Annals of Mathematics and Artificial Intelligence
https://doi.org/10.1007/s10472-024-09965-3

RESEARCH

Type-based computation of knowledge graph statistics

Iztok Savnik[1] · Kiyoshi Nitta[2] · Riste Skrekovski[3,4] · Nikolaus Augsten[5]

Accepted: 24 December 2024

Abstract
We propose a formal model of a knowledge graph (abbr. KG) that classifies the ground triples into sets that correspond to the triple types. The triple types are partially ordered by the sub-type relation. Consequently, the sets of ground triples that are the interpretations of triple types are partially ordered by the subsumption relation. The types of triple patterns restrict the sets of ground triples, which need to be addressed in the evaluation of triple patterns, to the interpretation of the types of triple patterns. Therefore, a *schema graph* of a KG should include all triple types that are likely to be determined as the types of triple patterns. The stored schema graph consists of the selected triple types that are stored in a KG and the complete schema graph includes all valid triple types of KG. We propose choosing the schema graph, which consists of the triple types from a strip around the stored schema graph, i.e., the triple types from the stored schema graph and some adjacent levels of triple types with respect to the sub-type relation. Given a selected schema graph, the statistics are updated for each ground triple t from a KG. First, we determine the set of triple types stt from the schema graph that are affected by adding a triple t to an RDF store. Finally, the statistics of triple types from the set stt are updated.

Keywords Knowledge graphs · RDF stores · Graph databases · Database statistics

Matjematics Subject Classification (2010) 68P20 · 68T30 · 68P15 · 03B70

✉ Iztok Savnik
iztok.savnik@famnit.upr.si

Kiyoshi Nitta
knitta@lycorp.co.jp

Riste Skrekovski
skrekovski@gmail.com

Nikolaus Augsten
nikolaus.augsten@plus.ac.at

1 Faculty of Mathematics, Natural Sciences and Information Technologies, University of Primorska, Koper, Slovenia

2 LY Corporation, Tokyo, Japan

3 Faculty of Information Studies, Novo Mesto, Slovenia

4 Faculty of Mathematics and Physics, University of Ljubljana, Ljubljana, Slovenia

5 Department of Computer Science, University of Salzburg, Salzburg, Austria

Published online: 17 January 2025

1 Introduction

The statistics of RDF stores is an essential tool used in the processes of query optimization [1–4]. They are used to estimate the size of a query result and the time needed to evaluate a given query. The estimations are required to find the most efficient query evaluation plan for a given input query. Furthermore, the statistics of large graphs are used to solve problems closely related to query evaluation; for example, they can be employed in algorithms for partitioning large graphs [5].

Many RDF stores treat graphs as simple sets of vertices and edges without a conceptual schema. The statistics in the schema-less RDF stores are based on the cardinality of the keys representing the constants in triple patterns. While some of the initial RDF stores are implemented on top of relational database systems [6, 7], most of them are implemented around a subset of seven indexes corresponding to the keys that are subsets of $\{S, P, O\}$, i.e., the subject, predicate, and object of the triples. The indexes can be either a B+ tree [1, 8], a custom-designed index [9–12], or some other index data structure (e.g., radix trie) [13–15]. Statistics at the level of relations (as used in relational systems) are not useful at the level of RDF graphs; therefore, statistics are obtained by sampling the indexes [6], or by creating separate aggregates of indexes [1, 8].

There are a few approaches to the computation of statistics of RDF stores that use some form of the semantic structure to which the statistics are attached. Stocker et al. [2] use RDF Schema information to precompute the size of all possible joins where the domain/range of some predicate is joined with the domain/range component of some other predicate. Wolff et al. [43] implement the statistics similarly as proposed in [2] to be utilized in the heuristic rules used for query optimization. Neumann and Moetkotte propose the use of the characteristic sets [3], i.e., the sets of predicates with a common S/O component, for the computation of statistics of star-shaped queries. Gubichev and Neumann further organize characteristic sets [4] in a hierarchy to allow precise estimation of joins in star queries.

In this paper, we assume that a knowledge graph (abbr. KG) [16], including a complete RDF schema, is stored in an RDF store. We propose a formal model of a KG that organizes the ground triples from a KG into sets of triples that correspond to their types. In this respect, the formal model is close to the relational view of data [17]: the tuples from a relational database are organized into separate relations defined by the relational schemata.

The main difference between the relational model and the RDF graph model is in the organization of triple types. The triple types are organized into a poset on the basis of the sub-typing relation. Such organization of triple types is the consequence of the ontologies of classes and predicates that form a part of the conceptual schema of a KG. On the other hand, one of the important advantages of the relational model over the network and hierarchical models is the value-based links between the tuples from different relations, which are contained in the ternary relations of an RDF store.

We propose the statistics of an RDF store based on the triple types. The main reason for this is the interpretations of triple types that are the targets of the queries composed of triple patterns. Each triple pattern has a type whose interpretation subsumes the result of the triple pattern evaluation. There are many possible valid types of a triple pattern from which the minimal type, having the smallest interpretation, is chosen. To process a triple pattern, solely the ground triples from the interpretation of the minimal type of a triple pattern need to be addressed.

A set of triple types, including the types of all ground triples from a knowledge graph, together with the triples expressing the sub-type (sub-class and sub-predicate) relations

among classes and predicates, form a conceptual schema of a knowledge graph that we refer to as a *schema graph*. The selection of the set of triple types that constitute the schema graph is important since only the triple types from the schema graph have statistics attached. Ideally, all the triple types that are the types of triple patterns from the workload of queries are a part of the schema graph.

The *stored schema graph* is the selected minimal schema graph of a given knowledge graph, including only the triple types that are stored in an RDF store in the form of the definitions of domains and ranges of predicates.[1] On the other hand, the *complete schema graph* includes all valid types of ground triples from a KG. The schema graphs that can be selected as the basis for the computation of KG statistics are defined in between the stored schema graph and the complete schema graph. We propose to use the schema graph that includes the triple types from a strip around the stored schema graph, i.e., the triple types from the stored schema graph and some adjacent levels of triple types with respect to the sub-type relation. We expect that the triple types that are close to some stored schema types are likely to be used as the type of a triple pattern.

Let us now sketch the algorithm for computing the statistics of a KG. Assume the schema graph sg of a KG has already been selected. For each ground triple t from a KG, we first determine a set of triple types stt, that include all valid types of t which are a part of the schema graph sg. The set stt depends on the stored types (via rdf:type) of the triple t's components. From this point of view, the set stt is the intersection between the set of all possible valid types of t and the triple types from sg. Finally, the statistics are updated for the selected triple types from stt as the consequence of adding the triple t to the statistics.

1.1 Contributions

The contributions of the work presented in this paper are as follows. First, we propose the definition of the conceptual schemata of a KG in the form of a schema graph, a graph that includes the triple types as its edges. The concept of a triple type is close to the relation schema, which separates the instances of the given relation schema from the other tuples from a database. Similarly, the triple types separate the instances of a given triple type from the other triples from an RDF store. The main difference between the n-ary relations of the relational model and ternary relations of the RDF graph model is in the rich structures that are formed by triple types and, consequently, the interpretations of triple types (i.e., ternary relations). Namely, the triple types are ordered by the relation sub-type to form a partially ordered set, which is reflected in the partially ordered set of the interpretations of triple types.

Second, to the best of our knowledge, this is the first proposal for using the triple types as a formal framework for attaching the statistics of the triple type instances. We introduce an algorithm for the computation of the statistics of the ground triples from a KG. The algorithm computes the statistics of the triple types from a schema graph. The schema graph includes the triple types that are either the stored triple types or the triple types close to some stored triple types with respect to the relation sub-type. The size of the selected strip around the stored schema graph can be controlled by limiting the number of levels above and below the stored schema graph.

Finally, we propose two ways of counting the instances and keys of a given triple type. When adding individual triples to the statistics, a triple type represents the seven types of

[1] As an example, Section 5.3 includes the description of the *stored schema graph* used in the experiments with the Yago KG.

keys. The *bound* type of counting a key respects underlying triple type (the complete type of the key), and the *unbound* type of counting is free from the underlying triple type.

1.2 Paper organization

The paper is organized as follows. Section 2 provides a formalization of a KG and introduces the denotational semantics of the concepts defined. Section 3 presents three algorithms for the computation of statistics. The first algorithm computes the statistics for the stored schema graph, and the second algorithm computes the statistics of the complete schema graph. The third algorithm focuses on the triple types from a strip around the stored schema graph, i.e., in addition to the stored schema graph, also the triple types that are some levels below and above the stored schema graph. Section 4 introduces the concepts of a key and a key type that are used for counting the instances of a triple type. Further, the concepts of bound and unbound counting are discussed. An empirical study of the algorithms for the computation of the statistics is presented in Section 5. Two experiments are presented, first, on a simple toy domain and second, on the core of the Yago2 KG. Related work is presented in Section 6. Finally, concluding remarks are given in Section 7.

2 Conceptual schema of a knowledge graph

A formal definition of the schema of a knowledge graph is based on the RDF [18] and RDF-Schema [19] data models. Let U be the set of IRI-s, B be the set of blanks and L be the set of literals. Let us also define sets $S = U \cup B$, $P = U$, and $O = U \cup B \cup L$.

A *RDF triple* is a triple $(s, p, o) \in S \times P \times O$, an *RDF graph* $g \subseteq S \times P \times O$ is a set of triples and the set of all RDF graphs is $G = \mathcal{P}(S \times P \times O)$. We state that an RDF graph g_1 is a *sub-graph* of g_2, denoted as $g_1 \subseteq g_2$, if all triples of g_1 are also triples of g_2.

2.1 Knowledge graph simple

A simple knowledge graph presented as an RDF graph [18] using the RDF-Schema vocabulary [19] is given in Fig. 1. The KG, referred to in the rest of the text as *Simple*, is used in the examples presented in the following sections. The part of *Simple* expressed using RDF Schema is colored blue, and the data part of the graph is colored black. The dataset *Simple* is available from the *epsilon* data repository [20].

The dataset *Simple* includes 33 triples. Four user-defined classes are included. The class person has sub-classes, scientist and philosopher. The class location represents the concept of a physical location. Further, *Simple* includes four predicates. The predicate wasBornIn relates a person with a location. The predicate hasAge has the domain class philosopher and the range type xsd:integer. The predicate influences relates the class person with itself. Finally, the predicate subjectStartRelation is a super-predicate of the predicate wasBornIn. It links owl:Thing with itself.

2.2 Identifiers

The definition of the sets S, P, and O given at the beginning of this section reflects the possible syntactical forms of triple components. To abstract away the details of the RDF

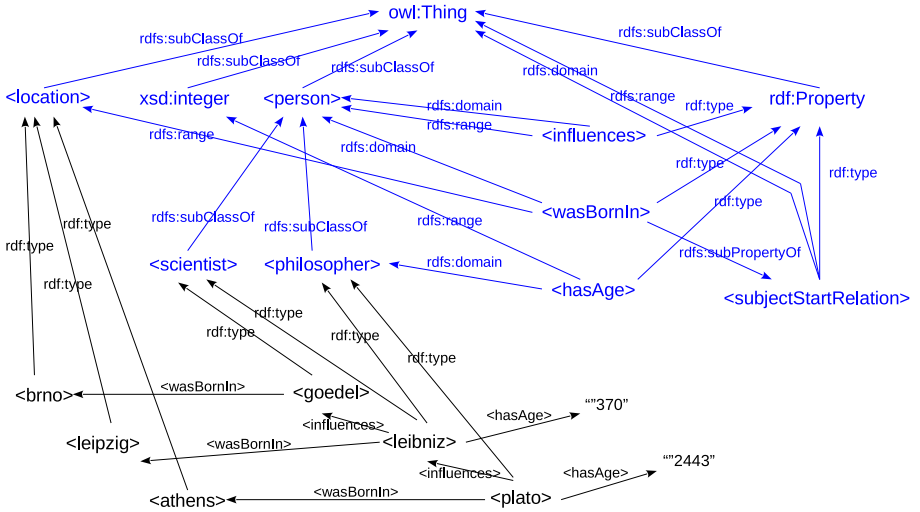

Fig. 1 Knowledge graph *Simple* represented by using the RDF and RDF Schema

data model [21], we refer to the elements of $U \cup B \cup L$ as *identifiers*. The set, including all identifiers, is denoted as $\mathcal{I}$.

