ECAI 2020
G.D. Giacomo et al. (Eds.)

doi:10.3233/FAIA200168

# Rewrite or Not Rewrite? ML-Based Algorithm Selection for Datalog Query Answering on Knowledge Graphs

**Unmesh Joshi**[1] and **Ceriel Jacobs** and **Jacopo Urbani**

**Abstract.** Query-driven reasoning techniques with Datalog rules, like Magic Sets (MS), are ideal for implementing query answering on Knowledge Graphs (KGs). For some queries, executing a rewriting procedure like MS is the best choice, but for others a non-rewriting procedure like Query-subquery (QSQ) can be faster. Choosing beforehand which procedure should be used is not trivial and mistakes can be costly. To address this problem, we describe a first-of-its-kind method that builds a Machine Learning (ML) model to predict whether a query should be answered with MS or with QSQ. Experiments on several well-known KGs show that our method can return accurate predictions, and this leads to a significant reduction of the response time of query answering.

## 1 Introduction

Knowledge bases in the form of Knowledge Graphs (KGs) have become a crucial asset for enhancing tasks like web search or data integration, and providing efficient query answering is a key problem in many of such tasks.

Recently, the rule-based language Datalog [1] has emerged as a good choice for implementing query answering due to its expressivity [18, 21], and the availability of engines that can reason over large KGs in a scalable manner [32, 28, 5, 37, 6, 9, 2, 33]. One key service offered by datalog engines consists of allowing the user to query the set $P^\infty(I)$ of all derivations that can be produced with a given ruleset $P$ and KG $I$. Currently, the most popular strategy for supporting this operation consists of first fully materializing $P^\infty(I)$, and then constructing appropriate index structures to facilitate query answering. While this approach can offer good response times, it is not ideal in several cases. First, it is inefficient if the input contains many nonsensical derivations, which occurs frequently in the Web of data [4]. Moreover, precomputing $P^\infty(I)$ can be unnecessarily expensive if the user is only interested in a subset of it, or if the rules are supposed to infer knowledge that should not be reused, like inconsistencies. Finally, computing $P^\infty(I)$ can be too slow if the input and/or rules change frequently, or if the input has a short time validity (e.g., see stream reasoning [25]). Note that this last case can be handled efficiently by procedures for incremental reasoning [15, 14, 26, 19], but these still require an initial full computation of $P^\infty(I)$.

A well-known alternative approach to materialization is to perform reasoning as part of the query answering procedure. In this case, we can employ techniques like Magic Sets (MS) [3] or Query-subquery (QSQ) [1, 34] to compute only the derivations that are relevant for answering the input query, avoiding, whenever possible, the computation of the entire $P^\infty(I)$.

These techniques do not suffer from the limitations outlined above but the runtime of MS or QSQ can be either instantaneous or even higher than computing $P^\infty(I)$ from scratch [3]. We empirically observed that if the query requires little or no reasoning, then a low-latency implementation of a non-rewriting procedure like QSQ is usually much faster than a rewriting procedure like MS. However, MS becomes a better choice for more expensive queries because it allows the usage of state-of-the-art materialization engines that are more suitable for such queries. We could take advantage of these differences if we would have a technique that can select beforehand the best algorithm for each query. Unfortunately, as far as we know, no such technique exists.

To fill this gap, we propose to use a Machine Learning (ML) model, and in particular a binary classifier, to perform dynamic algorithm selection, i.e., decide whether the query should be answered with QSQ or MS. The problem of performing algorithm selection is well-known in the literature [31] and machine learning has been shown to be effective in selecting the best algorithms for search problems [22] or the satisfiability problem (SAT) [38], but not yet for datalog query-driven reasoning. Some additional related work can be found in the field of databases where machine learning is used for various purposes (e.g., for cardinality estimation [20], join ordering [24], parameter tuning [35], or indexing [23]). Our setting is, however, different especially because of the support of recursion.

Algorithm selection for datalog query-driven reasoning is challenging for various reasons. First, we need to define some meaningful numerical features that capture the difficulty of reasoning so that the classifier can return accurate predictions. Second, the predictions should not only be accurate but also fast to compute to avoid a negative repercussion on the response time. Finally, we must obtain a large amount of training data (i.e., queries) to train the classifier.

To address the first challenge, we map each query to five quantifiable features like the estimated cardinality, the number of rules executions, etc., which are then used by the binary classifier to choose the best algorithm. To quickly compute values for these features (second challenge), we introduce a lightweight resolution-based procedure to estimate their values. Finally, we address the last challenge with an automatic procedure which, given $I$ and $P$, analyses the dependencies between atoms in the program and returns many example queries of different types.

We evaluated our technique using different classifiers (Support Vector Machines (SVM) [10], Naive Bayes classifiers [7], etc.), artificial and real-world KGs, and rulesets which were either extracted from OWL ontologies [12] or mined using association rule mining algorithms [11]. Our results are encouraging as they show that our technique can make quick and accurate predictions, and this results in a significant reduction of the response time of query answering

[1] Vrije Universiteit Amsterdam, The Netherlands, email: u.n.joshi@vu.nl

when compared to the case when only one algorithm is used.

An extended version of this paper with a more detailed evaluation, and additional experimental data is available in the Github repository at https://github.com/unmeshvrije/ecai2020.

## 2 Preliminaries

We start our discussion by introducing some background notions to specify the task of query answering with datalog rules and with a short description of the MS and QSQ algorithms.

**Datalog.** Let $\mathcal{P}, \mathcal{V}, \mathcal{C}$ be disjoints sets of predicates, variables, and constants symbols respectively. Each predicate $p \in \mathcal{P}$ has arity $\mathsf{ar}(p) \geq 0$. A *term* $t$ is a member of $\mathcal{C} \cup \mathcal{V}$ and an *atom* is an expression $p(\mathbf{t})$ where $p \in \mathcal{P}$ and $\mathbf{t}$ is a list of terms of length $\mathsf{ar}(p)$. If $\mathbf{t}$ does not contain variables, then $p(\mathbf{t})$ is a *fact*. Rules are expressions of the form: $r : e_1(\mathbf{u_1}) \wedge \ldots \wedge e_m(\mathbf{u_m}) \wedge q_1(\mathbf{s_1}) \wedge \ldots \wedge q_n(\mathbf{s_n}) \rightarrow h(\mathbf{t})$ where $e_1(\mathbf{u_1}), \ldots, e_m(\mathbf{u_m}), q_1(\mathbf{s_1}), \ldots q_n(\mathbf{s_n}), h(\mathbf{t})$ are atoms, $e_1, \ldots, e_m$ are extensional predicates (i.e., they never appear in the right-side of rules), and $q_1, \ldots, q_n$ are intensional ones. We denote the conjunction $e_1(\mathbf{u_1}) \wedge \ldots \wedge e_m(\mathbf{u_m})$ as $\mathsf{edb}(r)$, the conjunction $q_1(\mathbf{s_1}) \wedge \ldots \wedge q_n(\mathbf{s_n})$ as $\mathsf{idb}(r)$, and $\mathsf{edb}(r) \wedge \mathsf{idb}(r)$ as $\mathsf{body}(r)$. We write $\mathsf{head}(r)$ to refer to the atom at the right-side of $r$. When the order is irrelevant, we view $\mathsf{edb}(r), \mathsf{idb}(r), \mathsf{body}(r)$ as sets of atoms. We assume rule safety, i.e., every variable in $\mathsf{head}(r)$ must appear in $\mathsf{body}(r)$. A *database* (*program*) is a finite set of facts (rules). A *Datalog query* (or *query* henceforth) $Q = (G, P)$ is a pair where $G$ is an atom called *goal atom*, and $P$ is a program.

