Evaluating Negation with Multi-way Joins Accelerates Class Expression Learning

Nikolaos Karalis[0000−0002−0710−7180](✉), Alexander Bigerl[0000−0002−9617−1466], Caglar Demir[0000−0001−8970−3850], Liss Heidrich[0009−0006−2031−2548], and Axel-Cyrille Ngonga Ngomo[0000−0001−7112−3516]

DICE group, Department of Computer Science, Paderborn University, Germany
{nikolaos.karalis, alexander.bigerl, caglar.demir, liss.heidrich, axel.ngonga}@uni-paderborn.de

Abstract. Class expression learning based on refinement operators is a popular family of explainable machine learning approaches for RDF knowledge graphs with ontologies in description logics. However, most implementations of this paradigm fail to scale to the large knowledge graphs found on the Web. One common bottleneck of these implementations is the instance retrieval function. We address this drawback by introducing an algorithm inspired by worst-case optimal multi-way joins for the evaluation of SPARQL queries that correspond to $\mathcal{ALC}$ class expressions. The main characteristic of our algorithm is the inclusion of negation, which is prominent in SPARQL queries generated from $\mathcal{ALC}$ class expressions, in multi-way join plans. We evaluate the implementation of our approach on five benchmark datasets against four state-of-the-art graph storage solutions for RDF knowledge graphs. The results of our extensive evaluation show that our approach outperforms its competition across all datasets and that it is the only one able to scale to large datasets. With our approach, we enable learning algorithms to retrieve information from Web-scale knowledge graphs, hence making ante-hoc explainable machine learning easier to deploy on the Semantic Web.

Keywords: knowledge graphs · class expression learning · multi-way joins

1 Introduction

RDF knowledge graphs are now first-class citizens of the Web. Over 80 billion RDF assertions are found in the 2021 crawl of the Web Data Commons.[1] RDF data dumps on the Web cover a similar order of magnitude.[2] Learning on RDF data at this scale is hence crucial for the deployment of machine learning on the Web—the world's largest shared information source with over 5 billion users. The large proportion of the human population impacted by machine learning on

[1] http://webdatacommons.org/structureddata/\#results-2021-1
[2] http://lod-a-lot.lod.labs.vu.nl/

the Web entails the need for explainable machine learning because of its known societal advantages [8].

A popular family of ante-hoc explainable approaches for supervised learning on RDF knowledge graphs are class expression learning algorithms based on refinement operators [10, 18, 23]. Given a set of positive and negative examples, these approaches generate a class expression in a predefined description logic, which ideally describes the positive and not the negative examples. However, a large body of literature on learning class expressions [13, 17, 23] suggests that these approaches do not scale to the size of datasets found on the Web. This is mostly due to their instance retrieval function–which computes the instances of given class expressions [10, 18, 23]–being unable to retrieve instances in a time-efficient manner [18, 23]. The authors of [7] suggest that this weakness can be addressed by converting class expressions into SPARQL queries.

SPARQL[3] is the designated language for querying RDF knowledge graphs. A recent advancement in querying processing is the introduction of worst-case optimal multi-way join algorithms [19]. Worst-case optimal multi-way join algorithms have been adopted by the semantic web community [15] and are now being used by state-of-the-art knowledge graph storage solutions (e.g., triple stores) for the efficient evaluation of SPARQL queries [2, 5]. However, the use of multi-way joins for SPARQL is mostly limited to conjunctive queries. However, as demonstrated in Section 3, class expression learning with SPARQL does not only deal with conjunctive queries; in particular, it requires the efficient evaluation of queries containing *negation*.

The hypothesis behind this work is that we can exploit multi-way joins to implement a time-efficient retrieval function for class expression learning. To this end, as in previous works (e.g., [13, 17]), we set our focus on the description logic $\mathcal{ALC}$ and present a multi-way join algorithm for the efficient evaluation of SPARQL queries generated by $\mathcal{ALC}$ class expressions. Such SPARQL queries include union graph patterns and negation, which is captured by `FILTER NOT EXISTS` patterns. To the best of our knowledge, there have not been any works for SPARQL that consider the evaluation of negation in multi-way join plans.

The main contributions of this work are the following: (i) Inspired by worst-case-optimal multi-way join algorithms and their efficiency in evaluating conjunctive queries, we present a multi-way join algorithm for the evaluation of SPARQL queries generated from $\mathcal{ALC}$ class expressions. Our approach relies on theoretical foundations of SPARQL to enable the use of multi-way joins in queries involving negation. (ii) We have identified an issue with the mapping of class expressions to SPARQL queries presented in [7] and present a solution that addresses this particular issue. (iii) We have implemented the proposed algorithm in a state-of-the-art triple store and present the results of an extensive comparison of our implementation on multiple benchmark datasets of varying sizes with multiple state-of-the-art triple stores using SPARQL queries that correspond to $\mathcal{ALC}$ class expressions. In our evaluation, we use four datasets, ranging from 96K to 2.1M triples, that are commonly used to evaluate learning algorithms.

[3] `https://www.w3.org/TR/2013/REC-sparql11-query-20130321/`

To evaluate the scalabilty of our approach, we also use a dataset consisting of more than 40M triples. The results of our experimental evaluation show that our approach outperforms its competition across all datasets and that it is the only one that efficiently handles query workloads in the largest dataset.

The rest of this work is structured as follows: In Section 2, we provide background knowledge on the topics that are covered in this paper. In Section 3, we cover the translation of $\mathcal{ALC}$ class expressions to SPARQL queries. We present the proposed multi-way join algorithm for SPARQL queries generated from $\mathcal{ALC}$ class expressions in Section 4. Our experimental results are presented in Section 5. We discuss related works in Section 6 and conclude in Section 7.

2 Preliminaries

We begin first with a brief overview of the description logic $\mathcal{ALC}$. Second, we describe the problem of class expression learning. Third, we cover definitions and properties of SPARQL queries that are required in this work. Last, we briefly summarize worst-case optimal multi-way join algorithms.

2.1 The Description Logic $\mathcal{ALC}$

A description logic is a decidable fragment of first-order predicate logic that uses only unary and binary predicates [4]. The set of unary predicates, binary predicates and constants correspond to the set of named concepts N_C, roles N_R, and individuals N_I of a description logic, respectively. Like in recent works on class expression learning (e.g., [13, 17]), we focus on the description logic $\mathcal{ALC}$ [25]. Its syntax and semantics are provided in Table 1. In this paper, $\mathcal{C}$ denotes all valid $\mathcal{ALC}$ concepts C under the construction rules: $C ::= A \mid \neg C \mid C \sqcap C \mid C \sqcup C \mid \exists r.C \mid \forall r.C$, where $A \in N_C$ and $r \in N_R$.

Knowledge bases in $\mathcal{ALC}$ are often defined as $\mathcal{K} = (\mathit{Tbox}, \mathit{Abox})$. All axioms in Tbox are of the form $A \sqsubseteq B$ or $A \equiv B$. Abox contains the relationships between individuals $a, b \in N_I$ via roles $r \in N_R$ and membership relationships between N_I and $\mathcal{C}$. Following previous works on class expression learning [13, 17, 18], we adopt closed world semantics. Under the closed world assumption, the ABox of a knowledge base is treated as the knowledge base's model $\mathcal{I}$ [18]. In addition, under closed world semantics, checking whether an instance belongs to a particular concept or retrieving the individuals of a particular concept is similar to querying in classical databases [7, 14].

2.2 Class Expression Learning

Class expression learning is a family of supervised machine learning algorithms. Given a knowledge base $\mathcal{K} = (\mathit{Tbox}, \mathit{Abox})$, a set of positive examples E^+, and a set of negative examples E^-, with $E^+ \cup E^- \subseteq N_I$ and $E^+ \cap E^- = \emptyset$, class expression learning algorithms aim to learn a class expression H that ideally describes all of the individuals of E^+ and not any of the individuals of E^-

Table 1. Syntax and semantics of $\mathcal{ALC}$. $\mathcal{I}$ denotes an interpretation and $\Delta^{\mathcal{I}}$ its domain.

Construct	Syntax	Semantics
Top concept	$\top$	$\Delta^{\mathcal{I}}$
Bottom concept	$\bot$	$\emptyset$
Atomic concept	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
Role	r	$r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
Conjunction	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
Disjunction	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
Negation	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
Existential restriction	$\exists\, r.C$	$\{x \mid \exists\, (x^{\mathcal{I}}, y^{\mathcal{I}}) \in r^{\mathcal{I}} \wedge y^{\mathcal{I}} \in C^{\mathcal{I}}\}$
Universal restriction	$\forall\, r.C$	$\{x \mid \forall\, (x^{\mathcal{I}}, y^{\mathcal{I}}) \in r^{\mathcal{I}} \implies y^{\mathcal{I}} \in C^{\mathcal{I}}\}$

[18]. To find the most appropriate H for a particular pair of E^+ and E^-, a class expression learning problem is often transformed into a search problem within a quasi-ordered space $(\mathcal{C}, \preceq)$ [11, 18, 31], where $\preceq$ is often the subsumption relation $\sqsubseteq$ between concepts [17]. The traversal of the search problem's space is usually conducted using a downward refinement operator $\rho : \mathcal{C} \rightarrow 2^{\mathcal{C}}$ such that $\rho(C) \subseteq \{C' \in \mathcal{C} \mid C' \sqsubseteq C, C' \neq C\}$ for all $C \in \mathcal{C}$ [17].

The quality of a class expression H (i.e., finding whether H describes the individuals of E^+ and not the individuals of E^-) is commonly computed by a heuristic function, such as CELOE [18]. Internally, such heuristic functions use a retrieval function $\mathcal{R}: \mathcal{C} \rightarrow 2^{N_I}$, which returns all individuals in N_I that are instances of the provided class expression H. As discussed in Section 2.1, we assume that retrieval operations are carried out under closed world semantics. As the size of an input knowledge base grows, executing retrieval operations against reasoners becomes one of the main computational bottlenecks [7, 18, 23] of learning algorithms. The goal of this work is to accelerate class expression learning by reducing the runtimes of retrieval operations.

2.3 Semantics and Properties of SPARQL

Here, we provide the semantics and the properties of SPARQL queries that use those features of the language that are used in queries generated by $\mathcal{ALC}$ class expressions (Section 3). In particular, our focus is on SPARQL queries consisting of *triple patterns*, *basic graph patterns*, *union graph patterns*, and *negation* in the form of `FILTER NOT EXISTS` patterns. Note that SPARQL queries are defined under bag semantics. In this work, we adopt set semantics (i.e., queries contain `DISTINCT`), which are in line with the semantics of $\mathcal{ALC}$. Let $\mathbf{I}$ be an infinite set of IRIs, $\mathbf{B}$ an infinite set of blank nodes, and $\mathbf{L}$ an infinite set of literals. The sets $\mathbf{I}$, $\mathbf{B}$ and $\mathbf{L}$ are pairwise disjoint and their union, denoted as $\mathbf{T} = \mathbf{I} \cup \mathbf{B} \cup \mathbf{L}$, is the set of all terms. Furthermore, let $\mathbf{V}$ be an infinite set of variables. An RDF graph is a set of triples and is defined as $G = \{(s, p, o) \mid s \in (\mathbf{I} \cup \mathbf{B}), p \in \mathbf{I}, o \in \mathbf{T}\}$, where s, p, and o stand for *subject*, *predicate*, and *object*, respectively.

Triple, Basic and Union Graph Patterns The following is based on [15, 21]. A triple pattern *tp* is a triple (s, p, o), where $s \in (\mathbf{I} \cup \mathbf{L} \cup \mathbf{V})$, $p \in (\mathbf{I} \cup \mathbf{V})$, and $o \in (\mathbf{I} \cup \mathbf{L} \cup \mathbf{V})$. The set of variables of a triple pattern is denoted as $var(tp)$. Since blank nodes behave as variables, we do not consider them in triple patterns [15, 21]. A basic graph pattern (BGP) is a set of triple patterns. The semantics of SPARQL queries are defined using mappings, which are partial functions assigning terms to variables. A mapping is formally defined as $\mu : \mathbf{V} \to \mathbf{T}$ and its domain is denoted as $dom(\mu)$. With $\mu(tp)$, we denote the RDF triple obtained by replacing the variables of $var(tp)$ with their corresponding values in μ. Two mappings μ_1 and μ_2 are compatible ($\mu_1 \sim \mu_2$), iff $\mu_1(?v) = \mu_2(?v)$ for every variable $?v \in dom(\mu_1) \cap dom(\mu_2)$. Two mappings μ_1 and μ_2 are not compatible ($\mu_1 \nsim \mu_2$), if for any $?v \in dom(\mu_1) \cap dom(\mu_2)$, $\mu_1(?v) \neq \mu_2(?v)$. Given two sets of mappings Ω_1 and Ω_2 the join operation is defined as $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ and } \mu_1 \sim \mu_2\}$ and the union operation is defined as $\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\}$. The evaluation of a triple pattern *tp* and a BGP P over an RDF graph G is denoted as $[\![tp]\!]_G = \{\mu \mid dom(\mu) = var(tp) \text{ and } \mu(tp) \in G\}$ and $[\![P]\!]_G = [\![tp_1, \ldots, tp_n]\!]_G = [\![tp_1]\!]_G \bowtie \ldots \bowtie [\![tp_n]\!]_G$, respectively. In SPARQL, graph patterns are constructed recursively. Triple patterns and BGPs are graph patterns. The set of variables of a graph pattern P is denoted as $var(P)$. The conjunction and union between two graph patterns P_1 and P_2 are also graph patterns and are evaluated over G as $[\![P_1 \ AND \ P_2]\!]_G = [\![P_1]\!]_G \bowtie [\![P_2]\!]_G$ and $[\![P_1 \ UNION \ P_2]\!]_G = [\![P_1]\!]_G \cup [\![P_2]\!]_G$, respectively.

Negation The following definitions are based on [1]. In SPARQL, there are multiple ways to express negation [1]. Here, we focus on patterns of negation that are expressed using `FILTER NOT EXISTS` patterns, which are used for converting $\mathcal{ALC}$ class expressions to SPARQL queries (Section 3). `FILTER NOT EXISTS` can be provided with either a subquery or a graph pattern. In this work, we are interested only in the latter case (Section 3). Given a graph pattern $P = (P_1 \ FNE \ P_2)$, where *FNE* stands for `FILTER NOT EXISTS`, the set of correlated variables $corVars(P)$ is defined as $corVars(P) = var(P_1) \cap var(P_2)$; P_1 and P_2 are correlated if $corVars(P) \neq \emptyset$. In this work, we focus only on queries for which P_1 and P_2 are always correlated (Section 3). If P_1 and P_2 are correlated, P_1 is evaluated before P_2 and P_2 is evaluated after each correlated variable in P_2 is replaced with its corresponding value obtained in the evaluation of P_1 [1]. The set of certain variables of a graph pattern P, denoted as $cVars(P)$, consists of the variables that are always bound in the solution mapping of P [24]. The set of certain variables of a graph pattern P is defined recursively [1, 24]: (i) if P is a triple pattern *tp*, then $cVars(P) = var(tp)$, (ii) if $P = (P_1 \ AND \ P_2)$, then $cVars(P) = cVars(P_1) \cup cVars(P_2)$, (iii) if $P = (P_1 \ UNION \ P_2)$, then $cVars(P) = cVars(P_1) \cap cVars(P_2)$, (iv) if $P = (P_1 \ FILTER \ R)$, then $cVars(P) = cVars(P_1)$ with R being a built-in condition (e.g., integer addition), and (v) if $P = (P_1 \ FNE \ P_2)$, then $cVars(P) = cVars(P_1)$. A graph pattern is *fne-safe*, if for every subpattern $P = (P_1 \ FNE \ P_2)$ holds that $corVars(P) \subseteq cVars(P_2)$. Given two sets of mappings Ω_1 and Ω_2 the difference operation is defined as

Algorithm 1 Generic Join

```
// Execution is resumed after a yield operation
function GenericJoin(P, G, X)          ▷ P: BGP, G: RDF Graph, X: Mapping
    if var(P) = ∅ then yield X and return          ▷ All variables are evaluated
    ?x ← a variable from var(P)                    ▷ Select a variable to be evaluated
    K ← ⋂_{tp∈P|?x∈var(tp)} {μ(?x) | μ ∈ [[tp]]_G}          ▷ All possible values of ?x
    for all k ∈ K do                   ▷ Iterate over the possible values of ?x
        X(?x) ← k                                  ▷ Store k in the solution mapping
        P' ← assign k to all occurrences of ?x in P
        yield all GenericJoin(P', G, X)   ▷ proceeds with the next k afterwards
```

$\Omega_1 \setminus \Omega_2 = \{\mu_1 \in \Omega_1 \mid \forall \mu_2 \in \Omega_2, \mu_1 \nsim \mu_2\}$. For fne-safe patterns, the negation is defined as $[\![P_1 \ \mathit{FNE}\ P_2]\!]_G = [\![P_1]\!]_G \setminus [\![P_2]\!]_G$. The notion of fne-safety ensures that nested *FNE* patterns can be evaluated using the difference operation [1].

Filter Rewriting Rule and Union Normal Form A graph pattern $((P_1\ \mathit{FILTER}\ R)\ \mathit{AND}\ P_2)$ can be rewritten to a pattern $((P_1\ \mathit{AND}\ P_2)\ \mathit{FILTER}\ R)$, if every variable of R is also a certain variable of P_1 or if R and P_2 do not share any variables [24]. A graph pattern P is in *UNION normal form* if it is in the form $(P_1\ \mathit{UNION}\ P_2\ \mathit{UNION}\ \ldots\ \mathit{UNION}\ P_n)$ and each P_i, for $1 \leq i \leq n$, is UNION-free [21]. Every *AND-UNION-FILTER* graph pattern P is equivalent to a graph pattern P', which is in UNION normal form [21].

2.4 Worst-case Optimal Multi-way Join Algorithms

In recent years, worst-case optimal multi-way join algorithms [19] have been the subject of many research works in the database literature. This is due to their runtime complexity being bounded by the worst-case size of the result of the input query [3] and their ability to achieve state-of-the-art performance in the evaluation of conjunctive queries (e.g., [2, 5, 15]). Their main characteristic is that, contrary to conventional binary joins that carry out joins on two operands at a time, their evaluation follows a variable elimination process. This evaluation process resembles a backtracking search, does not store any intermediate results and enables the incremental output of solution mappings. A worst-case optimal multi-way join algorithm for the evaluation of BGPs (conjunctive queries) based on Generic Join [20] is shown in Algorithm 1.

3 Mapping $\mathcal{ALC}$ Class Expressions to SPARQL Queries

As discussed in Section 2.2, one of the main computational bottlenecks for class expression learning algorithms is the execution of retrieval operations against reasoners. To alleviate this issue, Bin et al. [7] propose a mapping for the conversion of class expressions to SPARQL queries. This mapping is shown in the first six entries of Table 2—including the struck through text. These conversions

Table 2. The mapping of $\mathcal{ALC}$ expressions to SPARQL queries. The struck through entry is the erroneous mapping for $\forall r.C$ class expressions proposed in [7]. Our proposed mapping for $\forall r.C$ class expressions is shown in the table's last entry.

$\mathcal{ALC}$ Class Expression C_i	SPARQL Graph Pattern $\tau(C_i, ?var)$
A	{ $?var$ `rdf:type` A . }
$\neg C$	{ $?var$ $?p$ $?o$. `FILTER NOT EXISTS` { $\tau(C, ?var)$ } }
$C_1 \sqcap \cdots \sqcap C_n$	{ $\tau(C_1, ?var)$. $\tau(C_2, ?var)$ . ... . $\tau(C_n, ?var)$ }
$C_1 \sqcup \cdots \sqcup C_n$	{ { $\tau(C_1, ?var)$ } `UNION` ... `UNION` { $\tau(C_n, ?var)$ } }
$\exists r.C$	{ $?var$ r $?s$. $\tau(C, ?s)$ }
~~$\forall r.C$~~	~~{ $?var$ r $?s0$. { `SELECT` $?var$ (`COUNT`($?s1$) `AS` $?c1$) `WHERE` { $?var$ r $?s1$. $\tau(C, ?vs1)$ } `GROUP BY` $?var$ }~~ ~~{ `SELECT` $?var$ (`COUNT`($?s2$) `AS` $?c2$) `WHERE` { $?var$ r $?s2$ } `GROUP BY` $?var$ } `FILTER`($?c1 = ?c2$) }~~
$\forall r.C$	{ $?var$ $?p$ $?o$. `FILTER NOT EXISTS` { $?var$ r $?s$. `FILTER NOT EXISTS` { $\tau(C, ?s)$ } } }

enable learning algorithms to carry out retrieval operations against a triple store instead of a reasoner.

During the development of our multi-way join algorithm for class expression learning (Section 4), we identified an issue with the mapping presented in [7]. In particular, the SPARQL queries corresponding to class expressions of the type $\forall r.C$ did not return the expected results. As per the semantics of $\mathcal{ALC}$, the set of instances corresponding to a class expression $\forall r.C$ should also contain those individuals that do not have any r-successors [22, Remark 18]. For example, assuming a knowledge base capturing the concepts of family members, the set of individuals corresponding to the concept $\forall \mathit{hasChild}.\mathit{Male}$ should also contain those individuals that do not have any children. However, this was not the case with the SPARQL queries corresponding to such class expressions, as they did not take individuals not having any r-successors into account [7, Section 3 of the corresponding technical report]. To alleviate this issue we propose a new mapping for $\forall r.C$ expressions, which is presented in the last entry of Table 2 and replaces the struck through rule. The proposed graph pattern for $\forall r.C$ class expressions is constructed using the concept equivalence rule $\forall r.C \equiv \neg\exists r.\neg C$ [22, Section 5.1]. For individuals that do not have any r-successors, the first `FILTER NOT EXISTS` is always evaluated to true. For individuals that have at least one r-successor, the first `FILTER NOT EXISTS` is evaluated to true, if all of their r-successors belong to C, which is tested by the seconds `FILTER NOT EXISTS`. Note that applying a combination of the transformation rules corresponding to $\neg C$ and $\exists r.C$ would yield a query that is semantically equivalent to the updated query.

The SPARQL queries shown in Table 2 use only those features of SPARQL that were discussed in Section 2.3. Furthermore, all of the queries that involve patterns of negation (`FILTER NOT EXISTS`) are fne-safe and hence, can be evalu-

ated using the difference operator for sets of mappings. To prove that the newly proposed mapping for $\forall r.C$ follows the semantics of $\mathcal{ALC}$, the existing proofs for $\neg C$ and $\exists r.C$, which are provided in the technical report of [7], can be used.

4 Negation in Multi-way Joins

The efficiency of worst-case optimal multi-way join algorithms in evaluating basic graph pattern queries has been demonstrated in recent works (e.g., [2, 5, 15]). However, to the best of our knowledge, there have not been any efforts for SPARQL that include patterns of negation in multi-way join plans. In this section, we present our algorithm for the efficient evaluation of SPARQL queries corresponding to $\mathcal{ALC}$ class expressions. Our algorithm follows the sketch provided in [28] for the evaluation of Datalog rules containing negation and incorporates the evaluation of `FILTER NOT EXISTS` patterns in multi-way join plans. The two main ideas behind the proposed algorithm are the following. First, the evaluation of union graph patterns should take advantage of multi-way joins and their efficiency in evaluating basic graph patterns [16]. To this end, we rely on the definitions provided in Section 2.3 to rewrite each query generated by $\mathcal{ALC}$ class expressions to a semantically equivalent query that is in UNION normal form. Second, instead of waiting for a graph pattern to be fully evaluated, `FILTER NOT EXISTS` patterns should be evaluated as soon as their correlated variables (Section 2.3) are bound to a particular term.

4.1 Rewriting Rule for Negation and UNION Normal Form

As discussed in Section 2.3, a graph pattern $((P_1\ \mathit{FILTER}\ R)\ \ \mathit{AND}\ P_2)$ can be rewritten to a semantically equivalent graph pattern $((P_1\ \mathit{AND}\ P_2)\ \mathit{FILTER}\ R)$, if every variable of R is also a certain variable of P_1 or R and P_2 do not share any variables [24]. The above rewriting rule is also applicable to `FILTER NOT EXISTS` patterns. Note that the SPARQL standard treats `FILTER NOT EXISTS` patterns as filter expressions.

Proposition 1. *Let $P = ((P_1\ \mathit{FNE}\ P_3)\ \ \mathit{AND}\ P_2)$ and $P' = ((P_1\ \mathit{AND}\ P_2)\ \mathit{FNE}\ P_3)$ be fne-safe graph patterns. P and P' are semantically equivalent, if $\mathit{corVars}(P_1\ \mathit{FNE}\ P_3) = \mathit{corVars}((P_1\ \mathit{AND}\ P_2)\ \mathit{FNE}\ P_3)$.*

Proof. For $[\![P]\!]_G = [\![P']\!]_G$, we need $(([\![P_1]\!]_G \setminus [\![P_3]\!]_G) \bowtie [\![P_2]\!]_G) = (([\![P_1]\!]_G \bowtie [\![P_2]\!]_G) \setminus [\![P_3]\!]_G)$. As per the semantics (Section 2.3), for the equality to hold, we need $(\mathit{dom}([\![P_1]\!]_G) \cap \mathit{dom}([\![P_3]\!]_G)) = (\mathit{dom}([\![P_1]\!]_G \bowtie [\![P_2]\!]_G) \cap \mathit{dom}([\![P_3]\!]_G))$, which holds because $\mathit{corVars}(P_1\ \mathit{FNE}\ P_3) = \mathit{corVars}((P_1\ \mathit{AND}\ P_2)\ \mathit{FNE}\ P_3)$.

Intuitively, for the equivalence to hold, any variable of P_2 that does not appear in P_1 should also not appear in P_3. To enable the rewriting of $P = ((P_1\ \mathit{FNE}\ P_3)\ \mathit{AND}\ P_2)$ to a semantically equivalent pattern $P' = ((P_1\ \mathit{AND}\ P_2)\ \mathit{FNE}\ P_3)$, we propose the replacement of each variable of P_3 that is not in $\mathit{corVars}(P_1\ \mathit{FNE}\ P_3)$ with a unique variable that is not used in P. This way, we ensure that P_2 and P_3 do not share any variables that do not appear in P_1.

Algorithm 2 Multi-Way Join for Class Expression Learning in $\mathcal{ALC}$

1: **function** MWJ(P, G, X) ▷ P: Graph Pattern, G: RDF Graph, X: Mapping
2: // P is a single graph pattern or a *UNION* of *UNION*-free graph patterns
3: **for all** UNION-free patterns P_i of P **do**
4: **if** P_i is a BGP **then yield all** GenericJoin(P_i, G, X) ▷ Algorithm 1
5: **else yield all** MWJFNE(P_i, G, X) ▷ P_i contains negation
6: **function** MWJFNE(P, G, X) ▷ P: Graph Pattern, G: RDF Graph, X: Mapping
7: $P' \leftarrow P$ ▷ Copy pattern to preserve original
8: **for all** *FNE* patterns P_i of P **do**
9: Let $P_i = P_l$ *FNE* P_r ▷ P_l is a set of triple patterns
10: **if** $corVars(P_i) = \emptyset$ **then** ▷ Correlated variables are evaluated (fne-safety)
11: **if** EvalFNE(P_r, G, X) is **false then**
12: **return** ▷ The current mapping X does not yield a solution mapping
13: **else** remove (*FNE* P_r) from P' ▷ successfully evaluated
14: // All *FNE* patterns that were evaluated returned **true** and were removed
15: **if** there are no more *FNE* patterns in P' **then**
16: **yield all** GenericJoin(P', G, X) and **return**
17: $?x \leftarrow$ a variable from $cVars(P')$ ▷ Select a certain variable to be evaluated
18: // Evaluate the selected variable $?x$ using the triple patterns of P'
19: $K \leftarrow \bigcap_{tp \in P' | ?x \in var(tp)} \{\mu(?x) \mid \mu \in [\![tp]\!]_G\}$
20: **for all** $k \in K$ **do** ▷ Iterate over the possible values of $?x$
21: $X(?x) \leftarrow k$ ▷ Store k in the solution mapping
22: $P'' \leftarrow$ assign k to all occurrences of $?x$ in the triple patterns of P'
23: **yield all** MWJFNE(P'', G, X) ▷ after yielding proceeds with the next k
24: **function** EvalFNE(P, G, X)
25: // Uses the values assigned to the correlated variables to evaluate P
26: $P' \leftarrow \forall ?v \in var(P) \cap dom(X)$, assign $X(?v)$ to $?v$ in the triple patterns of P
27: $X' \leftarrow$ new empty mapping ; $X'(?v) \leftarrow X(?v)$
28: **if** MWJ(P', G, X') yields a solution **then return false**
29: **else return true**

By following the SPARQL standard and treating `FILTER NOT EXISTS` patterns as `FILTER` expressions, we can apply the UNION normal form provided for *AND-UNION-FILTER* graph patterns (Section 2.3) to SPARQL queries generated by $\mathcal{ALC}$ class expressions. By applying the UNION normal form and the rewriting rule proposed above to the queries of Table 2, we end up dealing with queries that are disjunctions of graph patterns, where each graph pattern is either a BGP or a set of triple patterns alongside patterns of negation. Our proposed algorithm, which is presented below, is able to evaluate the resulting queries by carrying out a series of multi-way joins.

4.2 Multi-way Join Algorithm

The proposed algorithm for the evaluation of SPARQL queries generated by $\mathcal{ALC}$ class expressions is shown in Algorithm 2. The algorithm's entry point is the function `MWJ` (lines 1–5), which takes a graph pattern that is already in

UNION normal form as input. `MWJ` iterates over all UNION-free graph patterns P_i of the input graph pattern P and for each P_i calls the appropriate function for its evaluation. If P_i is a BGP, it is evaluated by Generic Join (line 4), otherwise it is evaluated by the function `MWJFNE` (line 5), which is responsible for the evaluation of graph patterns containing `FILTER NOT EXISTS` patterns.

The function `MWJFNE` (lines 6–23) iterates first over the `FILTER NOT EXISTS` patterns ($P_i = P_l$ *FNE* P_r) of the provided graph pattern P (lines 8–13). For each P_i having its correlated variables evaluated (lines 11–13), the function `EvalFNE` is called (line 11). `EvalFNE` (lines 24–29) checks whether the current solution mapping X is a solution of P_i. This is done by evaluating P_r (line 28) after all the occurrences of the correlated variables of P_i are replaced with their corresponding term in the triple patterns of P_r (line 26). If the evaluation of P_r yields at least one solution, `EvalFNE` is evaluated to false, which leads to the active solution mapping in `MWJFNE` to be discarded (line 13). Recall that the mappings of P_l and P_r should not be compatible (Section 2.3). If all `FILTER NOT EXISTS` patterns are successfully evaluated, `MWJFNE` proceeds with the evaluation of P' (lines 15–23). Note that successfully evaluated `FILTER NOT EXISTS` patterns are removed from P' (line 13) and hence, are not evaluated multiple times. If P' is a BGP, it is evaluated by Generic Join (line 16). Otherwise, the evaluation proceeds in a similar fashion to Generic Join (lines 17-23). If there are remaining `FILTER NOT EXISTS` patterns in P', the evaluation focuses on the certain variables of P', i.e., on the variables appearing only in triple patterns (line 19). For each possible value of the selected certain variable (lines 20–23), `MWJFNE` is called recursively.

The proposed algorithm integrates negation in multi-way join plans by calling `MWJ` within `EvalFNE` (line 28). In addition, provided a solution mapping, it does not completely evaluate the right hand side of `FILTER NOT EXISTS` patterns. Instead, it terminates its evaluation once a solution mapping is found, thus avoiding storing intermediate results. The use of the UNION normal form may result in a larger number of joins. However, this can be beneficial, as the joins may limit the intermediate mappings generated by the original union patterns.

4.3 Implementation

We have implemented the proposed algorithm within the tensor-based triple store Tentris [5]. We chose Tentris because it supports multi-way joins [5] and achieves state-of-the-art performance in the evaluation of basic graph patterns [6]. However, Tentris is not able to evaluate SPARQL queries generated by $\mathcal{ALC}$ class expressions, due to its missing support for `FILTER NOT EXISTS` patterns. By implementing our proposed algorithm within Tentris, we further improve the state of the art, as demonstrated by the experimental results (Section 5). The performance of multi-way joins is affected by the order in which variables are evaluated [15]. Tentris dynamically selects a variable at each recursive step of the algorithm using cardinality estimations. We modified its selection process to take the number of `FILTER NOT EXISTS` patterns a particular variable appears into account to break ties (i.e., provided two variables with the same cardinality

estimation, the one that appears in a `FILTER NOT EXISTS` pattern is selected). Last, we apply the UNION normal form to the provided queries while parsing them. Henceforth, we refer to our implementation as TentrisALC.

5 Experimental Results

We evaluated the performance of our implementation using SPARQL queries corresponding to $\mathcal{ALC}$ class expressions on five datasets of varying sizes. The experiments that are presented below were carried out on a Debian 10 server with an AMD EPYC 7282 CPU, 256GB RAM and a 2TB Samsung 970 EVO Plus SSD. Supplementary material—including datasets, binaries, queries, scripts, and configurations—is available online.[4]

5.1 Systems, Setup and Execution

As the learning of class expressions using SPARQL is carried out over HTTP [7], we compared the performance of TentrisALC against the performance of the following triple stores that provide a SPARQL compliant HTTP endpoint: (i) Blazegraph 2.1.6.RC, (ii) Fuseki 4.10.0, (iii) GraphDB 10.3.3, and (iv) Virtuoso 7.2.10. In our experiments, we also wanted to include MilleniumDB[5], commit: 442e650 [29], which uses multi-way joins for the evaluation of basic graph patterns. However, we did not include it, as it does not evaluate queries having union graph patterns within `FILTER NOT EXISTS` patterns correctly. Each triple store was configured following its respective documentation. The experiemnts were executed over HTTP using the benchmark execution framework IGUANA 3.3.3 [9]. For each dataset, each query was executed once during the warmup phase. After the warmup phase, the sets of queries were executed three consecutive times. The query timeout was set to three minutes. As in [6], we measured the performance of the triple stores using the following metrics: (i) QPS, i.e., the number of queries executed per second, (ii) pAvgQPS, i.e., the penalized average QPS and (iii) QMPH, i.e., the number of query mixes executed per hour. A query mix is a set or a multiset of queries. The metric QMPH captures the number of times a particular query mix is evaluated in an hour. This means that the lower the runtimes of individual queries are, the higher the QMPH value is. Queries that failed (e.g., timed out or returned an error code) are penalized with a runtime of three minutes.

5.2 Datasets and Queries

Our evaluation comprises five datasets (i.e., knowledge graphs). Four of these datasets, namely Carcinogenesis, Mutagenesis, Premier League and Vicodi, are frequently used to evaluate class expression learning algorithms (e.g., [13, 17])

[4] `https://github.com/dice-group/alc2sparql-bench`
[5] `https://github.com/MillenniumDB/MillenniumDB`

Table 3. The statistics of the datasets used in the evaluation

	#Triples	#Distinct Subjects	#Distinct Predicates	#Distinct Objects	# Queries \w Negation
Carcinogenesis	157K	22.5K	25	23.2K	169
Mutagenesis	96K	14.2K	16	15K	208
Premier League	2.1M	11.5K	217	12.5K	209
Vicodi	405K	33.4K	14	35.2K	137
YAGO4English	40.2M	7.2M	104	3M	209

and are available in [17, 30]. The largest of these datasets, namely Premier League, contains 2.1M triples. To evaluate the scalability of our approach, we used a subset of the English version of YAGO4 [26], which contains more than 40M triples. As in previous works that follow closed-world semantics (e.g., [7, 13]), we first materialized the inferences of the knowledge graphs. Table 3 reports the statistics of the knowledge bases after the materialization process. For the class expression learning datasets, we generated 300 unique $\mathcal{ALC}$ class expressions using a slightly modified version of the learning problem generator of [17]. This modified version does not prioritize simple class expressions. For YAGO4English, due to the generator not scaling to its size, we randomly created 300 unique class expressions by recursively applying the construction rules of $\mathcal{ALC}$. During the generation of class expressions for YAGO4English, we focused only on the properties of the dataset that come from schemas.org (i.e., we did not consider properties coming from bioschemas.org or the RDF/S vocabulary). Schema.org has a richer taxonomy than bioschemas.org, which resulted in $\forall r.C$ and $\exists r.C$ class expressions being more diverse. All class expressions were mapped to SPARQL queries using the mapping of Table 2. For YAGO4English, we ended up having to remove eight queries, as IGUANA was not able to handle them properly (non-escaped characters in IRIs). The last column of Table 3 shows the number of queries having at least one `FILTER NOT EXISTS` pattern.

5.3 Results and Discussion

The results of our evaluation on the datasets for class expression learning are shown in Tables 4 and 5. The results on YAGO4English are shown in Table 6. Many queries of YAGO4English return more than 7M results, due to their corresponding class expression covering the whole set of individuals. As Virtuoso has a limit of 2^{20} results [5], we did not evaluate it on YAGO4English.

The results show that TentrisALC achieves the highest pAvgQPS and QMPH values across all datasets. To ensure the that difference in the performance is statistically significant, we performed the Wilcoxon signed-rank test using the penalized QPS values achieved by each system in each query. The null hypothesis (i.e., the performances of TentrisALC and the baseline systems come from the same distribution) was rejected for all systems in all datasets (p-value $p < 0.001$).

Table 4. The results on class expression learning datasets (cold run). The column failed reports the number of queries for which the corresponding system failed (e.g., timed out) at least once.

	Carcinogenesis			Mutagenesis		
	QMPH	pAvgQPS	failed	QMPH	pAvgQPS	failed
Blazegraph	68.122	49.221	0	40.911	31.637	0
Fuseki	43.426	96.789	0	28.580	58.097	0
GraphDB	28.003	139.800	0	21.156	102.190	0
TentrisALC (ours)	**1410.674**	**792.274**	**0**	**1128.076**	**545.604**	**0**
Virtuoso	148.622	372.766	0	113.312	268.588	0
	Premier League			Vicodi		
	QMPH	pAvgQPS	failed	QMPH	pAvgQPS	failed
Blazegraph	16.832	44.261	0	2.508	48.152	0
Fuseki	3.254	85.898	2	1.399	101.890	9
GraphDB	5.708	139.975	0	1.756	128.904	5
TentrisALC (ours)	**2463.874**	**902.343**	**0**	**715.107**	**773.800**	**0**
Virtuoso	82.245	390.305	0	6.227	411.529	0

The scalability of our approach can already be observed in the datasets for class expression learning. Table 5 shows that, in the smaller datasets for class expression learning, TentrisALC achieves a value of QMPH that is 10 times higher than the second best system, namely Virtuoso. In Premier League and Vicodi, TentrisALC performs 37 and 126 times better than Virtuoso in terms of QMPH, respectively. This is due to TentrisALC being able to efficiently evaluate queries having `FILTER NOT EXISTS` patterns. More specifically, in Vicodi, TentrisALC achieves a QMPH value that is 7.6 times higher than the second best system (Virtuoso) in the set of queries that do not have any `FILTER NOT EXISTS` patterns. In the set of queries that contain negation, TentrisALC achieves a QMPH value that is 128 times higher than the second best value (Virtuoso). This shows that the overall results are mostly affected by the systems' ability to efficiently evaluate queries with `FILTER NOT EXISTS` patterns. This observation becomes more evident in YAGO4English (Table 6), where all systems apart from TentrisALC timed out in multiple queries. In particular, Fuseki and GraphDB timed out in more than half of the queries. More importantly, the timeouts occurred only in queries that contain `FILTER NOT EXISTS` patterns. As mentioned earlier, many queries in YAGO4English return more than 7M results. In queries with large result sets, the runtime can be heavily impacted by the results' enumeration. An example of a class expression that captures the efficiency of our approach is $\neg(\forall \mathit{exampleOfWork.Prueba\_Villafranca\_de\_Ordizia})$. The corresponding SPARQL query includes three nested `FILTER NOT EXISTS` patterns and returns only 161 solutions (i.e., the results' enumeration did not impact the query's execution time). TentrisALC required 8.3 seconds on average to evaluate this query, whereas all other systems timed out.

Table 5. The results on class expression learning datasets (warm runs). The column failed reports the number of queries for which the corresponding system failed (e.g., timed out) at least once.

	Carcinogenesis			Mutagenesis		
	QMPH	pAvgQPS	failed	QMPH	pAvgQPS	failed
Blazegraph	72.920	58.134	0	42.615	37.090	0
Fuseki	45.509	133.754	0	28.940	82.716	0
GraphDB	31.263	246.155	0	22.802	162.238	0
TentrisALC (ours)	**1605.557**	**1571.377**	**0**	**1224.473**	**1015.727**	**0**
Virtuoso	149.928	734.093	0	114.822	532.871	0
	Premier League			Vicodi		
	QMPH	pAvgQPS	failed	QMPH	pAvgQPS	failed
Blazegraph	17.303	51.415	0	2.524	58.075	0
Fuseki	3.237	110.457	2	1.381	144.729	14
GraphDB	5.823	257.314	0	1.763	220.353	5
TentrisALC (ours)	**3176.005**	**1984.633**	**0**	**761.052**	**1554.764**	**0**
Virtuoso	84.486	1150.088	0	5.991	762.763	0

Table 6. The results on the largest dataset, namely YAGO4English. The column failed reports the number of queries for which the corresponding system failed (e.g., timed out) at least once. Virtuoso is not included due to its hard limit of 2^{20} results.

	YAGO4English (cold run)			YAGO4English (warm runs)		
	QMPH	pAvgQPS	failed	QMPH	pAvgQPS	failed
Blazegraph	0.131	18.275	58	0.131	25.985	66
Fuseki	0.116	36.425	168	0.116	66.201	168
GraphDB	0.123	46.120	161	0.123	100.297	161
TentrisALC (ours)	**1.342**	**207.281**	**0**	**1.351**	**435.361**	**0**

The efficiency of our approach lies in its ability to evaluate `FILTER NOT EXISTS` patterns without having to materialize intermediate results and its ability to terminate the evaluation of such patterns once a single mapping is found. Regarding the size of the datasets, to the best of our knowledge, recent works on class expression learning have focused only on small scale datasets that contain up to a few million triples (e.g., Premier League). As demonstrated above, our approach is able to scale to large datasets and we believe that it will enable learning algorithms to be deployed on large scale knowledge graphs.

6 Related Work

Class expression learning has been extensively investigated (e.g., [12, 18, 27]). In recent years, several works have focused on accelerating the process of learning

class expressions. For instance, Westphal et al. [31] accelerate class expression learning by introducing a heuristic function that is based on simulated annealing. With their meta-heuristics, they are able to reduce the number of instance retrieval operations that are required to reach a goal concept. Kouagou et al. [17] accelerate class expression learning by accurately predicting lengths of possible goal concepts. By this, they avoid instance retrieval operations on lengthy concepts. In [11], the authors employ deep reinforcement learning to learn non-myopic heuristic functions, i.e., heuristic functions that take future rewards into account. These non-myopic heuristic functions accelerate class expression learning, as they are able to efficiently steer the search process towards goal states. Our work builds upon [7] that showed that class expression learning can be accelerated by converting class expressions to SPARQL queries. Our work focuses on improving the runtimes of retrieval operations and hence can be used in combination with the approaches described above.

Hogan et al. [15] were the first to formalize worst-case optimal join algorithms for the evaluation of basic graph pattern SPARQL queries. Several works have employed such algorithms for the evaluation of conjunctive queries [2, 5, 15] and demonstrated their efficiency. Recently, a multi-way join algorithm for the evaluation of conjunctive regular path queries based on the evaluation process of worst-case optimal join algorithms was proposed in [16]. This algorithm also makes use of the UNION normal form for AND-UNION patterns (i.e., it does not consider filters). One of the first worst-case optimal multi-way join algorithms was presented in [28]. As mentioned in Section 4, a sketch for the evaluation of Datalog rules containing negation is provided in [28]. However, to the best of our knowledge, there have not been any works for SPARQL that integrate the evaluation of negation in multi-way join plans.

7 Conclusion And Future Work

We presented a multi-way join algorithm for the evaluation of SPARQL queries corresponding to $\mathcal{ALC}$ class expressions. The main characteristic of our algorithm is the inclusion of `FILTER NOT EXISTS` patterns (i.e., negation) in multi-way join plans. Its purpose is to accelerate class expression learning in $\mathcal{ALC}$ by reducing the runtimes of instance retrieval operations. The experimental results on five datasets show that our implementation outperforms its competition across all datasets and that it is the only one scaling to the largest dataset, which contains more than 40M triples. In the future, we will extend our approach to support more expressive description logics (e.g., $\mathcal{SROIQ}$).

Acknowledgments. This work has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 860801 and Horizon Europe research and innovation programme under grant agreement No 101070305. It has also been supported by the Ministry of Culture and Science of North Rhine-Westphalia (MKW NRW) within the project SAIL under the grant no NW21-059D. This work has also

been supported by the German Federal Ministry of Education and Research (BMBF) within the project NEBULA under the grant no 13N16364

References

1. Angles, R., Gutierrez, C.: Negation in SPARQL. In: Proceedings of the 10th Alberto Mendelzon International Workshop on Foundations of Data Management, Panama City, Panama, May 8-10, 2016. CEUR Workshop Proceedings, vol. 1644 (2016)
2. Arroyuelo, D., Hogan, A., Navarro, G., Reutter, J.L., Rojas-Ledesma, J., Soto, A.: Worst-case optimal graph joins in almost no space. In: SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021. pp. 102–114 (2021)
3. Atserias, A., Grohe, M., Marx, D.: Size bounds and query plans for relational joins. In: 49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA. pp. 739–748 (2008)
4. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press (2003)
5. Bigerl, A., Conrads, F., Behning, C., Sherif, M.A., Saleem, M., Ngomo, A.N.: Tentris - A tensor-based triple store. In: The Semantic Web - ISWC 2020 - 19th International Semantic Web Conference, Athens, Greece, November 2-6, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12506, pp. 56–73 (2020)
6. Bigerl, A., Conrads, L., Behning, C., Saleem, M., Ngomo, A.N.: Hashing the hypertrie: Space- and time-efficient indexing for SPARQL in tensors. In: The Semantic Web - 21st International Semantic Web Conference, Virtual Event, October 23-27, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13489 (2022)
7. Bin, S., Bühmann, L., Lehmann, J., Ngomo, A.N.: Towards sparql-based induction for large-scale RDF data sets. In: ECAI 2016 - 22nd European Conference on Artificial Intelligence, 29 August-2 September 2016, The Hague, The Netherlands - Including Prestigious Applications of Artificial Intelligence (PAIS 2016). Frontiers in Artificial Intelligence and Applications, vol. 285, pp. 1551–1552 (2016)
8. Burnett, M.: Explaining ai: fairly? well? In: Proceedings of the 25th International Conference on Intelligent User Interfaces. pp. 1–2 (2020)
9. Conrads, F., Lehmann, J., Saleem, M., Morsey, M., Ngomo, A.N.: Iguana: A generic framework for benchmarking the read-write performance of triple stores. In: The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10588, pp. 48–65. Springer (2017)
10. d'Amato, C.: Machine learning for the semantic web: Lessons learnt and next research directions. Semantic Web **11**(1), 195–203 (2020)
11. Demir, C., Ngomo, A.N.: Neuro-symbolic class expression learning. In: Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, 19th-25th August 2023, Macao, SAR, China. pp. 3624–3632 (2023)
12. Fanizzi, N., d'Amato, C., Esposito, F.: DL-FOIL concept learning in description logics. In: Inductive Logic Programming, 18th International Conference, ILP 2008, Prague, Czech Republic, September 10-12, 2008, Proceedings. Lecture Notes in Computer Science, vol. 5194, pp. 107–121. Springer (2008)

13. Heindorf, S., Blübaum, L., Düsterhus, N., Werner, T., Golani, V.N., Demir, C., Ngomo, A.N.: Evolearner: Learning description logics with evolutionary algorithms. In: WWW '22: The ACM Web Conference 2022, Virtual Event, Lyon, France, April 25 - 29, 2022. pp. 818–828. ACM (2022)
14. Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., de Melo, G., Gutiérrez, C., Kirrane, S., Gayo, J.E.L., Navigli, R., Neumaier, S., Ngomo, A.N., Polleres, A., Rashid, S.M., Rula, A., Schmelzeisen, L., Sequeda, J.F., Staab, S., Zimmermann, A.: Knowledge graphs. ACM Comput. Surv. pp. 71:1–71:37 (2021)
15. Hogan, A., Riveros, C., Rojas, C., Soto, A.: A worst-case optimal join algorithm for SPARQL. In: The Semantic Web - ISWC 2019 - 18th International Semantic Web Conference, Auckland, New Zealand, October 26-30, 2019, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11778, pp. 258–275. Springer (2019)
16. Karalis, N., Bigerl, A., Heidrich, L., Sherif, M.A., Ngomo, A.N.: Efficient evaluation of conjunctive regular path queries using multi-way joins. In: The Semantic Web - 21st International Conference, ESWC 2024, Hersonissos, Crete, Greece, May 26-30, 2024, Proceedings, Part I. Lecture Notes in Computer Science, vol. 14664, pp. 218–235. Springer (2024)
17. Kouagou, N.J., Heindorf, S., Demir, C., Ngomo, A.N.: Learning concept lengths accelerates concept learning in ALC. In: The Semantic Web - 19th International Conference, ESWC 2022, Hersonissos, Crete, Greece, May 29 - June 2, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13261, pp. 236–252 (2022)
18. Lehmann, J.: Learning OWL Class Expressions, Studies on the Semantic Web, vol. 6. IOS Press (2010)
19. Ngo, H.Q., Porat, E., Ré, C., Rudra, A.: Worst-case optimal join algorithms. J. ACM pp. 16:1–16:40 (2018)
20. Ngo, H.Q., Ré, C., Rudra, A.: Skew strikes back: new developments in the theory of join algorithms. SIGMOD Rec. **42**(4), 5–16 (2013)
21. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM Trans. Database Syst. **34**(3), 16:1–16:45 (2009)
22. Rudolph, S.: Foundations of description logics. In: Reasoning Web. Semantic Technologies for the Web of Data - 7th International Summer School 2011, Galway, Ireland, August 23-27, 2011, Tutorial Lectures. Lecture Notes in Computer Science, vol. 6848, pp. 76–136. Springer (2011)
23. Sarker, M.K., Hitzler, P.: Efficient concept induction for description logics. In: The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019. pp. 3036–3043 (2019)
24. Schmidt, M., Meier, M., Lausen, G.: Foundations of SPARQL query optimization. In: Segoufin, L. (ed.) Database Theory - ICDT 2010, 13th International Conference, Lausanne, Switzerland, March 23-25, 2010, Proceedings. pp. 4–33. ACM International Conference Proceeding Series, ACM (2010)
25. Schmidt-Schauß, M., Smolka, G.: Attributive concept descriptions with complements. Artificial intelligence **48**(1), 1–26 (1991)
26. Tanon, T.P., Weikum, G., Suchanek, F.M.: YAGO 4: A reason-able knowledge base. In: The Semantic Web - 17th International Conference, ESWC 2020, Heraklion, Crete, Greece, May 31-June 4, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12123, pp. 583–596. Springer (2020)
27. Tran, A.C., Dietrich, J., Guesgen, H.W., Marsland, S.: Parallel symmetric class expression learning. J. Mach. Learn. Res. **18**, 64:1–64:34 (2017)

28. Veldhuizen, T.L.: Triejoin: A simple, worst-case optimal join algorithm. In: Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014. pp. 96–106. OpenProceedings.org (2014)
29. Vrgoč, D., Rojas, C., Angles, R., Arenas, M., Arroyuelo, D., Buil-Aranda, C., Hogan, A., Navarro, G., Riveros, C., Romero, J.: MillenniumDB: An Open-Source Graph Database System. Data Intelligence pp. 1–39 (2023)
30. Westphal, P., Bühmann, L., Bin, S., Jabeen, H., Lehmann, J.: Sml-bench - A benchmarking framework for structured machine learning. Semantic Web **10**(2), 231–245 (2019)
31. Westphal, P., Vahdati, S., Lehmann, J.: A simulated annealing meta-heuristic for concept learning in description logics. In: International Conference on Inductive Logic Programming. pp. 266–281. Springer (2021)