We define a set of concepts to be used for a semantic characterization of identifiers $\mathcal{I}$. *Individual identifiers*, denoted as $\mathcal{I}_i$, stand for the objects and things. They are identifiers that have specified their classes using the property rdf:type. *Class identifiers* denoted as set $\mathcal{I}_c$, are identifiers of all sub-classes of the top class $\top$—Yago, for instance, uses the top class owl:Thing. Finally, the *predicate identifiers*, denoted as a set $\mathcal{I}_p$, are identifiers that represent RDF predicates. The predicate identifiers are, from one perspective, very similar to the class identifiers. Indeed, the predicates can have sub-predicates in the same manner as the classes that have sub-classes. However, while classes have instances, predicates do not have instances. Predicates $\mathcal{I}_p$ are individual identifiers that are the instances of rdf:Property.

The partial ordering of identifiers $(\mathcal{I}, \preceq)$ is defined by using the relations rdf:type, rdfs:subClassOf, and rdfs:subPropertyOf. Let $i_1, i_2 \in \mathcal{I}$. If $i_1, i_2 \in \mathcal{I}_i$ then $i_1 \preceq i_2$ only if $i_1 = i_2$. If $i_1 \in \mathcal{I}_i$ and $i_2 \in \mathcal{I}_c$ then $i_1 \preceq i_2$ only if i_1 is an instance of i_2, i.e., $(i_1, \text{rdf:type}, i_2') \in g$ where i_2' is either i_2 or some subclass of i_2. Next, if $i_1, i_2 \in \mathcal{I}_c$, then $i_1 \preceq i_2$ holds if $i_1 = i_2$ or if i_1 is a subclass of i_2. Finally, $i_1, i_2 \in \mathcal{I}_p$ then $i_1 \preceq i_2$ holds only if $i_1 = i_2$ or if i_1 is a sub-property of i_2.

The meaning of $i_1 \preceq i_2$ can be described by saying that i_1 is more specific than or equal to i_2. The relationship $\preceq$ is reflexive, transitive, and anti-symmetric, i.e., it defines a partial ordering $(\mathcal{I}, \preceq)$ over all identifiers $\mathcal{I}$. The top of the partial ordering is the top-class $\top$ such that $\forall c \in \mathcal{I}_c : c \preceq \top$. The bottom of the partial ordering of identifiers comprises the individual identifiers from $\mathcal{I}_i$.

If we restrict the partial ordering relation $\preceq$ to link solely the class identifiers $\mathcal{I}_c$, and predicate identifiers $\mathcal{I}_p$, we obtain the poset of *type identifiers* $(\mathcal{I}_c \cup \mathcal{I}_p, \preceq)$. The classes and predicates have the role of *types* and the relation $\preceq$ has the role of the relation sub-type [22].

Example 1 Figure 2 presents the schema part of the KG *Simple*. The graph includes solely the type identifiers, i.e., the class and predicate identifiers. The class identifiers are location, person, scientist, philosopher, xsd:integer, rdf:Property, and owl:Thing. The predicated iden-

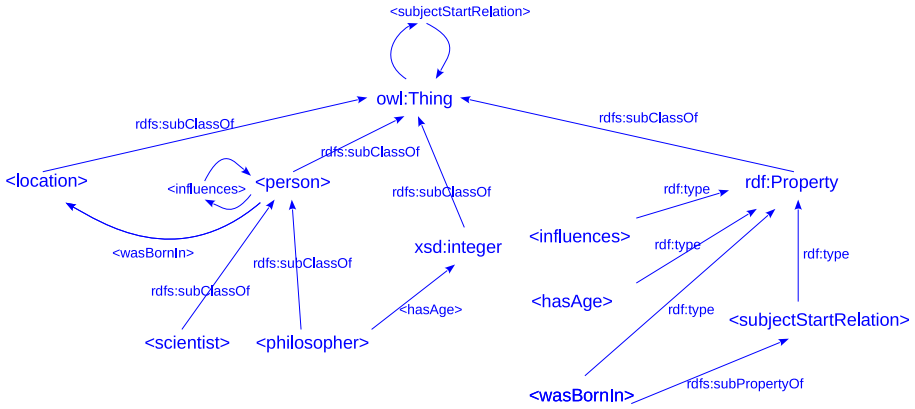

Fig. 2 The schema graph of the knowledge graph *Simple*

tifiers are influences, hasAge, wasBorinIn, and subjectStartRelation. We did not present the predicates defined in RDF-Schema vocabulary.

The sub-type relation $\preceq$, defined for the above classes and predicates, is the set {scientist$\preceq$person, philosopher$\preceq$person, person$\preceq$owl:Thing, location$\preceq$owl:Thing, rdf:Predicate$\preceq$owl:Thing, wasBornIn$\preceq$subjectStartRelation}. □

The predicates have a special role in knowledge representation. They are the instances of the class rdf:Property but they can also act as types. A predicate is the name of a binary relation between the interpretations of types of the subject and object. The poset of predicates then defines the poset of binary relations represented by the predicates.

A class and predicate identifiers can have three different denotations that can be useful in a knowledge base. First, the ordinary interpretation of a class c includes all *members* of c, formally $[\![c]\!]_g = \{i \mid i \in \mathcal{I}_i \wedge (i, \text{rdf:type}, c) \in g\}$. The ordinary interpretation of a predicate p is the predicate p itself.

Second, the inherited interpretation of a given class c are the *instances* of c, i.e., the members of the class c and all c's sub-classes. Formally, $[\![c]\!]^*_g = \{i \mid \exists c' \in \mathcal{I}_i (c' \preceq c \wedge i \in [\![c']\!]_g)$. The inherited interpretation of a predicate p includes a predicate p and all sub-predicates of p, i.e., $[\![p]\!]_g = \{p' \mid p' \in \mathcal{I}_p \wedge p' \preceq p\}$.

Finally, the natural interpretation of the class c include all identifiers that are more specific or equal to c, i.e., $[\![c]\!]^{\downarrow}_g = \{i \mid i \in \mathcal{I} \wedge i \preceq c\}$. The natural interpretation of a predicate p, $[\![p]\!]^{\downarrow}_g$, is the same as the inherited interpretation of p.

Example 2 Let us give some examples of the above-defined interpretations using the graph *Simple* presented in Fig. 2. The ordinary interpretation of the class person is $[\![person]\!]_g = \{\}$ since there are no members of the class person. The ordinary interpretation of the class scientist is $[\![scientist]\!]_g = \{leibniz, goedel\}$.

The inherited interpretation of the class person is $[\![person]\!]^*_g = \{leibniz, goedel, plato\}$ since it includes the interpretations of classes scientist and philosopher.

Finally, the natural interpretation of the class person is $[\![person]\!]^{\downarrow}_g = \{person, scientist, philosopher, leibniz, goedel, plato\}$. □

The partial ordering relationship $\preceq$ and the inherited interpretation function $[\![]\!]^*_g$ are *consistent*. The relationship $\preceq$ between two class identifiers, say $c_1 \preceq c_2$, implies subsumption

of the inherited interpretations of classes: $[\![c_1]\!]^*_g \subseteq [\![c_2]\!]^*_g$. The same holds also for the natural interpretation: $c_1 \preceq c_2$ implies $[\![c_1]\!]^{\downarrow}_g \subseteq [\![c_2]\!]^{\downarrow}_g$.

2.3 Triples and graphs

The set of all triples from a graph $g \in G$ is referred to as $\mathcal{T}$. We differentiate among the ground triples, abstract triples, and triple types. The *ground triples*, denoted as $\mathcal{T}_g$, can only include individual identifiers and predicates. The *abstract triples*, denoted as $\mathcal{T}_a$, include at least one ground identifier and at least one class identifier. The triples that include solely classes and predicates are called the *triple types*, denoted by the set $\mathcal{T}_t$. Note that $\mathcal{T} = \mathcal{T}_g \cup \mathcal{T}_a \cup \mathcal{T}_t$.

Example 3 The triple (plato,wasBornIn,athens) is a ground triple, and its type, the triple (person,wasBornIn,location), is a triple type. An example of an abstract triple is the triple (plato,ref:type,philosopher). The abstract triples define the relations among the ground and type identifiers. □

The partial ordering of triples $(\mathcal{T}, \preceq)$ is defined on the basis of the poset of identifiers $(\mathcal{I}, \preceq)$. Let $t_1 = (s_1, p_1, o_1)$ and $t_2 = (s_2, p_2, o_2)$ such that $t_1, t_2 \in \mathcal{T}_t$. Triple t_1 is more specific than or equal to triple t_2, denoted as $t_1 \preceq t_2$, iff $s_1 \preceq s_2$, $p_1 \preceq p_2$ and $o_1 \preceq o_2$.

If the poset of triples $(\mathcal{T}, \preceq)$ is restricted to the triple types from $\mathcal{T}_t$, then the relation $\preceq$ stands for a *sub-type relation* [22]. A triple type t_1 is a sub-type of t_2, i.e., $t_1 \preceq t_2$, if their components are in the sub-type relation $\preceq$.

A *triple type* is a triple (s_t, p_t, o_t) where the components s_t, p_t, and o_t are type identifiers such that $s_t, o_t \in \mathcal{I}_c$ and $p_t \in \mathcal{I}_p$. Apart from the syntactic constraints, a triple type must also meet some semantic constraints. A triple type must be *valid* with respect to the stored schema of a KG. A valid triple type is either a stored triple type, or a triple type that is related to some stored triple type using the relation $\preceq$. First, a stored triple type $t = (s_t, p_t, o_t)$ is a triple that includes a predicate p_t, a subject s_t that is a domain class of p_t, and an object s_t that is a range class of p_t. Second, a triple type t is related to a stored triple type t_t if $t \preceq t_t$ or $t \succeq t_t$.

Example 4 The stored schema graph of the KG *Simple* is presented in Fig. 2. The triple types from the schema graph include (person,wasBornIn,location), (person,influences,person) and (philosopher,hasAge,integer).

The triple types, which are either sub-types or super-types of some stored triple type, are (scientist,wasBornIn,location), (philosopher,wasBornIn,location), (scientist,influences,-scientist), etc. The triple (scientist,wasBornIn,location), for example, is a valid triple type because of the stored triple type (person,wasBornIn,location) where the class person is a super-class of the class scientist. Therefore, (scientist,wasBornIn,location) $\preceq$ (person,-wasBornIn,location). □

We can now define the interpretations of a triple type $t = (s_t, p_t, o_t)$ on the basis of the interpretations of the type identifiers. The *ordinary interpretation function* $[\![t]\!]_g$ maps a triple type t to the set of ground triples $(s, p, o) \in \mathcal{T}_g$, called the members of t, such that $p = p_t$, $s \in [\![s_t]\!]_g$, and $o \in [\![o_t]\!]_g$. The *inherited interpretation function* $[\![t]\!]^*_g$ maps a triple type t to the set of ground triples $(s, p, o) \in \mathcal{T}_g$, called the instances of t, such that $p \in [\![p_t]\!]^*_g$, $s \in [\![s_t]\!]^*_g$, and $o \in [\![o_t]\!]^*_g$. Finally, the *natural interpretation function* $[\![t]\!]^{\downarrow}_g$ maps t to a set of triples $(s, p, o) \in \mathcal{T}$ such that that $(s, p, o) \preceq t$.