In order to define the rule execution, let $\sigma$ be a *substitution*, i.e., a partial mapping from variables to terms. We use $\sigma$ as postfix operator to replace variables with the corresponding mapping in atoms, tuples or sets of them. For instance, if $\sigma = \{X \mapsto a, Y \mapsto b\}$ where $a, b \in \mathcal{C}$, then $p(X, Y)\sigma = p(a, b)$.

Let $I$ denote the input set of facts, $r(I) = \{\mathsf{head}(r)\sigma \mid \mathsf{body}(r)\sigma \subseteq I\}$ represent the application of rule $r$ on $I$, and $P(I) = \bigcup_{r \in P} r(I)$ be its extension to the program. Further, we set $P^0(I) = I$ and define $P^{i+1} = P(P^i(I)) \cup P^i(I)$ for $i \geq 0$. The *materialization* of $I$ with $P$ is the union $P^\infty(I) = \bigcup_{i \geq 0} P^i(I)$.

**Adornments and sequences.** An *adornment* is a finite string from the alphabet $\{\mathtt{b}, \mathtt{f}\}$. An *adorned atom* $p^\alpha(\mathbf{t})$ is an atom where the adornment $\alpha$ is of length $\mathsf{ar}(p)$. A character $c$ in $\alpha$ *complies* with $t$ if $t$ is a constant and $c$=$\mathtt{b}$ or if $t$ is a variable and $c$=$\mathtt{f}$. An adornment $\alpha$ complies with $\mathbf{t}$ if each $c$ in $\alpha$ complies with its correspondent term in $\mathbf{t}$. We denote with $\mathsf{adorn}(p(\mathbf{t}))$ the atom $p^\alpha(\mathbf{t})$ where $\alpha$ complies with $\mathbf{t}$. Finally, a rule $r$ is *adorned* if $q_1(\mathbf{s_1}), \ldots, q_n(\mathbf{s_n}), h(\mathbf{t})$ are adorned with $\alpha_1, \ldots, \alpha_n, \alpha$ and each character $c$ in $\alpha_i$ equals to $\mathtt{f}$ iff 1) its corresponding term $t$ is a variable that does not occur in $\mathbf{s_1}, \ldots, \mathbf{s_{i-1}}$ and 2) any possible occurrences of $t$ in $\mathbf{t}$ correspond to $\mathtt{f}$ in $\alpha$[2]. We introduce two auxiliary functions: $\mathsf{adorn}(r, \alpha)$ returns the adorned version of rule $r$ where the adornment of $\mathsf{head}(r)$ is $\alpha$, while $\mathsf{bnd}(\mathbf{t}, \alpha)$ returns the sublist of $\mathbf{t}$ that maps to $\mathtt{b}$ in $\alpha$.

A *sequence* $s = \langle a_1, \ldots, a_n \rangle$ is a tuple of $\mathsf{len}(s) = n$ generic elements. The postfix operator $[i]$ returns the $i^{th}$ element, i.e., $s[i] = a_i$ while $\mathsf{append}(s, s_1, \ldots, s_j)$ returns a sequence where $s_1, \ldots, s_j$ are appended to $s$. We also view $s$ as a set and write $a \in s$ to refer to an element $a$ in $s$.

**Magic Sets.** Given the database $I$ and query $Q = (G, P)$, our goal is to compute the set of facts $\mathsf{ans}(I, Q) = \{G\sigma \mid G\sigma \in P^\infty(I)\}$. Query-driven reasoning procedures speed up the computation of ans by avoiding, whenever possible, the entire computation of $P^\infty(I)$. We consider *Magic Sets* (MS) [3], one of the most popular techniques of this kind. Given $Q$ in input, MS *rewrites* $Q$ creating a new program from $P$. The new program contains special "magic" predicates, called $mgc^*_*$ below, to derive only facts relevant to answers of $G$. This procedure, outlined as MS in Algorithm 1, is described with the following example.

[2] Usually, extensional predicates are not adorned.

**Algorithm 1:** MS(Q)

```
1  P_M := ∅, A := {adorn(G)}, B := ∅
2  while A ≠ B do
3      Q' := p^α(t) from A \ B
4      B := B ∪ {Q'}
5      R := {adorn(r, α) | r ∈ P ∧ head(r)θ = p(t)θ}
6      while R ≠ ∅ do
7          Remove rule r from R
8          Let p^α(t) := head(r); u := bnd(t, α);
9          S := edb(r) ∧ mgc_p^α(u)
10         P_M := P_M ∪ {S ∧ idb(r) → head(r)}
11         foreach i ∈ {1, ..., |idb(r)|} do
12             Let q_i^{α_i}(s_i) be the i^th atom in idb(r)
13             P_M := P_M ∪ {S → mgc_{q_i}^{α_i}(bnd(s_i, α_i))}
14             S := S ∧ q_i^{α_i}(s_i)
15             A := A ∪ {q_i^{α_i}(s_i)}
16 return (G, P_M)
```

**Example 1.** Let us consider the query $Q = (G, P)$ where $G = q(a, Y)$ and $P$ contains the rules

$$p(X, Y) \wedge p(Z, Y) \rightarrow q(X, Z) \quad (1)$$

$$s(Y, X) \rightarrow p(X, Y) \quad (2)$$

Line 1 adds $q^{bf}(a, Y)$ to $A$. In lines 3 and 4, the algorithm sets $Q' := q^{bf}(a, Y)$ and adds $Q'$ to $B$. Then, it puts all adorned rules that can produce answers for $Q'$ in $R$. In our case, it is $r : p^{bf}(X, Y) \wedge p^{fb}(Z, Y) \rightarrow q^{bf}(X, Z)$. Lines 8-10 will add to $P_M$ the rewritten version of $r$

$$mgc_q^{bf}(X) \wedge p^{bf}(X, Y) \wedge p^{fb}(Z, Y) \rightarrow q^{bf}(X, Z) \quad (3)$$

