$. *For every partition* J_i, *the language of* $\mathsf{A}_{(q,\Pi)}$ *is filtered by Summary*(J_i). *This produces a finite set of queries* $Q_{|\mathsf{Summary}(J_i)}$ *to evaluate on* J_i.
Partitioning can be implemented in several ways. Here, we consider physical partitioning, where a distinct database collection is built for every partition. Using a *logical* partitioning strategy is discussed in [69] and proved to have similar effectiveness. Finally, with partitioning in place, *parallelization* can further be added to attack all of these partitions simultaneously by leveraging on the concurrent data access facilities of the underlying database.

Let us point out that assuming data-awareness is not a strong hypothesis in practice. For instance, data-awareness is implicit in the fact of running a materialization algorithm. Also note that a summary does not capture a single instance, but rather abstracts over a class of instances. Instance-aware query rewriting is evaluated in Section 7. The following sections (5 and 6) present our query rewriting algorithms.

$$\begin{tikzcd}
 & \mathsf{Summary}(J_1) \arrow[r, squiggly] & Q^{(q,\Pi)}_{|\mathsf{Summary}(J_1)} \arrow[r] & J_1 & \\
q \rightsquigarrow \mathsf{A}_{(q,\Pi)} \arrow[ur] \arrow[dr] & & \vdots & & J \arrow[ul] \arrow[dl] \\
 & \mathsf{Summary}(J_k) \arrow[r, squiggly] & Q^{(q,\Pi)}_{|\mathsf{Summary}(J_k)} \arrow[r] & J_k &
\end{tikzcd}$$

$\boxed{\text{Rewriting}}\ \boxed{\text{Filtering}}\ \boxed{\text{Evaluation}}\ \boxed{\text{Partitioning}}$

Figure 4: Instance-Aware Rewriting and Evaluation

5 SEMI-TWIG-BASED QUERY REWRITING

The goal of this section is to present semi-twig based unification and query rewriting. As already mentioned, we revisit techniques developed for existential rules, based on so-called *piece-unifiers* [16, 61]. To begin, we need to introduce a key notion, that of *separating node*. Let T be a tree and S be a subtree of T. We denote by $(T \setminus S)$ the forest obtained by removing from T all edges in S. Then, the nodes that belong to both S and $(T \setminus S)$ are said to *separate* S from T. In other words, these are the nodes of S that have their parent or one of their children in $(T \setminus S)$. Figure 5 pictures a tree T with four subtrees S_1, S_2, T', T''. The nodes marked with $\mathcal{S}$ are separating for at least one subtree. The subtrees S_1 (in blue), S_2 (in brown) and T' (in red) have the same root as T. For all of these subtrees the root is separating; indeed the root has other children that do not belong to the subtree. The root of T'' (in red) is separating, this time because its parent does not belong to T''. Note that S_2 also has a separating leaf and T' also has a separating internal node.

To simplify definitions, we now assume that the root of a rule head has a single child. This can be done without loss of generality since every rule can be decomposed into an equivalent set of rules satisfying this assumption [69].

Definition 5.1 (Semi-twig). A subtree S of T is called *semi-twig* if:

(1) the root of S has exactly one child
(2) any node separating S from T is either the root of S or a leaf of S (i.e., no internal node of S is separating).

A semi-twig without separating leaves is called a *twig*.

Consider again T in Figure 5. Then, S_1 is a twig as its only separating node is the root, and S_2 is a semi-twig as it has a separating leaf. Finally, T' and T'' are not semi-twigs, since T' has an internal separating node and the root of T'' has two children. The intuition behind semi-twigs is that these are the parts of the query that can be easily rewritten (Item (1)) while preserving tree-shaped rewritings (Item (2)). We are now ready to define semi-twig unification.

Definition 5.2 (Semi-Twig Unification). A *semi-twig-unifier* μ for a constrained query q and a (general) constrained rule r is a triple (S, H, v, u) where:

• S is a semi-twig of q
• H is a rooted subtree of Head(r)
• v is a leaf assignment for the nodes in $frontier_{leaves}(r)$
• u is a rooted homomorphism from S to $v(H)$ such that $u(S) = v(H)$ and u maps:
(1) each constrained leaf of S to a frontier leaf of r
(2) each separating leaf of S to an unconstrained frontier leaf of r.

Although the definition of semi-twig unification may seem somewhat involved, it is perfectly natural. The goal is to identify part of the query (i.e., S) that is entailed by the application of the rule r. Rewriting will then replace this part by a suitable specialization of the body of r that reflects the way in which the rule is applied. This is captured first by the rooted homomorphism from S to a leaf assignment of H. This leaf assignment may only assign to a frontier node in r a value from S, since $u(S) = v(H)$. By the homomorphism u, a leaf of S is necessarily mapped to a leaf of H. Then, Conditions (1) and (2) ensure the correctness of rewriting.

In the next definition of a direct rewriting, the union of the trees $u(q \setminus S)$ and $v(\text{Body}(r))$ is defined in the obvious way (i.e, by making the union of the node sets, the edge sets and the functions) and it can be checked that it results in a tree.

Definition 5.3 (Semi-Twig Rewriting). Let $\mu = (S, H, v, u)$ be a semi-twig-unifier for a constrained query q and a (general) constrained rule r. A *direct rewriting* of q with μ is a query $q^{rew} = (T, C, \mathbf{x})$ such that:

- $T = u(q \setminus S) \cup v(\text{Body}(r))$
- $C = u(\text{Constrained}(q)) \cup \text{Constrained}(r)$
- $\mathbf{x} = u(\text{AnswerSeq}(q))$.

Proposition 5.4 (Closedness). *q^{rew} is a constrained tree query.*

Let us come back to Conditions (1) and (2) of semi-twig unification and illustrate them with the query q and rules r_1 and r_2 from Figure 5. Both rules have a constrained leaf in their body. This leaf is frontier for r_1, but not for r_2. Hence, an application of r_1 only brings a new edge, while an application of r_2 also brings a new node. One could consider unifying the semi-twig S'_1 constituted by the sole edge labeled by a. Note that the leaf of S'_1 is separating, hence can only be mapped to an unconstrained frontier leaf. Unifying S'_1 with r_1 would violate Condition (2) because the frontier leaf of r_1 is constrained (and the associated rewriting would not be a well-formed query, as it would have an internal constrained node). Unifying S'_1 with r_2 would also violate Condition (2) because the leaf of Head(r_2) is not frontier (and the associated rewriting would not be a well-formed query, as it would be disconnected, and would furthermore be unsound). Now, consider the twig S'_2. It can be unified with r_1 thereby yielding the query $q^{rew}_{r_1}$. However, S'_2 cannot be unified with r_2 because Condition (1) would be violated (and the associated rewriting would be unsound).

Finally, we say that q^{rew} is a Π-*rewriting* of q if there is a sequence $q = q_0, \mu_1, q'_1, ..., \mu_k, q'_k = q^{rew}$ such that q_i is a direct rewriting of q_{i-1} with μ_i using a rule of Π and $q'_i = (q_i)^{\text{safe}}$, where $\cdot^{\text{safe}}$ is a function replacing all nodes of q_i by fresh nodes ($1 \leq i \leq k$). This is merely a technicality, but it is needed to avoid multiple uses of the same node if r is used multiple times in a rewriting sequence. The soundness and completeness of query rewriting ensures that for any query q, instance T, and set of constrained tree-rules Π, $\mathbf{a} \in \text{Ans}(q, T, \Pi)$ if and only if there is a Π-rewriting q' of q such that $\mathbf{a} \in \text{Ans}(q', T)$ (as stated by Theorem 3.1). From this, we can easily build a breadth-first query rewriting operator in the spirit of [61] that terminates if and only the query admits a finite sound and complete set of rewritings. However, this still leaves the case of infinite rewritings open. An important remark can be made about

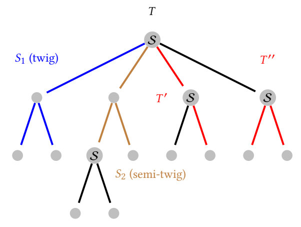

Figure 5: Separating Nodes and (Semi-)Twig Unification

the specific case of frontier-constrained rules (Definition 3.2) on which we will focus in the next section.

Remark 1. *Let (S, H, v, u) be a semi-twig unifier for a query q and a frontier-constrained rule r. Then, S must be a twig.*

For instance, in Figure 5, both rules are frontier-constrained, hence S'_1 can be disregarded for rewriting as it is not a twig.

