Querying Key-Value Stores Under Simple Semantic Constraints : Rewriting and Parallelization

Olivier Rodriguez, Corentin Colomier, Cecilie Rivière, Reza Akbarinia, Federico Ulliana

Querying Key-Value Stores Under Simple Semantic Constraints : Rewriting and Parallelization

Olivier Rodriguez
Université de Montpellier
olivier.rodriguez@etu.umontpellier.fr

Corentin Colomier
Université de Montpellier
corentin.colomier@etu.umontpellier.fr

Cecilie Rivière
Université de Montpellier
cecilie.riviere@etu.umontpellier.fr

Reza Akbarinia
INRIA Sophia Antipolis
reza.akbarinia@inria.fr

Federico Ulliana
Université de Montpellier
ulliana@lirmm.fr

ABSTRACT

We demonstrate a system for accessing Key-Value stores in the presence of semantic constraints, as considered in the setting of ontology-mediated query answering. The constraints we study are expressed in a native rule language for JSON records and their purpose is to establish a high level view over a collection of legacy Key-Value stores. To ensure correct and complete data access, constraints are taken into account via query rewriting techniques tailored for MongoDB queries. The considered setting also enable the deployment of parallelization techniques for optimizing query rewriting thereby accelerating the whole data access task.

During the demonstration, attendees will be able first to query JSON datasets enriched with semantic constraints. Then, it will be possible to make a detailed analysis of query rewriting algorithm and compare the parallel version with the baseline centralized one.

1 INTRODUCTION

Ontology-mediated query answering (OMQA) is currently at the center of many investigations. This paradigm for data access is designed to query incomplete databases by taking into account semantic constraints which are usually provided by means of an ontology. OMQA has many applications in data integration, especially with Semantic Web technologies [11]. Indeed, constraints can be used to establish a unified view of a collection of heterogenous datasets thereby allowing the user to formulate high-level queries employing a richer vocabulary than that of the single sources.

The paradigm so far has been widely studied for relational and RDF databases. The ontological knowledge is often expressed using description logics [9, 3] which also underpin the OWL 2 QL and EL profiles, or suitable fragments of existential rules (also called Datalog$^{\pm}$) akin to Tuple Generating Dependencies (TGDs) [4, 8, 5]. Yet, much less attention has been devoted to the case where the data to be accessed are JSON records sitting in a NOSQL system such as Key-Value stores like MongoDB [2] or CouchDB [1].

To illustrate the OMQA approach to Key-Value stores [6, 7], consider the JSON record r in Figure 1 which describes a computer science university department with two of its professors, and the MongoDB queries Q_1 and Q_2. The query Q_1 selects all department records where there exists a professor with a contact, while the query Q_2 selects all computer science departments having a director. It can be easily seen that these two queries have empty answer when evaluated on r. The reason is that they employ a vocabulary which is disjoint from that of r. Indeed, Q_1 asks for the key contact while Q_2 asks for the key director that are both not employed in r.

Here is where semantic constraints can come into play. Indeed, although the key contact is not present in the source r, this can be seen as a *high-level* key generalizing phone and mail. This is captured by rules σ_1 and σ_2. Hence, by taking into account the semantic constraints while accessing data, r would satisfy the query Q_1. Similarly, σ_3 says that whenever a professor is present, then a director exists. Again, if this constraint is considered, then r would also satisfy Q_2. This shows how semantic constraints allow the user to express a high-level query on a collection of data sources and also reason with incomplete information at the record level. Of course, the structure of a record can be enriched by more expressive constraints [6, 10], but in this work we focus on simple constraints built on pairs of keys, as those yet illustrated.