Note that here the atom $mgc_q^{bf}(X)$ obliges rule (3) to derive $q$-facts only if $X$ can be mapped to constants in $mgc_q^{bf}-$facts. Thus, if we add $mgc_q^{bf}(a)$ to $I$, then rule (3) will only derive answers for $Q$. The *for* loop in line 11 will process the idb atoms of $r$ left-to-right to restrict the derivation of $p-$facts in a similar way. It adds to $P_M$ the following rules

$$mgc_q^{bf}(X) \rightarrow mgc_p^{bf}(X) \quad (4)$$

$$mgc_q^{bf}(X) \wedge p^{bf}(X, Y) \rightarrow mgc_p^{fb}(Y) \quad (5)$$

Rules (4-5) are used to populate the magic predicates for $p$ so that only $p$-facts that are relevant for $Q$ are derived. The loop also adds the intensional body atoms of $r$ to $A$ so that further rules can be rewritten. Then, the algorithm returns to line 3 and selects $Q' = p^{bf}(X, Y)$. Lines 7 and 8 select the adorned version of rule (2) and lines 9 and 10 adds

$$s(Y, X) \wedge mgc_p^{bf}(X) \rightarrow p^{bf}(X, Y) \quad (6)$$

to $P_M$. Since $s$ is an extensional predicate, the loop in line 11 does not start. Processing the remaining atom in $A$ adds to $P_M$ the rule

$$s(Y, X) \wedge mgc_p^{fb}(Y) \rightarrow p^{fb}(X, Y) \quad (7)$$

At this point, $P_M$ contains rules (3-7) and the answers for $G$ can be computed by materializing $P_M^\infty(I \cup \{mgc_q^{bf}(a)\})$, which is likely to be faster than computing $P^\infty(I)$.

**QSQ.** Query-subquery (QSQ) [1] is another query-driven algorithm defined as a set-based variant of standard SLD resolution [36] with an additional *admissibility test* and *lemma resolution* to ensure termination [34]. QSQ restricts the reasoning process precisely in the same way as done by MS, namely with adorned rules and with special relations to restrict the number of derivations. The difference is that QSQ does not create a new program but proceeds instead in a top-down fashion with the original program. In Example 1, for instance, QSQ will first consider rule (1), and then use sideways information passing to propagate the constants in $G$ to the body atoms of rule (1). Then it will start evaluating the two body atoms one-by-one. The evaluation of the first body atom will start with the evaluation of the subquery $p(a, Y)$ which will trigger a recursive process until all its answers are computed. Then, the algorithm will move to the second body atom of rule (1), and execute a new subquery until all subqueries are computed.

## 3 Approach Rationale and Overview

Adding the magic predicates in the way outlined in Algorithm 1 is useful to restrict the reasoner to derive only relevant facts for $Q$. However, doing so might introduce other inefficiencies, which we categorize either as algorithmic or implementation-wise.

**Algorithmic inefficiencies.** MS rewrites the program without considering the input database, thus all possible rewritings must be included. In Example 1, for instance, the rewritten rule (7) is included in $P_M$ even if it might be excluded, e.g., if rule (5) does not produce any derivation (e.g., when no $p^{bf}$-fact joins with $mgc_q^{bf}$-facts). This inclusion is due to the fact that MS must produce a ruleset that is suitable for any input. While rewriting the rules is in practice a relatively fast procedure (in the order of milliseconds in our experiments), computing all possible rewritings might become a source of overhead if the query is selective and/or requires no reasoning. Moreover, evaluating a rule has often a bootstrap cost that is unconditional to the number of produced derivations. Therefore, unnecessary rules, like the ones that will not produce any derivation due to empty magic predicates in their bodies, should be avoided.

**Implementation-wise differences.** MS rewritings are elegant since they restrict the derivations in a declarative way. However, delegating the rule execution to a materialization engine can introduce additional overhead.

In contrast to MS, QSQ's top-down strategy can be implemented internally as a "pay-as-you-go" method where the adorned rules and the temporary relations are introduced only if needed. In Example 1, for instance, QSQ can avoid to adorn rule (2) with the head adornment `fb` if there are no $s$-facts that join with $mgc_p^{bf}$-facts (rule (6)), and thus would also avoid to create all temporary data structures necessary to store additional subqueries. This avoidance results in less overhead but the downside is that if we apply QSQ, then we cannot rely on advanced techniques used in state-of-the-art materialization engines for speeding up the rule execution (e.g., parallelism in RDFox [28], or compression in VLog [32]). Therefore, this approach should be used only if the query requires little or no reasoning.

**Approach Overview.** In order to take the best of both approaches, our proposal is to train a binary classifier to decide whether the query should be answered with MS or with QSQ, using previous query executions as training evidence. After the training is completed, query answering is implemented as follows: First, the input query is translated into a numerical feature vector that can serve as input for the classifier. Then, we use the classifier to identify what technique is likely to be faster, and finally execute the query with it.

In order to implement our approach, we must address the challenge of translating the input query into a vector of numerical features that somehow estimates the expected cost of executing the query. Moreover, we also need a procedure to extract a large number of sample queries to construct suitable amount of training evidence for the classifier. We describe these two procedures in Sections 4 and 5 respectively.

## 4 Feature Estimation

The performance of the classifier depends on the quality of the features: If they are not good indicators of the difficulty of reasoning, then the classifier would not be able to make an accurate prediction. After profiling the runtime of MS and QSQ on multiple example queries, we identified five quantifiable features that can give an indication of the difficulty of reasoning. These are:

- $f_1$*: number of substitutions.* This feature estimates the number of substitutions computed to infer $P_M^\infty(I)$. Substitutions are computed by performing a series of natural joins over the relations that store the body facts. Since joins can be time-consuming, a high value of this feature can predict a longer response time.
- $f_2$*: number of relevant facts.* This feature estimates the number of facts with non-magic predicates in $P_M^\infty(I)$. Since each derived fact requires additional computation and storage, a high value of $f_2$ indicates longer runtimes.
- $f_3$*: number of subqueries*. This feature corresponds to the number of subqueries that are produced during the computation of $P_M^\infty(I)$. Note that with MS the subqueries are the facts with magic predicates, while QSQ uses ad-hoc relations. Since subqueries can potentially trigger further reasoning, a high value of $f_3$ can also indicate longer runtimes.
- $f_4$*: number of rules applications and* $f_5$*: number of unique rules.* These two features correspond to the number of rules applications ($f_4$) and rules ($f_5$) that are triggered by the reasoning process. For instance, the computation of $P_M^\infty(I)$ could trigger the application of $r_1$ twice and of $r_2$ once. In this case, $f_4 = 3$ and $f_5 = 2$. These two features are included because more rules applications introduce additional computation, while the fact that a query triggers the execution of different rules indicates that it is potentially less selective, and hence take longer to be executed.

The classifier trained with features $f_1, \ldots, f_5$ returned a good accuracy (see Section 6 for the empirical evaluation). To further study whether other features could give equal or better results, we have experimented with other two types of features, called $f_6$ and $f_7$ below.

- $f_6$*: boundedness.* We added as extra features a vector of boolean features which are activated i.e., equal to true) if the query contains constants in specific locations. For example, if $q(\mathbf{t}) = p(X, a)$, where $X$ is a variable and $a$ a constant, then the corresponding vector is $\langle 0, 1\rangle$. The rationale is that constants can potentially reduce the number of answers and hence improve the runtime of reasoning.
- $f_7$*: involved intensional predicates.* Instead of counting the number of subqueries as with $f_3$, we constructed a boolean vector of features where each feature maps to an intensional predicate and it is activated if there is a subquery with the corresponding predicate. For instance, the feature that maps to the predicate $p_1$ is activated if there is a subquery with predicate $p_1$. The motivation for adding them is that the subset of facts with certain predicates in $P^\infty(I)$ can be small

**Algorithm 2:** Feature estimation functions for query $Q$ on database $I$. $maxd$ is a global constant (default is 5).

```
1  function est (Q, I)
2      f1, f2, f3, rules := estQuery(Q, I, 0)
3      f4 := len(rules)      f5 := |{r | r ∈ rules}|
4      return ⟨f1, f2, f3, f4, f5⟩
5  function estQuery (Q := (G, P), I, d)
6      if d ≥ maxd then return ⟨1, 1, 0, ⟨⟩⟩
7      f1 := f2 := |{θ | Gθ ∈ I}|
8      f3 := 1      rules := ⟨⟩
9      foreach r ∈ P do
10         r is of the form B1 ∧ ... ∧ Bn → H
11         if ∃θ | Hθ = Qθ then
12             g1, g2, g3, r4 := estRule(θ, r, I, P, d + 1)
13             f1 := f1 + g1      f2 := f2 + g2      f3 := f3 + g3
14             rules := append(rules, r, r4)
15     return ⟨f1, f2, f3, rules⟩
16 function estRule (θ, r, I, P, d)
17     tmp := 1      f1, f2, f3 := 0      rules := ⟨⟩
18     foreach B ∈ body(r) do
19         q1, q2, q3, q4 := estQuery((Bθ, P), I, d)
20         tmp := tmp * q2
21         f1 := f1 + q1      f2 := f2 + q2      f3 := f3 + q3
22         rules := append(rules, q4)
23     return ⟨f1 + tmp, f2, f3, rules⟩
```

(large) and hence reasoning over them can be quick (slow).

Eventually, features $f_6$ and $f_7$ were discarded since they returned lower accuracies in all but one case, and we retained only $f_1, \ldots, f_5$.

**Estimating the features.** An exact calculation of $f_1, \ldots, f_5$ can be time consuming, and this may cancel the advantage of using a faster query-driven reasoning algorithm. To avoid this problem, we introduce a procedure, represented by function est in Algorithm 2, to provide a *quick estimation* of $f_1, \ldots, f_5$ (note that this procedure does not describe the computation of $f_6, f_7$ since these were discarded features). This function proceeds in a top-down fashion that mimics the functioning of QSQ without producing any derivation.

The procedure works as follows: Function est first invokes the subroutine estQuery, which compute the estimations for an input query. Function estQuery selects all rules that can produce some answers for $Q$ (line 12) and, for each of them, invokes function estRule which returns the list of estimates that characterize execution of rule $r$. These estimates are added together (line 13), while $r$ and the rule applications invoked during the call of estRule ($r_4$) are appended to the list of rule applications (line 14).

Function estRule traverses each body atom of rule $r$ (line 18) and invokes estQuery for each of them. Variable $tmp$ stores the product of the number of facts that instantiate the body atoms of $r$ (line 20), which is a number that approximates the maximum number of possible derivations. The number of substitutions ($f_1$) returned by estRule is computed as the sum of $tmp$ and all values of $f_1$ which are returned by estQuery on each body atom (line 19).

Note that est does not use blocking mechanisms, like tabling, to ensure termination. It relies instead on the counter $d$ that is increased at every call of estRule, and recursive calls are blocked after $d \geq maxd$ (line 6). After experimenting with different values, we observed that $maxd := 5$ is a good compromise between runtime and accuracy. Finally, $est$ returns a tuple with the approximations of the five features: $f_1, f_2, f_3$ are returned by estQuery, while $f_4$ and $f_5$ are computed from the multiset of triggered rules (line 3).

**Algorithm 3:** Functions for creating goal atoms for database $I$ and program $P$. $maxd$ and $maxs$ are global constants (defaults are 5 and 50 respectively.

```
1  function dfs (s, path, G)
2      if len(path) > maxd then return {path}
3      B := ∅
4      foreach a_{s,q} ∈ A(G) do
5          newpath := append(path, a_{s,q})
6          B := B ∪ dfs(q, newpath, G)
7      return B
8  function goals (I, P)
9      Let G be the dependency multigraph of P
10     O := {}      A := ∅
11     foreach p ∈ P do A := A ∪ dfs(p, [], G)
12     foreach path ∈ A s.t. len(path) > 0 do
13         ⟨h, b⟩ := lbl(last(path))
14         Σ := sample of maxs facts from {bσ | bσ ∈ I}
15         foreach f ∈ Σ do
16             f' := f      i := len(path)
17             while i > 0 do
18                 ⟨h, b⟩ := lbl(path[i])
19                 if ∃σ s.t. bσ = f' then
20                     f' := hσ
21                     O := O ∪ {f'}
22                 else break
23                 i := i − 1
24     return O
```

## 5 Goal Atoms Generation

In this section, we describe a method for obtaining example goal atoms for a given program $P$ and database $I$ that we can use as training evidence. Our method constructs a directed labelled multigraph that represents the dependencies between the predicates, and then creates example goal atoms by traversing it with samples from $I$. We use this method because if we would simply create random goal atoms then there will be a high probability that they will produce no answer. In contrast, the multigraph can guide the selection of atoms that trigger some reasoning.

We construct the graph as follows. Let $\mathcal{G}$ be the directed labelled multigraph where vertices map one-to-one to predicates in $\mathcal{P}$ and arcs represent the dependencies between the head and the body predicates in the rule. Throughout, we denote with $\mathsf{A}(\mathcal{G})$ the set of arcs in $\mathcal{G}$, $v_p$ the vertex that maps to the predicate $p$, and $a_{q,p}$ the arc from $v_q$ to $v_p$. The multigraph contains all dependencies between the head and the body atoms that share some variables, i.e., $a_{q,p} \in \mathsf{A}(\mathcal{G})$ iff $q(\mathbf{t}) = \mathsf{head}(r)$, $p(\mathbf{s}) \in \mathsf{body}(r)$, and $\mathsf{vars}(\mathbf{t}) \cap \mathsf{vars}(\mathbf{s}) \neq \emptyset$.

We label each arc with the pair of atoms that define the dependency. Let $a_{q,p} \in \mathsf{A}(\mathcal{G})$ be the arc that represents the dependency in rule $r$ between the head atom $q(\mathbf{t})$ and body atom $p(\mathbf{s})$. In this case, we assign to $a_{q,p}$ the label $\mathsf{lbl}(a_{q,p}) := \langle q(\mathbf{t}), p(\mathbf{s}) \rangle$. Note that if the body of the rule contains multiple body atoms with the same predicate, then there will be multiple arcs unless the labels are equal.

| Dataset | # Triples | # Terms | # Rules ( # Derived facts) | | |
|---|---|---|---|---|---|
| | | | L | LE | AMIE |
| LUBM1K | 133M | 33M | 170 (293M) | 182 (444M) | |
| DBpedia | 112M | 18M | 9396(146M) | | |
| Claros | 19M | 6M | 2689 (108M) | 2749 (482M) | |
| Freebase | 14M | 6M | | | 1449 (26M) |
| YAGO | 1M | 800K | | | 127 (44M) |

**Table 1.** Statistics about the used databases and rule sets.

Procedure goals in Algorithm 3 shows how we can traverses $\mathcal{G}$ to compute examples of goal atoms. The algorithm considers each predicate (line 11) and computes all paths by performing a depth first search (function dfs, lines 1–7) up to a maximum depth ($maxd$). Then, for each path, it considers the label of the last arc in the path (line 13) and retrieves from $I$ up to $maxs$ facts that instantiate the body atom in the label (line 14). For each fact, it traverses the arcs in the path (lines 17–22), propagating the substitutions for the body atom to the head of the rule (if such substitutions exist, line 19). Example goal atoms are obtained by passing the substitutions to the head atoms of the labels (line 20).

All generated goal atoms are returned in $O$. If $O$ contains multiple equal atoms (modulo variable renaming), then only one is retained.

**Example 2.** Consider the program $P$ that consists of rules (1) and (2) of Example 1 and the database $I = \{s(u,a), s(v,b)\}$. The directed multigraph of $P$ will contain three arcs: $a_1$ with label $\langle q(X,Z), p(X,Y)\rangle$, $a_2$ with label $\langle q(X,Z), p(Z,Y)\rangle$ from rule (1), and $a_3$ with label $\langle p(X,Y), s(Y,X)\rangle$ from rule (2). Let us now assume that at some point $path \coloneqq \langle a_2, a_3\rangle$ in Algorithm 3 (line 12). This path has the length 2. Thus, it enters in the loop at line 13, we access the label of the last arc ($a_3$) of this path which is $h = p(X,Y)$ and $b = s(Y,X)$.

In line 14, we find all the facts that match the body atom $s(Y,X)$. Thus, $\Sigma = \{s(u,a), s(v,b)\}$. In line 15, we start traversing all facts. Then at line 17, we start traversing the path backwards.

Let $f' = s(u,a)$. In line 19, we get substitution $\sigma$ ($\{X \mapsto a, Y \mapsto u\}$) that can be applied to the atom $s(Y,X)$ in order to get a fact $f'$. Then, line 20 makes $f' = p(X,Y)\sigma$ i.e., $f' = p(a,u)$. This is added to the set of goal atoms. In the next iteration of the while loop, we extract the label of the next arc ($a_2$) in line 18 leading to $h = q(X,Z)$ and $b = p(Z,Y)$. Now $\sigma = \{Z \mapsto a, Y \mapsto u\}$ (in line 19). In line 20, $f' = q(X,Z)\sigma$ i.e., $f' = q(X,a)$ which is then added to the set of goal atoms. The algorithm continues similarly for the paths obtained from other predicates.

## 6 Empirical Evaluation

**Experimental Setup.** To test our approach, we used the implementations of MS and QSQ provided by the system VLog [32]. We chose VLog for its state-of-the-art runtimes with these algorithms.

As inputs, we used the artificial benchmark tool LUBM [13] to generate an artificial KG with 1K universities (LUBM1K), and four subsets of real-world KGs: DBpedia [8], Freebase[3], YAGO2 [17], and Claros [28]. The first three KGs contain encyclopedic-like facts primarily extracted from Wikipedia and other sources while Claros is an ontology in the domain of cultural heritage. The content of these datasets can be represented using $\langle s,p,o\rangle$ RDF triples [16]; hence each dataset can be seen as a set of Datalog facts with one ternary predicate. LUBM1K, DBpedia, and Claros have been used previously [32] and are publicly available. The samples of YAGO and Freebase are in our data repository.

As rulesets, we considered the ones obtained by translating the underlying ontologies in the KG, and the ones produced by applying mining tools. The first types of rulesets encode the knowledge provided in the ontologies, while the seconds capture some refined correlations in the KG. For LUBM1K, Claros, and DBpedia, we used incomplete translations of the corresponding OWL ontologies that cover the expressivity of OWL2 RL [12]. These rulesets were initially provided by [27] and were also used in [32]. We considered the translations "L" for all three KGs while for LUBM1K and Claros we also considered the translations "LE". The "LE" rulesets contain more rules, have rules with more body atoms (thus more joins) and produce more derivations than the "L" rulesets.

| Dataset (Ruleset) | # Goal Atoms QSQ | # Goal Atoms MS | Runtimes (seconds) Q | Runtimes (seconds) F | Runtimes (seconds) R |
|---|---|---|---|---|---|
| LUBM1K (L) | 2919 | 350 (27) | 0.4 | 81 | 996 |
| LUBM1K (LE) | 3019 | 367 (36) | 0.5 | 74 | 1283 |
| DBpedia (L) | 14071 | 1256 (14) | 2.06 | 3680 | 4192 |
| Claros (L) | 1081 | 126 (70) | 0.31 | 203 | 2026 |
| Claros (LE) | 1324 | 151 (71) | 1.03 | 259 | 2871 |
| Freebase (AMIE) | 9923 | 77 (0) | 0.4 | 330 | 1637 |
| YAGO (AMIE) | 1018 | 378 (74) | 0.08 | 228 | 2462 |

**Table 2.** Statistics about the generated training data.

For YAGO and Freebase, we used the rulesets mined by AMIE [11], a state-of-the-art mining tool. AMIE mines rules from KGs based on the number of facts that support them. For example, AMIE mined $livesIn(Y,C) \wedge marriedTo(X,Y) \rightarrow livesIn(X,C)$ because many entities that are married live in the same location as their partners. Table 1 reports details for each input.

We used the implementations of the statistical models provided by the library *scikit-learn*[4]. Tests were run on a Macbook Pro with 2.2GHz i7 CPU and 16GB RAM.

**Training Data Generation.** To obtain training data, we used Algorithm 3 to generate training goal atoms. Then, for each goal atom, we executed the corresponding $\langle G,P\rangle$ query both with QSQ and MS. Since the number of atoms was large, we set a timeout: If the query timed out with both methods, we labeled it with MS since it is the method that can handle expensive atoms more efficiently due to its usage of materialization engines. Otherwise, we selected the algorithm with the lowest runtime. After experimenting with different timeouts, we selected 10s as a good compromise between training speed and accuracy.

Details about the training data are in Table 2. The second and third columns report the number of atoms assigned to QSQ and MS respectively. The number of atoms assigned to MS due to timeout is reported between parentheses. The fourth column (Q) is the runtime of Algorithm 3 while the fifth column (F) is the sum of the runtimes of executing Algorithm 2 to compute the features for the queries. The last column (R) reports the cumulative runtime of executing both QSQ and MS on each training query to compute the correct label. The sum Q+F+R is the total runtime for creating the training set for the given KG and ruleset. In the worst case, creating the training data took about 2h6m (DBPedia).

**Training.** We considered four binary classifiers: A Support Vector Machine (SVM) [10], Naïve Bayes [7, chapter 8], Decision Trees [7, chapter 14] and Logistic Regression (LR) with 0.5 as threshold [7, chapter 3]. These models are among the most common ones used for classification. After the training data was generated, training these models was rather fast: It took 3 milliseconds (YAGO) and 7 seconds (DBpedia) in the best and worst cases respectively.

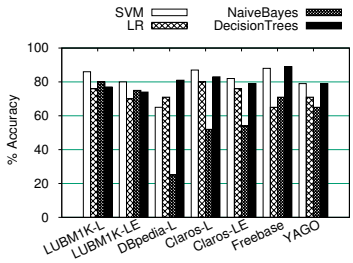
**Predictions.** We tested the performance of our method both with SPARQL BGP queries [30] and with atomic queries over the intensional predicates in the program. Each SPARQL query is executed by 1) adding an extra rule which has the triple patterns as body and a fresh predicate with the projected variables as head, and 2) use the head atom as goal atom. This is a well-known conversion strategy to

[3] https://developers.google.com/freebase/data

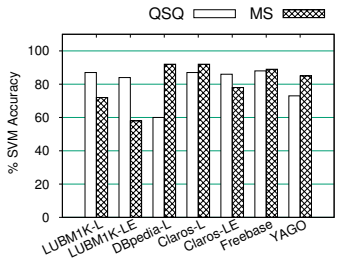
[4] http://scikit-learn.org/stable/index.html

| Dataset (Ruleset) | # Goal Atoms QSQ | # Goal Atoms MS | # Atom in Subsets B | # Atom in Subsets M | # Atom in Subsets G |
|---|---|---|---|---|---|
| LU (L) | 895 | 105 (0) | 576 | 407 | 17 |
| LU (LE) | 797 | 143 (6) | 513 | 403 | 24 |
| DBp (L) | 878 | 169 (6) | 464 | 425 | 158 |
| Cl (L) | 709 | 36 (7) | 344 | 356 | 45 |
| Cl (LE) | 612 | 74 (5) | 328 | 314 | 44 |
| Fr (A) | 420 | 119 (0) | 180 | 180 | 179 |
| YA (A) | 154 | 146 (142) | 152 | 137 | 11 |

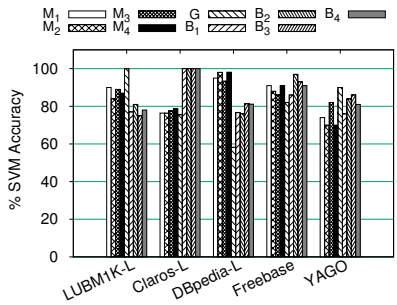
(a) Statistics about the generated test data.




(b) Accuracy with different classifiers.




(c) Comparison QSQ/MS.




(d) SVM Accuracy on atom subsets.

| Dataset (Ruleset) | Runtimes other methods Only QSQ | Only MS | Ideal | Total = | Our runtimes Feat. + | Pred. + | Reasoning |
|---|---|---|---|---|---|---|---|
| LU (L) | 31.4h | 13.2h | 10.3h | 11.5h | 49s | 10ms | 11.5h |
| LU (LE) | 65.0h | 39.0h | 36.0h | 37.4h | 55s | 22ms | 37.4h |
| DBp (L) | 131.1h | 82.2h | 19.5h | 33.6h | 262s | 100ms | 33.5h |
| Cl (L) | 184.3h | 379.6h | 67.4h | 75.8h | 137s | 60ms | 75.8h |
| Cl (LE) | 184.1h | 357.3h | 63.6h | 83.3h | 126s | 68ms | 83.3h |
| Fr (A) | 28.0h | 10.2h | 9.5h | 9.6h | 42s | 53ms | 9.6h |
| YA (A) | 162.2h | 236.4h | 26.5h | 31.1h | 34s | 34ms | 31.1h |

(e) Cumulative runtimes of our method vs. other approaches on the test goal atoms.

**Figure 1.** Statistics, accuracies, and runtimes test atoms.

execute SPARQL BGP queries with Datalog [29].

As SPARQL queries, we considered the 14 official queries provided by LUBM and manually created 10 additional SPARQL queries for DBpedia. For DBpedia, we created the queries manually because we wanted to ensure they trigger reasoning and that are ideal for both approaches (the queries are available in the repository).

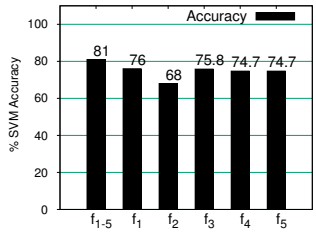
| Q. | LUBM1K (LE) Runtime (sec.) QSQ | MS | Ours | Pred | DBpedia (L) Runtime (sec.) QSQ | MS | Ours | Pred |
|---|---|---|---|---|---|---|---|---|
| 1 | 0.8 | 0.7 | 0.83 | N | TO | 90.3 | 90.6 | **Y** |
| 2 | 4.1 | 4.1 | 4.20 | **Y** | TO | 55.2 | 55.3 | **Y** |
| 3 | 2.6 | 10.2 | 2.61 | **Y** | 0.9 | 0.4 | 0.5 | **Y** |
| 4 | TO | 33.3 | 33.35 | **Y** | 33.3 | 3.8 | 4.0 | **Y** |
| 5 | TO | 32.8 | 32.82 | **Y** | 43.1 | 12.9 | 13.0 | **Y** |
| 6 | TO | 28.5 | 28.52 | **Y** | 0.4 | 1.3 | 1.4 | N |
| 7 | TO | 27.7 | 27.75 | **Y** | 33.4 | 457.5 | 33.5 | **Y** |
| 8 | TO | 28.8 | 28.82 | **Y** | 5.4 | 28.3 | 5.5 | **Y** |
| 9 | TO | 27.9 | 27.96 | **Y** | 11.2 | 36.3 | 11.4 | **Y** |
| 10 | TO | 29.6 | 29.64 | **Y** | 94.5 | 293.6 | 94.6 | **Y** |
| 11 | 6.4 | 28.6 | 6.48 | **Y** | | | | |
| 12 | TO | 29.2 | TO | N | | | | |
| 13 | TO | 35.4 | TO | N | | | | |
| 14 | 10.3 | 2.5 | 2.53 | **Y** | | | | |

**Table 3.** Performance of our approach with SPARQL queries. "QSQ" ("MS") is the runtime with QSQ (MS), TO: Timeout (10 minutes).

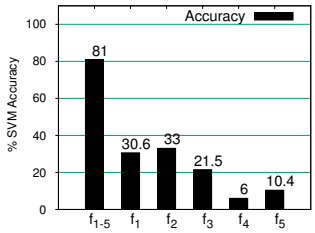
Table 3 reports the accuracy and runtimes of our method. From the table, we observe that we are able to pick the faster algorithm most of the times, and this results in a significant saving against the less ideal alternative. In four cases (marked with 'N' in the table), the model makes the wrong choice. In one case, the two algorithms return the same runtime (LUBM Query 2) and our approach is slightly slower due to the overhead of making the prediction. In general, however, we observe that the overhead necessary to extract the features and making the prediction is small. In several cases, our algorithm avoided to pick the slower algorithm which hit the timeout of ten minutes.

The results with SPARQL queries are representative for a possible practical use case, but may not cover all types of queries. To increase the coverage of our tests, we artificially created goal atoms. We pre-materialized each dataset $I$ with program $P$ and randomly picked facts from $P^1(I), P^2(I)$, etc. Then, we replaced some constants with variables, ensuring that the resulting goal atoms were not considered during training. We partitioned the atoms into three subsets: "G" contains all goal atoms with only variables; "B" contains all atoms with only constants while "M" contains mixed atoms. We further split two sets $X$ in $X_1, \ldots, X_4$ where $X \in \{B, M\}$ such that queries from $X_1$ have all answers in $P^1(I)$, $X_2$ in $P^2(I)$, $X_3$ in $P^3(I)$, and $X_4$ in $P^{\geq 4}(I)$. The rationale was to separate atoms with answers that require more or less inference steps. Details with the number of goal atoms labeled with each method, and cardinalities of the various subsets are reported in Table 1a.

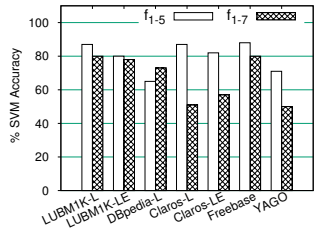
Figure 1b depicts the prediction accuracy obtained with the four models. SVM emerges as the best choice as it returns an accuracy that is close or greater than 80% for all but one dataset. To gain more insights, Figure 1c shows the accuracy of SVM either only with atoms labeled with QSQ or MS. We observe that SVM is generally accurate in predicting both types of atoms. While some fluctuations in the accuracy should be expected due to the stochastic nature of the process, with LUBM1K (LE) and DBpedia (L) the difference is more prominent. With LUBM1K (LE), we observed that the runtime difference between choosing QSQ and MS is not high. Therefore, similar atoms can be classified with different labels. With DBpedia (L), we noticed that the difference between the values of the features of QSQ and MS atoms is much smaller than with the other inputs and this makes the prediction more challenging.




(a) Feature Ablation Study.




(b) Feature Inclusion Study.




(c) Accuracy also with $f_6, f_7$.

| Dataset (Ruleset) | $\lvert f_7 \rvert$ |
|---|---|
| LUBM1K (L) | 73 |
| LUBM1K (LE) | 76 |
| DBpedia (L) | 2137 |
| Claros (L) | 527 |
| Claros (LE) | 532 |
| Freebase (AMIE) | 644 |
| YAGO (AMIE) | 35 |

(d) Extra features with $f_7$.

**Figure 2.** Feature ablation and inclusion study and accuracy with extra features.

The accuracy of SVM with different subsets of goal atoms is shown in Figure 1d. The accuracy is often above 80% and it never decreases below 70% with the exception of DBpedia (G). This indicates that the model is accurate also if the answers require multiple inference steps and/or with atoms that have more or less constants.

Table 1e reports the cumulative runtimes necessary to execute all the atomic test queries with various configurations. The second column reports the runtime if we only use QSQ. Analogously, the third column reports the runtime if we only use MS. The fourth column reports the (unrealistic) best runtime we could achieve if we always pick the best algorithm. This last runtime represents the best possible scenario and it is useful for evaluating the margin for improvement. The fifth column reports the total runtime produced by our method (SVM). This runtime is the sum of the time taken for generating the features (sixth column), making the prediction (seventh column), and of reasoning, i.e., executing either QSQ or MS (eighth column). Note that the first two runtimes are much smaller than the reasoning runtime, and essentially all the runtime is spent in reasoning.

From Table 1e, we observe that our approach is better than using a fixed strategy (either MS or QSQ). The improvement is particular significant with YAGO (ours is 31.1h vs. 162.2h of QSQ) and with DBPedia (ours is 33.5h vs. 82.2h of MS). Moreover, we observe that the runtime of our method is closed to the ideal one with LUBM1K and Freebase, which indicates that we are close to the maximum gain that we can obtain. It is important to note, however, that with LUBM1K and Freebase a fixed strategy of always choosing MS would perform well. The reason is that with these inputs the runtime difference between QSQ and MS is relatively small and in these cases choosing the wrong algorithm does not impact the performance. With the other inputs, choosing the wrong algorithm might lead to much longer runtimes (up to a few hours in the case of YAGO). It is in such cases that our technique is most effective.

Finally, Figure 2a reports the results on a feature ablation study, which is a procedure that is commonly applied to understand to what extent the features contribute for improving the accuracy. The first column reports the average accuracy obtained on all inputs when we use all five features, while the remaining five columns report the accuracy if we remove one feature (e.g., the second column reports the accuracy if we exclude $f_1$). As we can see from the figure, all five features contribute to increase the accuracy and the best value is obtained when we include them all. Figure 2b reports the results of a similar experiment where we measured instead the accuracy if we use only one feature at the time. For instance, the second column reports the accuracy if we use only $f_1$. From this experiment, we see that there is no feature that can return an accuracy more than 50%. From the results shown in these two figures, we conclude that combining all five features is crucial to obtain a high prediction accuracy.

Figure 2c reports the accuracy if we also include the set of features $f_6$ and $f_7$. We observe that these additional features decrease the accuracy on all datasets except DBpedia where most of the goal atoms with the same predicate are labeled with the same algorithm and the binary features in $f_7$ can capture this more effectively. In addition to producing a lower accuracy, features $f_6$ and $f_7$ also increase significantly the size of the model (Table 2d reports the number of extra binary features if we include $f_7$). For these two reasons, we decided to exclude them and use only features $f_1, \ldots, f_5$.

## 7 Conclusion

In order to exploit KGs to the fullest, it is important to implement efficient query answering procedures. Existing datalog query-driven procedures are ideal for this task, but there are cases when one technique is better than another and detecting this beforehand is not easy.

To improve the efficiency of query answering, this paper proposes a method to perform an automatic algorithm selection for query-driven procedures. More in particular, our method uses a ML binary classifier to choose (or help the user in choosing) whether the query at hand should be answered with QSQ (non-rewriting procedure) or MS (rewriting procedure). Using a ML classifier for this task is not straightforward, and we had to address the challenges of defining a way to (quickly) translate each query into numerical features and to obtain queries for training the model.

So far, algorithm selection has been successfully applied in other fields, like SAT solvers, but never for datalog query-driven reasoning. Our experiments show that this is an effective type of optimization as the predictions resulted in significant reductions of the response times; up to 80% in the best case.

Our work builds from the intuition that the insights extracted from past evidence can improve significantly the efficiency of query-driven reasoning. It is interesting, as a direction for future research, to study whether our approach can be extended to other reasoning techniques as well (e.g., incremental materialization procedures). Moreover, it is also interesting to further research how statistical methods can be integrated more deeply into the reasoning process (e.g., to optimize the join ordering or the selection of subqueries), or how they can provide more fine-grained estimations of the runtime or of the potential number of answers.

**Acknowledgments.** This project was partly funded by the NWO research programme 400.17.605 (VWData) and NWO VENI project 639.021.335.

## REFERENCES


[1] Serge Abiteboul, Richard Hull, and Victor Vianu, *Foundations of databases*, Addison Wesley, 1995.

[2] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn, 'Design and Implementation of the LogicBlox System', in *Proceedings of SIGMOD*, pp. 1371–1382, (2015).

[3] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman, 'Magic Sets and Other Strange Ways to Implement Logic Programs (Extended Abstract)', in *Proceedings of SIGMOD*, pp. 1–15, (1986).

[4] Hamid R. Bazoobandi, Jacopo Urbani, Frank van Harmelen, and Henri Bal, 'An Empirical Study on How the Distribution of Ontologies Affects Reasoning on the Web', in *Proceedings of ISWC*, pp. 69–86, (2017).

[5] Luigi Bellomarini, Emanuel Sallinger, and Georg Gottlob, 'The Vadalog System: Datalog-based Reasoning for Knowledge Graphs', *Proceedings of VLDB*, **11**(9), 975–987, (2018).

[6] Barry Bishop, Atanas Kiryakov, Damyan Ognyanoff, Ivan Peikov, Zdravko Tashev, and Ruslan Velkov, 'OWLIM: A family of scalable semantic repositories', *Semantic Web*, **2**(1), 33–42, (2011).

[7] Christopher M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2006.

[8] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellman, 'DBpedia - a crystallization point for the Web of Data', *Web Semantics*, **7**(3), 154–165, (2009).

[9] Brian Chin, Daniel von Dincklage, Vuk Ercegovac, Peter Hawkins, Mark S. Miller, Franz Och, Christopher Olston, and Fernando Pereira, 'Yedalog: Exploring Knowledge at Scale', in *SNAPL*, volume 32, pp. 63–78, (2015).

[10] Corinna Cortes and Vladimir Vapnik, 'Support-vector networks', *Machine learning*, **20**(3), 273–297, (1995).

[11] Luis Galárraga, Christina Teflioudi, Katja Hose, and Fabian M Suchanek, 'AMIE : Association Rule Mining under Incomplete Evidence in Ontological Knowledge Base', *Proceedings of WWW*, 413–422, (2013).

[12] Bernardo Cuenca Grau, Ian Horrocks, Boris Motik, Bijan Parsia, Peter Patel-Schneider, and Ulrike Sattler, 'OWL 2 : The next step for OWL', *Web Semantics*, **6**(4), 309–322, (2008).

[13] Y. Guo, Z. Pan, and J. Hefflin, 'LUBM: A benchmark for OWL knowledge based systems', *Web Semantics*, **3**(2), 158–182, (2005).

[14] Ashish Gupta, Dinesh Katiyar, and Inderpal Singh Mumick, 'Counting solutions to the view maintenance problem.', in *JICSLP*, pp. 185–194, (1992).

[15] Ashish Gupta, Inderpal Singh Mumick, and Venkatramanan Siva Subrahmanian, 'Maintaining views incrementally', *ACM SIGMOD*, **22**(2), 157–166, (1993).

[16] Patrick Hayes. RDF Semantics, 2004. W3C.

[17] J. Hoffart, F. Suchanek, K. Berberich, and G. Weikum, 'YAGO2: A spatially and temporally enhanced knowledge base from wikipedia', *Artificial Intelligence*, **194**, 28–61, (2013).

[18] Ian Horrocks, Mark Kaminski, Bernardo Cuenca Grau, Egor V. Kostylev, and Boris Motik, 'Foundations of Declarative Data Analysis Using Limit Datalog Programs', in *Proceedings of IJCAI*, pp. 1123–1130, (2017).

[19] Pan Hu, Boris Motik, and Ian Horrocks, 'Optimised Maintenance of Datalog Materialisations', in *Proceedings of AAAI*, pp. 1871–1879, (2018).

[20] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper, 'Learned Cardinalities: Estimating Correlated Joins with Deep Learning', in *Proceedings of CIDR*, (2019).

[21] Nikolaos Konstantinou, Martin Koehler, Edward Abel, Cristina Civili, Bernd Neumayr, Emanuel Sallinger, Alvaro A.A. Fernandes, Georg Gottlob, John A. Keane, Leonid Libkin, and Norman W. Paton, 'The VADA Architecture for Cost-Effective Data Wrangling', in *Proceedings of SIGMOD*, pp. 1599–1602, (2017).

[22] Lars Kotthoff, Ian P Gent, and Ian Miguel, 'An evaluation of machine learning in algorithm selection for search problems', *AI Communications*, **25**(3), 257–270, (2012).

[23] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis, 'The Case for Learned Index Structures', in *Proceedings of SIGMOD*, pp. 489–504, (2018).

[24] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica, 'Learning to Optimize Join Queries With Deep Reinforcement Learning', *CoRR*, **abs/1808.03196**, (2018).

[25] Alessandro Margara, Jacopo Urbani, Frank Van Harmelen, and Henri Bal, 'Streaming the web: Reasoning over dynamic data', *Web Semantics*, **25**, 24–44, (2014).

[26] Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks, 'Incremental Update of Datalog Materialisation: The Backward/Forward Algorithm', in *Proceedings of AAAI*, pp. 1560–1568, (2015).

[27] Boris Motik, Yavor Nenov, Robert Piro, Ian Horrocks, and Dan Olteanu, 'Parallel Materialisation of Datalog Programs in Centralised, Main-Memory RDF Systems', in *Proceedings of AAAI*, pp. 129–137, (2014).

[28] Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee, 'RDFox: A Highly-Scalable RDF Store', in *Proceedings of ISWC*, pp. 3–20, (2015).

[29] Axel Polleres, 'How (well) do datalog, sparql and rif interplay?', in *International Datalog 2.0 Workshop*, pp. 27–30. Springer, (2012).

[30] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF, 2008. W3C.

[31] John R Rice, 'The algorithm selection problem', *Advances in computers*, **15**, 65–118, (1976).

[32] Jacopo Urbani, Ceriel Jacobs, and Markus Krötzsch, 'Column-Oriented Datalog Materialization for Large Knowledge Graphs', in *Proceedings of AAAI*, pp. 258–264, (2016).

[33] Jacopo Urbani, Spyros Kotoulas, Jason Maassen, Frank Van Harmelen, and Henri Bal, 'WebPIE: A Web-scale Parallel Inference Engine using MapReduce', *Web Semantics*, **10**, 59–75, (2012).

[34] Jacopo Urbani, Robert Piro, Frank van Harmelen, and Henri Bal, 'Hybrid reasoning on OWL RL', *Semantic Web*, **5**(6), 423–447, (2014).

[35] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang, 'Automatic Database Management System Tuning Through Large-scale Machine Learning', in *Proceedings of SIGMOD*, pp. 1009–1024, (2017).

[36] Maarten H Van Emden and Robert A Kowalski, 'The semantics of predicate logic as a programming language', *JACM*, **23**(4), 733–742, (1976).

[37] Zhe Wu, George Eadon, Souripriya Das, Eugene Inseok Chong, Vladimir Kolovski, Melliyal Annamalai, and Jagannathan Srinivasan, 'Implementing an Inference Engine for RDFS/OWL Constructs and user-defined Rules in Oracle', in *Proceedings of ICDE*, pp. 1239–1248, (2008).

[38] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown, 'Satzilla: portfolio-based algorithm selection for sat', *Journal of artificial intelligence research*, **32**, 565–606, (2008).