6 AUTOMATA-BASED QUERY REWRITING

Frontier-constrained rules do not ensure the finiteness of rewriting sets (nor of saturation). However, they have the following key property: *the rewriting set of a query under frontier-constrained rules can be characterized by a regular tree language.* This means that it can be compactly represented by a tree automaton. Frontier-constrained tree rules generalize context-free path rules in [23] which are in turn related to suffix-rewriting systems [42, 72]. What paves the way for regularity is that frontier-constrained rules make any semi-twig unifier for the query to be necessarily a *twig-unifier* (Remark 1). Another important insight is that adding relabeling rules (which, in contrast, requires semi-twig unifiers that may not be twig unifiers) still preserves regularity, and so both types of rules can be taken into account jointly. The goal of this section is to present the construction of the automaton $\mathsf{A}_{(q,\Pi)}$ recognizing a sound and complete set of rewritings for a query q against a set of rules Π.

We start by posing the notion of tree automaton needed to capture the rewritings of a query. This follows standard models [44] adapted to the case of unordered trees.

Definition 6.1 (Automaton). A bottom-up automaton for *unordered* trees is a tuple $\mathsf{A} = (\Sigma, \mathcal{S}, \mathcal{L}, \mathcal{F}, \Delta)$ where:

- Σ is an alphabet
- $\mathcal{S}$ is a set of states made of
  $\mathcal{S}^E$ a set of edge-states and $\mathcal{S}^N$ a set of node-states
- $\mathcal{L}, \mathcal{F} \subseteq \mathcal{S}^N$ are sets of initial (leaves) and final (root) states
- Δ is a set of transitions made of
  $\Delta^{\mathcal{L}} \subseteq \left(\Sigma \cup \{\epsilon\}\right) \times \mathcal{L}$ a set of leaf nodes transitions
  $\Delta^E \subseteq \mathcal{S}^N \times \Sigma \times \mathcal{S}^E$ a set of edge transitions
  $\Delta^N \subseteq 2^{\mathcal{S}^E} \times \mathcal{S}^N$ a set of (non-leaf) node transitions

A *run* of A on $T = (\mathcal{N}, \mathcal{E}, t, \lambda_{\mathcal{E}}, \lambda_{\mathcal{N}})$ is a function $\rho : \mathcal{N} \to \mathcal{S}^N$ that agrees with the transition rules of A, in the following sense.

- Every unlabeled leaf ℓ of T is recognized by a $\Delta^{\mathcal{L}}$-transition
$$\emptyset \xrightarrow{\epsilon} \rho(\ell)$$
- Every labeled leaf ℓ of T is recognized by a $\Delta^{\mathcal{L}}$-transition
$$\emptyset \xrightarrow{\lambda_{\mathcal{N}}(\ell)} \rho(\ell)$$

- Every (non-leaf) node n of T with children $n_1, \ldots, n_k$ is recognized by the following Δ-transitions

  $\{\mathsf{s}_1 \ldots \mathsf{s}_k\} \longrightarrow \rho(n) \in \Delta^N \qquad \rho(n_i) \xrightarrow{\lambda_{\mathcal{E}}(n,n_i)} \mathsf{s}_i \in \Delta^E$

  provided that $\mathsf{s}_i \neq \mathsf{s}_j$ for all $1 \leq i \neq j \leq k$
- The root t of T is such that $\rho(t)$ is a final state in $\mathcal{F}$

An automaton defines a language (or set) of unordered trees $\mathsf{L}(\mathsf{A})$.

The automata framework represents tree languages, but is insensitive to query features such as *constrained* and *answer* nodes. So, to proceed, we must define an encoding of a query q as a "plain" tree $\mathsf{encode}(q)$ that can manipulated by automata. The idea of the encoding is to store information on constrained and answer leaf nodes as *values*. The encoding must essentially cover three cases. The first is where the leaf is unconstrained. The second is where the leaf is constrained but does not belong to the answer sequence. The third is where the leaf belongs to the answer sequence (so, by definition, is also constrained). Figure 6 (left) shows a query q' as well as its corresponding automaton $\mathsf{A}_{\mathsf{encode}(q')}$. The query seeks for forked projects (`forkee`) for all records describing a `ForkEvent` which also includes the project from which fork originated (`forkedFrom`). The query has three leaf nodes, which are recognized by three initial states of the automaton. Each initial state is used to define a non-leaf node transition $\emptyset \xrightarrow{v_i} \mathsf{s}_i^{\mathsf{initial}}$ where v_i is the encoding of a query leaf. State $\mathsf{s}_1^{\mathsf{initial}}$ recognizes the query leaf labeled by value `ForkEvent`. By definition, this is a constrained node. Hence its encoding is $v_1 = (\$ \cdot \mathtt{ForkEvent})$ where the reserved symbol \$ is used to denote a constrained node. State $\mathsf{s}_2^{\mathsf{initial}}$ recognizes the answer variable x, which by definition is also constrained. This is encoded as $v_2 = (\$ \cdot 1)$, where 1 denotes the position of the answer variable in the answer node sequence. State $\mathsf{s}_3^{\mathsf{initial}}$ recognizes an existential node. This is encoded as $v_3 = (\#)$ where # denotes that the node is not constrained. Moving on, the initial states are used to define three edge transitions of the form $\mathsf{s}_i^{\mathsf{initial}} \xrightarrow{k_i} \mathsf{e}_i$ where $k_1 = \mathtt{type}$, $k_2 = \mathtt{forkee}$, and $k_3 = \mathtt{forkedFrom}$. Each transition recognizes an edge of the query. Finally, the non-leaf node transition $\{\mathsf{e}_1, \mathsf{e}_2, \mathsf{e}_3\} \longrightarrow \mathsf{s}^{\mathsf{final}}$ allows $\mathsf{A}_{\mathsf{encode}(q')}$ to recognize the whole tree. Due to space constraints, the formal construction of the automata encoding of queries is detailed in [69]. The example of Figure 6 illustrates that it is straightforward to build an automaton A_T recognizing a tree T which is also minimal (i.e., without useless state or transition). We can do so by creating distinct states and transitions for every (1) node label, (2) edge, and (3) internal node of T. This handling of (un)constrained, valued, and answer nodes, also extends to the encoding of rule bodies introduced from single rewriting steps [69]. In the remainder of the section, the encoding of a query q and of a rule body B specialized by a unifier μ are denoted by $\mathsf{encode}(q)$ and $\mathsf{encode}(\mu(B))$, respectively. This is at the basis of the rewriting process described next.

Building the Rewriting Automaton Remind that our goal is to construct an automaton $\mathsf{A}_{(q,\Pi)}$ recognizing the set of rewritings of a query q against a set of rules Π. For clarity, we present the construction of $\mathsf{A}_{(q,\Pi)}$ in two steps. First, we provide a declarative construction, showing (i) the main steps of the process as well as its connections to the general query rewriting from Section 5 and (ii) the finiteness of $\mathsf{A}_{(q,\Pi)}$. In the second step, we outline how to also achieve a terminating algorithm.

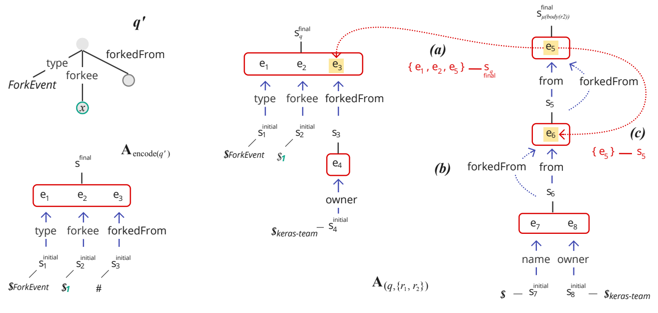

Figure 6: Query and Rewriting Automata

Figure 7 presents the construction of the rewriting automaton $\mathsf{A}_{(q,\Pi)}$. In there, we do some standard assumptions. First, when constructing an automaton A_B from a specialized rule body, we use a fresh set of states. The Extend operation makes the union of two automata $\mathsf{A}_{(q,\Pi)} = (\Sigma, \mathcal{S}, \mathcal{L}, \mathcal{F}, \Delta)$ and $\mathsf{A}_B = (\Sigma_B, \mathcal{S}_B, \mathcal{L}_B, \mathcal{F}_B, \Delta_B)$ but gives precedence to the final state of $\mathsf{A}_{(q,\Pi)}$ by returning $(\Sigma \cup \Sigma_B, \mathcal{S} \cup \mathcal{S}_B, \mathcal{L} \cup \mathcal{L}_B, \mathcal{F}, \Delta \cup \Delta_B)$. Finally, as we work with automata that have a single final state, we denote by $\mathsf{s}_{\mathsf{A}}^{\mathsf{final}}$ the final state of A.