Demo contribution

This demonstration showcases *i*) *a system for accessing Key-Value stores in the presence of semantic constraints based on query rewriting as well as ii*) *a parallelization technique for optimizing data access.* Query rewriting is an algorithmic approach to account for semantics whose goal is to incorporate the constraints *into the query*. This process yields a set (or union) of rewritings whose answers set over the input database is exactly the same as the initial query under constraints. This *virtual* approach to query answering has many advantages. First, it is well suited for accessing legacy databases even with read-only access rights. Second, it is resilient to updates, being the rewritings of a query independent from the data sources.

```
(r)     { dept :
            {  name : "CS",
               prof : [  { name : "Bob",   phone : "5-256" } ,
                         { name : "Charles",   mail : "charles@um.fr" } ]
            } }

(Q1)    find( { dept :  { prof : { contact : { $exists : true } } } } )

(Q2)    find( { dept :  {
                  $elemMatch : {  name : "CS",  director : { $exists : true } }
                  } } )

(σ1)  phone ⟶ contact   (σ2)  mail ⟶ contact   (σ3)  prof ⟶ ∃director
```

Figure 1: Record example

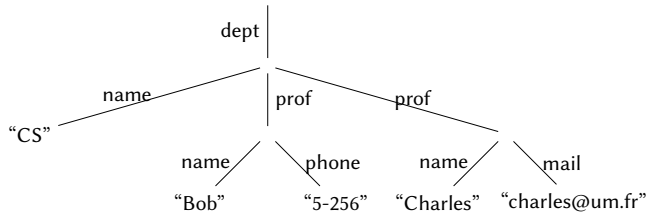


Figure 2: Tree representation of a record

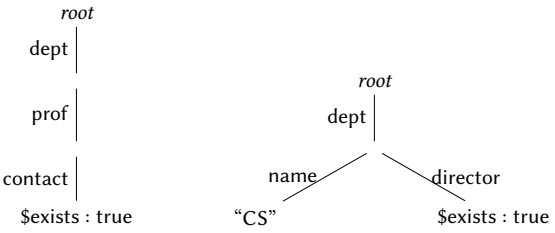


Figure 3: Tree representation of queries

The alternative approach to query rewriting is *materialization*. Intuitively speaking, for the constraints we consider of the form $k \longrightarrow k'$, materialization means to create *fresh copies* of all values coupled with a key k and then associating these copies with a new key k'. The *tree-shape* of JSON records may however hinder compact (graph) representations of data thereby making materialization suboptimal. This is exacerbated by the fact that data is voluminous.

In contrast, queries are usually small. From an OMQA perspective, it is thus interesting to explore query rewriting approaches that can take into account semantic constraints while accessing data without modifying the data sources.

In spite of the many advantages of virtual approaches we already mentioned, query rewriting can still suffer from combinatorial explosions, which often happen in practice. To mitigate this problem, our system features a parallel query rewriting method that can be used to distribute the computing of rewritings over several nodes as well as serve as basis for distributed query evaluation.

Our system currently enables access to MongoDB stores. However, the extension to other Key-Value stores systems is conceptually straightforward. The source of the code used for this demonstration is available at `github.com/zuri66/keyval-qrewriting`.

2 PRELIMINARIES

In this section, we present the preliminaries about the data, queries, rules and query rewriting.

2.1 Data

A JSON record, or *key-value record*, is a finite set of key-value pairs of the form $r = \{(k_1, e_1) \dots (k_n, e_n)\}$. A *value* is recursively defined as (i) a constant or a null value (ii) a sequence $e = [e_1 \dots e_n]$ where each e_i is a value and $n \geq 1$, or (iii) a record r, where each $k_i \in$ Keys and each e_i is a value, with $n \geq 1$ and $k_i \neq k_j$ whenever $i \neq j$. A KV-store I is a set of records.

We see a record r as a *rooted labeled tree*, denoted by $tree(r)$, in which edges are labeled by keys, leaves are labeled by constant or null values, and all other internal nodes are unlabeled. Of course, a key-value pair $(k, [e_1 \dots e_n])$ involving a sequence is represented by several edges labeled by k leading to the nodes that represent the elements of e [1]. Figure 2 provides a tree representation of the record given in the previous example.

[1]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.

2.2 Queries

We focus on the tree pattern queries that are supported by MongoDB [2]. These are of the form `find(`ϕ`)` where ϕ is an expression generated by the following grammar.

$$\begin{array}{rcl} \phi & ::= & \{\,\texttt{key} : cond\,\} \mid conj \\ cond & ::= & \phi \mid terminal \mid \{\,\texttt{\$elemMatch} : \{\,conj\,\}\,\} \\ conj & ::= & \phi \mid conj, conj \\ terminal & ::= & a \mid null \mid \{\,\texttt{\$exists} : \texttt{true}\,\} \end{array}$$

It is easy to see that the set of queries expressible by this grammar corresponds to a form of tree pattern queries without joins. [2] The grammar should be self explicative, we just comment on the main constructs. The `$elemMatch` clause is used to specify a *conjunction* of conditions that the subrecord has to match. [3] The condition `$exists : true` is used to verify the existence of any value associated with a key, while a to specify the constant that has to be matched like the string "CS" for the query Q_2 in Figure 1. Recall that a KV-store query is always evaluated starting from the *root* of the record. As for records, we see a query `find(`ϕ`)`, as a tree. This is denoted by $tree(\phi)$. Figure 3 provides a tree representation of the queries given in the previous example.

Following the MongoDB semantics, a `find` query evaluated on a record r yields an answer which is either the record r itself or the emptyset. Accordingly, we say that a record r answers (or satisfies) a query `find(`ϕ`)` if there exists a (tree-) homomorphism $h : tree(\phi) \rightarrow tree(r)$ mapping the root of $tree(\phi)$ to the root of $tree(r)$. With a little abuse of notation, we denote this by $r \models \phi$. [4] For a KV store I, we write $I \models \phi$ whenever $r \models \phi$ for some $r \in I$.

2.3 Rules

The constraints we are interested in are key inclusions, as introduced in [10, 6]. These are expressed as rules σ of the form

$$k \longrightarrow k' \quad (\forall\text{-rule}) \qquad k \longrightarrow \exists k' \quad (\exists\text{-rule})$$

and allow to define *hierarchies* of keys ($\forall$-rules) as well as the existence of mandatory keys ($\exists$-rule). The semantics of rules is now described. For the sake of formalization, we lift the setting of trees to that of *multi-trees*. That is, we consider trees where there can be more than one edge between a pair of nodes. To illustrate, $r = \{\mathsf{a} : \{\mathsf{c} : \text{"alice"}\}, \mathsf{b} : \{\mathsf{c} : \text{"alice"}\}\}$ can be seen either as the tree $\{(n_0, n_1, \mathsf{a}), (n_0, n_1', \mathsf{b}), (n_1, n_2, \mathsf{c}), (n_1', n_2', \mathsf{c})\}$ where the root is

[2]In passing, note also that ϕ is a JSON record.
[3]As a subtlety, MongoDB matches only records that belong to a sequence.
[4]This stands for $tree(r) \models tree(\phi)$.

n_0 and the leaves n_2, n_2' are labelled with "alice", or as the multi-tree $\{(n_0, n_1, \mathsf{a}), (n_0, n_1, \mathsf{b}), (n_1, n_2, \mathsf{c})\}$. This allows us to represent in a compact way the fact that a value (i.e., the terminal of the edge) is common to two keys. In the formal development below, we always consider rules of the form $k \longrightarrow [\exists]k'$ (the optional part being between brackets).

We say that a multi-tree t satisfies a $\forall$-rule $\sigma : k \longrightarrow k'$ if for each edge of the form (u, v, k) belonging to t there exists an edge (u, v, k') also belonging to t. Similarly, a multi-tree t satisfies an $\exists$-rule $\sigma : k \longrightarrow \exists k'$ if for each edge of the form (u, v, k) in t there exists an edge (u, v', k') in t, with possibly $v \neq v'$. [5] We denote this by $t \models \sigma$. For a set of constraints Σ we have that $t \models \Sigma$ if and only if $t \models \sigma$ for all $\sigma \in \Sigma$. Then, we say that t is a *model* of $r \wedge \Sigma$ when both $t \models tree(r)$ and $t \models \Sigma$ hold. Finally, a query ϕ is a *certain answer* of r and Σ, denoted by $r \wedge \Sigma \models \phi$, if whenever $t \models r \wedge \Sigma$ then $t \models \phi$. We are here interested at computing certain answers.

2.4 Query Rewriting

We now present an algorithmic procedure for query rewriting allowing us to take into account the semantics of constraints while accessing data. We say that a query ϕ can be rewritten by a $\forall$-rule σ (of the form $k \longrightarrow k'$), if there exists an edge $(u, v, k') \in tree(\phi)$. The rewriting process simply replaces the key k' with k. We say that a query ϕ can be rewritten by a $\exists$-rule σ (of the form $k \longrightarrow \exists k'$), if there exists an edge $(u, v, k') \in tree(\phi)$ such that i) v is a leaf and ii) v is not labelled with a constant or null (that is, an `$exists` leaf). The rewriting process replaces the edge (u, v, k') with the edge (u, v_0, k), with v_0 a fresh (leaf) node. In both cases, we say that the new obtained query ϕ' is a direct rewriting of ϕ with σ. For example the following queries Q_1' and Q_1'' are direct rewritings of Q_1 with σ_1 and σ_2 respectively, while Q_2' is a direct rewriting of Q_2 with σ_3.

(Q_1') `find( { dept : { prof : { phone : { $exists : true } } } } )`
(Q_1'') `find( { dept : { prof : { mail : { $exists : true } } } } )`
(Q_2')
```
find( { dept : {
    $elemMatch : { name : "CS", prof : { $exists : true } }
  } } )
```

It is easy to see that r in Figure 1 answers all of these rewritings. Note that $\exists$-rules apply only the leaves of a query that are *not* associated with a constant. To see why consider the query (Q_3) `find( { director : "charles" } )` where σ_3 does not apply. Indeed, the query `find( { professor : { $exists : true } } )` is not a valid rewriting, as it does not not imply an answer to Q_3.

Then, we define breadth-first query rewriting as $Rew_0(\phi, \Sigma) = \{\phi\}$ and $Rew_{i+1}(\phi, \Sigma) = Rew_i(\phi, \Sigma) \bigcup \{\phi'\}$ for all ϕ' which is a direct rewriting of $\phi'' \in Rew_i(\phi, \Sigma)$ with $\sigma \in \Sigma$. In contrast to the general case [6], this procedure always terminates, that is for all ϕ and Σ there exists an integer k such that $Rew_k(\phi, \Sigma) = Rew_{k+1}(\phi, \Sigma)$. Hence, we denote by $Rew(\phi, \Sigma)$ the set of all rewritings of a query. We say that r answers $Rew(\phi, \Sigma)$, denoted by $r \models Rew(\phi, \Sigma)$ when $r \models \phi$ for some $\phi \in Rew(\phi, \Sigma)$. Soundness and completeness of query rewriting is as follows: $r \wedge \Sigma \models \phi \iff r \models Rew(\phi, \Sigma)$.

[5] From a first order logic perspective, these rules correspond to universally quantified formulae of the form $\forall x, y(\mathsf{k}(x, y) \longrightarrow \mathsf{k}'(x, y))$ and existential quantified formula $\forall x, y(\mathsf{k}(x, y) \longrightarrow \exists z.\, \mathsf{k}'(x, z))$

3 PARALLELIZATION

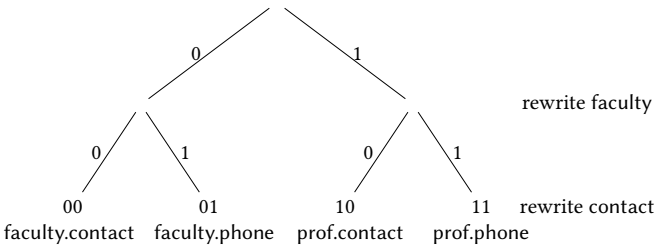
We now present a parallel method for computing $Rew(\phi, \Sigma)$. The idea is to exploit a set of independent computing units $u_1, \ldots, u_n$ being each a local thread or a machine of a computing cluster. Each unit u_i will compute a (sub)set of the rewritings of the input query $\Phi_i \subseteq Rew(\phi, \Sigma)$. It is important that every unit computes different rewritings, that is for all $u_i \neq u_j$ we have that $\Phi_i \cap \Phi_j = \emptyset$.

For the sake of efficiency, the rewriting load should be well balanced among the computing units. Formally, given a query ϕ, a set of rules Σ, and n computing units, our goal is to find a balanced partitioning of the set of rewritings $[\Phi_1, \ldots, \Phi_n]$ making each node u_i generating a set Φ_i such that $|\Phi_i| \approx \frac{|Rew(\phi, \Sigma)|}{n}$. After determining the partitions, each unit starts doing the local rewritings for its partition, and the global result is the union of the local results.

Encoding the space of rewritings. Our solution for load balanced rewriting proceeds in two steps. First, we define a encoding of the space of rewritings suitable for this task. Second, we partition the space over the computing units.

To encode $Rew(\phi, \Sigma)$ we build a decision tree $\mathcal{T}_{Rew(\phi,\Sigma)}$ such that: i) any path from the root to a leaf of the tree represents a (unique) rewriting of ϕ with Σ; and ii) all rewritings in $Rew(\phi, \Sigma)$ have a corresponding path in $\mathcal{T}_{Rew(\phi,\Sigma)}$. A balanced partitioning of the rewritings can be obtained by equally partitioning these "rewriting paths" over the n units. We outline the main technical ideas of our approach then generalize it to the whole framework.

Dealing with single rule applications. Consider the query $\phi = \{\text{faculty} : \{\text{contact} : \{\$\text{exists} : \text{true}\}\}\}$ and the rules $\sigma_1 : \text{phone} \rightarrow \text{contact}$ and $\sigma_2 : \text{prof} \rightarrow \text{faculty}$. In this case, $Rew(\phi, \{\sigma_1, \sigma_2\}) = \{\text{faculty.contact}, \text{faculty.prof}, \text{prof.contact}, \text{prof.phone}\}$. This set can be represented with a binary tree whose depth ℓ depends on the number of *rewritable edges* of the query (in this case 2) and where an edge from level i to level $i + 1$ has a binary label corresponding to the fact that the edge has been rewritten (1) or not (0) by the respective rule. The absolute path of every leaf of the tree corresponds to a rewriting. With this approach we have at most $2^{|\phi|}$ binary codes, one for each rewriting. Note that 0^ℓ denotes the original query.



Bounded size communication. Note that the encoding and partitioning of the rewritings space is done in one of the computing units u, which is chosen randomly. After doing the partitioning, u should communicate to each other unit its encoded partition. For

the sake of efficiency, it is important to have a bounded communication between u and each other unit.

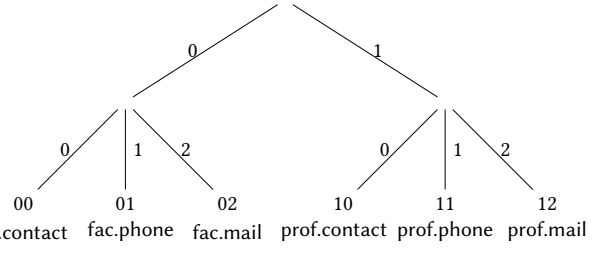
An important property of our encoding approach is that the obtained codes can be put in bijection with the integers $\{0, \ldots, k-1\}$. Therefore, the unit u can communicate to each unit u_i in a succinct way the *interval* of rewritings that it has to generate. In fact, u sends to u_i only two values $(N, N+\lambda)$ where N is a positive integer and $\lambda \approx k/n$. For instance, if $n = 1$ the interval $[0, 3]$ representing the set of (binary) codes $\{00, 01, 10, 11\}$ will be sent to u_1. This ensures a $O(1)$ bound the size for communicating the interval to a single computing unit.

Dealing with multiple rule applications. In the general case, the rule application is more complex than what we saw in the previous example, since not only two rules σ_1 and σ_2 can rewrite the same edge of the query, but also the application of σ_1 can enable that of σ_2. For these cases, the approach yet described must be properly extended. To do so, we look at *transitive dependencies* between the keys in Σ.

We say that k' is $(\exists)-$*derivable* from k, denoted by $k \leq_\exists k'$, if there exists a sequence of rules $\sigma_1, \ldots, \sigma_n$ of the form $\sigma_i = k_i \longrightarrow [\exists]k_i{}'$ such that $k'_i = k_{i+1}$ for all $1 \leq i < n$, with $k_1 = k$ and $k'_n = k'$. When all rules in the sequence are $(\forall)-$rules we say that k' is $(\forall)-$*derivable* from k, and we denote it by $k \leq_\forall k'$. It follows that $k \leq_\forall k'$ implies $k \leq_\exists k'$, but the converse does not hold. It is important to distinguish the two relations because, as we mentioned, $(\exists)-$rules must be applied only on the existential leaves of the query. Finally, note that we can compute the relations $\leq_\forall$ and $\leq_\exists$ in a preprocessing step with a single traversal of Σ.

Let us denote by the integers $\{1, \ldots, \rho\}$ the edges of the query ϕ that can be rewritten by a rule of Σ. Then, each edge j labelled with k', is rewritten with a key k such that *i*) $k \leq_\exists k'$ if the edge is incident to an existential leaf and *ii*) $k \leq_\forall k'$ otherwise. Importantly, we denote by b_j the number of ways that the edge j of the query can be rewritten. For instance, from the example of Figure 1 we have phone $\leq_\forall$ contact and mail $\leq_\forall$ contact and thus for an edge j of the query labelled with contact we have $b_j = 2$.

The set $Rew(\phi, \Sigma)$ can therefore be represented by an *unranked* decision tree where *i*) each level j corresponds to the rewritings of the edge j of the query, and *ii*) the degree of all of nodes at level j is exactly $(b_j + 1)$, that is, there exists an outgoing edge from the node for each possible rewriting of the edge (plus one denoting that the edge is not rewritten). As before, the leaves of the decision tree are labelled with their absolute path $c_1 c_2 \ldots c_\rho$ which describe a unique rewriting of ϕ. Note that $c_j \leq (b_j + 1)$ for all $1 \leq j \leq \rho$.



To preserve the bounded-size communication when moving from binary to unranked trees, we need to put again the set of k codes we obtained in bijection with the integers $\{0, \ldots, k-1\}$, in order to easily compute the partitioning of the rewritings over the n computing units. To do so, we see a code as a number in a multiple base $(b_\rho, \ldots, b_1)$ where as before each b_j denotes the number of elements of the base (corresponding to the number of possible rewritings of the edge j in the query). A code $(c_1 \ldots c_\rho)$ in the base $(b_\rho, \ldots, b_1)$ is equivalent to the integer p where

$$p = c_\rho * B_1 + c_{\rho-1} * B_2 + \ldots + c_1 * B_\rho$$

where $B_1 = 1$ and for all $i \geq 2$ it holds that $B_i = b_{i-1} * B_{i-1}$. In the above example, the codes $\{00, 01, 02, 10, 11, 12\}$ will correspond to the interval $[0, 5]$. Indeed, we consider each code in the base[6] $(b_2, b_1) = (2, 3)$ we have $B_1 = 1$ (by definition) and $B_2 = b_1 * B_1 = 3$. For example, the code $(c_1 c_2) = (12)$ stands for $c_2 * B_1 + c_1 * B2 = 5$. At this point, by knowing the base $(b_n, \ldots, b_1)$ it is possible to determine the number of rewritings $k = b_\rho * B_1 + \cdots + b_1 * B_\rho$, and then assign to each of the n units an interval of size k/n.

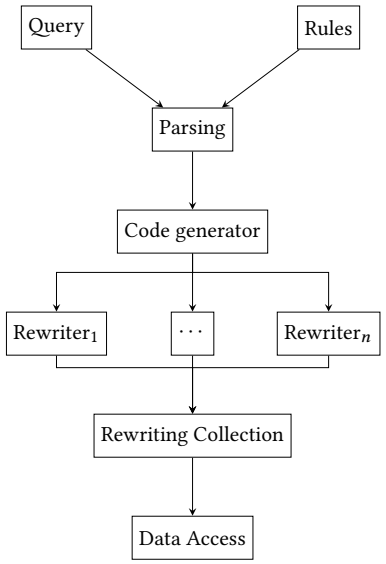


Figure 4: System architecture

4 SYSTEM ARCHITECTURE

In this section, we describe the global functioning of our system, whose architecture is depicted in Figure 4.

Our solution can be implemented in any shared nothing parallel framework. In our demo, we use different cores of a machine and parallelize our solution by executing threads in the cores. However, the threads can also be executed in the nodes of a distributed cluster, if such a cluster is available.

The first component of our system is a *Parser* of the queries and the rules. After that, the *Code Generator* is in charge of computing the encoding of the rewritings. The Code Generator associates to every edge of the query, which can be rewritten, all of the possible keys and their associated codes for the rewriting. This module is also in charge of properly taking into account the ($\exists$-rule) constraints which, as said before, must apply only to existential leaves. The set of codings is then partitioned into intervals. Each

[6] we have two possibilities for c_1 (i.e., 0,1) and tree for c_2 (i.e., 0,1,2)

task (in a unit) takes its interval, and generates the corresponding rewriting results. When all parallel rewriting tasks are completed, the results are combined in order to obtain the final rewriting set. Finally, the Data Access module is in charge of evaluating the whole query set on the available MongoDB instances.

5 DEMO OVERVIEW

The goal of our demonstration is *i*) to showcase the benefits of accessing data under semantic constraints as well as *ii*) to show the efficiency of the parallelization for the query rewriting system.

The demonstration scenario is constituted of two JSON datasets. The first, *University*, has been manually built and concerns university employees, faculty members, and students,. The second, *XMark* [12], is a standard benchmark for semi-structured data that we used to generate JSON records. Both datasets are associated with queries of different complexities ranging from path queries to tree-pattern queries. Similarly semantic constraints adapted to the structure of the datasets have been devised.

The demonstration will start by showing how thanks to semantic constraints we can bring novel and pertinent answer to users queries as well as formulate high-level queries over heterogenous records. Then, it will be possible to analyze the functioning of the query rewriting module of the system, by exploring the rewriting set generated for a given input query and constraint set. Finally, it will be shown how, for cases where the number of rewriting is huge, the rewriting taks can be parallellized across multiple threads so as to optimise the overall data access.

The demonstration will take place on a graphical interface allowing the attendees to choose predefined queries and rules, but also to write their own queries and constraints. Similarly, the number of rewriting threads can be configured so as to compare the effectiveness of the parallelized version of the algorithm with the baseline centralized one.

REFERENCES

[1] Couchdb.

[2] MongoDB.

[3] Baader, F., Lutz, C., and Brandt, S. Pushing the EL envelope further. In *OWLED (Spring)* (2008), K. Clark and P. F. Patel-Schneider, Eds., vol. 496 of *CEUR Workshop Proceedings*, CEUR-WS.org.

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

[5] Beeri, C., and Vardi, M. Y. A proof procedure for data dependencies. *J. ACM 31*, 4 (Sept. 1984), 718–741.

[6] Bienvenu, M., Bourhis, P., Mugnier, M., Tison, S., and Ulliana, F. Ontology-mediated query answering for key-value stores. In *International Joint Conference on Artificial Intelligence* (2017).

[7] Botoeva, E., Calvanese, D., Cogrel, B., Rezk, M., and Xiao, G. Obda beyond relational dbs: A study for mongodb. *birth 1926*, 08–27.

[8] Calì, A., Gottlob, G., and Pieris, A. Towards more expressive ontology languages: The query answering problem. *Artificial Intelligence 193* (2012), 87–128.

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

[10] Mugnier, M., Rousset, M., and Ulliana, F. Ontology-mediated queries for NOSQL databases. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.* (2016).

[11] Poggi, A., Lembo, D., Calvanese, D., De Giacomo, G., Lenzerini, M., and Rosati, R. Linking data to ontologies. *J. Data Semantics 10* (2008), 133–173.

[12] Schmidt, A., Waas, F., Kersten, M., Carey, M. J., Manolescu, I., and Busse, R. Xmark: A benchmark for xml data management. In *Proceedings of the 28th international conference on Very Large Data Bases* (2002), VLDB Endowment, pp. 974–985.