Example 5 We now present an example of the ordinary, inherited, and natural interpretation functions, $[\![]\!]_g$, $[\![]\!]_g^*$ and $[\![]\!]_g^{\downarrow}$, within the context of the example graph *Simple* from Fig. 2. The ordinary, inherited, and natural interpretations of the type t =(person,wasBornIn,location) are as follows.

$$
\begin{aligned}
[\![t]\!]_g &= \{\} \\
[\![t]\!]_g^* &= \{(\text{plato,wasBornIn,athens}), (\text{leibniz,wasBornIn,leipzig}), \\
&\quad (\text{goedel,wasBornIn,brno})\} \\
[\![t]\!]_g^{\downarrow} &= \{(\text{person,wasBornIn,location}), (\text{scientist,wasBornIn,location}), \\
&\quad (\text{philosopher,wasBornIn,location}), (\text{plato,wasBornIn,athens}), \\
&\quad (\text{leibniz,wasBornIn,leipzig}), (\text{goedel,wasBornIn,brno})\}
\end{aligned}
$$

The ordinary interpretation of t is the empty set since the class person does not have any members. The inherited interpretation includes the instances of the types (scientist,wasBornIn,location) and (philosopher,wasBornIn,location), both subtypes of (person,wasBornIn,location). The natural interpretation of t includes all triples from Simple that are more specific than or equal to t. □

The consistency of the partially ordered set of triples with the subsumption hierarchy of the interpretations of the triple types can be shown similarly to the consistency of identifiers. The semantic relationship $\preceq$ among triple types implies the subsumption among the inherited and natural interpretations of types. The space of triples from a given KG is ordered into the subsumption hierarchy that reflects the partial ordering of triple types.

The set of all stored triple types together with the triples that define the ontologies of classes and predicates form the *stored schema graph* (abbr. ssg). The ontologies of classes and predicates are defined using the predicates rdfs:subClassOf and rdfs:subPropertyOf, respectively. The stored schema graph defines the structure of ground triples in g. Each ground triple is an instance of at least one triple type $t_s \in ssg$.

Example 6 The stored schema of the KG *Simple* is represented in Fig. 2. The first part of the stored schema graph is derived from the RDF-Schema model by replacing the triples using predicates rdfs:domain and rdfs:range with the triple types. The stored triple types included in *Simple* are (person,wasBornIn,location), (person,influences,person), (philosopher,hasAge,-integer) and (owl:Thing,subjectStartRelation,owl:Thing).

Second, the triple types that define the types of properties include (wasBornIn,rdf:-type,rdf:Property), (influences,rdf:type,rdf:Property), (hasAge,rdf:type,rdf:Property), and (subjectStartRelation,rdf:type,rdf:Property).

Finally, the triple types that define the ontology of classes and properties in *Simple* are (location,rdfs:subClassOf,Owl:Thing), (person,rdfs:subClassOf,Owl:Thing), (xsd:integer,-rdfs:subClassOf,Owl:Thing), (rdf:Property,rdfs:subClassOf,Owl:Thing), (scientist,rdfs:sub-ClassOf,person), (philosopher,rdfs:subClassOf,person),
(philosopher,rdfs:subClassOf,person), and (wasBornIn,rdfs:subPropertyOf,subject-StartRelation). □

A *complete schema graph* csg of g includes the stored schema graph of g as well as all triple types that are either more general or more specific than the stored triple types. A *schema graph* sg can now be characterized as $ssg \subseteq sg \subseteq csg$. Every schema graph includes at least the stored schema graph. Additionally, it may include a subset of the complete schema graph that is not part of ssg. These triple types can be either more specific or more general than the stored triple types. The triple types selected from csg/ssg should appear as types of triple patterns so that the statistics of these triple types is useful.

2.4 Triple patterns

We assumed the existence of a set of variables V. A *triple pattern* $(s, p, o) \in (S \cup V) \times (P \cup V) \times (O \cup V)$ is a triple that includes at least one variable.

To present the semantics of triple patterns, we define an alternative way of accessing the components of triples. The components of triple $t = (s, p, o)$ can be accessed, similarly to the elements in an array, as $t[1] = s$, $t[2] = p$ and $t[3] = o$.

Let $tp = (s, p, o)$ be a triple pattern and the set $tp_v \subseteq \{1, 2, 3\}$ be a set of indices of components that are variables, i.e., $\forall j \in tp_v : tp[j] \in V$. The interpretation of a triple pattern tp is the set of triples $t \in g$ such that t includes a value in place of variables indexed by the elements of tp_v, and has the values of other components equal to the corresponding tp components. Formally, $[\![tp]\!]_g = \{t \mid t \in g \wedge \forall j \in \{1, 2, 3\} \setminus tp_v : t[j] = tp[j]\}$.

The type of triple pattern is a triple type such that the interpretation of the triple type subsumes the interpretation of the triple pattern. Let $tp = (s, p, o)$ be a triple pattern and $t_{tp} \in \mathcal{T}_t$ be a triple type. The triple type t_{tp} is the type of tp iff $[\![tp]\!]_g \subseteq [\![t_{tp}]\!]_g^*$. Finally, the minimal type of a triple pattern is a type $t_{min} \in \mathcal{T}_t$ such that $\forall t'_{tp} \in \mathcal{T}_t (t_{min} \preceq t'_{tp})$ and $[\![tp]\!]_g \subseteq [\![t_{min}]\!]_g^*$.

Note that we only specify the semantic conditions for the definition of the minimal type of a triple pattern. The typing rules and the algorithms for the derivation of the type of a triple pattern are presented in [23].

3 Computing statistics

The statistical index is implemented as a dictionary where keys represent triple types, and the values represent the statistical information for the given keys. For instance, the index entry for the triple type (person,wasBornIn,location) represents the statistical information about the ground triples that have the instance of a person as the first component, the property wasBornIn as the second component, and the instance of location as the third component.

The main procedure for the computation of the statistics of a KG is presented as Algorithm 1. The statistic is computed for each triple t from a given knowledge graph. The function STATISTICS- TRIPLE(t) represents one of three functions: STATISTICS- STORED(t), STATISTICS- ALL(t) and STATISTICS- LEVELS(t, l, u), which are presented in detail in the sequel.

Algorithm 1 Procedure COMPUTE- STATISTICS(kg : knowledge-graph).

```
1: procedure COMPUTE- STATISTICS(kg : knowlege-graph)
2:     for all t ∈ kg do
3:         STATISTICS- TRIPLE(t)
4:     end for
5: end procedure
```

In the following Sections 3.2-3.4, we present the algorithms for the computation of the statistics of KG. Each of the algorithms computes a *schema graph* that is used as a basis for the computation of the statistics of a KG. Since a schema graph can include a large number of triple types that are not useful in computing the types of triple patterns from the query workload, we restrict the schema graph to include only the selected classes that can

participate in a schema graph. For example, the classes from the Wordnet taxonomy [24] usually define the level of a KG where most of the predicates are defined. The function SG-CLASS(c: CLASS): BOOL is provided to return $true$ if the class c is selected to be part of the schema graph, and $false$ otherwise. Let us now introduce the algorithms.

The procedure STATISTICS- STORED, which computes the statistics for the stored schema graph is described in Section 3.2. The second procedure STATISTICS- ALL, presented in Section 3.3, computes statistics for all legal triple types of a given KG. The problem with storing the statistics for all valid triple types is, despite the restricted use of classes in the schema graph, the size of such a set. To provide control of the size of the schema graph, the third procedure STATISTICS- LEVELS computes the statistics for the triple types included in a strip around the stored schema graph. This procedure is presented in Section 3.4.

Before the presentation of the algorithms, let us present some motivations for the computation of the statistics of types by means of examples in the followining Section 3.1.

3.1 Motivations

The following Example 7 demonstrates that typing a triple pattern restricts the set of triples that need to be addressed by the query.

Example 7 The triple pattern (?x,wasBornIn,?y) has the type (person,wasBornIn,location) since we only have one schema triple with the predicate wasBornIn (see Fig. 2). Therefore, solely the instances of (person,wasBornIn,location) are addressed by the triple pattern. □

Example 8 shows that it is beneficial to store the statistics not only for the triple types from the stored schema graph but also for more specific triple types. This allows a more precise estimation of the size of the triple patterns that form a query.

Example 8 A join can be defined by triple patterns (?x,wasBornIn,?y) and (?x,hasAge,?z). T he types of triple patterns are (person,wasBornIn,location) and (philosopher,hasAge,integer), respectively.

We see that the variable ?x has the type person as well as philosopher. The type of ?x is (person AND philosopher), which equals the type philosopher. The logical AND of two types requires that the selected instances are from the interpretations of both types (classes). Therefore, we can safely use the triple type (philosopher,wasBornIn,location) as the type of (?x,wasBornIn,?y). Instead of accessing triples of the type (person,wasBornIn,location) we can only access the triples of the type (philosopher,wasBornIn,location). □

Finally, the following example shows the case when a predicate has two possible triple types from which one is selected when merging the types for a join operation [25].

Example 9 Suppose the dataset *Simple* has one additional class book together with the triple type (book,influences,book). Then the triple pattern (?x,influences,?y) has the type (person,influences,person) OR (book,influences,book). A given ground triple with the predicate influences can have one of the two types.

The join of the triple patterns (?x,influences,?y) and (?x,hasAge,?z) has the type {(philosopher,influences,person), (philosopher,hasAge,xsd:integer)}. The variable ?x can initially have two possible types: person OR book. When merging the schema triples as the consequence of a join, we get as the type of ?x the expression (philosopher AND (person OR book)). This expression is reduced to the type philosopher since classes philosopher and book are not related by the sub-class hierarchy and do not have common instances [23].

Hence, the triple pattern (?x,influences,?y) can, in the context of above query, only access the triples that are the instances of (philosopher,influences,person) and no instances of (book,influences,book) or (person\philosopher,influences,person). □

3.2 Statistics of the stored schema graph

The first procedure for computing the statistics is based on the stored schema graph, which is a part of the knowledge graph. The stored schema graph includes all triple types stored in a KG that contain solely the pre-selected classes as the domains and ranges of the predicates. The membership test on the set of pre-selected classes is implemented by a function SG-CLASS(c: CLASS): BOOL.

Algorithm 2 Statistics of the stored schema graph.

```
1: function STATISTICS- STORED(t = (s, p, o) : triple)
2:     g_p ← {p} ∪ {c_p | (p, rdfs:subPropertyOf+, c_p) ∈ g}
3:     for all p_g ∈ g_p do
4:         d_p ← {t_s | (p_g, rdfs:domain, t_s) ∈ g ∧ SG- CLASS(t_s)}
5:         r_p ← {t_o | (p_g, rdfs:range, t_o) ∈ g ∧ SG- CLASS(t_o)}
6:         for all t_s ∈ d_p, t_o ∈ r_p do
7:             UPDATE- STATISTICS((t_s, p_g, t_o), t)
8:         end for
9:     end for
10: end function
```

Let us now describe the procedure STATISTICS- STORED presented as Algorithm 2. We assume that $t = (s, p, o)$ is an arbitrary ground triple from graph $g \in G$. First, the algorithm initializes set g_p in the line 2 to include the element p, and the transitive closure of $\{p\}$ computed with respect to the relationship rdfs:subPropertyOf to obtain all more general predicates of p. Note that '+' denotes one or more applications of the predicate rdfs:subPropertyOf.

After the set g_p is computed, the domains and the ranges of each particular property $p_g \in g_p$ are retrieved from a KG in lines 4-5. Note that the domains and ranges can include solely the classes selected by the function SG- CLASS. After we have computed all the sets holding the types of t's components, the triple types are enumerated by taking property p_g and a pair of the domain and range classes t_s and t_o of p_g. The statistics are updated using the procedure UPDATE- STATISTICS for each of the generated triple types in line 7. A detailed description of the procedure UPDATE- STATISTICS is given in Section 4.4.

3.3 Statistics of a complete schema graph

A *complete schema graph* of a knowledge graph includes all valid triple types of ground triples, derived from the already selected stored schema graph and the stored typings of ground triples. The complete set of triple types of a given ground triple $t = (s, p, o)$ is obtained by first computing the sets of all valid types of the components s and o, as well as the set including p and all super-predicates of p. Finally, the computed sets are used to enumerate a complete set of the triple types of t.

Let us now discuss the details of computing a complete set of triple types selected to have statistics updated for a given ground triple $t = (s, p, o)$. First, the predicates that can appear in valid types of t include p together with all predicates that are more general than p. This set of predicates is referred to as g_p. A predicate that is more specific than p can not serve as a predicate of a type of the triple t. For example, we can judge from the stored type $(person, wasBornIn, location)$ that $(person, subjectStartRelation, location)$ also holds, but not vice versa.

The types of the subject s and object o are treated differently from the predicates. The sets of all valid types (classes) of s and o are computed and stored in the sets g_s and g_o, respectively. Let's focus on the elements of the set g_s while g_o is treated in the same way. The required types (classes) of s are the domains of the predicate p. Assume they are stored in a set d_p. The classes $c_s \in g_s$, which are more general than some class $c \in d_p$, are valid types of s since c is a valid type of s and $c_s \succeq c$. On the other hand, the classes $c_s \in g_s$ that are more specific than some $c \in d_p$ are not valid types of s as the consequence of $c_p \preceq c$ and c being the type of s. However, c_s is the valid type of s since it is computed by generalizing the stored types (obtained via rdf:type) of s. Consequently, the valid types of t are all types (t_s, t_p, t_o) such that $t_s \in g_s$, $t_p \in g_p$, and $t_o \in g_o$.

Algorithm 3 Statistics of a complete schema graph.

```
1: function STATISTICS- ALL(t = (s, p, o) : triple)
2:     g_s ← {t_s | (s, rdf:type, t_s) ∈ g}
3:     g_o ← {t_o | (o, rdf:type, t_o) ∈ g}
4:     g_s ← g_s ∪ {c'_s | ∃c_s ∈ g_s((c_s, rdfs:subClassOf+, c'_s) ∈ g)}
5:     g_p ← {p} ∪ {c_p | (p, rdfs:subPropertyOf+, c_p) ∈ g}
6:     g_o ← g_o ∪ {c'_o | ∃c_o ∈ g_o((c_o, rdfs:subClassOf+, c'_o) ∈ g)}
7:     for all c_p ∈ g_p do
8:         d_p ← {c | (c_p, rdfs:domain, c) ∈ g ∧ SG- CLASS(c)}
9:         r_p ← {c | (c_p, rdfs:range, c) ∈ g ∧ SG- CLASS(c)}
10:        s_s ← {c_s | c_s ∈ g_s ∧ SG- CLASS(c_s) ∧ ∃c ∈ d_p(c_s ⪯ c ∨ c_s ⪰ c)}
11:        s_o ← {c_o | c_o ∈ g_o ∧ SG- CLASS(c_o) ∧ ∃c ∈ r_p(c_o ⪯ c ∨ c_o ⪰ c)}
12:        for all c_s ∈ s_s, c_o ∈ s_o do
13:            UPDATE- STATISTICS((c_s, c_p, c_o), t)
14:        end for
15:    end for
16: end function
```

The procedure for the computation of statistics for a triple $t = (s, p, o)$ is presented in Algorithm 3. The procedure STATISTICS- ALL computes in lines 1-6 the sets of types g_s, g_p and g_o of the triple components s, p and o, respectively. The set of types g_s is, in line 2, initialized with the set of the stored types of s. The set g_s is then closed by means of the relationship rdfs:subClassOf in line 4. Set g_o is computed in the same way as set g_s. Note that the sets of classes g_s and g_o are not restricted in any way. The set of predicates g_p is computed differently. The set g_p includes the closure of the set $\{p\}$ by using the relationship rdfs:subPropertyOf, in line 5. Predicates play a similar role to classes in knowledge representation systems [26].

The FOR loop in lines 7-15 goes through all the predicates c_p from the set g_p. In lines 8-9, the domain and range classes of c_p are gathered in the sets d_p and r_p, respectively. The classes from d_p and r_p are restricted to the pre-selected classes by using the function SG- CLASS.

Further, in lines 10-11, we select from g_s and g_o only those classes that are related by $\preceq$ to some classes from d_p and r_p, respectively. Those classes from g_s and g_o that are not backed by the predicate p's legal domains and ranges are removed. The removed classes represent an error in the stored typing of s or o. The classes in g_s and g_o are also filtered by using the function SG- CLASS resulting in the sets s_s and s_o, respectively.

Finally, in lines 12-14, the triple types of t are enumerated by using sets s_s, s_p, and a predicate $c_p \in g_p$. Since s_s and s_o include all valid classes of s and o, and the predicates $c_p \in g_p$ are valid, then the complete set of t's types is $s_s \times g_p \times s_o$. The procedure UPDATE-STATISTICS is applied for each type $t_t \in s_s \times g_p \times s_o$.

3.4 Statistics of a strip around the stored schema graph

In the procedure STATISTICS- STORED, we did not compute the statistics of the triple types that are either more specific or more general than the triple types from the stored schema graph. For instance, we have the statistics for the triple type (person,wasBornIn,location) since this triple type is part of the stored schema graph, but we do not have the statistics for the triple types (scientist,wasBornIn,location) and (philosopher,wasBornIn,location).

On the other hand, the procedure STATISTICS- ALL updates statistics for *all* valid triple types, whose interpretation includes the ground triple t. The number of all valid triple types may be much too large for a KG with a rich conceptual schema. However, the interesting triple types are around the stored schema graph, i.e., some levels of more specific and some levels of more general triple types.

Algorithm 4 Statistics of a strip around the stored schema graph.

1: **function** STATISTICS- LEVELS($t = (s, p, o)$: triple, l, u : integer)
2: $g_s \leftarrow \{t_s \mid (s, \text{rdf:type}, t_s) \in g\}$
3: $g_o \leftarrow \{t_o \mid (o, \text{rdf:type}, t_o) \in g\}$
4: $g_s \leftarrow g_s \cup \{c'_s \mid c_s \in g_s \wedge (c_s, \text{rdfs:subClassOf+}, c'_s) \in g\}$
5: $g_p \leftarrow \{p\} \cup \{t_p \mid (p, \text{rdfs:subPropertyOf+}, t_p) \in g \wedge \text{DIST}(p, t_p) \leq u\}$
6: $g_o \leftarrow g_o \cup \{c'_o \mid c_o \in g_o \wedge (c_o, \text{rdfs:subClassOf+}, c'_o) \in g\}$
7: **for all** $c_p \in g_p$ **do**
8: $d_p \leftarrow \{c \mid (c_p, \text{rdfs:domain}, c) \in g \wedge \text{SG- CLASS}(c)\}$
9: $r_p \leftarrow \{c \mid (c_p, \text{rdfs:range}, c) \in g \wedge \text{SG- CLASS}(c)\}$
10: $s_s \leftarrow \{c_s \mid c_s \in g_s \wedge \text{SG- CLASS}(c_s) \wedge \exists c \in d_p((c_s \preceq c \wedge \text{DIST}(c_s, c) \leq l) \vee$
11: $(c_s \succeq c \wedge \text{DIST}(c_s, c) \leq u))\}$
12: $s_o \leftarrow \{c_o \mid c_o \in g_o \wedge \text{SG- CLASS}(c_o) \wedge \exists c \in r_p(((c_o \preceq c \wedge \text{DIST}(c_o, c) \leq l) \vee$
13: $(c_o \succeq c \wedge \text{DIST}(c_o, c) \leq u))\}$
14: **for all** $c_s \in s_s, c_o \in s_o$ **do**
15: UPDATE- STATISTICS($(c_s, c_p, c_o), t$)
16: **end for**
17: **end for**
18: **end function**

The algorithm STATISTICS- LEVELS is very similar to the algorithm STATISTICS- ALL. In fact, the only difference between the algorithms is in the selection of the sets of classes and predicates s_s, g_p, and s_o that contain the types of s, p, and o, respectively. In the case of the algorithm STATISTICS- LEVELS, the sets g_s, g_p, and g_o are filtered to include only the

classes with limited distance to the stored schema graph. Let us present the computation of the distance between the schema triples.

First, we define the distance between classes. Suppose we have classes c and c'. The function DIST(c, c') computes the number of edges labeled rdfs:subClassOf on the path from c to c', if such a path exists, or returns ∞ if there is no such path. The distance between two predicates is defined in the same way, but the relation rdfs:subPropertyOf is used instead of rdfs:subClassOf. We can now define the distance between triple types. The distance between the triple types $t = (s_t, p_t, o_t)$ and $t' = (s'_t, p'_t, o'_t)$ is smaller or equal n if the distances between all components of t and t' are smaller than or equal to n.

Let us now comment on the parts of the algorithm that are different from the procedure STATISTICS- ALL. The set of predicates g_p to be used as the basis for the computation of the types of $t = (s, p, o)$ is computed in line 5. The set g_p includes p and all the predicates that are more general than p and have a distance to p, which is less or equal to the upper bound u.

The sets g_s and g_o, which include all types of s and o, are filtered in lines 10-13. The result of filtering g_s is a set of classes s_s which includes classes $c_s \in g_s$ such that their distance to at least one class $c \in d_p$ is less or equal to l if $c_s \preceq c$, and less than or equal to u if $c_s \succeq c$. The set s_o is computed in the same way as s_s.

3.4.1 Retrieving statistics of a triple type

Let us suppose that we have the statistical index of a knowledge graph computed using the algorithm STATISTICS- LEVELS, and, that the statistics are based on the triple types from u levels above the stored schema graph and l levels below the stored schema. The statistical index is implemented as a key-value dictionary where the key is a triple type, and the value represents the statistical entry for a given key. We call the set of triple types that represent the keys of the statistical index the *schema of statistics* S.

When we want to retrieve the statistical data for a triple type $t = (s, p, o)$, we have two possible scenarios. The first is that the triple type t is stored in the dictionary. The statistics are, in this case, directly retrieved from the statistical index. This is the expected scenario, i.e., we expect that the statistical index includes all triple types that can represent the types of triple patterns. The second scenario covers cases when the triple types t are either above or below S. Here, the statistics of t have to be approximated from the data computed for the triple types S that are included in the statistics.