Figure 6 (right) illustrates the rewriting of the query q presented in Figure 3 with rules r_1 and r_2. Recall that query q searches for projects `forkedFrom` a repository `owned` by the `keras-team`. The query has an infinite rewriting set, which is captured as follows. The process starts by initializing $\mathsf{A}_{(q,\Pi)}$ as the automaton for $\mathsf{encode}(q)$. A twig-unifier μ for r_2 and q is found (steps 3 and 4). Remark that μ unifies a twig of the query (actually, the path `forkedFrom.owner`) with a twig of the rule head. This leads to the extension of $\mathsf{A}_{(q,\Pi)}$ with a fresh sub-automaton for $\mathsf{encode}(\mu(\mathsf{Body}(r_2)))$ (step 5). Also this yields ⓐ a novel non-leaf node transition $\{\mathsf{e}_1, \mathsf{e}_2, \mathsf{e}_5\} \longrightarrow \mathsf{s}_q^{\mathsf{final}}$ (step 6) resulting from replacing e_3 by e_5 in $\{\mathsf{e}_1, \mathsf{e}_2, \mathsf{e}_3\} \longrightarrow \mathsf{s}_q^{\mathsf{final}}$. In Figure 6, abusing of notation, we picture this with an edge from e_5 to e_3. At this point, ⓑ the relabeling rule r_1 is applied to the fresh sub-automaton (step 2). This in turn yields a new twig-unifier μ' for r_2 (steps 3, 4). As $\mathsf{encode}(\mu(\mathsf{Body}(r_2))) = \mathsf{encode}(\mu'(\mathsf{Body}(r_2)))$, no extension of $\mathsf{A}_{(q,\Pi)}$ is performed at this time (step 5). However, ⓒ a novel non-leaf node transition $\{\mathsf{e}_5\} \longrightarrow \mathsf{s}_5$ (step 6) capturing the infinite recursion stemming from the rules r_1 and r_2 is added. The correctness of the construction is now stated [69].

Theorem 6.2 (Soundness and Completeness). *Let Π be a set of frontier-constrained and relabeling rules. For every tree T and query q we have that* $\bigcup_{\mathsf{encode}(q') \in \mathsf{L}(\mathsf{A}_{(q,\Pi)})} \mathsf{Ans}(T, q') = \mathsf{Ans}(T, q, \Pi)$. *Also, $\mathsf{A}_{(q,\Pi)}$ is finite and its language is made by trees of bounded rank.*

All-at-Once Twig-Unification To turn our declarative construction into a terminating algorithm we have to provide a finite procedure for the enumeration of all possible queries and twig-unifiers (steps 3 and 4) which may loop when $\mathsf{A}_{(q,\Pi)}$ recognizes an infinite language. And indeed, it is possible to manipulate these unifiers *all-at-once* by directly working on the states of $\mathsf{A}_{(q,\Pi)}$. So, instead of considering every possible query q' recognized by $\mathsf{A}_{(q,\Pi)}$ and every possible twig S of q' we do the following. First, we iterate on the (finite) set of rooted twigs H that belong to the head of the rules in Π. Then, for each H, we iterate on the (finite) set of node-states s of $\mathsf{A}_{(q,\Pi)}$. For every H and s, we check if $\mathsf{A}_{(q,\Pi)}(\mathsf{s})$

Construction of $A_{(q,\Pi)}$ from q and Π

```
(1) Initialize $A_{(q,\Pi)}$ to be equal to $A_{encode(q)}$
(2) For every edge transition $s \xrightarrow{a} e$ in $\Delta^E$ and rule $r \in \Pi$ relabeling b into a Add the transition $s \xrightarrow{b} e$ in $\Delta^E$
(3) For every tree encode($q'$) recognized by $A_{(q,\Pi)}$ via a run $\rho$
(4)     For every twig-unifier $\mu = (S, H, v, u)$ for $q'$ and rule $r \in \Pi$
(5)         Extend $A_{(q,\Pi)}$ with an automaton for $A_{encode(\mu(Body(r)))}$
                                                        (if not already done)
(6)         Add to $\Delta^N$ a (non-leaf) node transition
                $(\mathcal{U} \setminus \mathcal{T}) \cup \mathcal{B} \longrightarrow \rho(Root(S))$    where
      • $\mathcal{U} = \{\rho(n_1), \ldots, \rho(n_k)\}$ is a set of node-states where $n_1, \ldots, n_k$ are the children of Root($S$) within the query $q'$
      • $\mathcal{T} = \{n_j\}$ is a singleton node-state where $n_j$ is the (only) child of Root($S$) within the twig $S$
      • $\mathcal{B}$ is a set of states such that $\mathcal{B} \longrightarrow s^{final}_{encode(\mu(Body(r)))}$
(7) Repeat (2-6) until no more transitions can be added.
```

Figure 7: Rewriting Automaton Construction

(that is, the automaton $A_{(q,\Pi)}$ where the final state is set to be s) recognizes a tree T such that S is a rooted twig of T and there are u and v such that $u(S) = v(H)$. Because H is finite this can be done in a finite number of steps. In this way, we can capture all unifiers μ representing a possibly infinite class of unifiers. Once $A_{(q,\Pi)}$ built, it is possible to perform a single post-order traversal of an instance T to compute answer to queries [69]. This adapts classical validation algorithms from [44]. Follows a PTime data-complexity upper-bound for query answering. The construction is independent from data. Moreover, it can be boostrapped independently from queries. Indeed, the unifiers that hold between the rules only, as those for r_1 and r_2 in Figure 6, can be precomputed *off-line* thus reducing rewriting costs at query time.

7 EXPERIMENTAL ANALYSIS

Implementation We implemented our approach for reasoning over document stores in a Java 11 tool: TreeForce (TF). This library can be seen as a general toolbox for implementing reasoning techniques for tree-shaped data and rules. The tool has been coded from scratch. It is composed of two main modules. The first includes generic data structures and algorithms for trees and tree-automata. The second includes our instance-aware query rewriting and evaluation methods. TreeForce also includes a translation module tied to the target DBMS. We deployed our system on top of a well-known NoSQL database: MongoDB (v5.0.8). Our approach can be ported to systems supporting constrained tree-queries, which is a basic requirement for many stores [1, 3, 4, 25, 58, 70]. Query translation as well as the deployment on other systems is discussed in [69].

Environment We performed all experiments on a machine with an AMD Ryzen 9 3900XT CPU (4.7 Ghz, 12 cores), 128GB DDR4 2400Mhz memory and 2TB SSD disk, running KDE neon on ext4 FS. By relying on MongoDB, TreeForce has modest memory requirements. We allocated only 10GB memory to the JVM (jdk16). We used MongoDB standard configuration with 8GB cache size.

Benchmarks used for our study are shown in Table 1. We defined three benchmarks from known benchs: DBLP^{JR}, GitHub^{JR} and XMark^{JR} (JR, for JSON Reasoning). DBLP and GitHub are large corpus of *real* data. XMark, in contrast, is a *synthetic* benchmark but equipped with a data generator that helped us in better understanding the scalability question. Both DBLP and XMark come as XML; they have been translated into JSON to feed document stores. These systems are oriented towards the exploitation of limited size records (e.g., 16MB for MongoDB). So, XML trees have been shredded into a collection of JSON records in a standard way, that is, by recording the main objects of the original data in different records (e.g., one record per publication in DBLP^{JR}). **DBLP^{JR}**and **GitHub^{JR}** are tests over several million real world records. For DBLP^{JR} we used 22 queries containing a mix of tree queries of different complexity inspired from those in [30, 32, 52]. We manually defined a set of 51 rules with 15 frontier-constrained rules and 36 relabeling rules inspired both from the DBLP ontology (dblp.org/rdf/schema) and [30]. For GitHub^{JR} we defined 5 queries of different complexity and 54 tree-rules including 40 frontier-constrained and 14 relabeling. Overall, these allowed us to gauge our approach on real voluminous data. **XMark^{JR}** includes a set of 115 distinct JSON collections (up to several million records), 23 rulesets (containing 5 to 62 relabeling rules), and 10 queries. XMark^{JR} is an extension of the well-known XMark [75] we designed to dispose of a *rule-based query answering benchmark over trees* to understand the scalability of our approach. First, XMark^{JR} allows us to control the number of rewritings of a query - which is a crucial parameter. We consider 23 rulesets *each creating a larger number of rewritings per query* (from 1 to 500). Second, it allows us to control the size and variability of data. So for, each ruleset, we considered five collections (see Table 1), with the larger collection having 1.4B edges when data is seen as trees. We refer to [69] for its detailed presentation. Overall, XMark^{JR} is meant to push the limits of rewriting-based query answering.

Table 1: Dataset Size

			XMark^{JR}				
	DBLP^{JR}	GitHub^{JR}	[η=0.1]	[η=1]	[η=10]	[η=100]	[η=500]
#records	8.9 M	1.2 M	7.1 K	71 K	710 K	7.1 M	35.5 M
#edges	145 M	107 M	290 K	2.9 M	28.9 M	289 M	1.4 B
(JSON)	3.7 GB	5GB	10 MB	100 MB	1 GB	10 GB	50 GB

Other systems. We are not aware of any system for reasoning with constrained-tree rules on top of JSON data. However, as our rules can be encoded as Datalog$^{\pm}$ rules [16, 34] we used reasoners for Datalog$^{\pm}$ designed for knowledge-graphs as a baseline. *Materialization-based approaches.* We used VLog [41] via Rulewerk [5] to study in-memory materialization. VLog is a high-performance state-of-the-art reasoner [11]. As it runs in-memory, it gives us a point on absolute performance to evaluate DBMS-based approaches. *Rewriting-based approaches.* Graal is the only rewriting-based system for Datalog$^{\pm}$ we are aware of [15, 62]. We used a recent major version of the tool [6]. Graal is a *modular system* that allows one to parametrize the storage and reasoning algorithms. So, we leveraged on Graal's modularity to study three rewriting-based strategies, as follows. In all three cases, we used Graal to compute (finite) rewritings of queries - whenever possible. Then, rewritings were evaluated in two different ways. The first is in-memory, again with VLog. The second is with PostgreSQL (v14.7); data was stored as a

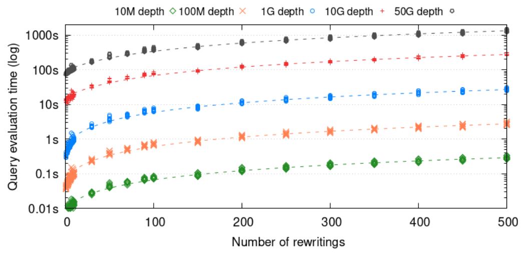

Figure 8: Answering Time (baseline) - XMark^{JR}

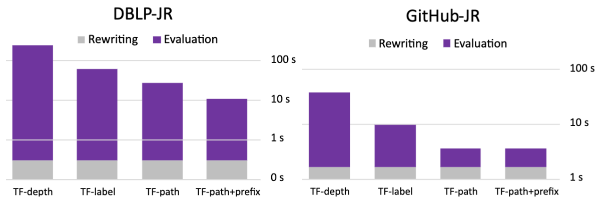

Figure 9: Answering Time (average) vs Summaries

knowledge graph by using indexed property-tables [52] and integer encoding [81] and tree-queries were translated as SQL queries. As scaling integer encoding can be challenging, we also considered a PostgreSQL storage without integer encoding.

MongoDB Wrapper Queries have been translated into MongoDB with the `find()` facility which answers true on every record of the collection satisfying a query by returning the record itself. It is also worth noting that for all benchmarks we considered tree-queries without answer variables. However, this does not make evaluation easy, as we still ask MongoDB to check if each record satisfies the query. To gauge performances across different systems we asked the competitors compute the set of trees matching a query as for our system. We focused on the computational effort needed for evaluating constrained tree-queries: no input time (i.e., data loading) nor output time (i.e. result serialization) were considered. Inputs were translated into formats recognizable by the other systems (JSON data as CSV, tree queries and rules as DLGP [14]).

Overview of the Results We will focus on two aspects. First, a study of the query answering techniques we proposed, then a comparison with the other systems. For space limits, we focus on the main findings and report average query times. Detailed times are reported in [69]. Results are robust averages over 5 repetitions. *All times are in seconds.* Experiments with the TreeForce system will be denoted by "TF". The baseline approach for our system uses the depth-summary, this is denoted by *TF-depth*. Variants of our system with different summaries introduced in Section 4 are denoted *TF-{label,path,path+prefix}*. We denote by *TF-part-parall-path* our method with partitioning and parallelization, with the path summary; this is our best resulting approach. The variant of this technique *without* parallelization is denoted by *TF-part-path*.

Linear Scalability of the Underlying Database. Figure 8 illustrates how MongoDB reacts to the evaluation of 1) larger rewriting sets on 2) larger volumes of data produced by XMark^{JR}. Every data point indicates the time taken for evaluating a rewriting set. We can clearly see that the response time is linear both in the size of the rewriting sets and in the size of the data. This is expected as the evaluation algorithm of MongoDB is based on a database scan. This observation is also confirmed on DBLP^{JR} and GitHub^{JR}.

Summary-based Filtering is Critical. Figure 9 shows average query answering times for the depth, label, path and path+prefix summaries for DBLP^{JR} and GitHub^{JR}. It reports both the query rewriting and the evaluation time. For the path+prefix summary, we fixed $k = 5$, i.e., prefixes have length 5. We also tested longer prefixes but this did not bring any sensible improvement. For DBLP^{JR}, the label summary leads to an average speedup of 3x on a single query. This raises to 5x for the path summary and to 23x when k-prefixes are added. For GitHub^{JR}, the label summary leads to a speedup of 4x on average, which raises to 17x for the path summary. While for GitHub^{JR} prefixes did not improve much things, for DBLP^{JR} queries with *text values*, gains with k-prefixes can be in orders of magnitude for single queries [69]. Overall, it is unavoidable to use data summaries for efficiency as the underlying database system is *unable to discard* empty queries based on their structure, which slows down performances. Summaries have no effect on XMark^{JR}, which by design produces almost no empty queries, and forces the underlying database to evaluate all queries in a rewriting set. However, summaries will have again the same powerful filtering role *combined with partitioning*, as discussed next. As a last remark, summaries were relatively lightweight for DBLP^{JR} and GitHub^{JR}. For DBLP^{JR}, they included 50 labels, 188 paths, and 957K 5-prefixes while GitHub^{JR} had 250 labels, 991 paths, and 3.3M 5-prefixes.

Evaluation Dominates Query Rewriting Costs. Recall that query rewriting is divided in 1) automaton construction and 2) summary-based filtering and query generation. As expected, the query rewriting time depends on the complexity of rules and the size of the resulting rewriting set [69]. For DBLP^{JR}, the average automaton construction time was 300ms and within 1 second even for thousands of rewritings. For GitHub^{JR} the average automaton construction time was 1.5s and within 3.5 seconds. For XMark^{JR}, it was within 0.1 seconds in general. After the automaton construction follow filtering and query generation. Rewriting generation was on average in the order of tens of milliseconds and within 200ms for large rewritings, across all benchmarks. Figure 9 allows one to compare the rewriting time (automaton construction+query generation) and the evaluation time of a rewriting set, with different data summaries. The main observation is that for large volumes of data query answering is likely to be dominated by the evaluation of a rewriting-set rather than by rewriting itself. Also, automaton construction could be boostrapped off-line as discussed in Section 6 further reducing rewriting costs at query time.

Partitioning and Parallelization Bring Big Improvements. For partitioning, we used the Λ_d function with $d = 2$ (Section 4). This led to 8 partitions for DBLP^{JR}, 30 for GitHub^{JR}, and between 6 and 130 for XMark^{JR}. We report the results for the path summary *without prefixes*. We use this as a yardstick to show that partitioning and parallelization have sensible effects; gains cannot decrease when prefixes are added. Recall we focus on *physical* partitioning; we found logical partitioning having similar effectiveness [69].

For DBLP^{JR}, we start by looking at the gain provided by partitioning alone (i.e., without parallelization), as illustrated in Figure 10 (*TF-part-path*). The advantage of partitioning alone is that *i*) a query is evaluated on smaller portions of the database and *ii*) partitions can be *skipped* when the data summary rules out any answer on a specific partition. In this experiment, partitions are queried sequentially and times added up. For DBLP^{JR} queries that took more than 200ms (20 out 22) the average speedup for (sequentially) querying the partition was of 16.1x; the value increases if all 22 queries are considered [69]. For GitHub^{JR}, the average speedup was 30x. By introducing *parallelization* (*TF-part-parall-path*) the runtime is dominated by the time of querying the "slowest" partition (to which we add the management of the parallel data access). So, the gains further raise to 22.9x for (again: 20 out of 22) DBLP queries and 53x for GitHub^{JR} wrt the baseline TF-depth. For comparison, recall that the path summary had an average speedup of 5.5x (on all 22 queries) on DBLP^{JR}, and of 17.6x on GitHub^{JR}, which means partitioning and parallelization allow one to go 2x to 4x faster wrt the path summary alone on these datasets. As said above, we reported on the path summary to have a yardstick for performances. Interestingly, we found the label summary on DBLP^{JR} to provide essentially the same results [69]. This is because partitions have a reduced variability of data, hence labels are used with less ambiguity. So, summaries can be both very small and efficient when combined with partitioning. As expected, adding prefixes to the path summary improves the filtering power and reduces the size of rewritings. This can again further improve the speed-up by at least 3x on DBLP^{JR} with respect to using partitioning, parallelization and the path summary *without* k-prefixes. Overall, these results show the gains provided by partitioning and parallelization.

Partitioning Leads Towards Horizontal Scalability. Partitioning and parallelization can be extremely efficient in distributing the query answering effort on large databases. Figure 11 (left) reports answering times on an XMark^{JR} ruleset generating rewritings of size 100 over several instances. Data points indicate the average query time for the 10 XMark^{JR} queries. Let us focus on two variants of our system: the baseline (*TF-depth*) and with partitioning, parallelization, and the path summary (*TF-part-parall-path*). While for small data the two are closer, gains can reach peaks of two orders of magnitude for single queries on large JSON collections. The average gain on 100M is 2x (over all 23 rulesets), and raises 13.5x on 1G and to 68.2x for 10GB and to 88.8x for 50G. Figure 11 (right) reports single query time with *TF-part-parall-path* for 10GB and 50G XMark^{JR} data (note, across all rulesets). We observe that the greater is the number of partitions attacked by a query, the lower is the query evaluation time, as the querying effort can be better distributed. For 10GB data we can answer queries with 100 rewritings in 1.3 seconds, which is encouraging. This suggests that other partitioning functions could achieve better load balancing leading towards horizontal scalability; this is left for future work.

No Stored Inferences and Low Memory Consumption Our method achieves both efficiency and scalability thanks the underlying database. Its persistent storage enables querying on large data, and query rewriting avoids to store any inference. Let us go back to Figure 11 (left), showing average query times for rewriting sets of size 100. Experiments confirm that in-memory materialization with VLog is extremely efficient, especially on small to medium instances, because of its dedicated algorithms and data-structures. TreeForce with partitioning, parallelization, and path summary is

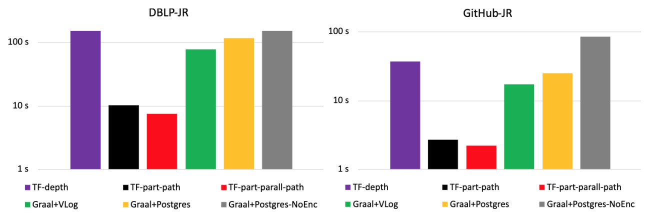

Figure 10: Answering Time (average) - Rewriting Systems

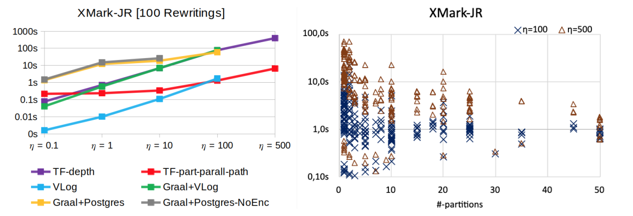

Figure 11: Answering Time: (left) Systems (right) Partitions

slower on small instances but, as data increases, it reaches similar performances (already for $\eta = 100$). This was confirmed also on XMark^{JR} ruleset generating rewritings of size 10 and 500 [69]. For $\eta = 500$, VLog could not load the data as it consumed all of the RAM we allocated. This shows that in-memory solutions may require powerful hardware for their deployment on large datasets, while relying on database solutions allows one to scale with modest memory resources. For DBLP and GitHub the average query time for VLog was 5.9s on DBLP (against 7.4s for *TF-part-parall-path*) and 0.7s on GitHub^{JR} (against 2.2s for *TF-part-parall-path*). For a fair comparison, this data does *not* include the time taken by VLog to apply the rules alone (without computing answers to queries) which can be considered either a one-of operation or not. We report however that rule application time was 160s for DBLP and 65s for GitHub^{JR} (no input nor output time). We conclude that for one-of operations our approach can be relevant also on smaller instances. Finally, note that VLog rule-application is negligible for XMark^{JR} as relabeling rules are kept intensional [57]; this makes the comparison of Figure 11 meaningful. All of this is interesting also because a DBMS can be considered at disadvantage when compared with main-memory solutions in terms of pure response time.

Rewriting Algorithms Tailored for Data-Tree. For GitHub^{JR} and XMark^{JR}, we could use Graal to compute rewritings of queries. However, this was not possible for all DBLP^{JR} queries as the translation of DBLP^{JR} tree also yields Datalog$^{\pm}$ rules which are recursive. Indeed, rewriting with Graal terminated only on 6 queries out of 22. Note that this is not a limitation of this tool, but of any general rewriting algorithm that makes no hypothesis on the shape of data. In contrast, our rewriting algorithm tailored for tree rules terminated in all cases. Hence, times reported next for rewriting-based approaches on DBLP^{JR} only consider those 6 queries.

The main feature of interest of rewriting-based systems is their frugality in terms of main memory. However, they may face other challenges when scaling out reasoning. The first, is the evaluation of large rewriting sets, which is a well-known issue [33]. The second is computing dictionary encoding over large collections, which is a question in itself [81]. Figure 10 illustrates rewriting-based approaches on DBLP$^{\mathrm{JR}}$ andGitHub$^{\mathrm{JR}}$ showing the effectiveness of our optimizations. XMark$^{\mathrm{JR}}$ results for rewriting-based systems are reported in Figure 11. We include only average times for the cases where all of the 10 XMark$^{\mathrm{JR}}$ queries were answered by a system [69]. For $\eta = 500$, again data could not be loaded in main memory by VLog. On the other side, Graal could not encode the data, as the dictionary encoding computation ran out of memory. By disabling dictionary encoding data was loaded into PostgreSQL. However, already for $\eta = 100$, disabling encoding did not allow to answer all the queries of the benchmark (recall, each with 100 rewritings). Our approach, in both its baseline and optimized versions, was able to provide answers across all benchmarks. Overall, we believe these results on rewriting over MongoDB to be very promising.

8 RELATED WORK AND CONCLUSION

There has been little work on reasoning on top of JSON databases. Prior work studied path-rules for JSON [23, 67]; we focus on tree-rules. More specifically, our *frontier-constrained* tree-rules extend the context-free fragment of suffix-*path* rules from [23], while keeping suitability for query rewriting on document-stores. Still on the theoretical level, the work of [26, 56] focuses on the computational complexity of some logical query languages for JSON. Note that the frameworks from [23, 26, 56, 67] have not been implemented. Another line of work considers the definition of virtual RDF views of JSON data as well as reasoning with RDFS/OWL ontologies at the RDF level [24, 31, 32]; we do reasoning at the JSON level.

With respect to the area of knowledge graphs and hypergraphs, constrained tree-rules allow for *existential nodes* in the rule head: these are analogous to *existentially quantified variables* in Datalog$^{\pm}$ [16, 19, 34]. As already said, our decidable fragment can be seen as a strict subset of body-acyclic frontier-guarded existential rules [17], as well as of the description logic $\mathcal{ELHV}$ [63]. Many reasoners have been implemented for knowledge graphs. State-of-the-art materialization-based systems include in-memory solutions like VLog [41], RDFox [68] and Vadalog [20] (the latter being restricted to a specific rule fragment) and RDBMS-based solutions like LLunatic [51] and DLV [12]. Note that materialization does not always halt for frontier-constraint tree-rules. State-of-the-art rewriting-based systems include OnTop [37] and Mastro [39], both devoted to lightweight OWL ontologies incomparable with our rules, and Graal for existential rules [15]. Note that Graal accepts any set of existential rules and conjunctive query, but query rewriting terminates only if the rewriting set is finite [61], a property that is not fulfilled by frontier-constrained tree-rules (recall Figure 3). Also, our query rewriting technique based on semi-twig-unifiers exploits the fact that data is tree-shaped and generates only tree queries.

Various formalisms for reasoning on XML and object-oriented databases have been studied [7, 18, 27, 29, 45, 60, 65, 74]. A difference between our framework and most of these proposals is that they are tied to XML and XPath. Among the rule-based languages for XML, decidable fragments such as XPathLog [65], Elog [18] and F-Logic [59, 60], which do not allow for existential nodes, are incomparable with our frontier-constrained tree-rules in terms of expressivity. Another sharp difference is that [7, 18, 60, 65] have been designed with a forward-chaining reasoning in mind; our focus is query rewriting. Xcerpt [74] is the closest language to our tree-rules for which backward chaining (over ordered trees) has been studied [73]. Our work differs as [73, 74] provides no guarantees for the termination of Xcerpt rules, while we identify a decidable fragment via query rewriting (over unordered trees). Active XML [7] is not based on rules but on function calls. UnQL [29] and XML-QL [45] are not rule-languages but query-languages based on recursive functions. Although these approaches can be assimilated with ours, their technical development is quite different. A last crucial difference with all these proposals is that our work is the first to consider (and implement) the use of NoSQL document stores to support rule-based query answering over data-trees.

Rewriting-based techniques have been considered for answering queries over virtual XML views defined either from SQL data [49, 50, 77] or XML data [47, 48, 79] to support data-integration and secure access to information. Note first that view languages are different from rule languages, which are typically more general. Automata have been proved effective for tackling problems related to XML queries in many settings [44, 76]. Very few work considers the presence of recursion in the definition of views. From the automata perspective, the work of [48] is the closest to ours in spirit as it considers automata-based rewriting for regular XPath queries over recursive XML views. However, the decidable language stemming from recursive view definitions is much closer to [18, 60, 65] which makes it orthogonal to frontier-constrained tree-rules. Further, [48] produces *regular*-XPath rewritings which are not supported by any system and calls for an ad-hoc implementation. In contrast, our instance-aware method outputs tree-queries and relies on efficient NoSQL systems. To conclude on this matter, note that the problem of answering queries with materialized views [54] (the input query targets the source database, rewritings target the views) is opposite to the one with virtual views (the input query targets the virtual database, rewritings target the source database). Our approach can nevertheless benefit from the presence of materialized views containing certain answers to queries over the original database. Having materialized views opens for two types of opportunities. First, evaluating a *subset* of the rewriting-set of a query on the materialized views. Second, the development of early stop techniques for our rewriting algorithms when answers to single rewritings (over views) are found. The study of these questions is however left as the subject of future work.

In conclusion, we have shown that NoSQL stores can be used to build an efficient and scalable tool for reasoning on JSON databases. We have outlined the importance of instance-aware query rewriting techniques and shown that the combination of data summaries, partitioning and parallelization result in high performances on large data. We believe that some of these ideas can be transposed to the case of knowledge graphs and hypergraphs. Future work also includes the extension to more expressive rule languages, for example including suitable forms of equality constraints and contexts.

ACKNOWLEDGMENTS

We are grateful to Florent Tornil for his help with the experiments. This work was financially supported by the ANR project CQFD (ANR-18-CE23-0003).

REFERENCES

[1] 2009. (Software) MongoDB. Retrieved 2023-05-01 from www.mongodb.com

[2] 2011. GitHub (GH) Archive. Retrieved 2023-05-01 from www.gharchive.org

[3] 2011. (Software) ArangoDB. Retrieved 2023-05-01 from www.arangodb.com

[4] 2016. (Software) AsterixDB. Retrieved 2023-05-01 from https://asterixdb.apache.org

[5] 2018. (Software) Rulewerk. Retrieved 2023-05-01 from www.github.com/knowsys/rulewerk

[6] 2022. (Software) InteGraal. Retrieved 2023-05-01 from https://gitlab.inria.fr/rules/integraal

[7] Serge Abiteboul, Omar Benjelloun, and Tova Milo. 2004. Positive Active XML. In *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 14-16, 2004, Paris, France*, Catriel Beeri and Alin Deutsch (Eds.). ACM, 35–45. https://doi.org/10.1145/1055558.1055564

[8] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases: The Logical Level* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[9] Serge Abiteboul and Victor Vianu. 1999. Regular Path Queries with Constraints. *J. Comput. Syst. Sci.* 58, 3 (June 1999), 428–452. https://doi.org/10.1006/jcss.1999.1627

[10] Alsayed Algergawy, Marco Mesiti, Richi Nayak, and Gunter Saake. 2011. XML Data Clustering: An Overview. *ACM Comput. Surv.* 43, 4, Article 25 (oct 2011), 41 pages. https://doi.org/10.1145/1978802.1978804

[11] Afnan Alhazmi, Tom Blount, and George Konstantinidis. 2022. ForBackBench: A Benchmark for Chasing vs. Query-Rewriting. *Proc. VLDB Endow.* 15, 8 (jun 2022), 1519–1532. https://doi.org/10.14778/3529337.3529338

[12] Mario Alviano, Wolfgang Faber, Nicola Leone, Simona Perri, Gerald Pfeifer, and Giorgio Terracina. 2010. The Disjunctive Datalog System DLV. In *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers (Lecture Notes in Computer Science)*, Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Jon Sellers (Eds.), Vol. 6702. Springer, 282–301. https://doi.org/10.1007/978-3-642-24206-9_17

[13] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1371–1382. https://doi.org/10.1145/2723372.2742796

[14] Jean-François Baget, Alain Gutierrez, Michel Leclère, Marie-Laure Mugnier, Swan Rocher, and Clément Sipieter. 2015. Datalog+, RuleML and OWL 2: Formats and Translations for Existential Rules. In *Proceedings of the RuleML 2015 Challenge, the Special Track on Rule-based Recommender Systems for the Web of Data, the Special Industry Track and the RuleML 2015 Doctoral Consortium hosted by the 9th International Web Rule Symposium (RuleML 2015), Berlin, Germany, August 2-5, 2015 (CEUR Workshop Proceedings)*, Nick Bassiliades, Paul Fodor, Adrian Giurca, Georg Gottlob, Tomás Kliegr, Grzegorz J. Nalepa, Monica Palmirani, Adrian Paschke, Mark Proctor, Dumitru Roman, Fariba Sadri, and Nenad Stojanovic (Eds.), Vol. 1417. CEUR-WS.org, 15 pages. https://ceur-ws.org/Vol-1417/paper9.pdf

[15] Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, Swan Rocher, and Clément Sipieter. 2015. Graal: A Toolkit for Query Answering with Existential Rules. In *Rule Technologies: Foundations, Tools, and Applications - 9th International Symposium, RuleML 2015, Berlin, Germany, August 2-5, 2015, Proceedings (Lecture Notes in Computer Science)*, Nick Bassiliades, Georg Gottlob, Fariba Sadri, Adrian Paschke, and Dumitru Roman (Eds.), Vol. 9202. Springer, 328–344. https://doi.org/10.1007/978-3-319-21542-6_21

[16] Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, and Eric Salvat. 2011. On rules with existential variables: Walking the decidability line. *Artif. Intell.* 175, 9-10 (2011), 1620–1654. https://doi.org/10.1016/j.artint.2011.03.002

[17] Jean-François Baget, Marie-Laure Mugnier, Sebastian Rudolph, and Michaël Thomazo. 2011. Walking the Complexity Lines for Generalized Guarded Existential Rules. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, Toby Walsh (Ed.). IJCAI/AAAI, 712–717. https://doi.org/10.5591/978-1-57735-516-8/IJCAI11-126

[18] Robert Baumgartner, Sergio Flesca, and Georg Gottlob. 2001. Supervised Wrapper Generation with Lixto. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass (Eds.). Morgan Kaufmann, 715–716. http://www.vldb.org/conf/2001/P715.pdf

[19] Catriel Beeri and Moshe Y. Vardi. 1984. A Proof Procedure for Data Dependencies. *J. ACM* 31, 4 (Sept. 1984), 718–741. https://doi.org/10.1145/1634.1636

[20] Luigi Bellomarini, Emanuel Sallinger, and Georg Gottlob. 2018. The Vadalog System: Datalog-based Reasoning for Knowledge Graphs. *Proc. VLDB Endow.* 11, 9 (2018), 975–987. https://doi.org/10.14778/3213880.3213888

[21] Michael Benedikt, George Konstantinidis, Giansalvatore Mecca, Boris Motik, Paolo Papotti, Donatello Santoro, and Efthymia Tsamoura. 2017. Benchmarking the Chase. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Chicago, Illinois, USA) *(PODS '17)*. Association for Computing Machinery, New York, NY, USA, 37–52. https://doi.org/10.1145/3034786.3034796

[22] Michael Benedikt, Boris Motik, and Efthymia Tsamoura. 2018. Goal-Driven Query Answering for Existential Rules With Equality. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, Sheila A. McIlraith and Kilian Q. Weinberger (Eds.). AAAI Press, 1761–1770. https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16927

[23] Meghyn Bienvenu, Pierre Bourhis, Marie-Laure Mugnier, Sophie Tison, and Federico Ulliana. 2017. Ontology-Mediated Query Answering for Key-Value Stores. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, Carles Sierra (Ed.). ijcai.org, 844–851. https://doi.org/10.24963/ijcai.2017/117

[24] Elena Botoeva, Diego Calvanese, Benjamin Cogrel, Martin Rezk, and Guohui Xiao. 2016. OBDA Beyond Relational DBs: A Study for MongoDB. In *Proceedings of the 29th International Workshop on Description Logics, Cape Town, South Africa, April 22-25, 2016.* 12 pages. http://ceur-ws.org/Vol-1577/paper_40.pdf

[25] Elena Botoeva, Diego Calvanese, Benjamin Cogrel, and Guohui Xiao. 2018. Expressivity and Complexity of MongoDB Queries. In *21st International Conference on Database Theory, ICDT 2018, March 26-29, 2018, Vienna, Austria (LIPIcs)*, Benny Kimelfeld and Yael Amsterdamer (Eds.), Vol. 98. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:23. https://doi.org/10.4230/LIPIcs.ICDT.2018.9

[26] Pierre Bourhis, Juan L. Reutter, Fernando Suárez, and Domagoj Vrgoc. 2017. JSON: Data model, Query languages and Schema specification. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts (Eds.). ACM, 123–135. https://doi.org/10.1145/3034786.3056120

[27] François Bry, Norbert Eisinger, Thomas Eiter, Tim Furche, Georg Gottlob, Clemens Ley, Benedikt Linse, Reinhard Pichler, and Fang Wei. 2007. Foundations of Rule-Based Query Answering. In *Proceedings of the Third International Summer School Conference on Reasoning Web* (Dresden, Germany) *(RW'07)*. Springer-Verlag, Berlin, Heidelberg, 1–153.

[28] Peter Buneman, Wenfei Fan, and Scott Weinstein. 2000. Path constraints in semistructured databases. *J. Comput. System Sci.* 61, 2 (2000), 146–193.

[29] Peter Buneman, Mary Fernandez, and Dan Suciu. 2000. UnQL: a query language and algebra for semistructured data based on structural recursion. *The VLDB Journal* 9 (2000), 76–110.

[30] Maxime Buron. 2020. *Raisonnement efficace sur des grands graphes hétérogènes.* These de doctorat. Institut polytechnique de Paris. http://www.theses.fr/2020IPPAX061

[31] Maxime Buron, François Goasdoué, Ioana Manolescu, and Marie-Laure Mugnier. 2020. Obi-Wan: Ontology-Based RDF Integration of Heterogeneous Data. *Proc. VLDB Endow.* 13, 12 (2020), 2933–2936. https://doi.org/10.14778/3415478.3415512

[32] Maxime Buron, François Goasdoué, Ioana Manolescu, and Marie-Laure Mugnier. 2020. Ontology-Based RDF Integration of Heterogeneous Data. In *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, Angela Bonifati, Yongluan Zhou, Marcos Antonio Vaz Salles, Alexander Böhm, Dan Olteanu, George H. L. Fletcher, Arijit Khan, and Bin Yang (Eds.). OpenProceedings.org, 299–310. https://doi.org/10.5441/002/edbt.2020.27

[33] Damian Bursztyn, François Goasdoué, and Ioana Manolescu. 2016. Teaching an RDBMS about ontological constraints. *Proc. VLDB Endow.* 9, 12 (2016), 1161–1172. https://doi.org/10.14778/2994509.2994532

[34] Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. 2009. A general datalog-based framework for tractable query answering over ontologies. In *Proceedings of the Twenty-Eigth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2009, June 19 - July 1, 2009, Providence, Rhode Island, USA*, Jan Paredaens and Jianwen Su (Eds.). ACM, 77–86. https://doi.org/10.1145/1559795.1559809

[35] Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. 2012. A general Datalog-based framework for tractable query answering over ontologies. *J. Web Semant.* 14 (2012), 57–83. https://doi.org/10.1016/j.websem.2012.03.001

[36] Andrea Calì, Georg Gottlob, and Andreas Pieris. 2012. Towards more expressive ontology languages: The query answering problem. *Artif. Intell.* 193 (2012), 87–128. https://doi.org/10.1016/j.artint.2012.08.002

[37] Diego Calvanese, Benjamin Cogrel, Sarah Komla-Ebri, Roman Kontchakov, Davide Lanti, Martin Rezk, Mariano Rodriguez-Muro, and Guohui Xiao. 2017. Ontop: Answering SPARQL queries over relational databases. *Semantic Web* 8, 3 (2017), 471–487.

[38] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. 2007. Tractable Reasoning and Efficient Query Answering in Description Logics: The *DL-Lite* Family. *J. Autom. Reasoning* 39, 3 (2007), 385–429. https://doi.org/10.1007/s10817-007-9078-x

[39] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, Mariano Rodriguez-Muro, Riccardo Rosati, Marco Ruzzi, and

Domenico Fabio Savo. 2011. The MASTRO system for ontology-based data access. *Semantic Web* 2, 1 (2011), 43–53. https://doi.org/10.3233/SW-2011-0029

[40] Diego Calvanese, Magdalena Ortiz, and Mantas Simkus. 2016. Verification of Evolving Graph-structured Data under Expressive Path Constraints. In *19th International Conference on Database Theory, ICDT 2016, Bordeaux, France, March 15-18, 2016 (LIPIcs)*, Wim Martens and Thomas Zeume (Eds.), Vol. 48. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 15:1–15:19. https://doi.org/10.4230/LIPIcs.ICDT.2016.15

[41] David Carral, Irina Dragoste, Larry González, Ceriel J. H. Jacobs, Markus Krötzsch, and Jacopo Urbani. 2019. VLog: A Rule Engine for Knowledge Graphs. In *The Semantic Web - ISWC 2019 - 18th International Semantic Web Conference, Auckland, New Zealand, October 26-30, 2019, Proceedings, Part II (Lecture Notes in Computer Science)*, Chiara Ghidini, Olaf Hartig, Maria Maleshkova, Vojtech Svátek, Isabel F. Cruz, Aidan Hogan, Jie Song, Maxime Lefrançois, and Fabien Gandon (Eds.), Vol. 11779. Springer, 19–35. https://doi.org/10.1007/978-3-030-30796-7_2

[42] Didier Caucal. 2000. On word rewriting systems having a rational derivation. In *Foundations of Software Science and Computation Structures: Third International Conference, FOSSACS 2000 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2000 Berlin, Germany, March 25–April 2, 2000 Proceedings 3*. Springer, 48–62.

[43] Brian Chin, Daniel von Dincklage, Vuk Ercegovac, Peter Hawkins, Mark S. Miller, Franz Och, Chris Olston, and Fernando Pereira. 2015. Yedalog: Exploring Knowledge at Scale. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Dagstuhl, Germany, 63–78. http://drops.dagstuhl.de/opus/frontdoor.php?source_opus=5017

[44] Hubert Comon, Max Dauchet, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. 1997. *Tree automata techniques and applications*. http://www.grappa.univ-lille3.fr/tata

[45] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. 1999. A query language for XML. *Computer networks* 31, 11-16 (1999), 1155–1169.

[46] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. 2005. Data exchange: semantics and query answering. *Theor. Comput. Sci.* 336, 1 (2005), 89–124. https://doi.org/10.1016/j.tcs.2004.10.033

[47] Wenfei Fan, Chee-Yong Chan, and Minos Garofalakis. 2004. Secure XML querying with security views. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. 587–598.

[48] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. 2007. Rewriting regular XPath queries on XML views. In *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 666–675.

[49] Wenfei Fan, Jeffrey Xu Yu, Jianzhong Li, Bolin Ding, and Lu Qin. 2009. Query translation from XPath to SQL in the presence of recursive DTDs. *The VLDB journal* 18 (2009), 857–883.

[50] Mary Fernández, Yana Kadiyska, Dan Suciu, Atsuyuki Morishima, and Wang-Chiew Tan. 2002. SilkRoute: A framework for publishing relational data in XML. *ACM Transactions on Database Systems (TODS)* 27, 4 (2002), 438–493.

[51] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. 2014. That's All Folks! Llunatic Goes Open Source. *Proc. VLDB Endow.* 7, 13 (Aug. 2014), 1565–1568. https://doi.org/10.14778/2733004.2733031

[52] François Goasdoué, Ioana Manolescu, and Alexandra Roatiş. 2013. Efficient query answering against dynamic RDF databases. In *Proceedings of the 16th International Conference on Extending Database Technology (EDBT '13)*. Association for Computing Machinery, New York, NY, USA, 299–310. https://doi.org/10.1145/2452376.2452412

[53] Roy Goldman and Jennifer Widom. 1997. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB '97)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 436–445. http://dl.acm.org/citation.cfm?id=645923.671008

[54] Alon Y Halevy. 2001. Answering queries using views: A survey. *The VLDB Journal* 10 (2001), 270–294.

[55] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. 1995. Computing Simulations on Finite and Infinite Graphs. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS '95)*. IEEE Computer Society, USA, 453.

[56] Jan Hidders, Jan Paredaens, and Jan Van den Bussche. 2017. J-Logic: Logical foundations for JSON querying. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 137–149.

[57] Pan Hu, Jacopo Urbani, Boris Motik, and Ian Horrocks. 2019. Datalog Reasoning over Compressed RDF Knowledge Bases. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management, CIKM 2019, Beijing, China, November 3-7, 2019*, Wenwu Zhu, Dacheng Tao, Xueqi Cheng, Peng Cui, Elke A. Rundensteiner, David Carmel, Qi He, and Jeffrey Xu Yu (Eds.). ACM, 2065–2068. https://doi.org/10.1145/3357384.3358147

[58] Murtadha AI Hubail, Ali Alsuliman, Michael Blow, Michael Carey, Dmitry Lychagin, Ian Maxon, and Till Westmann. 2019. Couchbase Analytics: NoETL for Scalable NoSQL Data Analysis. *Proc. VLDB Endow.* 12, 12 (aug 2019), 2275–2286. https://doi.org/10.14778/3352063.3352143

[59] Michael Kifer. 2005. Rules and Ontologies in F-Logic. In *Reasoning Web, First International Summer School 2005, Msida, Malta, July 25-29, 2005, Tutorial Lectures (Lecture Notes in Computer Science)*, Norbert Eisinger and Jan Maluszynski (Eds.), Vol. 3564. Springer, 22–34. https://doi.org/10.1007/11526988_2

[60] Michael Kifer, Georg Lausen, and James Wu. 1995. Logical Foundations of Object-Oriented and Frame-Based Languages. *J. ACM* 42, 4 (1995), 741–843. https://doi.org/10.1145/210332.210335

[61] Mélanie König, Michel Leclère, Marie-Laure Mugnier, and Michaël Thomazo. 2015. Sound, complete and minimal UCQ-rewriting for existential rules. *Semantic Web* 6, 5 (2015), 451–475. https://doi.org/10.3233/SW-140153

[62] Mélanie König, Michel Leclere, and Marie-Laure Mugnier. 2015. Query rewriting for existential rules with compiled preorder. In *IJCAI: International Joint Conference on Artificial Intelligence*. 3006–3112.

[63] Markus Krötzsch and Sebastian Rudolph. 2014. Nominal Schemas in Description Logics: Complexities Clarified. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference, KR 2014, Vienna, Austria, July 20-24, 2014*, Chitta Baral, Giuseppe De Giacomo, and Thomas Eiter (Eds.). AAAI Press, 10 pages.

[64] Carsten Lutz, David Toman, and Frank Wolter. 2009. Conjunctive Query Answering in the Description Logic EL Using a Relational Database System. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, Craig Boutilier (Ed.). 2070–2075. http://ijcai.org/Proceedings/09/Papers/341.pdf

[65] Wolfgang May. 2002. A rule-based querying and updating language for XML. In *Database Programming Languages: 8th International Workshop, DBPL 2001 Frascati, Italy, September 8–10, 2001 Revised Papers 8*. Springer, 165–181.

[66] Walaa Eldin Moustafa, Vicky Papavasileiou, Ken Yocum, and Alin Deutsch. 2016. Datalography: Scaling datalog graph analytics on graph processing systems. In *2016 IEEE International Conference on Big Data (IEEE BigData 2016), Washington DC, USA, December 5-8, 2016*, James Joshi, George Karypis, Ling Liu, Xiaohua Hu, Ronay Ak, Yinglong Xia, Weijia Xu, Aki-Hiro Sato, Sudarsan Rachuri, Lyle H. Ungar, Philip S. Yu, Rama Govindaraju, and Toyotaro Suzumura (Eds.). IEEE Computer Society, 56–65. https://doi.org/10.1109/BigData.2016.7840589

[67] Marie-Laure Mugnier, Marie-Christine Rousset, and Federico Ulliana. 2016. Ontology-Mediated Queries for NOSQL Databases. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, Dale Schuurmans and Michael P. Wellman (Eds.). AAAI Press, 1051–1057. http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12395

[68] Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. 2015. RDFox: A highly-scalable RDF store. In *International Semantic Web Conference*. Springer, 3–20.

[69] Rodriguez Olivier, Ulliana Federico, and Mugnier Marie-Laure. 2023. *Scalable Reasoning on Document Stores via Instance-Aware Query Rewriting (with Appendix)*. Technical Report. https://gitlab.inria.fr/boreal-artifacts/pvldb2023

[70] Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. 2014. The SQL++ Semi-structured Data Model and Query Language: A Capabilities Survey of SQL-on-Hadoop, NoSQL and NewSQL Databases. *CoRR* abs/1405.3631 (2014). arXiv:1405.3631 http://arxiv.org/abs/1405.3631

[71] Rachel Pottinger and Alon Y. Levy. 2000. A Scalable Algorithm for Answering Queries Using Views. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB '00)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 484–495.

[72] J Richard Büchi. 1964. Regular canonical systems. *Archiv für mathematische Logik und Grundlagenforschung* 6, 3-4 (1964), 91–111.

[73] Finn Sebastian Schaffert. 2004. *Xcerpt: A rule-based query and transformation language for the Web*. Ph.D. Dissertation. Dissertation, LMU München: Faculty of Mathematics, Computer Science and Statistics.

[74] Sebastian Schaffert and François Bry. 2002. A gentle introduction to Xcerpt, a rule-based query and transformation language for XML. In *RuleML 2002, Proceedings of the International Workshop on Rule Markup Languages for Business Rules on the Semantic Web, 14 June 2002, Sardinia, Italy (In conjunction with the First International Semantic Web Conference ISWC2002 and hosted by SIG2 of the OntoWeb Network (CEUR Workshop Proceedings)*, Michael Schroeder and Gerd Wagner (Eds.), Vol. 60. CEUR-WS.org, 22 pages. https://ceur-ws.org/Vol-60/bry_schaffert.pdf

[75] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. 2002. XMark: A Benchmark for XML Data Management. In *Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, Hong Kong, August 20-23, 2002*. Morgan Kaufmann, 974–985. https://doi.org/10.1016/B978-155860869-6/50096-2

[76] Thomas Schwentick. 2007. Automata for XML—a survey. *J. Comput. System Sci.* 73, 3 (2007), 289–315.

[77] Jayavel Shanmugasundaram, Jerry Kiernan, Eugene J. Shekita, Catalina Fan, and John E. Funderburk. 2001. Querying XML Views of Relational Data. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass (Eds.). Morgan Kaufmann, 261–270. http://www.vldb.org/conf/2001/P261.pdf

[78] K Tuncay Tekle and Yanhong A Liu. 2011. More efficient datalog queries: subsumptive tabling beats magic sets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. 661–672.

[79] Alex Thomo and Srinivasan Venkatesh. 2008. Rewriting of visibly pushdown languages for xml data integration. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management*. 521–530.

[80] Efthymia Tsamoura, David Carral, Enrico Malizia, and Jacopo Urbani. 2021. Materializing knowledge bases via trigger graphs. *Proceedings of the VLDB Endowment* 14, 6 (2021), 943–956.

[81] Jacopo Urbani, Sourav Dutta, Sairam Gurajada, and Gerhard Weikum. 2016. KOGNAC: Efficient Encoding of Large Knowledge Graphs. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, Subbarao Kambhampati (Ed.). IJCAI/AAAI Press, 3896–3902. http://www.ijcai.org/Abstract/16/548