Querying Key-Value Stores under Single-Key Constraints: Rewriting and Parallelization

Olivier Rodriguez, Reza Akbarinia, Federico Ulliana

Querying Key-Value Stores under Single-Key Constraints: Rewriting and Parallelization

Olivier Rodriguez, Reza Akbarinia, and Federico Ulliana

INRIA & LIRMM, Univ. Montpellier, France
{firstname.lastname}@inria.fr

Abstract. We consider the problem of querying key-value stores in the presence of semantic constraints, expressed as rules on keys, whose purpose is to establish a high-level view over a collection of legacy databases. We focus on the rewriting-based approach for data access, which is the most suitable for the key-value store setting because of the limited expressivity of the data model employed by such systems. Our main contribution is a parallel technique for rewriting and evaluating tree-shaped queries under constraints which is able to speed up query answering. We implemented and evaluated our parallel technique. Results show significant performance gains compared to the baseline sequential approach.

1 Introduction

Semantic constraints are knowledge on the structure and on the domain of data which are used in contexts such as data integration and ontology mediated query answering to establish a unified view of a collection of a database. Constraints allow users to better exploit their data thanks to the possibility of formulating high-level queries, which use a vocabulary richer than that of the single sources. In the last decade, the use of constraints in the form of ontologies has been intensively studied in the knowledge representation domain [10,4,3]. A key factor in the rise of the paradigm has been the reuse of off-the-shelf data management systems as the underlying physical layer for querying data under constraints. This resulted in a successful use of the paradigm especially on top of relational and RDF systems [5]. However, the use of constraints to query NOSQL systems like key-value stores (e.g., MongoDB [1], CouchDB [2]) has just begun to be investigated [6,7,8]. Key-value stores are designed to support data-intensive tasks on collections of JSON records, this last one being a language which is becoming the new de facto standard for data exchange.

To illustrate the use of semantic constraints for querying key-value records, consider the records in Example 1 which describe university departments. Query Q_1 selects all department records having a professor with some contact details. Query Q_2 selects all computer science departments with a director. It can be easily seen that these two queries do not match any of the records. Indeed, Q_1 asks for the key contact which is not used in both r_1 and r_2, while Q_2 asks, on the one hand, for the key director, which is not used in r_1 and, on the other hand, for the value "CS" for the department name, which does not match that of r_2.

(r_1) { dept : {
name : "CS" ,
prof : { name : "Bob" ,
mail : "$bob@uni.com$" } } }

(r_2) { dept : {
name : "$Math$" ,
director : { name : "$Alice$" ,
phone : null } } }

(σ_1) phone → contact
(σ_2) mail → contact
(σ_3) director → prof
(σ_4) prof → ∃director

(Q_1) find({ dept : { prof : { contact : $exists } } })

(Q_2) find({ dept : { name : "CS", director : $exists } })

Example 1. Data, queries, and rules.

This is where semantic constraints come into play. Indeed, although the key contact is not used in the records, this can be seen as a *high-level* key generalizing both phone and mail, as captured by rules σ_1 and σ_2. Therefore, by taking into account these semantic constraints, r_1 satisfies the query Q_1. Moreover, since σ_3 says that the director of a department must be a professor, also r_2 satisfies Q_2. Finally, σ_4 says that whenever a professor is present, then a director exists. Again, with this rule in hand, r_1 would also satisfy Q_2. This example outlines how semantic constraints allow users to better exploit their data.

The two main algorithmic approaches usually considered to account for semantic constraints during data access are materialization and query rewriting. Intuitively speaking, for constraints of the form $\mathsf{k} \longrightarrow \mathsf{k}'$, materialization means creating a *fresh copy* of the value of k and then associating this copy to the key k'. It is important to notice that, being the JSON data model based on trees, materialization can result in exponential blowups of the data. Also, not only it is computationally expensive to repeatedly copy subrecords, but it is also hard to efficiently implement such mechanism on top of key-value stores whose primitives, despite handling bulk record insertions, are not oriented towards the update of a single record. This is exacerbated by the fact that data is voluminous. In contrast, queries are usually small. From this 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. The idea of query rewriting is to propagate constraints "into the query". This process yields a set (or a union) of rewritings whose answers over the input database is exactly the same as the initial query on the database where materialization would have been done. Being rewritings independent from the sources, this approach is well suited for accessing legacy databases, in particular with read-only access rights.

The query facilities of key-value stores systems include primarily a language for selecting records matching several conditions based on tree-shaped queries called find() queries [1,2]. The MongoDB store also includes an expressive language for aggregate queries which is equivalent to nested relational algebra [9]. In this work, we focus our attention on the evaluation of find() queries under single-key constraints built on pairs of keys, as those of Example 1. It is worth noting that NOSQL systems still lack the standardization of a common query language and therefore of a standard syntax and semantics for queries. Therefore, in the formal development presented in Section 2, we chose to abstract away from the conventions of existing systems and adopt a syntax for queries akin to that of key-value records and a natural semantics based on tree-homomorphisms.

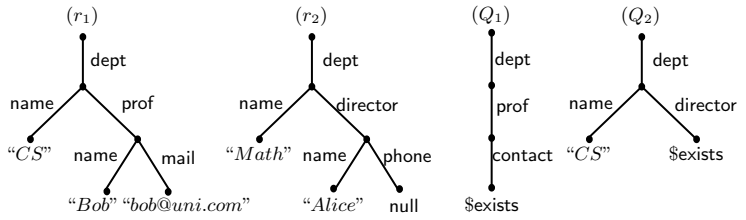

Fig. 2. Tree representation of records and queries of Example 1

In spite of the many advantages we already mentioned it is understood that, depending on the target language chosen for the rewritings, the query rewriting approach can suffer from combinatorial explosions - even for rule languages with a limited expressivity [4]. This happens as well for the key-value store languages. This means that the rewriting set of a query to generate may be large, which has a consequent impact on the slowing down of query answering.

To mitigate this problem, we present a novel technique for parallelizing both the generation and the evaluation of the rewriting set of a query serving as the basis for distributed query evaluation under constraints. Our solution is presented in Section 3, and relies on a schema for encoding the possible rewritings of a query on an integer interval. This allows us to generate equi-size partitions of rewritings, and thus to balance the load of the parallel working units that are in charge of generating and evaluating the queries. The experimental evaluation of our technique reported in Section 4 shows a significant reduction of query rewriting and execution time by means of parallelization.

2 Single-Key Constraints and Query Rewriting

This section is dedicated to the formalization of the setting we consider, and follows the lines of [8]. For concision, we will focus on a simplified JSON model. However, this is w.l.o.g, as the technique we present applies to the full language.

Data. A *key-value record* is a finite (non-empty) set of key-value pairs of the form $\mathsf{r} = \{(\mathsf{k}_1, \mathsf{v}_1), \ldots, (\mathsf{k}_n, \mathsf{v}_n)\}$ where all k_i are distinct keys and each v_i is a value. A *value* is defined as *i*) a constant or the null value, *ii*) a record r, or *iii*) a non-empty sequence $\mathsf{v} = [\mathsf{v}_1 \ldots \mathsf{v}_n]$ where each v_i is a constant, null, or a record. A value v in a key-value record can be associated with a *rooted labelled forest* T_v, where each tree of the forest has a root and nodes and edges can be labelled. If v is a constant or null then T_v is a single (root) node labelled by that value. If v is a sequence $\mathsf{v} = [\mathsf{v}_1 \ldots \mathsf{v}_n]$ then T_v is a forest of n rooted trees $T_1, \ldots, T_n$ where T_i is the tree associated to the value v_i. Finally, if v is a record, then T_v is as follows. Let, k be a key of v and T_k be the forest associated to the value of k. Then T_v contains *i*) all nodes and edges of each tree T_k, *ii*) a root node s, and *iii*) an edge from s to s_k labelled by k whenever s_k is the root of a tree in T_k. Clearly, T_v is a rooted tree whenever v is a record. In the remainder, we will see a key-value record as its associated tree, as illustrated in Figure 2. Note the order of the elements of a sequence is not represented in the associated trees. Also, we will assume a fixed way to associate a tree with a unique (representative) record.

Rules. We focus on semantic constraints we call *single-keys*, also studied in [6,8]. These are expressed as rules σ of the form

$$\mathsf{k} \longrightarrow \mathsf{k}' \quad \text{(key inclusion)} \qquad \mathsf{k} \longrightarrow \exists \mathsf{k}' \quad \text{(mandatory key)}$$

enabling the definition of hierarchies of keys as well as the existence of mandatory keys. The semantics of constraints is defined on the tree associated to a record. Next, we denote by (u, w, k) an edge from u to w labelled by k. We say that a tree T satisfies a constraint $\sigma : \mathsf{k} \longrightarrow \mathsf{k}'$ if for each of its edges of the form (u, w, k) there also exists an edge (u, z, k') and an isomorphism φ from the subtree rooted at w to that rooted at z such that $\varphi(w) = z$. Then, T satisfies $\sigma : \mathsf{k} \longrightarrow \exists \mathsf{k}'$ if for each of its edges of the form (u, w, k) there also exists an edge (u, z, k'), whatever the subtree rooted at z. Let Σ be a set of constraints. Then, we say that a tree T is a model of r and Σ when $i)$ T_r and T have the same root, $ii)$ T_r is a subtree of T, and $iii)$ T satisfies all constraints of Σ. For single-key constraints, it can be easily shown that every pair (r,Σ) admits a finite model.

Queries. We consider the problem of answering $\mathsf{find}()$ queries that are integrated in the facilities of popular key-value stores such as MongoDB and CouchDB [1,2]. These queries select all records satisfying some structural and value conditions, and can be seen as Boolean queries, in that query evaluation on a record yields an answer which is either the empty set or the record itself. A query is thus of the form $\mathsf{find}(\phi)$ where ϕ is a key-value record. Importantly, we assume that 1) queries do not use the null value and 2) the reserved constant $\$\mathsf{exists}$ is used to require the existence of any value associated with a key, as illustrated by Q_1 in Example 1. As for records, queries can be associated with labelled trees. Figure 2 illustrates the tree representation of the queries of Example 1.

Then, a query $\mathsf{find}(\phi)$ answers *true* on a record r if there exists a mapping h from the nodes of T_ϕ to that of T_r such that $i)$ the root of T_ϕ is mapped to the root of T_r, $ii)$ for every edge (u, w, k) of T_ϕ, $(h(u), h(w), \mathsf{k})$ is an edge of T_r, and $iii)$ every leaf node u of T_ϕ which is labelled by a constant different from $\$\mathsf{exists}$ is mapped to a node $h(u)$ with the same label. Finally, with constraints, a query $\mathsf{find}(\phi)$ answers true on r and Σ if it answers true on all models of r and Σ.

Query Rewriting Query rewriting is an algorithmic procedure for taking into account a set of semantic constraints Σ that starts from a query Q and produces a set of rewritings $Rew(Q, \Sigma)$ such that, for all records r, Q answers true on r and Σ if and only if there exists a query $Q' \in Rew(Q, \Sigma)$ that answers true on r. As for rules, we define the rewriting of a query $\mathsf{find}(\phi)$ on its associated tree T_ϕ. So, $\mathsf{find}(\phi)$ can be rewritten with $\sigma : \mathsf{k} \rightarrow \mathsf{k}'$, if T_ϕ contains an edge (u, w, k'). Similarly, $\mathsf{find}(\phi)$ can be rewritten with $\sigma : \mathsf{k} \rightarrow \exists \mathsf{k}'$, if T_ϕ has an edge (u, w, k') where w is a leaf node labelled by $\$\mathsf{exists}$. In both cases, the rewriting consists at replacing the edge (u, w, k') in T_ϕ with (u, w, k). Let $T_{\phi'}$ be the resulting tree, whose representative record is ϕ'. Then we say that $\mathsf{find}(\phi')$ is a direct rewriting of $\mathsf{find}(\phi)$ with σ. We denote by $Rew(Q, \Sigma)$ the set of Q' for which there exists a (possibly empty) sequence of direct rewritings from Q to Q' using the rules of Σ. This means that Q belongs to $Rew(Q, \Sigma)$. The size of $Rew(Q, \Sigma)$ is bounded by $|\Sigma|^{|Q|}$, where $|Q|$ is the number of edges of Q. The correctness of the rewriting algorithm can be shown by extending the proofs of [8].

rewriting	*array*	*integer*
(Q_1^1) find({ dept : { director : { contact : $exists } } })	$\langle 0,1,0\rangle$	**1**
(Q_1^2) find({ dept : { prof : { phone : $exists } } })	$\langle 0,0,1\rangle$	**2**
(Q_1^3) find({ dept : { director : { phone : $exists } } })	$\langle 0,1,1\rangle$	**3**
(Q_1^4) find({ dept : { prof : { mail : $exists } } })	$\langle 0,0,2\rangle$	**4**
(Q_1^5) find({ dept : { director : { mail : $exists } } })	$\langle 0,1,2\rangle$	**5**
(Q_2^1) find({ dept : { name : "CS" , prof : $exists } })	$\langle 0,0,1\rangle$	**1**

Fig. 3. Rewritings of the queries in Example 1 (left) and their encoding (right)

Figure 3 illustrates the rewritings of queries given in Example 1. Here, Q_1^1–Q_1^5 are rewritings of Q_1 with $\sigma_1, \sigma_2, \sigma_3$, while Q_2^1 is a direct rewriting of Q_2 with σ_4. It holds that Q_1^4 selects r_1, Q_1^3 selects r_2, and Q_2^1 selects r_1. Note that rules for mandatory keys apply only on the leaves of a query that are labelled with $\mathsf{exists}. To see why consider the query $\mathsf{find}(\{\mathsf{dept} : \{ \mathsf{director} :$ "*Alice*" $\}\})$. Here, if σ_4 is used, we get $\mathsf{find}(\{\mathsf{dept} : \{ \mathsf{prof} : \$\mathsf{exists} \}\})$ which is not a valid query rewriting.

3 Parallelization

We now present a parallel method that can be used to distribute both the generation and the evaluation of the rewriting set of a query to a set of independent computing units $u_1, \ldots, u_m$, each being a local thread or a machine of a cluster. Our approach relies on an *interval-encoding* of the rewritings. The general idea is to establish a bijection between $Rew(Q, \Sigma)$ and the integers in $[0, \ldots, N-1]$, where $N = |Rew(Q, \Sigma)|$. Then each of the m computing units is communicated an interval $[i, j]$ of size $\lambda \approx N/m$ corresponding to the subset of rewritings it has to generate. This will result in a parallel rewriting method that enjoys the following three properties.

1) *partitioning*: no rewriting is computed twice by two distinct units
2) *load balance*: the number of rewritings is equally distributed across all units
3) *bounded-communication*: units receive a constant size interval information

Encoding Queries. In contrast to the general case [6,8], when considering single-key constraints, one can exploit the fact that, the rewriting process we described in the previous section, yields queries that are structurally similar. This enables a compact representation of (the edges of) a query as fixed size arrays, which we now describe. Let $\mathsf{find}(\phi)$ be a query. By fixing a total order on the edges of T_ϕ, we can see the query as an array $\langle \mathsf{k}_1, \ldots, \mathsf{k}_n \rangle$, where k_i is the key labelling the i-th edge of T_ϕ. Thus, to reconstruct a rewriting from an array it just suffices to replace the i-th edge of Q with the i-th key of the array. Moreover, given that an edge can be rewritten only in a finite number of ways, we can even use integers to denote the possible labels of the query edges. These ideas are the basis of the definition of an encoding function $\gamma_{Q,\Sigma}$ which is illustrated next.

Consider the query Q_1 and $\sigma_1, \sigma_2, \sigma_3$ of Example 1 yielding rewritings Q_1^1–Q_1^5 as in Figure 3. For simplicity, assume the edges of Q_1 being ordered by depth.

So the edges labelled by dept, prof, and contact are indexed by *1*, *2*, and *3*, respectively. To begin, we represent the query Q_1 with $\langle 0, 0, 0\rangle$ where the value 0 at position *1*, *2*, and *3*, of the array denote the fact that no edge is rewritten. Then, the rewritings Q_1^2, Q_1^4 can be represented by the arrays $\langle 0, 0, 1\rangle$, $\langle 0, 0, 2\rangle$, denoting the fact that the edge labelled by contact has been rewritten either by phone or mail while the rewritings Q_1^1, Q_1^3, Q_1^5 can be represented by the arrays $\langle 0, 1, 0\rangle$, $\langle 0, 1, 1\rangle$ and $\langle 0, 1, 2\rangle$ where prof is replaced by director and the edge labelled with contact is rewritten (or not) as before. The resulting encoding function is $\gamma_{Q,\Sigma} = \{(1, 0, \mathsf{dept}), (2, 0, \mathsf{prof}), (2, 1, \mathsf{director}), (3, 0, \mathsf{contact}), (3, 1, \mathsf{phone}), (3, 2, \mathsf{mail})\}$.

It is important to notice that at this point $\gamma_{Q,\Sigma}$ establishes a bijection from arrays to the rewritings of a query. The next step towards our goal of mapping rewritings to integers is to map the arrays encoding the rewritings to a sequence of successive integers. To do so, we see an array as a number in a multiple base $(b_1, \ldots, b_n)$ where each b_i denotes the number of possible rewritings of the i-th edge of Q. An array $\langle \mathsf{c}_1 \ldots \mathsf{c}_n\rangle$ in the base $(b_1, \ldots, b_n)$ corresponds to the integer $\mathbf{p} = \mathsf{c}_1 + \mathsf{c}_2 * B_1 \ldots + \mathsf{c}_n * B_{n-1}$ with $B_1 = b_1$ and $B_i = b_i * B_{i-1}$ for $i \geq 2$. In the example, the arrays $\langle 0, 0, 0\rangle, \langle 0, 1, 0\rangle, \langle 0, 0, 1\rangle, \langle 0, 1, 1\rangle, \langle 0, 0, 2\rangle, \langle 0, 1, 2\rangle$ in base $(b_1, b_2, b_3) = (1, 2, 3)$ correspond to the integers in the interval $[\mathbf{0}, \mathbf{5}]$, respectively. For instance, $\langle \mathsf{c}_1, \mathsf{c}_2, \mathsf{c}_3\rangle = \langle 0, 1, 1\rangle$ correspond to the integer $\mathbf{3}$ as, given that $B_1 = b_1 = 1$ and $B_2 = 2$, we have $0 + 1 * B_1 + 1 * B_2 = \mathbf{3}$. Conversely, the integer $\mathbf{p}$ in base $(b_1, \ldots, b_n)$ corresponds to the array $\langle \mathsf{c}_1, \ldots, \mathsf{c}_n\rangle$ where $\mathsf{c}_i = (d_i \mod b_i)$ where $d_1 = \mathbf{p}$ and $d_i = (d_{i-1} \div b_{i-1})$ for all $i \geq 2$. Of course, it must be that $0 \leq \mathbf{p} < B_n$. The correspondence between rewritings and integers is outlined in Figure 3. Finally note that by using the same formula we can compute the size of the rewriting set of a query, which is B_n, with n the number of edges of Q.

Building the encoding function. In the general case 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. Hence, the number of alternative keys for a single edge has to be inferred by looking at the dependencies between the keys in Σ. In doing so, we have to distinguish between the different types of edges of the query. For every edge of the query labelled by k the set of possible rewritings is made of all k' for which there exists a sequence of rules $\sigma_1, \ldots, \sigma_n$ of the form $\sigma_i = \mathsf{k}'_i \longrightarrow \mathsf{k}_i$ such that $\mathsf{k}_i = \mathsf{k}'_{i+1}$ for all $1 \leq i < n$, with $\mathsf{k}'_1 = \mathsf{k}'$ and $\mathsf{k}_n = \mathsf{k}$. For every edge of the query labelled by k ending on a node labelled by $exists the set of possible rewritings is made of all k' for which there exists a sequence of rules this time either of the form $\sigma_i = \mathsf{k}'_i \longrightarrow \exists \mathsf{k}_i$ or $\sigma_i = \mathsf{k}'_i \longrightarrow \mathsf{k}_i$ satisfying the same condition as before. Note that it is possible to analyze Σ *independently of any query*, and therefore compute once the possible rewritings of a key depending on the cases described before. Then, the construction of $\gamma_{Q,\Sigma}$ follows by fixing any total order on the edges of T_Q. The size of $\gamma_{Q,\Sigma}$ is bounded by $|Q| \times |\Sigma|$. This avoids to communicate to the units the whole $Rew(Q, \Sigma)$, whose size can be exponential.

In conclusion, the key properties achieved with our interval encoding are that 1) we avoid a "centralized" enumeration of the rewritings (which is parallelized) and 2) minimize communication costs by sending to each unit only a pair of values (i, j) denoting a (possibly exponentially large) query set it has to handle.

4 Performance Evaluation

We implemented our approach in Java and parallelized query rewriting and evaluation by executing concurrent threads and using different cores of a machine. Nevertheless, our approach is suitable for any shared nothing parallel framework. For example, the threads can also be executed in the nodes of a distributed cluster, if such a cluster is available. The three main modules of our tool are dedicated to *i*) the interval encoding, *ii*) rewriting generation, and *iii*) query evaluation. Next, we use the term *query answering* for the combination of the three tasks, which amounts to the whole task of answering queries under constraints.

We performed an experimental evaluation whose goal is to show the benefits of parallelization when querying key-value stores under semantic constraints. We deployed our tool on top of key-value store MongoDB version 3.6.3. Our experiments are based on the XMark benchmark which is a standard testing suite for semi-structured data [11]. XMark provides a document generator whose output was translated to obtain JSON records complying with our setting. Precisely, we performed our experiments on a key-value store instance created by shredding XMark generated data in JSON records. The results reported here concern an instance created from 100MB XMark and split in ∼60K records of size ∼1KB. XMark also provides a set of queries that were translated to our setting.To test query evaluation in the presence of constraints, we then extended the benchmark by manually adding a set of 68 rules on top of the data. These are "specialization" rules of the form $\mathsf{k}_{new} \rightarrow \mathsf{k}_{xmark}$ where k_{xmark} is a key of the XMark data vocabulary and k_{new} is a fresh key that does not appear in XMark. The benchmark data employs a vocabulary made of 91 keys and the rules define the specialization of 40 among them. More precisely, 20 keys have 1 specialization, 14 keys have two specializations, 5 have three specializations and 1 key has 5. Accordingly, the generated XMark data has been modified by randomly replacing one of such keys by one of its specializations thereby mimicking the fact that datasets use more specific keys while the user asks a high-level query.

Tests were performed on an Ubuntu 18.10 64-bit system, running on a machine that provides an Intel Core i7-8650U CPU 4 cores, 16 GB of RAM, and an Intel SSD Pro 7600p Series. Figure 4 summarises *i*) the query answering time under constraints and *ii*) the speed up of our parallel technique for 10 XMark queries, by varying the number of threads. The speed-up is defined as the ratio between the case of 1 thread (*i.e.*, without parallelization) and the case with n threads. As expected, our results show that the query answering time depends on the size of the rewriting set, and the queries are thus sorted according to this criterion. Query answering time with *one thread* takes up to 1.3s for queries with less 150 rewritings (i.e., q_4, q_{10}, q_1, q_2) and increases to $2.8s$ for q_3, which has 324 rewritings. However, by using four threads, answering time for q_3 drops to $1.3s$ (55% time reduction). Answering q_7, which has 1296 rewritings, takes $11s$. This falls to $4.7s$ by using four threads (58% time reduction). The same can be observed for q_8 and q_9. More generally, our results show that already by only using *two* threads, there is a 1.5x speedup (33% reduction) of query answering time for almost all queries. This increases to a 2/2.3x speedup (50-58% time re-

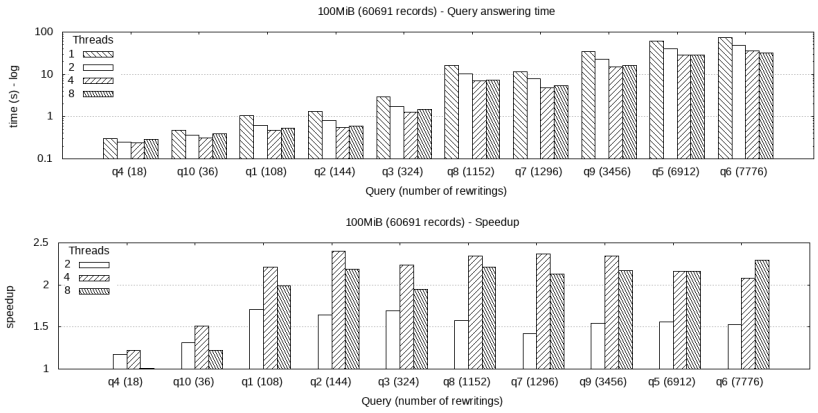

Fig. 4. Evaluation time and speedup of our method for XMark queries on MongoDB

duction) when *four* threads are used. Interestingly, this is the maximum number of concurrent physical threads of our test machine, and we observe that when using eight virtual threads essentially no improvement can be further remarked. Naturally, when the number of rewritings of a query is too small, the impact of parallelization is less important. For example, as illustrated by q_4, which has only 18 rewritings, only a 1.2x speedup is achieved with four threads. Summing up, this shows the interest of parallelization in querying key value stores under semantic constraints.

Conclusion. In this paper, we proposed a parallel technique for the efficient rewriting and evaluation of tree-shaped queries under constraints based on an interval encoding of the rewriting set of a query. We implemented our solution and measured its performance using the XMark benchmark. The results show significant performance gains compared to the baseline sequential approach.

Acknowledgements This work has been partially supported by the ANR CQFD Project (ANR-18-CE23-0003).

References

1. *MongoDB www.mongodb.com.*
2. *CouchDB couchdb.apache.org.*
3. M.-L. Mugnier and M. Thomazo An introduction to ontology-based query answering with existential rules. *Reasoning Web International Summer School 2014*
4. 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 2007*
5. Poggi, A., Lembo, D., Calvanese, D., De Giacomo, G., Lenzerini, M., and Rosati, R. Linking data to ontologies. *J. Data Semantics 2008*
6. Mugnier, M., Rousset, M., and Ulliana, F. Ontology-mediated queries for NOSQL databases. AAAI 2016
7. Botoeva, E., Calvanese, D., Cogrel, B., Rezk, M., and Xiao, G. OBDA beyond relational DBs: A study for MongoDB. *2016.*
8. Bienvenu, M., Bourhis, P., Mugnier, M., Tison, S., and Ulliana, F. Ontology-mediated query answering for key-value stores. In *IJCAI 2017*
9. E. Botoeva, D. Calvanese, and B. Cogrel and X. Guohui, Expressivity and complexity of MongoDB queries, ICDT, 2018
10. G. Xiao, D. Calvanese, R. Kontchakov, D. Lembo, A. Poggi, R. Rosati, and M. Zakharyaschev, Ontology-based data access: A survey, IJCAI, 2018
11. Schmidt, A., Waas, F., Kersten, M., Carey, M. J., Manolescu, I., and Busse, R. Xmark: A benchmark for XML data management. VLDB 2002