The triple type t is *above* the schema of the statistics S when there is at least one $t' \in S$: $t \succ t'$, and for all $t' \in S$ either $t \succ t'$, or t and t' are not related. The relationship *below* can be defined similarly to the definition of the relationship *above*. The triple type t is *below* the schema of the statistics S when there is at least one $t' \in S : t \prec t'$, and for all $t' \in S$ either $t \prec t'$ or t and t' are not related.

The retrieval of the statistical entry for a given triple type $t = (s, p, o)$ is presented in Algorithm 5. The first part of the algorithm, stated in lines 3-4, retrieves the statistics for the triple types t that are stored in the statistical index.

The computation of the statistics, when the triple type t is above the schema of the statistics, is presented in lines 5-11. The set b_u includes the upper bounds t_u from S such that $t_u \preceq t$, i.e., the triple types $t_u \in S : t_u \preceq t$ and there is no other $t'_u \in S$ which would be between t_u and t. Therefore, b_u contains unrelated triple types that are the specializations of t. We approximate the statistics of a schema triple in lines 9-11 by summing the statistics of the triple types from b_u. We may consequently make mistakes because we count triples that belong to more than one triple type from b_u multiple times.

Algorithm 5 Function RETRIEVE- STATISTICS(t : triple) $\rightarrow$ integer.

1: **function** RETRIEVE- STATISTICS((s, p, o) : triple) $\rightarrow$ integer
2: st $\leftarrow 0$
3: **if** EXISTS- STATISTICS((s, p, o)) **then**
4: **return** GET- STATISTICS((s, p, o))
5: **else if** ABOVE- STAT- SCHEMA((s, p, o), S) **then**
6: $b_u \leftarrow \{(s_u, p_u, o_u) \mid (s_u, p_u, o_u) \in S \wedge (s_u, p_u, o_u) \preceq (s, p, o) \wedge$
7: $\nexists (s'_u, p'_u, o'_u) \in S\ ((s_u, p_u, o_u) \preceq (s'_u, p'_u, o'_u) \wedge$
8: $(s'_u, p'_u, o'_u) \preceq (s, p, o))\}$
9: **for all** $t \in b_u$ **do**
10: st $\leftarrow$ st + GET- STATISTICS(t)
11: **end for**
12: **else if** BELOW- STAT- SCHEMA((s, p, o), S) **then**
13: $b_l \leftarrow \{(s_l, p_l, o_l) \mid (s_l, p_l, o_l) \in S \wedge (s, p, o) \preceq (s_l, p_l, o_l) \wedge$
14: $\nexists (s'_l, p'_l, o'_l) \in S\ ((s'_l, p'_l, o'_l) \preceq (s_l, p_l, o_l) \wedge$
15: $(s, p, o) \preceq (s'_l, p'_l, o'_l))\}$
16: **for all** $t \in b_l$ **do**
17: st $\leftarrow$ MIN(st,GET- STATISTICS(t))
18: **end for**
19: **end if**
20: **return** st
21: **end function**

The computation of the statistics when t is below the schema of the statistics S is presented in lines 12-19. It is defined similarly to the case that t is above S. The set of the triple types b_l is computed by selecting the most specific triple types $t_l \in S : t \preceq t_l$ that are not related by the relationship $\preceq$. The statistics of the parameter triple type t are approximated by using the statistics of the triple type $t_l \in b_l$ with the smallest interpretation. Indeed, the instances of t are also instances of $t_l \in b_l$. Therefore, they represent the intersection of the interpretations of all $t_l \in b_l$. Hence, selecting t_l with the smallest interpretation provides an upper bound of the size of the interpretation of t. The actual size of t may be smaller.

Example 10 Suppose that we have computed the statistics for the stored schema of *Simple* by using the procedure STATISTICS- STORED. Observe that the same schema would be obtained by using the procedure STATISTICS- LEVELS if the number of levels above and below the stored schema is zero. The stored triple types, which represent the types of ground triples from Simple are (person,wasBornIn,location), (person,influences,person) and (philosopher,hasAge,xsd:integer) and (owl:Thing,subjectStartRelation,owl:Thing).

Assume the statistical index SI solely includes the number of triples for the given triple types.[2] The entries of the statistical index for the stored triple types are SI{(person,wasBornIn, location)}= 3, SI{(person,influences, person)}= 2, SI{(philosopher,hasAge,xsd:integer)}= 2 and SI{(owl:Thing,subjectStartRelation,owl:Thing)}= 3.

In case we need the statistics of a triple type, that is included in the index, it can simply be retrieved. For example, the number of instances of (person,wasBornIn,location) is 3. However, in the case that we need the statistics for a triple type, that is either more specific or more general than the keys of the statistical index, then the statistics are approximated as presented in Algorithm 5.

[2] The complete treatment of the statistics for all types of keys is presented in detail in the following Section 4.

For example, the triple type (scientist,wasBornIn,location) is more specific than (person,wasBornIn, location); therefore, as described in Algorithm 5, (scientist,wasBornIn, location) is below the schema of statistics. In this case, the statistics of (scientist,wasBornIn,location) are approximated by the statistics of the least more general triple types from the computed schema of statistics. The triple type (person,wasBornIn, location) is the only candidate for the closest more general schema triple of (scientist,wasBornIn,location). Hence, the statistics are approximated by using the index entry for (person,wasBornIn,location). □

4 On counting keys

Let $g \in G$ be a knowledge graph stored in a triple store, and let $t = (s, p, o)$ be a triple. All algorithms for the computation of the statistics of g presented in the previous section enumerate a set of triple types S such that for each triple type $t_s \in S$ the interpretation $[\![t_s]\!]^*_g$ includes the triple t. For each of the computed triple types $t_s \in S$ the procedure UPDATE-STATISTICS() updates the statistics of the key t_s in the statistical index as the consequence of the insertion of the triple t into the graph g.

The triple $t = (s, p, o)$ includes seven keys: s, p, o, sp, so, po and spo. These keys are the targets to be queried in the triple patterns. Let us give an example of a triple pattern to present the data needed for the estimation of the size of the triple pattern interpretation.

Example 11 Let $t_p = (?x,\text{wasBornIn},\text{athens})$ be a triple pattern including the variable $?x$ in the position of s. To compute the number of the triples in $[\![t_p]\!]^*_g$ we first need to know the type of the triple pattern t_p, which can be obtained by retrieving the types of the domain and range of the predicate wasBornIn from the graph g. The type of t_p is the triple type (person,wasBornIn,location).

The number of triples from $[\![t_p]\!]^*_g$ can be computed by first computing the size of the interpretation $[\![(\text{person},\text{wasBornIn},\text{location})]\!]^*_g$, i.e., the number of *all* instances of (person,-wasBornIn,location). Afterward, we need to know the number of different pairs of instances of the predicate wasBornIn and the objects of type location. The type of the key composed of the predicate wasBornIn, and an instance of the class location is called *key type*—it is written as (person,wasBornIn,location). The keys and the types of keys are defined formally in the sequel.

The size of $[\![(?x,\text{wasBornIn},\text{athens})]\!]^*_g$ can be now estimated as the number of *all possible* triples from $[\![(\text{person},\text{wasBornIn},\text{location})]\!]^*_g$ divided by the number of *different* keys of the type (person,wasBornIn,location). In this way, we obtain the estimation of the number of triples that have p=wasBornIn, o=athens, and, arbitrary value of s. □

4.1 Keys and key types

Let us now define the concepts of the key and the key type. A *key* is a triple (s_k, p_k, o_k) composed of $s_k \in S \cup \{\_\}$, $p_k \in P \cup \{\_\}$ and $o_k \in O \cup \{\_\}$. Note that we use the notation presented in Section 2. The symbol "_" denotes a missing component.

A *key type* is a triple type that includes the types of the key components as well as the *underlined types* of components that are not parts of keys. However, the underlined types can restrict keys.[3] Formally, a key type of a triple (s_k, p_k, o_k) is a triple type (s_t, p_t, o_t), such that $s_k \in [\![s_t]\!]^*_g$, $p_k \in [\![p_t]\!]^*_g$ and $o_k \in [\![o_t]\!]^*_g$ for all the components s_k, p_k and o_k that are not "_".

[3] See bound/unbound keys in Section 4.3.

The underlined types of the missing components are computed from the stored schema of a knowledge graph. The type inference algorithm is not the subject of the research presented in this paper.

The *complete* type of the key represents the key type where the underlines are omitted, i.e., the complete type of the key is a bare triple type.

Example 12 The examples of the above-defined concepts are presented here. The triple (_,wasBornIn,athens) is a key with the components p=wasBornIn and o=athens, while the component s does not contain a value. The triple type (person,wasBornIn,location) is the type of the key (_,wasBornIn,athens). Finally, the complete type of (person,wasBornIn,location) is the triple type (person,wasBornIn,location). □

4.2 Computing statistics for keys

Let us now present the procedure for updating the statistical index for a given triple $t = (s, p, o)$ and the corresponding triple type $t_t = (t_s, t_p, t_o)$. In order to store the statistics of a triple t of type t_t, we split t_t into the seven key types: (t_s, t_p, t_o), $(\underline{t_s}, t_p, \underline{t_o})$, ..., $(\underline{t_s}, t_p, t_o)$, (t_s, t_p, t_o). Furthermore, the triple t is split into the seven keys: $(s, \_, \_)$, $(\_, p, \_)$, ..., $(\_, p, o)$, and (s, p, o). The statistics is updated for each of the selected *key types*.

There is more than one way of counting the instances of a given *key type*. For the following discussion, we select an example of the key type, say $t_k = (\underline{t_s}, t_p, t_o)$, and consider all the ways we can count the key $(\_, p, o)$ for a given key type t_k. The conclusions that we draw in the following paragraphs are general in the sense that they hold for all key types.

1. Firstly, we can either count all keys, including the repeating keys or, we can count only the different keys. We denote these two options by using the descriptive parameters *all* or *distinct*, respectively.
2. Secondly, we can either count triples of the type (t_s, t_p, t_o), or, the triples of the type $(\top, t_p, t_o)$. In the first case we call counting *bound*, and in the second case, we call it *unbound*. A more detailed description of bound and unbound ways of counting is presented in the following Section 4.3.

The above-stated choices are specified as parameters of the generic procedure UPDATE-KEYTYPE(all|distinct,bound|unbound,t_k,t). The first parameter specifies if we count all or distinct triples. The second parameter defines the domain of counters, which is either restricted by a given complete type t_t of key type t_k, or unrestricted, i.e., the underlined component is not bound by any type. The third parameter is a key type t_k, and, finally, the fourth parameter represents the triple t.

Note that we do not need to pass the triple t to the procedure in the case that we count all keys since one is always added to the current statistics of a given key type t_k regardless of the value of t. In the case that we count distinct keys, we need the parameter triple t to extract the key to be inserted in the statistical index for a given key type t_k.

4.3 Counting bound/unbound

We use the example of a key type $t_k = (\underline{t_s}, t_p, t_o)$ to present some details about counting bound and unbound. The keys that are counted for the key type t_k are of the form $(\_, p, o)$. In the case that solely the instances of the type (t_s, t_p, t_o) are used, i.e., the *bound* case, then we do not count instances of (t'_s, t_r, t_o) where t'_s is not related to t_s. In the case that instances

of type $(\top, t_p, t_o)$ are used, that is the *unbound* case, then s in $t = (s, p, o) \in g$ can be of any type $\top$. Note that an unbound case is a necessity if the complete type of a triple pattern is not known.

Example 13 Let us give an example of the bound and unbound counting. The restriction of the keys by the underlined types of components in the case of bound counting requires each key to be part of some triple, which is included in the interpretation of the complete type of the key. For example, the key (_,wasBornIn,athens) of type (person,wasBornIn,location) is included in the triple (plato,wasBornIn,athens) that is an element of the interpretation $[\![(\text{person,wasBornIn,location})]\!]^*_g$.

Suppose that the KG from Fig. 2 contains the triple (tom,wasBornIn,paris), where tom is not of the type person but of the type cat. The triple would be taken into account when counting the instances of the key type (person,wasBornIn,location) in the unbound way. However, the triple (tom,wasBornIn,paris) is not included in $[\![(\text{person,wasBornIn,location})]\!]^*_g$ and is not taken into account when counting in a bound way. □

4.4 Procedure update-statistics

Let us now present the procedure UPDATE- STATISTICS(t_t,t) for updating the entry of the statistical index that corresponds to the key type t_t and a triple t. The procedure is presented in Algorithm 6.

Algorithm 6 Procedure UPDATE- STATISTICS(t_t:schema-triple, t : triple).

```
1: procedure UPDATE- STATISTICS(t_t: schema-triple, t : triple)
2:     for all key types t_k of t_t do
3:         UPDATE- KEYTYPE(all,unbound,t_k);
4:         UPDATE- KEYTYPE(all,bound,t_k);
5:         UPDATE- KEYTYPE(dist,unbound,t_k, t);
6:         UPDATE- KEYTYPE(dist,bound,t_k, t);
7:     end for
8: end procedure
```

The parameters of the procedure UPDATE- STATISTICS are the triple type t_t and the triple t such that t_t is a type of t. The FOR statement in line 1 generates all seven key types of the triple type t_t. The procedure UPDATE- KEYTYPE is applied to each of the generated key types with the different values of the first and the second parameters.

We have enumerated the calls of all possible types of procedure UPDATE- KEYTYPE. However, we expect that the subset of these calls will be used to compute the statistics of a knowledge graph.

5 Experimental evaluation

In this section, we present the evaluation of the algorithms for the computation of the statistics for the two example knowledge graphs. First of all, we present the experimental environment used to compute statistics. Secondly, the statistics of a simple knowledge graph introduced

in Fig. 2 is presented in Section 5.2. Finally, the experiments with the computation of the statistics of the Yago are presented in Section 5.3.

5.1 Testbed description

The algorithms for the computation of the statistics of graphs are implemented in the open-source system for querying and manipulation of graph databases *epsilon* [27]. *epsilon* is a lightweight RDF store based on Berkeley DB [28]. It can execute *basic graph-pattern queries* on datasets that include up to 1G (10^9) triples.

epsilon was primarily used as a tool for browsing ontologies. The operations implemented in *epsilon* are based on the sets of identifiers I, as they are defined in Section 2.2. The operations include computing transitive closures based on some relationship (e.g., the relationship rdfs:subClassOf), level-wise computation of transitive closures, and computing the least upper bounds of the set elements with respect to the stored ontology. These operations are used in the procedures for the computation of the RDF store statistics.

5.2 Knowledge graph *Simple*

Table 1a) describes the properties of the schema graphs computed by the three algorithms for the computation of the statistics. Each line of the table represents an evaluation of the particular algorithm on the KG *Simple*. The algorithms denoted by the keywords *ST*, *AL* and *LV* refer to the algorithms STATISTICS- STORED, STATISTICS- ALL and STATISTICS- LEVELS presented in the Sections 3.2-3.4, respectively. The running time of all the algorithms used in the experiments was below 1ms.

The columns *#ulevel* and *#llevel* represent the number of levels above and below the stored schema graph that are relevant solely for the algorithm *LV*. The columns #bound and #ubound store the number of triple types included in the schema graph of the computed statistics. The former is computed with the *bound* type of counting, and the latter with the *unbound* type of counting.

Algorithm *ST* computes the statistics solely for the stored schema graph. This algorithm can be compared to the relational approach, where the statistics are computed for each relation. Algorithm *AL* computes the statistics for all possible triple types. The number of triple types computed by this algorithm is significant, even for this small instance. In algorithm *LV*, there are only three levels of the schemata, i.e., the maximal #ulevel and #dlevel both equal 2. The statistics of the triple types that are above the stored triple types are usually computed since we are interested in having the global statistics based on the most general triple types from the schema graph. Note that the algorithm *ST* gives the same results as the algorithm *LV* with the parameters #ulevel=0 and #dlevel=0.

5.3 Knowledge graph Yago-S

In this section, we present the evaluation of the algorithms for the computation of the statistics of the Yago-S knowledge graph—we use the core of the Yago 2.0 knowledge base [29], including approximately 25M (10^6) triples. The Yago-S dataset is available, together with the working environment for the computation of the statistics, from the *epsilon* data repository [20].

Table 1 The evaluation of the Algorithms 1-3 on: **a)** *Simple* **b)** Yago-S

algorithm	#ulevel	#dlevel	#bound	#unbound
ST	–	–	63	47
AL	–	–	630	209
LV	0	0	63	47
LV	0	1	336	142
LV	0	2	462	173
LV	1	0	147	72
LV	1	1	476	175
LV	1	2	602	202
LV	2	0	161	76
LV	2	1	490	178
LV	2	2	616	205

algorithm	#ulevel	#dlevel	#bound	time-b	#unbound	time-u
ST	–	–	532	1	405	1
AL	–	–	>1M	>24	>1M	>24
LV	0	0	532	4.9	405	4.9
LV	1	0	3325	6.1	1257	5.4
LV	2	0	8673	8.1	2528	6.2
LV	3	0	12873	9.1	3400	6.8
LV	4	0	15988	10.9	3998	7.1
LV	5	0	18669	11.7	4464	7.5
LV	6	0	20762	12.5	4819	7.8
LV	7	0	21525	13.1	4939	7.9
LV	0	1	116158	5.9	27504	5.4
LV	1	1	148190	7.8	34196	6.3
LV	2	1	183596	10.5	40927	7.4
LV	3	1	207564	12.7	45451	7.8
LV	4	1	225540	14.4	48545	8.3
LV	5	1	239463	15.6	50677	8.7
LV	6	1	250670	17.3	52376	9.1
LV	7	1	255906	19.8	53147	9.2
LV	0	2	969542	7.6	220441	6.6
LV	1	2	>1M	>24	>1M	>24

Yago includes three types of classes: the Wordnet classes, the Yago classes, and the Wikipedia classes. There are approximately 68000 Wordnet classes used in Yago that represent the top hierarchy of the Yago taxonomy. The classes introduced within the Yago dataset are mostly used to link the parts of the datasets. For example, they link the Wikipedia classes to the Wordnet hierarchy. There are less than 30 newly defined Yago classes. The Wikipedia classes are defined to represent group entities, individual entities, or some properties of the entities. There are approximately 500000 Wikipedia classes in Yago 2.0. While there are altogether around 571000 classes in Yago, it includes only a small number of predicates. There are 133 predicates [29], and there are only a few sub-predicates defined; therefore, the structure of the predicates is almost flat.

The stored schema graph of the Yago KG includes solely the Wordnet taxonomy and newly defined Yago classes. We do not use the Wikipedia classes since, in most cases, they are very specific. The newly defined stored triple types of the Yago KG are selected solely to get the global statistics of the KG. In general, the stored schema graph should include all stored triple types, i.e., the types explicitly defined in a given KG, that are the types of the triple patterns from the query workload. The height of the taxonomy from the stored schema graph of the Yago KG is 18 levels, but a small number of branches are higher than 10.

Table 1b) presents the execution of the algorithms STATISTICS- STORED, STATISTICS- ALL and STATISTICS- LEVELS on the Yago-S KG. It includes the same columns as defined for

Table 1a) (see Section 5.2), and two additional columns. The two additional columns, named time-b and time-u, represent the time in hours used for the computation of the statistics with either *bound* or *unbound* ways of counting, respectively.

The computation time for updating the statistics for a given triple t depends on the selected algorithm and the complexity of the triple enrollment into the conceptual schemata. In general, the more schema triples that include t in their interpretation, the longer the computation time is. For example, the computation time increases significantly in the case that lower levels of the ontology are used to compute the statistics, simply because there are many schema triples (approx. 450K) on the lower levels of the ontology. Finally, the computation of the statistics for the triples that describe people and their activities takes much more time than the computation of the statistics for the triples that represent links between websites, since the triples describing people have richer relationships with the schema than the triples describing links among URIs.

Let us now give some comments on the results presented in Table 1b). The algorithm ST for the computation of statistics of a KG, including 25M triples, takes about 1 hour to complete in the case of the bound and unbound types of counting. The computation of the algorithm AL is not complete because it consumes more than 32GB of main memory after generating more than 2M triple types.

The algorithm STATISTICS- LEVELS can regulate the amount of the triple types that serve as the framework for the computation of the statistics. The number of the generated triple types increases when more levels, either above or below the stored schemata, are taken into account. We executed the algorithm with the parameter #dlevel= 0, 1, 2. For each fixed value of #dlevel, the second parameter #ulevel value varied from 0 to 7. The number of generated triple types increases by about a factor of 10 when the value of #dlevel changes from 0 to 1 and from 1 to 2. Note also that varying #ulevel from 0 to 7 for each particular #dlevel results in an almost linear increase of the generated schema triples. This is because the number of triple types is falling fast as they are closer to the most general triple types from the KG.

6 Related work

The presented work started as a part of the project on the design and implementation of the prototype distributed RDF store, big3store [5]. The distribution of an RDF store is computed by partitioning the schema graph into strongly connected sub-graphs. These sub-graphs include strongly connected triple types with respect to the distance among nodes and the strength of edges represented by statistics of the triple types. The extensional part of the graph database is partitioned by taking the interpretations of the triple types from each partition of the schema graph.

The statistics of KGs are primarily used for the optimization of queries on the KGs. The queries in RDF stores can include a large number of joins; therefore, the query optimization depends heavily on the accuracy of statistics to estimate the selectivity of an operation and the size of the operation result. The approaches to capture the statistics of RDF graphs include the aggregate indexes of all SPO keys, the use of sampling to estimate the size of triple patterns, and the statistics based on the fragments of the conceptual schema of an RDF store. In the following Section 6.1 we review the methods for the computation and use of the statistics for query optimization in RDF stores.

Because of the rich conceptual schemata of KGs, the presented statistical index of a KG can be employed as a tool for analyzing the structure and contents of KGs. For example, the

study of the relationships among the triple types and their statistics in the conceptual schema of Yago 2 [29] can contribute to the analysis of the knowledge graph.

In this respect, our statistical index is close to the research on the integrity constraints in RDF databases that include shape graphs [30, 31] used for validating RDF graphs and the functional dependencies that can specify the functional constraint among the concepts defined in the context of a graph pattern. The research on SHACL [30] and ShEx [31] proposes to use shape graphs as a graph schemata. In Section 6.2 we compare the shape graph with the schema graph and present a review of the important works on the integrity constraints in RDF stores.

This paper is based on the work on the statistics of the knowledge graphs presented in [32]. The paper extends the work as follows.

- A formal definition of the knowledge graph, presented in Section 2, is extended and completed to provide a formal basis for the description of the algorithms for the computation of the statistics of knowledge graph.
- The presentation of the algorithms for the computation of the statistics, given in Section 3, is extended. The presentations of STATISTICS- ALL and STATISTICS- LEVELS are unified to show that the latter is a natural extension of the former. We show formally how the set of types of an input triple is determined in the algorithms STATISTICS- ALL and STATISTICS- LEVELS, and we present a formal definition of a strip around the stored schema graph and the function DIST used in the algorithm STATISTICS- LEVELS.
- A detailed description of the procedure for updating the statistics for a given triple is presented in Section 4. We introduce the concepts of a *key* and a *key type* to present how the statistics of keys are updated when a function UPDATE- STATISTICS is called.
- Numerous examples have been added to the paper to give more insight into the formal definition of KGs, the proposed algorithms for the computation of statistics and the method for counting keys of a given key type.
- The related work is extended to cover the recent research on the integrity constraints in knowledge graphs. We compare our definition of the schema graph to the shape graphs introduced by ShEx and SHACL [30, 31], and we review recent contributions to the research on integrity constraints in knowledge graphs.

6.1 Statistics for query optimization in RDF stores

In relational systems, the statistics are used for the estimation of the selectivity of simple predicates and join queries in a query optimization process [33]. For each relation, the gathered statistics include the cardinality, the number of pages, and the fraction of the pages that store tuples. Further, for each index, a relational system stores the number of distinct keys and the number of pages of the index. The equidistant histograms [34] were proposed to capture the non-uniform distribution of the attribute values. Piatetsky-Shapiro and Connell show in [35] that the equidistant histograms fail very often to give a precise estimation of the selectivity of simple predicates. To improve the precision of the selectivity estimation, they propose the use of histograms that have equal heights instead of equal intervals.

Most of the initially designed RDF stores treated stored triples as the edges of the schemaless graphs [6, 7, 9, 12, 13, 36, 37]. To efficiently process triple patterns, these RDF stores use either a subset of the six SPO indexes or a set of tables (a single triple-table or property tables) stored in a relational DBMS. However, the RDF data model [18] was extended with the RDF-Schema [19] that allows for the representation of knowledge bases [38, 39]. Consequently, RDF graphs can separate the conceptual and instance levels of the representation,

i.e., between the TBox and ABox [40]. Moreover, RDF-Schema can serve as the means for the definition of taxonomies of classes (or concepts) and predicates, i.e., the roles of a knowledge representation language [39]. Let us now present the existent approaches to collect the statistics of RDF stores.

Virtuoso [6] is based on relational technology. Since relational statistics can not capture well the semantics of SPARQL queries, the statistics are computed in real-time by inspecting one of 6 SPO indexes [41]. The size of conjunctive queries that can include one comparison operation ($\geq$, $>$, $<$, or $\leq$) is estimated by counting the pointers of index blocks satisfying the conditions on the complete path from the root to the leaf block of the index. In the case there are no conditions in a query, then sampling (1% of triples) is used to estimate the size of the query—the pointers of the index blocks are chosen at random. The results of query estimation are always stored in the index so that they are available for the following requests.

RDF-3X [1] is a centralized RDF store. RDF-3X uses six indexes for each ordering of columns S, P, and O. B+ tree indexes are customized in the following ways. Firstly, triples are stored directly in the leaves of B+ trees. Secondly, each index uses the lexicographic ordering of triples, which provides the opportunity to compress triples in leaves by storing only the differences between the triples. RDF-3X includes additional aggregated indexes where the number of triples is stored for each particular instance (value) of the prefix for each of the six indexes. Aggregate indexes can be used for the selectivity estimation of arbitrary triple patterns. They are converted into selectivity histograms that can be stored in the main memory to improve the performance of selectivity estimation. Furthermore, to provide a more precise estimation of the size of queries in the presence of correlated predicates, frequent paths are determined, and their cardinality is computed and stored.

TriAD [8] is a distributed RDF store implemented on shared-nothing servers running centralized RDF-3X [1]. The RDF store is partitioned utilizing a multilevel graph partitioning algorithm [42] that generates graph summarizations. These are further used for query optimization as well as during query execution to enable join-ahead pruning. Six distributed indexes are generated for each of the SPO permutations. Partitions are stored and indexed on slave servers, while the summary graph is stored and also indexed at the master server. The statistics of the RDF store are stored locally for the local partitions and globally for the summary graph. In both cases, the cardinality is stored for each value of S, P, and O, and, for the pairs of values SP, SO, and PO. Furthermore, the selectivity of joins between P_1 and P_2 predicates is also stored in the distributed index on all the slave servers and for the summary graph on the master server.

Stocker et al. [2] was the first to use statistics based on some form of semantic data. The statistics are used to compute the selectivity estimations in the query optimization algorithm that chooses the ordering of joins for basic graph patterns. For the triple patterns, the statistics are gathered for the bound (concrete) S, P, and O components. The number of triples with bound S component is approximated with the total number of triples and the number of distinct S values. Next, the number of triples with a given concrete predicate P is computed. Finally, equidistant histograms are computed for each particular predicate. The histograms represent the O component value distribution. Further, to compute the statistics of joins, RDF schema statements are used to enumerate all pairs of predicates that match in the domain/range of the first predicate with the domain/range of the second predicate. The number of triples is computed for each such pair of related predicates and employed as the upper bound of any basic graph pattern that includes a given pair of predicates.

Wolff, Fletcher, and Lu propose in [43] the framework for query optimization in RDF stores based on the heuristic rules. First, the qualitative rules originate from the insight into the characteristics of data stored in RDF stores. For example, the S triple component is more

selective than the O component which is more selective than the P component. Based on this insight, the rules are defined to prioritize the most selective access methods and the joins producing the smallest output. Second, the heuristic rules based on the statistics are used to prioritize the selection of the access methods, and the logical joins that, according to the statistics, produce the smallest result sets. The search for the efficient query plan is guided by the *collapse graph* of the query, composed of nodes representing atoms (access paths) and variables interconnected by links that represent joins. The query optimization method searches for an efficient query plan by selecting access paths and joins from a collapse graph to minimize the intermediate results of joins in the query plan. Similarly to the approach of Stocker et al. [2], the statistics used for the optimization include the information for estimating the sizes of triple patterns and joins between two triple patterns.

Neumann and Moetkotte propose the use of the characteristic sets [3] (abbr. CS) for the computation of statistics of star-shaped queries. For a given graph g and the subject s, a CS includes the predicates $\{p|(s, p, o) \in g\}$. The statistics are computed for each CS of a given RDF store. Furthermore, the CS-s of data and the CS-s of queries are computed. A star-shaped SPARQL query retrieves instances of all CS-s that are the super-sets of some CS from a query. Besides the statistics of CS-s, the number of triples with a given predicate is computed for each CS and each predicate. The size of joins in star-shaped queries can be accurately estimated in this way. Gubichev and Neumann further extended the work on characteristic sets (abbr. CS) in [4]. CS-s are organized in a hierarchy. The first level includes all CS-s of a given RDF store. The next level includes the cheapest CS-s, which are the subsets of CS-s from the previous level. The hierarchical characterization of CS-s allows precise estimation of joins in star queries.

Finally, Sagi et al. [44] propose a method for optimizing data representation in RDF triple stores based on three design principles: Subdivision, Compression, and Redundancy. Their approach focuses on improving performance by managing data division, compression, and redundancy according to query workload patterns. While their work emphasizes optimizing data structures for efficient query access, our study takes a different approach. Our study leverages schema-based type classification and the computation of statistics within the query space to optimize query processing. Therefore, both studies aim to enhance the performance of RDF stores, but they adopt distinct strategies—Sagi et al. focus on data representation, while our study emphasizes the utilization of schema information for query optimization.

6.2 Integrity constraints in RDF graphs

Our definition of a *schema graph* has been influenced by the relational data model [17]. A triple type can be compared to a relation schema of a relational database. The triple types are uniform since they all have the same structure, and they are also of the same shape as the ground triples. Finally, the schema triples are ordered in a poset with respect to the provided ontology of classes and predicates. The constraints are in a schema graph expressed solely in the form of the types that require from their instances, i.e., the ground triples, to respect the types of the triple type components.

Despite the simplicity, a schema graph, being a reformulation of the RDF [18] and RDF Schema [19], can express the structure of any set of relation schemata. Moreover, the predicates of an RDF schema are identifiers and can have properties defined by other predicates. Hence, the expressive power of a schema graph is equivalent to the structural part of a knowledge base [26, 38].

Let us now review the research on the integrity constraints defined for knowledge graphs. First, Shape Expressions (ShEx) is a schema language designed for RDF graphs [31]. It enables users to define and validate the structure and constraints of data in RDF graphs. ShEx offers a high-level syntax that is user-friendly and intuitive, allowing for the expression of complex constraints. Key features of ShEx include its ability to specify allowed values for properties of a node, cardinality constraints for the number of occurrences of a property, and recursive shape definitions. Additionally, ShEx supports negation, which adds further flexibility in defining constraints. The validation process in ShEx involves checking whether a node in an RDF graph adheres to a defined shape, where a shape is essentially a set of constraints. ShEx allows for post-hoc validation, ensuring that data complies with the defined schema after it has been added to the graph.

Shapes Constraint Language (SHACL) is a W3C recommendation for defining and validating constraints on RDF graphs [30]. SHACL allows users to define shapes, which represent a set of constraints that RDF nodes must satisfy. These constraints can include conditions on the data type, cardinality, and relationships between nodes within an RDF graph. SHACL's validation process involves specifying shapes that target specific nodes in an RDF graph and ensuring that the nodes conform to the given constraints. These constraints can be applied to individual nodes, as well as to the relationships between nodes. SHACL also supports advanced features such as recursion and logical operators, making it highly versatile for defining complex data structures and validation rules.

Now, let us compare the shape graphs of ShEx and SHACL to schema graphs. A *shape graph* is a structure depicting properties that describe an entity, together with the integrity constraints defined as the node and/or triple constraints. The data model provided by the shape graphs is an entity data model defined on top of the RDF data model. The entity model defines entities for the carefully selected nodes n of a KG and the properties of entities by specifying types and integrity constraints of the selected edges of KG linked to the entity node n. The data model of shape graphs is different from the graph model of the RDF and RDF-Schema.

The bare structure of a shape graph, including the definition of properties, is a subset of our schema graph since a schema graph, by assumption, includes a complete set of triple types of a knowledge graph. The data model provided by our formal definition of a knowledge graph is an upgrade of the RDF graph data model to represent the conceptual schemata of KGs. The schema graph includes, besides the triple types that specify the types of domains and ranges of each predicate, the ontology of classes and predicates. By using the ontology of classes and predicates, we define the poset of triple types based on a sub-type relation. We can infer, for example, that the triple type (scientist,influences,scientist) is a sub-type of the type (person,influences,person).

The ontology of classes and predicates implies that some entities may be specializations or generalizations of other entities since the nodes representing entities are linked by a sub-class relation. The shape graphs do not take into account the inheritance hierarchy of classes and predicates and, consequently, they do not capture the inheritance and sub-typing of entities. For example, the shape graphs that are defined for the class person and the class scientist are not related, and the shape graph of the node scientist may include the same triple constraints as the node person.

Let us now return to the presentation of the related works. Rabbani et al. [45] present an efficient tool for extracting validation shapes from large-scale KGs. The algorithm for the extraction of validating shapes begins with extracting the entities, types of entities and their frequencies. Second, the constraints are collected for the previously identified entities in the form of property shapes, represented similarly to our triple types. Third, the support and

confidence are computed for each property shape of an entity, and the min-max constraints are computed. Finally, in the shape extraction phase, the shape graphs are extracted, including the property shapes that have enough support and confidence computed from the statistics. An approximate variant of the presented (exact) algorithm is proposed. This algorithm can extract shape graphs from huge RDF collections by using sampling. The proposed algorithms outperform the existing tools for shape graph extraction.

Finally, the introduction of functional constraints in an RDF data model is important for achieving consistency and correctness of RDF graphs. The study of functional constraints of a KG may well provide hints for alternative design of the schema of an RDF database. Hellings et al. [46] present a general form of functional constraints on relations of arbitrary arity that can express functional dependencies, context-dependent functional dependencies, and constraints on the structure of graphs. A functional constraint $(P, L \rightarrow R)$ specifies a matching pattern P and a functional dependency $L \rightarrow R$ that holds if P matches some specific context. The sound and complete axiomatization of the functional constraints on relations of arbitrary arity is proposed. It has been shown that, in the case of ternary relations, we obtain the sound and complete axiomatization of the functional constraints in the RDF data model. The standard chase algorithm is adapted to the RDF data model, yielding a symmetry-preserving chase algorithm that provides reasoning power over functional constraints.

7 Conclusions

This paper presents a new method for the computation of the statistics of knowledge graphs. The statistics are based on the schema graph. We propose a technique to tune the size of the statistics; the user can choose between small, coarse-grained statistics and different levels of larger, finer-grained statistics. The smallest schema graph that can be used as the framework for the computation of the statistics is the stored schema graph, while the largest schema graph corresponds to all possible triple types that can be induced from the knowledge graph. In between, we can set the number of levels above and below the triple types from the stored schema graph to be included.

The computation of the statistics of the core of Yago, including around 25M triples, takes from 1-24 hours, depending on the size of the schema graph of the statistics. The experiments were run on a low-cost server with a 3.3GHz Intel Core CPU, 16GB of RAM, and 7200 RPM 1TB disk. Therefore, it is feasible to compute statistics even for large KGs. Note that KGs are not frequently updated so the statistics can be computed offline in a batch job.

Finally, the current implementation of algorithms for the computation of the statistics of KGs is not tuned for performance. While it includes some optimizations, for instance, the transitive closures of specific sets of classes are cached in the main memory index, there is a list of additional tuning options at the implementation level that can speed up the computation of the statistics in practice. For example, the complete schema graph can be stored in the main memory to speed up the operations that involve the schema graph solely.

Acknowledgements The authors would like to thank the anonymous referees for their valuable comments and suggestions to the initial version of this paper.

Author Contributions The authors Iztok Savnik, Kiyoshi Nitta, and Nikolaus Augsten made substantial contributions to the conception of the research work, the development of the formal framework, the design of the algorithms, the implementation of the algorithms, the design and implementation of the experiments, the analysis of the research results, and writing of the paper. In particular, Iztok Savnik was involved in all activities of the presented research. Kiyoshi Nitta was primarily involved in the design and implementation of the

algorithms. Riste Skrekovski was involved in the design and definition of the mathematical framework and algorithms. Finally, Nikolaus Augsten was involved in the conception of the research work, the design of the formalization and algorithms, the review of the results, and writing the paper.

Funding This work was partially supported by the Slovenian Research Agency (research core funding No. P1-00383) and the Federal State of Salzburg under grant number 20102-F2101143-FPR (Digital Neuroscience Initiative).

Data Availability Data for the experiments presented in the paper are available at https://osebje.famnit.upr.si/~savnik/epsilon/datasets/.

Declarations

Competing Interests The authors declare no competing interests.

References

1. Neumann, T., Weikum, G.: The rdf-3x engine for scalable management of rdf data. VLDB J. **19**(1), 91–113 (2010)
2. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D.: Sparql basic graph pattern optimization using selectivity estimation. In: WWW 2008, Semantic Web II. WWW 08, pp. 595–604. ACM, New York, NY, USA (2008)
3. Neumann, T., Moerkotte, G.: Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins. In: Proceedings of the 2011 IEEE 27th International Conference on Data Engineering. ICDE '11, pp. 984–994. IEEE Computer Society, Washington, DC, USA (2011). https://doi.org/10.1109/ICDE.2011.5767868
4. Gubichev, A., Neumann, T.: Exploiting the query structure for efficient join ordering in SPARQL queries. In: Amer-Yahia, S., Christophides, V., Kementsietsidis, A., Garofalakis, M.N., Idreos, S., Leroy, V. (eds.) Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014, pp. 439–450. OpenProceedings.org, Konstanz, Germany (2014). https://doi.org/10.5441/002/EDBT.2014.40
5. Savnik, I., Nitta, K.: In: Milutinovic, V., Kotlar, M. (eds.) Method of Big-Graph Partitioning Using a Skeleton Graph, pp. 3–39. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-13803-5_1
6. OpenLink Software Documentation Team: OpenLink Virtuoso Universal Server: Documentation. (2009). OpenLink Software Documentation Team
7. Wilkinson, K., Sayers, C., Kuno, H., Reynolds, D.: Efficient RDF Storage and Retrieval in Jena2
8. Gurajada, S., Seufert, S., Miliaraki, I., Theobald, M.: Triad: A distributed shared-nothing rdf engine based on asynchronous message passing. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data. SIGMOD '14, pp. 289–300. ACM, New York, NY, USA (2014).https://doi.org/10.1145/2588555.2610511. http://doi.acm.org/10.1145/2588555.2610511
9. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. Proc. VLDB Endow. **1**(1), 1008–1019 (2008)
10. Zou, L., Özsu, M.T., Chen, L., Shen, X., Huang, R., Zhao, D.: gstore: A graph-based sparql query engine. VLDB J. **23**(4), 565–590 (2014). https://doi.org/10.1007/s00778-013-0337-7
11. Harth, A., Decker, S.: Optimized index structures for querying rdf from the web. **2005**, 71–80 (2005). https://doi.org/10.1109/LAWEB.2005.25
12. Harth, A., Umbrich, J., Hogan, A., Decker, S.: Yars2: A federated repository for querying graph structured data from the web. In: Aberer, K., Choi, K.-S., Noy, N., Allemang, D., Lee, K.-I., Nixon, L., Golbeck, J.,

Mika, P., Maynard, D., Mizoguchi, R., Schreiber, G., Cudré-Mauroux, P. (eds.) The Semantic Web, pp. 211–224. Springer, Berlin, Heidelberg (2007)
13. Harris, S., Lamb, N., Shadbolt, N.: 4store: The design and implementation of a clustered rdf store. In: Proceedings of the The 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (2009)
14. Webber, J.: A programmatic introduction to neo4j. In: Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity. SPLASH '12, pp. 217–218. ACM, New York, NY, USA (2012). https://doi.org/10.1145/2384716.2384777
15. Harth, A., Hose, K., Schenkel, R.: Database techniques for linked data management. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. SIGMOD '12, pp. 597–600. ACM, New York, NY, USA (2012). https://doi.org/10.1145/2213836.2213909. http://doi.acm.org/10.1145/2213836.2213909
16. Hogan, A., Blomqvist, E., Cochez, M., D'amato, C., Melo, G.D., Gutierrez, C., Kirrane, S., Gayo, J.E.L., Navigli, R., Neumaier, S., Ngomo, A.-C.N., Polleres, A., Rashid, S.M., Rula, A., Schmelzeisen, L., Sequeda, J., Staab, S., Zimmermann, A.: Knowledge graphs. ACM Comput. Surv. **54**(4) (2021). https://doi.org/10.1145/3447772
17. Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM **13**(6), 377–387 (1970). https://doi.org/10.1145/362384.362685
18. Resource Description Framework (RDF). http://www.w3.org/RDF/ (2014)
19. RDF Schema 1.1. https://www.w3.org/TR/rdf-schema/ (2014)
20. Savnik, I., Nitta, K.: Datasets: Simple and Yago-S (2021)
21. Savnik, I., Nitta, K.: Algebra of rdf graphs for querying large-scale distributed triple-store. In: Proc. of CD-ARES. Lecture Notes in Computer Science. Springer, Berlin (2016)
22. Pierce, B.C.: Types and Programming Languages, 1st edn. MIT Press (2002)
23. Savnik, I.: Type-checking knowledge graphs. University of Primorska, Technical Report (In preparation), FAMNIT (2024)
24. WordNet: An electronic lexical database. Christiane Fellbaum (Ed.). Cambridge, MA: MIT Press, 1998. Pp. 423. Applied Psycholinguistics **22**(01), 131–134 (2001)
25. Pierce, B.C.: Programming With Intersection Types, Union Types, and Polymorphism (1991)
26. Ramachandran, D., Reagan, P., Goolsbey, K.: First-orderized researchcyc: Expressivity and efficiency in a common-sense ontology. In: AAAI Reports. AAAI, Washington, DC, USA (2005)
27. Savnik, I., Nitta, K.: epsilon: data and knowledge graph database system (2021)
28. Oracle Corporation: Oracle Berkeley DB 11g Release 2. (2011). Oracle Corporation
29. Hoffart, J., Suchanek, F.M., Berberich, K., Weikum, G.: Yago2: A spatially and temporally enhanced knowledge base from wikipedia. Artif. Intell. **194**(0), 28–61 (2013). https://doi.org/10.1016/j.artint.2012.06.001. Artificial Intelligence, Wikipedia and Semi-Structured Resources
30. Knublauch, H., Kontokostas, D.: Shapes Constraint Language (SHACL). Accessed 07 Sept 2024. https://www.w3.org/TR/shacl/
31. Boneva, I., Gayo, J.E.L., Prud'hommeaux, E.G., Staworko, S.: Shape Expressions Schemas. https://arxiv.org/abs/1510.05555
32. Savnik, I., Nitta, K., Skrekovski, R., Augsten, N.: Statistics of RDF store for querying knowledge graphs. In: Varzinczak, I. (ed.) Foundations of Information and Knowledge Systems - 12th International Symposium, FoIKS 2022, Helsinki, Finland, June 20-23, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13388, pp. 93–110. Springer, Berlin, Germany (2022). https://doi.org/10.1007/978-3-031-11321-5_6
33. Griffiths-Selinger, P., Astrahan, M.M., Chamberlin, D.D., Lorie, A., Price, T.G.: Access path selection in a relational database management system. In: Proceedings of SIGMOD '79. SIGMOD, pp. 23–34. ACM, New York, NY, USA (1979)
34. Christodoulakis, S.: Estimating block transfers and join sizes. In: Proceedings of SIGMOD '83. SIGMOD, pp. 40–54. ACM, New York, NY, USA (1983)
35. Piatetsky-Shapiro, G., Connell, C.: Accurate estimation of number of tuples satisfying condition. In: Proceedings of SIGMOD '84. SIGMOD, pp. 256–276. ACM, New York, NY, USA (1984)
36. Harris, S., Gibbins, N.: 3store: Efficient bulk rdf storage. In: 1st International Workshop on Practical and Scalable Semantic Systems (PSSS'03), pp. 1–15 (2003). Event Dates: 2003-10-20. http://eprints.soton.ac.uk/258231/
37. McBride, B.: Jena: A semantic web toolkit. IEEE Internet Comput. **6**(6), 55–59 (2002). https://doi.org/10.1109/MIC.2002.1067737
38. Lenat, D.B.: Cyc: A large-scale investment in knowledge infrastructure. Commun. ACM **38**(11), 33–38 (1995). https://doi.org/10.1145/219717.219745
39. Brachman, R.J., Levesque, H.J.: Knowledge representation and reasoning. Elsevier, Amsterdam (2004)

40. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P.: Description Logic Handbook. Cambridge University Press, England (2002)
41. Erling, O.: Implementing a SPARQL Compliant RDF Triple Store Using a SQL-ORDBMS. OpenLink Software, (2009). OpenLink Software. http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/VOSRDFWP/
42. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**(1), 359–392 (1999)
43. