Ontology-Mediated Query Answering for Key-Value Stores

Meghyn Bienvenu, Pierre Bourhis, Marie-Laure Mugnier, Sophie Tison, Federico Ulliana

Ontology-Mediated Query Answering for Key-Value Stores

Meghyn Bienvenu CNRS France
Pierre Bourhis CNRS France
Marie-Laure Mugnier Univ. Montpellier France
Sophie Tison Univ. Lille 1 France
Federico Ulliana Univ. Montpellier France

Abstract We propose a novel rule-based ontology language for JSON records and investigate its computational properties. After providing a natural translation into first-order logic, we identify relationships to existing ontology languages, which yield decidability of query answering but only rough complexity bounds. By establishing an interesting and non-trivial connection to word rewriting, we are able to pinpoint the exact combined complexity of query answering in our framework and obtain tractability results for data complexity. The upper bounds are proven using a query reformulation technique, which can be implemented on top of key-value stores, thereby exploiting their querying facilities.

1 Introduction

Ontology-mediated query answering (OMQA) is a paradigm for accessing legacy data while taking into account knowledge expressed by a domain ontology. OMQA has received a lot of attention in the last decade, driven by the increasing importance of data integration [Poggi *et al.*, 2008] and the growing interest in semantic technologies. This paradigm can be seen as a family of techniques for querying a *virtual knowledge base* composed of factual knowledge (the data) and ontological knowledge. So far, most of the attention on OMQA has been on providing techniques to access factual knowledge held by relational and RDF databases. The ontological knowledge is often expressed using data-tractable description logics [Calvanese *et al.*, 2007; Baader *et al.*, 2008; Eiter *et al.*, 2012] or suitable fragments of existential rules / Datalog$^{\pm}$ [Baget *et al.*, 2011; Calì *et al.*, 2012].

In data-intensive applications, however, the database to access can be composed of key-value (KV) records in JSON format. This data can be sourced from the Web or produced and then analyzed within an organization. In addition to widespread of JSON as a format for data storage and exchange, the recent development of key-value stores (a family of NOSQL data-management systems) makes possible today to efficiently query JSON records [Cattell, 2010]. These data-management systems feature a set of low-level optimizations tailored for the format, which results in fast and scalable data access without requiring any translation to a relational system, which was one of the prominent solutions to handle semi-structured data [Chasseur *et al.*, 2013].

From an OMQA perspective, a natural question is how to integrate JSON records in the setting and, more precisely, how to design OMQA systems directly on top of KV-stores, thereby exploiting the store's native query engine capabilities. The evaluation of ontological queries over key-value stores has received only little attention so far, with the exceptions of [Mugnier *et al.*, 2016; Botoeva *et al.*, 2016] (see last section).

A prominent family of OMQA techniques adapted to query JSON records are those based on *query reformulation*[1]. The idea is to reformulate the input query by incorporating the relevant information from the ontology in such a way that the answers to the reformulated query over the input database are exactly the answers to the initial query over the virtual knowledge base. By contrast, the approach of *materialization* involves the generation of the whole virtual knowledge base. The main advantage of query reformulation is that it leaves the data untouched, which is often a desirable characteristic.

Contributions. We propose a rule language for JSON records that $i)$ allows one to express ontological knowledge in terms of hierarchies of properties, path inclusions, and mandatory paths $ii)$ admits tractable query answering, and $iii)$ enables a query reformulation mechanism implementable over KV-stores. We consider core navigational queries supported by KV-stores, namely get and check queries [Mugnier *et al.*, 2016]. Our setting has a natural first-order logic semantics. This allows us to identify relationships to existing ontology languages, namely existential rules and description logics, which yield decidability of query answering, but only provide rough upper complexity bounds because of the specificities of the shape of JSON records and ontology rules.

A distinctive feature of our language are *contextual rules*. Because JSON records are tree structures, a given value can assume a different meaning depending on the context (of the record) where it is employed. From an ontology modeling perspective, this means that it is interesting to have available rules with context that apply selectively on the record structure. Nevertheless, we show that this important feature makes the combined complexity of query answering jump from NLogSpace-complete (without contexts) to PSpace-complete (with contexts). Then, independently from contextual rules, the tree shape of JSON records also impacts the data complexity of query answering, which is PTime-hard if we con-

[1] We use 'query reformulation' rather than the more common 'query rewriting' to avoid any confusion with 'word rewriting'.

sider arbitrary data instances but in NLogspace in our setting. While PTime-hard problems are considered as probably inherently sequential, this last result gives us hope that OMQA can be efficiently parallelized.

Finally, we are able to establish an interesting and non-trivial connection with word rewriting systems. This allows us to import and extend existing techniques, thereby deriving tight complexity bounds for query answering. Beside the theoretical interest, these techniques allow us to represent the (possibly infinite) set of reformulations of an input query with a regular expression, which in turn can be evaluated over KV-stores by unfolding or using simple auxiliary structures like data-guides.

2 The Framework

We start by formalizing JSON records, core queries, and rules we consider, together with their first-order logic semantics.

JSON records. Let CONST and NULLS be (infinite) sets of constants and (labeled) nulls, respectively, and let KEYS be a finite set of keys. A JSON record, or *key-value record*, is a finite set of key-value pairs. A *value* is recursively defined as (i) an element of CONST∪NULLS (i.e., a terminal value), (ii) a sequence $[e_1 \dots e_n]$ where each e_i is a value and $n \geq 1$, or (iii) a record r of the form $\{(k_1, e_1) \dots (k_n, e_n)\}$ where each $k_i \in$ KEYS and each e_i is a value, with $n \geq 1$ and $k_i \neq k_j$ for $i \neq j$. A KV-store I is a set of records. A record can be associated with a *rooted labeled tree*, in which edges are labeled by keys, leaves are labeled by constant or null values, and all other internal nodes are unlabeled. A key-value pair (k, e) where e is a sequence is represented by several edges labeled by k leading to the nodes that represent the elements of e. [2] To illustrate, consider the following JSON record which describes a teaching department, containing its name, a professor, and two courses.

```
{ dept :
  { name : "CS",
    prof : {name : "Bob" , boss : "Alice", phone : "5-256" } ,
    course : ["AI", "Logic"] } }
```

The tree associated with this record is depicted in Figure 1. From now on, a record will be identified with its associated tree. Hence a *path* in a record is a path in the associated tree, which is defined as usual. A path $(v_0 \dots v_n)$ is *rooted* if v_0 is the root of the tree. It is *valued* if v_n is a leaf labeled by a constant. Its associated sequence of keys is $K = k_1 \dots k_n$ where k_i is the label of the edge (v_{i-1}, v_i). For convenience, we will sometimes denote a path by $v_0 K v_n$ (emphasizing the associated sequence of keys) and, when the path is valued, replace v_n by its label, e.g. $v_0 K c$ where $c \in$ CONST. We denote by $paths(r)$ the set of all paths in a record r.

Queries. We consider two types of queries, which capture the core of the native query languages of KV-stores, namely get and $check$ queries. These are evaluated from the *root* of a record. A get query is used to retrieve all constant values that lie at the end of a *rooted* and *valued* path. A $check$ query is a Boolean expression yielding true when a rooted path is found. On the example, the query $get(\texttt{dept.course})$ returns all courses provided by the department (here: *"Logic"*, *"AI"*), while the query $check(\texttt{dept.prof.phone})$ verifies whether at least a professor in the department has a phone.

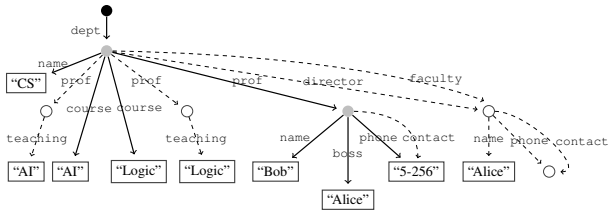

Figure 1: Tree associated with the record and rule application.

Formally, a query Q has the form $get(K)$ or $check(K)$ with K a sequence of keys. The set of answers to Q over a record r is denoted by $Q(r)$ and defined as follows. Let r_0 be the root of r. Then, $get(K)(r) = \{c \in \text{CONST} \mid r_0 K c \in paths(r)\}$ and $check(K)(r) = \{true\}$ if there is $r_0 K v \in paths(r)$, otherwise $check(K)(r) = \emptyset$. Finally, the set of answers to Q over a store I, denoted by $Q(I)$, is the union of $Q(r)$ for all $r \in I$.

Rule language. We now present a novel rule language for expressing ontological knowledge on key-value records. In the OMQA setting we define, KV-records are naturally the language for expressing factual knowledge on the application domain. We make an important distinction between the *internal* and the *leaf* nodes of the record's tree. We consider that internal nodes represent individuals, while leaf nodes hold the constants associated by properties. This choice is similar to that of the RDF language, where entities are URIs and values are literals. Building on this, we see the keys within records as binary relations (like roles in description logics), which hold either between a pair of individuals, or between individuals and constants. Another important aspect to consider when dealing with hierarchically organized information is the fact that the semantics of a value is determined by its position within the tree structure. We thus provide a mechanism to select a node based on the key-path defined by its ancestors, that we call a *context*.

The general form of a *KV-rule* is **Context : Body → Head**. Specifically, KV-rules can be of three types:

(A) $K : k_1 \to k_2$ (inclusion between keys)

(B) $K : K_1.val \to K_2$ (inclusion between valued paths)

(C) $K : K_1 \to \exists K_2$ (mandatory path)

where k_1, k_2 are keys and K, K_1, K_2 are sequences of keys. We allow the context K to be empty, which is denoted by ϵ. Moreover we also allow the body path K_1 of type (C) rule to be empty, but at the only condition that the context K is not. The context K defines where the rule can be possibly applied. Note that K does *not* need to start at the root of a record as (in contrast with queries) rules apply at any level.

[2] In this simplified data model, we do not represent the ordering on the elements of a sequence, since the considered queries will not exploit this order. Moreover, a sequence nested in a sequence is seen as a constant as core queries will not go through them either.

Type (A) rules define inclusions between keys, hence allowing to organize them hierarchically. Type (B) rules define inclusions between *valued* paths. They specify multiple ways of accessing a constant value in the record. Finally, type (C) rules are mandatory path assertions. These allow one to express that some path exists in the structure, even if its extremity is unknown. Importantly, since the body of the type (C) rules is possibly empty, these can be used to add properties to *all* individuals within a certain context.

To illustrate, consider the following KV-rules, whose evaluation on the example record is illustrated in Figure 1.

$\sigma_1 =$ `phone` $\rightarrow$ `contact`
$\sigma_2 =$ `course`.*val* $\rightarrow$ `prof.teaching`
$\sigma_3 =$ `dept : prof.boss`.*val* $\rightarrow$ `director.name`
$\sigma_4 =$ `dept : director` $\rightarrow$ `faculty`
$\sigma_5 =$ `dept.faculty` $: \epsilon \rightarrow \exists$`phone`

Rule σ_1 says that the key `phone` is a particular case of the key `contact` (key inclusion). The application of this rule creates an edge labeled with `contact` with the same end nodes as the edge labeled with `phone`. In the example, this allows us to positively answer the query $check($`dept.prof.contact`$)$ whose answer otherwise would be false. Rule σ_2 states that each course is taught by a professor. The *.val* in the rule body indicates that this applies only to valued paths (i.e., `course` keys pointing only towards constant values). The application of this rule creates a new valued path in the tree ending with a node labeled with the course name. Because of σ_2, the answer set of the query $get($`dept.prof.teaching`$)$ includes that of $get($`dept.course`$)$. Contextual rules are exemplified by the last three rules. Rules σ_3 and σ_4 model the fact that, within a department, the boss of a professor is always the director, and that `director` is a particular case of `faculty`, respectively. It is then possible to answer *"Alice"* to the query $get($`dept.faculty.name`$)$. Rule σ_5 is a mandatory path rule asserting that a key `phone` is always present within the context `dept.faculty`. Rules σ_1 and σ_5 make true the answer to the query $check($`dept.director.contact`$)$. Note that σ_5 has an empty body, as indicated by the presence of ϵ, hence it applies regardless of the actual sub-record content.

Finally, we conclude remarking that our language can also be seen as a constraint language, as studied in databases (see the related work section), in which case enforcing constraints on incomplete data may also lead to infer new data. In the same way, tuple-generating dependencies [Abiteboul *et al.*, 1995] can be seen either as constraints, or as ontological knowledge under the name of existential rules.

Rule application. As illustrated in Figure 1, rule application does not strictly preserve the tree-shape of records, since key inclusions replace keys by sets of keys (e.g., `phone, contact`). To formally define rule application we will thus consider trees with multi-edges (i.e., allowing for several edges with the same end nodes), called *multi-trees*. Similarly, paths with multi-edges will be called *multi-paths*.

We say that a rule $\sigma = K : K_1[.val] \rightarrow [\exists]K_2$ (optional parts between brackets) is *applicable* to a multi-tree r, if there exists a path $vKv_0K_1v_1 \in paths(r)$, which must be valued if [*.val*] is used. The effective application of σ at v_0 (with respect to v_1) consists of adding to the multi-tree a new path $v_0K_2v_2$, where $i)$ $v_2 = v_1$ if σ is of type (A), $ii)$ v_2 is a fresh node labeled as v_1 if σ is of type (B), otherwise $iii)$ v_2 is a fresh node labeled by a fresh null (σ is of type (C)).

Due to space restriction, we do not define the semantics of KV-stores and rules in terms of interpretations (those can, however, be obtained from the logical translation provided later). Instead, we define the notion of the saturation of a record r by a set of rules Σ, which can be seen as a canonical model of (Σ, r), representative of all models of (Σ, r), similarly to the chase on existential rules (see later). The breadth-first *saturation* of a multi-tree r by a set of KV-rules Σ, denoted by $Sat(\Sigma, r)$, is inductively defined as follows: $Sat^0(\Sigma, r) = r$ and, for any $i > 0$, $Sat^i(\Sigma, r)$ is obtained from $r^{i-1} = Sat^{i-1}(\Sigma, r)$ by performing in parallel all possible rule applications on r^{i-1}; finally, $Sat(\Sigma, r) = \bigcup_{i \geq 0} Sat^i(\Sigma, r)$. This notion is well-defined since the order in which the rule applications are performed at each level has no incidence on the result (up to the choice of labeled nulls).

Ontological query answering. A *knowledge base* (KB) is a pair $\mathcal{K} = (I, \Sigma)$, where I is a KV-store and Σ is a set of KV-rules. The notion of a (certain) answer is defined as follows. Given a query Q and a tree r, a value e belongs to $Q((\{r\}, \Sigma))$ if $e \in Q(Sat^k(\Sigma, r))$, for some $k \geq 0$. The set of answers to a query Q over $\mathcal{K}$, is defined as $Q(\mathcal{K}) = \bigcup_{r \in I} Q((\{r\}, \Sigma))$.

Note that materializing the saturation of a tree has several disadvantages of both practical and theoretical nature. First, it produces multi-trees, that are supported by some systems only. Second, the saturated tree may be infinite (e.g., the record $r = \{k : a\}$ with rule $\sigma : k.val \rightarrow k.k$ generates an infinite set of paths). In the remainder of the paper we will therefore consider the following *query reformulation problem*. Given a KB $\mathcal{K} = (I, \Sigma)$ and a $check$ or get query Q, compute a set of queries $\{Q_1, \ldots, Q_n\}$ such that $Q(\mathcal{K}) = \bigcup_i Q_i(I)$. Before tackling this question, we give a logical counterpart of our framework.

First-order logic translation. The framework translates naturally into first-order logic (FO). To a record we assign an existentially closed formula. E.g., for the example record: $\exists x_0, x_1(root(x_0) \wedge dept(x_0) \wedge name(x_0, CS) \wedge const(CS) \ldots \wedge prof(x_0, x_1) \wedge name(x_1, Bob) \wedge const(Bob) \wedge \ldots)$. Note that the root and the constant leaves are marked by atoms of the form $root(-)$ and $const(-)$ respectively.

A query $get(K)$ translates into a conjunctive query of the form $\exists x_0(root(x_0) \wedge FO(K, x_0, x_n) \wedge const(x_n))$, where $FO(K, x_0, x_n)$ is the translation of the key sequence K into a FO formula with free variables x_0 and x_n. A query $check(K)$ translates into a Boolean conjunctive query of the form $\exists x_0 \exists x_n(root(x_0) \wedge FO(K, x_0, x_n))$. Finally, we illustrate the translation of KV-rules on the example, omitting universal quantifiers for conciseness:

1. $phone(x_0, x) \rightarrow contact(x_0, x)$
2. $course(x_0, x) \wedge const(x) \rightarrow \exists z_0\ prof(x_0, z_0) \wedge teach(z_0, x)$
3. $dept(x_1, x_0) \wedge prof(x_0, y_1) \wedge boss(y_1, x) \wedge const(x)$
   $\rightarrow \exists z_0 director(x_0, z_0) \wedge name(z_0, x)$
4. $dept(x_1, x_0) \wedge director(x_0, x) \rightarrow faculty(x_0, x)$
5. $dept(x_1, x_2) \wedge faculty(x_2, x_0) \rightarrow \exists z phone(x_0, z)$

We observe that KV-rules are actually translated into *existential rules*, which are positive and conjunctive rules of the form $\forall \mathbf{x} \forall \mathbf{y}(Body[\mathbf{x}, \mathbf{y}] \rightarrow \exists \mathbf{z} Head[\mathbf{x}, \mathbf{z}])$. Moreover, the notions of a rule application and of an answer in the KV framework coincide with the same notions in the existential rule framework. This allows us to obtain the following result (where $\models$ denotes classical logical entailment, and $FO(Q)$ and $FO(\mathcal{K})$ are the FO translations of Q and $\mathcal{K}$).

Proposition 1 (Soundness and Completeness)**.** *Let $\mathcal{K} = (I, \Sigma)$ be a KB. For any query $Q{=}check(K)$, we have $Q(\mathcal{K}) \neq \emptyset$ iff $FO(\mathcal{K}) \models FO(Q)$. For any query $Q{=}get(K)$ and constant value a, we have $a \in Q(\mathcal{K})$ iff $FO(\mathcal{K}) \models FO_{x_n \mapsto a}(Q)$, where $x_n \mapsto a$ is the substitution of x_n by a.*

As already mentioned, our saturation notion corresponds to the fundamental (breadth-first) *chase* tool on existential rules.

To study decidability and complexity, we consider the following decision problems: Given a KB $\mathcal{K}$ and a $check$ query Q, is $Q(\mathcal{K}) \neq \emptyset$? Given a KB $\mathcal{K}$, a get query Q and a constant a, does $a \in Q(\mathcal{K})$ hold?

We first point out that the restriction of path inclusions to valued paths is crucial for decidability. Were we to remove this restriction, query answering would become undecidable, as already observed in several contexts [Abiteboul and Vianu, 1999; Buneman *et al.*, 2000; Calvanese *et al.*, 2016; Mugnier *et al.*, 2016].

Relation to existing ontology languages. By slightly modifying the FO translation of path inclusion rules (without incidence on query entailment), we see that KV-rules can be translated into a fragment of existential rules with decidable query answering, namely *frontier-guarded existential rules*, in which an atom from the rule body (called a guard) contains all the frontier variables, which are the variables shared by the body and the head of the rule [Baget *et al.*, 2011]. To do so, for any rule σ of type B, $FO(\sigma) = \forall x_0 \forall x(B \rightarrow H)$ is replaced by $|\textsc{Const}[I]|$ rules obtained by substituting x with each constant $c \in \textsc{Const}[I]$ (where $\textsc{Const}[I]$ is the restriction of $\textsc{Const}$ to I). A closer look at our objects allows to recast our problem as atomic query answering over fixed-arity guarded rules, which provides an upper bound for combined complexity, namely ExpTime [Calì *et al.*, 2013]. Since the translation to frontier-guarded rules is not data-independent, it does not yield any useful upper bound for data complexity.

Even though KV-rules are based on paths, their logical translation does not fit into any known DL dialect, due to the presence of contextual key inclusions (rules of type A). Were we to consider only context-free key inclusions, then the preceding translation into frontier-guarded existential rules would be expressible in the DL $\mathcal{ELHIO}$ (where role inclusions capture the key inclusions, nominals allow us to speak of constants, and inverse roles are used to capture contexts in rules of type B and C). However, this correspondence does not yield improved complexity results.

3 Resolution through Word Rewriting System

Our approach to the query answering problem is to reformulate an input query into a *pointed regular path query* (PRPQ) [Abiteboul and Vianu, 1999] which represents in a succinct way the (possibly infinite) set of reformulations of a check or get query. A PRPQ is defined by a regular language, which also is pointed, in the sense that it is evaluated from a fixed node (for JSON, this is always the root of the record). The semantics of a PRPQ over JSON records is as follows. Let L be (the language of) a PRPQ and r be a tree corresponding to a JSON record, then we say that a node v in r is an answer to the query if there exists a word K belonging to L such that $root(r).K.v$ is a path in r. In the remainder, we suppose the reader to be familiar with regular languages.

We want to show that the problem of computing the PRPQ reformulations of check and get queries can be reduced to that of computing the (regular) language of ancestors in a word rewriting system. Towards this goal, starting from a set of KV-rules Σ, we first define a word rewriting system $\mathcal{R}_\Sigma$ which simulates the forward-chaining application of the rules in Σ. Then, computing the language of ancestors of words derived by $\mathcal{R}_\Sigma$ will mimic the reformulation into PRPQs.

Word rewriting systems. A word (or string) rewriting system $\mathcal{R}$ over a finite alphabet A consists of a finite set of word rewriting rules of the form $(u_1, u_2) \in A^* \times A^*$, where u_1 and u_2 are respectively the left and right hand side of a rule. We say that a word w rewrites to w' following the rule (u_1, u_2) if $w = w_1 \cdot u_1 \cdot w_2$ and $w' = w_1 \cdot u_2 \cdot w_2$, where "$\cdot$" denotes word concatenation. We denote by $\rightarrow_\mathcal{R}$ the single-step rewriting relation and by $\rightarrow^*_\mathcal{R}$ its reflexive and transitive closure. We write $w \rightarrow^*_\mathcal{R} w'$ if there exists a (possibly empty) word sequence $(w =) w_1, \ldots, w_n (= w')$ such that w_j rewrites to w_{j+1} following a rule of $\mathcal{R}$. Given a regular language L, we denote by $Anc_\mathcal{R}(L)$ the set of *ancestors* of L by $\mathcal{R}$, which is defined as $Anc_\mathcal{R}(L) = \{w \mid \exists w' \in L, w \rightarrow^*_\mathcal{R} w'\}$.

An important family of rewriting systems we are interested in are *suffix* rewriting systems. Intuitively, these are rewriting systems restricting the application of rules thereby allowing them to rewrite only the suffixes of words. Formally, we say that $w_1 \cdot u_2 \cdot w_2$ is a suffix rewriting of $w_1 \cdot u_1 \cdot w_2$ with the rule (u_1, u_2) whenever w_2 is empty. It is well known that, if only suffix rewritings are allowed, then $Anc_\mathcal{R}(L)$ is a regular language [Büchi, 1962; Caucal, 2000].

At this point, it is not difficult to see that, for the case of *context-free* KV-rules, word rewriting can mimic the reformulation of KV-rules of type (A) and suffix word rewriting that of KV-rules of type (B). To capture the full reasoning capabilities of KV-rules we need however to introduce a more general notion of rewriting system, that we call an *extended rewriting suffix system* (*ERS*). An *ERS* contains two types of rules, namely relabeling and suffix rules. Intuitively, the former type of rules replace a single letter, while the latter are used for suffix rewriting. Furthermore, we consider suffix rules made of pairs of regular languages (rather than simply pairs of words) which will allow for a more concise representation of rewriting rules. Conceptually, a rule (L, R) with L and R regular languages means that any word belonging to L can be replaced by any word of R. Finally, all types of rules in an *ERS* are associated with a context restricting their application. This is necessary to mimic the behaviour of KV-rules with contexts. As before, contexts are written as regular languages and make a rule selectively applicable only to subwords that are preceded by a word defined by the context.

Formally, an *ERS* is made of:

- contextual relabeling rules of the form $C : a \mapsto b$ where C is a regular language, and a and b two letters. Their semantics is a follows: $u_1 \cdot a \cdot u_2$ rewrites to $u_1 \cdot b \cdot u_2$ by such type of rule if u_1 belongs to C;
- contextual extended suffix rules of the form $C : L \mapsto R$ with C, L, R regular languages. Their semantics is a follows: $u_1 \cdot l$ rewrites to $u_1 \cdot r$ by such type of rule if u_1 belongs to C, l belongs to L, and r belongs to R.

To illustrate, $(c_1 \mid c_2)^+ : a \mapsto b$ is a contextual relabeling rule replacing a with b in a word, whenever this is preceded by any (non-empty) sequence of c_1 and c_2. By this rule, $c_1 \cdot c_1 \cdot c_2 \cdot a$ rewrites to $c_1 \cdot c_1 \cdot c_2 \cdot b$, while $a \cdot a$ does not rewrite to any word. To illustrate suffix rewriting rules, consider $c : b \mapsto a^*$. With this rule, $c \cdot b \cdot c$ has no rewriting, while $c \cdot b \cdot c \cdot b$ rewrites to all words of the form $c \cdot b \cdot c \cdot a \cdots a$.

Simulating KV-rules with *ERS*. We now build an *ERS* which simulates the forward-chaining application of KV rules. This is possible because KV-rules actually reason on the paths of the input tree, and therefore we can see a path $root(r).k_1 \dots k_n.v$ of the saturated tree as a word $k_1 \cdot k_2 \cdots k_n$ obtained by applying the *ERS* rules. We have however to be careful that KV-rules of type (B) and (C) do not interact, because the former rules cannot be applied on a branch generated by the latter rules. We do so by introducing two novel symbols "\$" and "♯" to be employed for words within our *ERS*. Any path $root(r).K.v$ in the tree will be represented by the word $K \cdot \$$ if v is (a leaf) labeled by a constant, and by the word $K \cdot \sharp$ otherwise (that is, v is an internal node or a leaf labeled with a null possibly produced by a type (C) rule). When simulating type (B) KV-rules with *ERS* rules, we will make these apply only to suffix rules ending with \$.

A technical aspect of our construction is that to model the structure of multi-trees, we consider that letters in an *ERS* are *sets* of keys. We will consider a new alphabet $\mathcal{P} = 2^{\text{KEYS}} \setminus \emptyset$. From now on, for clarity we denote by $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathcal{P}$ the (complex) letters belonging to this set and by $\mathbf{u}, \mathbf{v}, \mathbf{w} \in \mathcal{P}^*$ the (complex) words made with these letters. For instance, let $\mathbf{a} = \{a_1, a_2\}$, $\mathbf{b} = \{b\}$, and $\mathbf{c} = \{c_1, c_2\}$ then $\mathbf{u} = \mathbf{a} \cdot \mathbf{b} \cdot \mathbf{c} = \{a_1, a_2\} \cdot \{b\} \cdot \{c_1, c_2\}$. The set of complex words that "enable" a word $w = a_1 \cdots a_n$, is defined as $\mathit{En}(w) = \{\mathbf{a}_1 \cdots \mathbf{a}_n \in \mathcal{P}^* | a_i \in \mathbf{a}_i\}$. For example, we have that $\mathit{En}(a \cdot b) = \{(\mathbf{a} \cup \{a\}) \cdot (\mathbf{b} \cup \{b\}) \mid \mathbf{a}, \mathbf{b} \in \mathcal{P}\}$ (note that we may have $\mathbf{a} = \{a\}$ or $\mathbf{b} = \{b\}$). Note that $\mathcal{P}^*$ and $\mathit{En}(w)$ are regular languages over $\mathcal{P}$, which we will employ within *ERS* rules.

Given a word over KEYS^* it will be handful to represent its corresponding (complex) element over $\mathcal{P}^*$. We define the function $\mathit{Sg}()$ (standing for "singleton") which given a (non-empty) word of the form $w = a_1 \cdots a_n$ yields its complex version with sets of letters made of a single element $\mathit{Sg}(w) = \{a_1\} \cdots \{a_n\}$. Finally, we denote by $\mathcal{P}^* \cdot (\sharp + \$)$ the set of words of $\mathcal{P}^*$ extended with the ♯ or \$ symbol. Here "+" denotes the union operator for regular languages.

A set of KV-rules Σ is associated with an *ERS* $\mathcal{R}_\Sigma = \Psi(\Sigma)$ containing all possible rules obtained by instantiating the following rules.

(A) inclusion between keys: a rule $K : k_1 \to k_2$ is associated with the relabeling (meta-)rule

$$(\mathcal{P}^* \cdot \mathit{En}(K)) : \mathbf{a} \cup \{k_1\} \mapsto \mathbf{a} \cup \{k_1, k_2\} \qquad (\forall \mathbf{a} \in \mathcal{P})$$

(B) inclusion between valued paths: a rule $K{:}K_1.val \to K_2$ is associated with the suffix "valued" rule

$$(\mathcal{P}^* \cdot \mathit{En}(K)) : (\mathit{En}(K_1) \cdot \$) \mapsto \mathit{Sg}(K_2) \cdot \$$$

(C) mandatory path: a rule $K : K_1 \to \exists K_2$ is associated with the suffix rule

$$(\mathcal{P}^* \cdot \mathit{En}(K)) : (\mathit{En}(K_1) \cdot \mathcal{P}^* \cdot (\sharp + \$)) \mapsto \mathit{Sg}(K_2) \cdot \sharp$$

To illustrate the construction, we give the *ERS* associated with the example used in the previous section.

$$\Psi(\sigma_1) = \begin{cases} \mathcal{P}^* : \{phone\} \mapsto \{phone, contact\} \\ \mathcal{P}^* : \{prof, phone\} \mapsto \{prof, phone, contact\} \\ \dots \end{cases}$$

$$\Psi(\sigma_2) = \mathcal{P}^* : \mathit{En}(course) \cdot \$ \mapsto \{prof\}.\{teaching\}.\$$$

$$\Psi(\sigma_3) = \mathcal{P}^* \cdot \mathit{En}(dept) : \mathit{En}(prof \cdot boss) \cdot \$ \mapsto \{director\} \cdot \{name\} \cdot \$$$

$$\Psi(\sigma_4) = \begin{cases} \mathcal{P}^* \cdot \mathit{En}(dept) : \{dir.\} \mapsto \{dir., faculty\} \\ \mathcal{P}^* \cdot \mathit{En}(dept) : \{prof, dir.\} \mapsto \{prof, dir., faculty\} \\ \dots \end{cases}$$

$$\Psi(\sigma_5) = \mathcal{P}^* \cdot \mathit{En}(dept \cdot faculty) : \mathcal{P}^* \cdot (\sharp + \$) \mapsto \{phone\}.\sharp$$

We can easily see that *ERS* rules mimic KV-rules. Consider for example the word $\{dept\} \cdot \{prof\} \cdot \{boss\} \cdot \$$ which corresponds to a path in the input tree of Figure 1 ending on the value "*Alice*". This rewrites to $\{dept\} \cdot \{director\} \cdot \{name\} \cdot \$$ by $\Psi(\sigma_3)$ and then to $\{dept\} \cdot \{director, faculty\} \cdot \{name\} \cdot \$$ by $\Psi(\sigma_4)$. Then, $\Psi(\sigma_5)$ gives $\{dept\} \cdot \{director, faculty\} \cdot \{phone\} \cdot \sharp$ and $\Psi(\sigma_1)$ gives $\{dept\} \cdot \{director, faculty\} \cdot \{phone, contact\} \cdot \sharp$. All of these words exactly correspond to some multi-paths inferred in Figure 1. Note also that the initial word representing a valued path ends with \$ while the final word ends with the ♯ symbol. This is correct, since only *check* queries have some answers on this multi–path of the saturated tree in Figure 1.

The following proposition formally states the correspondence between KV-rules and *ERS*.

Proposition 2. *Let Σ be KV-rule set, $\mathcal{R}_\Sigma$ its corresponding ERS, and $w \in \text{KEYS}^*$ a word. Let L_c be the set of words u such that $Sg(u) \cdot \sharp$ or $Sg(u) \cdot \$$ is in $Anc_{\mathcal{R}}(\mathit{En}(w) \cdot \mathcal{P}^* \cdot (\$ + \sharp))$. Then, for each tree r, $check(w)(r, \Sigma) = \cup_{u \in L_c} check(u)(r)$. Let L_g be the set of words u such that $Sg(u) \cdot \$$ is in $Anc_{\mathcal{R}}(\mathit{En}(w) \cdot \$)$. Then, $get(w)(r, \Sigma) = \cup_{u \in L_g} get(u)(r)$.*

Proof sketch. First note that a word in $\mathcal{P}^* \cdot (\sharp + \$)$ corresponds to a (multi-)path in a (multi-)tree. We take all maximal paths in the tree, together with their proper prefixes, and we associate them with a word ending by ♯. Moreover, for all maximal paths in the tree ending on a leaf node labeled with a constant, we add a word ending by \$. By transforming w in $\mathit{En}(w)$, we generate all multi-paths enabling w. We can check that each rewriting step is correct, in the sense that if a word w corresponds to a multi-path in $Sat(r, \Sigma)$ then its *ERS* rewritings by $\mathcal{R}$ do as well. Conversely, applying one saturation step can be simulated by rewriting the word corresponding to a multi-path. □

Computing the language of ancestors. We now turn to the problem of effectively computing the language of ancestors. In the following, we suppose that the reader is familiar with finite automata (NFAs). Let us first recall that the size of the alphabet $\mathcal{P}$ is exponential w.r.t. the size of KEYS. An NFA $\mathcal{A}$ is represented by a 5-tuple, $(\mathcal{P}, \mathrm{Q}, \Delta, \mathrm{I}, \mathrm{F})$, with $\mathcal{P}$ the alphabet, Q the set of states, Δ the set of transitions, and I and F the set of initial and final states, respectively. We denote ϵ-transitions with $(\mathrm{q}, \epsilon, \mathrm{q}')$. We omit $\mathcal{P}$ when it is clear from the context and denote by $\mathcal{L}(\mathcal{A})$ (resp. $\mathcal{L}_{\mathcal{A}}(\mathrm{q})$) the language recognized by $\mathcal{A}$ (resp. by $\mathcal{A}$ with q as initial state).

The automaton defining L_c (as defined in Proposition 2) can be computed in polynomial time in the size of the NFAs defining $Anc_{\mathcal{R}}(En(w) \cdot \mathcal{P}^* \cdot (\$ + \sharp))$ by considering only transitions labeled with letters of the form $Sg(a) \in \mathcal{P}$, making final all the states reaching $\sharp$ or \$. Finally, to get back to the alphabet KEYS, each letter $Sg(a)$ is transformed into a. Similarly, the automaton defining L_g can be computed from $Anc_{\mathcal{R}}(En(w) \cdot \$)$, with this time the final states reaching \$. The next proposition presents how to compute the two languages.

Proposition 3. *Let Σ be KV-rule set, $\mathcal{R}_\Sigma$ its corresponding ERS, and $w \in \text{KEYS}^*$ a word. Then there exist NFAs defining $Anc_{\mathcal{R}}(En(w) \cdot \mathcal{P}^* \cdot (\$ + \sharp))$ and $Anc_{\mathcal{R}}(En(w) \cdot \$)$ whose number of states is in $O(2^{|\text{KEYS}|*(M+1)} * (|w| + |\Sigma|))$, where M is the maximal length of contexts in Σ, $|\Sigma|$ is the size of Σ, and $|w|$ the size of w.*

Proof sketch. The construction consists in four steps:

First, starting from the word $w = w_1 \cdots w_n$ in the input query, we construct an automaton $\mathcal{A}_w = (\mathrm{Q}_w, \Delta_w, \mathrm{s_I}, \mathrm{F}_w)$ recognizing $(En(w) \cdot \mathcal{P}^* \cdot (\$ + \sharp))$ (or similarly $(En(w) \cdot \$)$). The construction is standard: the set of states is $\{i \mid 0 \leq i \leq |w| + 1\}$, $\Delta_w = \{(i, \mathbf{a}, i+1) \mid i < |w|, w_{i+1} \in \mathbf{a}\} \cup \{(|w|, \mathbf{a}, |w|) \mid \mathbf{a} \in \mathcal{P}\} \cup \{(|w|+1, \$, |w|+1), (|w|+1, \sharp, |w|+1)\}$; 0 (resp. $|w| + 1$) is the initial (resp. final) state.

Secondly, we define the notion of a *context automaton* for a set of KV-rules Σ: it is an NFA $\mathcal{A}_\mathrm{c} = (\mathrm{Q_c}, \Delta_\mathrm{c}, \mathrm{s_c}, \mathrm{Q_c})$ able to check $(\mathcal{P}^* \cdot En(K))$ for each key sequence K defining a context. So, for each K there exists a set of states $\mathrm{F}_K \subseteq \mathrm{Q_c}$ such that $\mathcal{L}(\mathrm{Q_c}, \Delta_\mathrm{c}, \mathrm{s_c}, \mathrm{F}_K)$ recognizes $(\mathcal{P}^* \cdot En(K))$. The set of states $\mathrm{Q_c}$ corresponds to the set of words over $\mathcal{P}$ whose length is at most M and we write $\mathrm{q}_\mathbf{u}$ for the state associated with the word $\mathbf{u}$. So, the size of $\mathrm{Q_c}$ is bounded by $2^{|\text{KEYS}|*(M+1)}$. We use $suffix_M(\mathbf{u})$ to denote the suffix of the word $\mathbf{u}$ whose length is M, and $\mathbf{u}$ itself when $|\mathbf{u}| \leq M$. Then, we write $\mathrm{q}_\mathbf{u}$ for the state associated with the word $\mathbf{u}$. Then, $\Delta_\mathrm{c} = \{(\mathrm{q}_\mathbf{u}, \mathbf{a}, \mathrm{q}_{suffix_M(\mathbf{u}\cdot\mathbf{a})}) \mid \mathrm{q}_\mathbf{u} \in \mathrm{Q_c}, \mathbf{a} \in \mathcal{P}\}$. The initial state s_c corresponds then to q_ϵ. F_K is the set of states $\mathrm{q}_\mathbf{u}$ such that $\mathbf{u} \in \mathcal{P}^* \cdot En(K)$.

Thirdly, in the same way, for each suffix rule $(C_i : L_i \mapsto R_i)$ we build an NFA $\mathcal{A}_{L_i}$ recognizing L_i with $O(|L_i|)$ states.

The last step is the most technical one, and requires extended automata completion techniques for constructing in polynomial time a new automaton $\mathcal{B}$ which can recognize $Anc_{\mathcal{R}}(En(w) \cdot \mathcal{P}^* \cdot (\$ + \sharp))$ and $Anc_{\mathcal{R}}(En(w) \cdot \$)$.

We first initialize $\mathcal{B}$ as the product of $\mathcal{A}_\mathrm{c}$ with the union of $\mathcal{A}_w$ together with all of the $\mathcal{A}_{L_j}$:

$$\mathcal{B} = (\ (\mathrm{Q}_w \cup (\cup_j \mathrm{Q}_{L_j})) \times Q_\mathrm{c}\ ,\ (\mathrm{s}, \mathrm{s_c})\ ,\ \Delta\ ,\ (\mathrm{F}_w \cup (\cup_j \mathrm{F}_{L_j})) \times Q_\mathrm{c}\ )$$

We initialize the automaton with the following transitions $\Delta = \{((\mathrm{q}, \mathrm{q_c}), \mathbf{a}, (\mathrm{q}', \mathrm{q}_\mathrm{c}')) | (\mathrm{q}, \mathbf{a}, \mathrm{q}') \in \Delta_w \cup (\cup_j \Delta_j), (\mathrm{q_c}, \mathbf{a}, \mathrm{q}_\mathrm{c}') \in \Delta_\mathrm{c}\}$ and complete it as follows:

1. for all relabeling rules $C : \mathbf{a} \mapsto \mathbf{b}$ in $\mathcal{R}_\Sigma$, $\mathrm{q_c}$ in F_C, and $((\mathrm{q}, \mathrm{q_c}), \mathbf{b}, (\mathrm{q}', \mathrm{q}_\mathrm{c}'))$ in Δ, we add $((\mathrm{q}, \mathrm{q_c}), \mathbf{a}, (\mathrm{q}', \mathrm{q}_\mathrm{c}'))$;
2. for all suffix rules $C_j : L_j \mapsto R_j$ in $\mathcal{R}_\Sigma$, $\mathrm{q_c}$ in F_{C_j}, add $((\mathrm{q}, \mathrm{q_c}), \epsilon, (\mathrm{s}_j, \mathrm{q_c}))$ when $\mathcal{L}_\mathcal{B}((\mathrm{q}, \mathrm{q_c})) \cap R_j$ is non empty.

As the number of states stays the same, the construction is polynomial in $\mathcal{A}$ and $\mathcal{B}$. □

When there is no relabeling, the construction is much simpler: the alphabet $\mathcal{P}$ is not needed and the number of states is in $O(|\Sigma|) \times (|w| + |\Sigma|))$. Similarly, when there are relabeling rules but no context, a context automaton is no more needed and the number of states is in $O(|w| + |\Sigma|)$.

4 Complexity of OMQA with KV-Rules

In the preceding section, we showed how to construct, using word rewriting techniques, query reformulations in the form of pointed RPQs. By analyzing the construction, we obtain the following complexity upper bounds:

Theorem 1. *Query answering on KV-stores is:*

- *in NLSpace for data complexity;*
- *in PSpace for combined complexity;*
- *in NLSpace for combined complexity without contexts.*

Proof sketch. Membership in NLSpace for data complexity follows directly from the (data-independent) reformulation into RPQs (which can be evaluated in NLSpace).

For the PSpace upper bound, we cannot explicitly construct the automata provided by the proof of Prop. 3, as they may be exponentially large. However, the states of the NFA can be stored in polynomial space, and we can test on the fly whether a given transition belongs to the automaton. The idea is then to simulate, using a non-deterministic PSpace Turing machine, a run of the NFA, keeping only the current position and state in memory. The only real difficulty is handling epsilon transitions, created by a suffix rule (Point 2 in the construction of the automaton $\mathcal{B}$). Indeed, we need to verify whether $\mathcal{L}((q, q_c)) \cap R_j$ is non-empty. Such non-emptiness checks can be carried out in PSpace, as emptiness testing is in NLSpace in the size of the automaton (which here is of exponential size). The NLSpace upper bound for combined complexity in the context-free case proceeds similarly, except that now we are working with a polysize automaton. □

We have matching lower bounds for combined complexity.

Theorem 2. *Query answering on KV-stores is PSPACE-hard for combined complexity, and NLSpace-hard for combined complexity when restricted to context-free rules.*

Proof sketch. NLSpace hardness is a simple reduction from directed reachability, so we concentrate on the PSpace hardness result. The proof is by reduction from the word problem for polynomially space-bounded deterministic Turing machines. Let $M = (\mathrm{Q}, \mathrm{A}, \delta, \mathrm{q}_i, \mathrm{q}_f)$ be such a Turing machine,

where Q is the set of states, A is the alphabet, δ is the transition function, q_i is the initial state and q_f the final one. Given M and an input word w, the problem is whether M accepts w. Let n be the bound on the number of tape cells used by M on input w. Configurations can be captured by words of length n over the alphabet $\Theta = \mathrm{A} \cup (\mathrm{A} \times \mathrm{Q})$, where $a \in \mathrm{A}$ signifies that a tape cell contains a, and (a, q) signifies that the cell contains a, is under the head, and the current state is q. Using the keys (i, β) where i is in $\{1, \cdots, n\}$ and $\beta \in \Theta$, we can straightforwardly encode configurations as sequences of keys $(1, \sigma_1) \dots (n, \sigma_n)$.

Our initial KV-store is a record whose only path encodes the initial configuration, i.e., it is of the form $r_0 K v$, with $K = (1, (w_1, \mathrm{q}_i)).(2, w_2) \dots (k, w_k).(k{+}1, b) \dots (n, b)$ where $w = w_1 \dots w_k$ and b denotes the blank symbol.

We then define KV-rules that allow us to build the full sequence of configurations of the run on w. First, for every key (i, a), we include the type (A) rule $(i, a) \rightarrow C$, where C is a new key, denoting 'any cell'. We next include rules that, from the previous configuration, construct the current configuration cell by cell, by 'appending' it at the end of the previous configuration. Note that the label of the *i-th* cell in the current configuration only depends on the three cells at positions $(i-1)$, i and $(i+1)$ in the previous configuration (with the first and last cells depending on only two cells). Therefore, we add the following contextual rules for each $i \in \{2, \dots, n-1\}$:
$(i-1, \alpha).(i, \beta).(i+1, \gamma).C^{n-3} : (i-1, \xi) \rightarrow$
$\exists (i-1, \xi).(i, Trans(\alpha, \beta, \gamma))$
where $Trans(\alpha, \beta, \gamma)$ returns the new label for cell i assuming that cells $i-1, i, i+1$ are respectively labelled α, β, γ in the previous configuration. Intuitively, the context goes from cell $(i-1)$ in the previous configuration to the cell that precedes cell $(i-1)$ in the current configuration, and a new path that encodes cell $i-1$ followed by cell i in the current configuration is created. Similar rules are introduced to handle the first and last tape cells.

Finally, we use rules of the form $(i, (\alpha, \mathrm{q}_f)) \rightarrow \exists Ok$ together with $C.Ok \rightarrow \exists Ok$ to propagate upwards the fact that we have reached an accepting configuration. It is easily verified that M accepts w iff $check(Ok)$ returns true. □

The question of the lower bound for data complexity remains open, as the specific shape of the data may play a role. Indeed, the following results show that the complexity is higher over arbitrary data instances than over KV-stores (assuming NLSpace $\neq$ PTime):

Proposition 4. *Query answering in the logical fragment associated with KV-rules (resp. context-free KV-rules) and check/get queries is PTime-hard for data complexity (resp. combined complexity) over arbitrary datasets.*

5 Discussion and Related Work

Evaluating ontology-mediated queries over KV-stores. We have shown that our query answering task can be recast as (1) computing a regular path expression corresponding to the ancestors of a word encoding the query in an extended rewriting suffix system associated with the KV-rules (2) evaluating this expression on the KV-store. However, the evaluation of regular expressions is not directly supported by KV-stores native languages. Nevertheless, as records have bounded depth d, one can still generate all check/get queries whose length is bounded by d, thereby making query answering complete. In practice, however, this may be inefficient when the regular expression contains a Kleene-$*$ or large disjunctions, as many queries could be generated. This is a known issue for semi-structured data-management systems, which is tackled by relying on a concise structural summary of the tree, such as data-guides [Goldman and Widom, 1997]. In our case, by relying on a structure that simply lists all maximal paths in the tree, one can select those belonging to the language of the regular expression, which suffices to ensure complete query answering. Note that the list of maximal paths in a record can be computed off-line in a single traversal of the record. Moreover, path-membership for regular expressions can be tested in linear time.

Related work. Two different approaches to OMQA over KV-stores have been recently proposed by [Mugnier *et al.*, 2016] and [Botoeva *et al.*, 2016]. The approach of [Mugnier *et al.*, 2016] is similar to ours in the sense that they propose a rule language, with rules of the form $K_1 \rightarrow K_2$, that are directly applicable to KV-stores. However, the expressivity of this language makes OMQA undecidable in general, hence restrictions are imposed: either the rules have a body restricted to a single key, or they have a head restricted to a single key, both kinds of rules being incompatible. Moreover, there is no correspondence with first-order-logic, hence the semantics remains operational. The work of [Botoeva *et al.*, 2016] extends the data integration-oriented approach known as Ontology-Based Data Access to KV-stores (and specially MongoDB). The ontology and the queries are expressed at a 'conceptual level' (in OWL 2 and SPARQL) and JSON-to-RDF mappings allow one to define a *virtual* RDF view of JSON data. Query reformulation is performed at the conceptual level, then the rewritten query is translated into a KV-query via the mappings.

In the domain of semi-structured databases, several constraint languages have been proposed. In particular, the works of [Abiteboul and Vianu, 1999] and [Andre *et al.*, 2007] study the implication problem for inclusion constraints expressed by regular-path expressions. These capture as a special case the context-free rules of type (B) studied here. In [Buneman *et al.*, 2000] path inclusions with contexts and inverses are studied. Finally, [Calvanese *et al.*, 2016] studied the extension of Description Logics with path-constraints.

As for future work, we plan to extend the rule language to represent concepts associated with nodes and investigate query reformulation with richer KV-queries, such as tree queries as provided for instance in MongoDB.

Acknowledgements. We thank the reviewers for their insightful reviews. This work was supported by ANR projects ASPIQ (ANR-12-BS02-0003), DeLTA (ANR-16-CE40-0007) and PAGODA (ANR-12-JS02-0007), CPER Nord-Pas de Calais/FEDER DATA 2015-2020, and STIC AMSUD project FoG.

References

[Abiteboul and Vianu, 1999] Serge Abiteboul and Victor Vianu. Regular path queries with constraints. *J. Comput. Syst. Sci.*, 58(3):428–452, June 1999.

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

[Andre *et al.*, 2007] Yves Andre, Anne-Cécile Caron, Denis Debarbieux, Yves Roos, and Sophie Tison. Path constraints in semistructured data. *Theor. Comput. Sci.*, 385(1-3):11–33, 2007.

[Baader *et al.*, 2008] Franz Baader, Carsten Lutz, and Sebastian Brandt. Pushing the el envelope further. In Kendall Clark and Peter F. Patel-Schneider, editors, *OWLED (Spring)*, volume 496 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[Baget *et al.*, 2011] J.-F. Baget, M. Leclère, M.-L. Mugnier, and E. Salvat. On Rules with Existential Variables: Walking the Decidability Line. *Artificial Intelligence*, 175(9-10):1620–1654, 2011.

[Botoeva *et al.*, 2016] Elena Botoeva, Diego Calvanese, Benjamin Cogrel, Martin Rezk, and Guohui Xiao. 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.*, 2016.

[Büchi, 1962] J. Richard Büchi. Regular canonical systems. *Archiv für mathematische Logik und Grundlagenforschung*, 1962.

[Buneman *et al.*, 2000] Peter Buneman, Wenfei Fan, and Scott Weinstein. Path constraints in semistructured databases. *Journal of Computer and System Sciences*, 61(2):146–193, 2000.

[Calì *et al.*, 2012] Andrea Calì, Georg Gottlob, and Andreas Pieris. Towards more expressive ontology languages: The query answering problem. *Artificial Intelligence*, 193:87–128, 2012.

[Calì *et al.*, 2013] Andrea Calì, Georg Gottlob, and Michael Kifer. Taming the infinite chase: Query answering under expressive relational constraints. *J. Artif. Intell. Res. (JAIR)*, 48:115–174, 2013.

[Calvanese *et al.*, 2007] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable Reasoning and Efficient Query Answering in Description Logics: The DL-Lite Family. *J. Autom. Reasoning*, 39(3):385–429, 2007.

[Calvanese *et al.*, 2016] Diego Calvanese, Magdalena Ortiz, and Mantas Šimkus. Verification of evolving graph-structured data under expressive path constraints. In *19th International Conference on Database Theory*, 2016.

[Cattell, 2010] Rick Cattell. Scalable SQL and nosql data stores. *SIGMOD Record*, 39(4):12–27, 2010.

[Caucal, 2000] Didier Caucal. On word rewriting systems having a rational derivation. In Jerzy Tiuryn, editor, *FOSSACS 2000*, volume 1784 of *Lecture Notes in Computer Science*, pages 48–62. Springer, 2000.

[Chasseur *et al.*, 2013] Craig Chasseur, Yinan Li, and Jignesh M. Patel. Enabling json document stores in relational systems. In Angela Bonifati and Cong Yu, editors, *WebDB*, pages 1–6, 2013.

[Eiter *et al.*, 2012] T. Eiter, M. Ortiz, M. Simkus, T.-K. Tran, and G. Xiao. Query Rewriting for Horn-SHIQ Plus Rules. In *AAAI.*, 2012.

[Goldman and Widom, 1997] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 436–445, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[Mugnier *et al.*, 2016] Marie-Laure Mugnier, Marie-Christine Rousset, and Federico Ulliana. Ontology-mediated queries for NOSQL databases. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, pages 1051–1057, 2016.

[Poggi *et al.*, 2008] Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Linking data to ontologies. *J. Data Semantics*, 10:133–173, 2008.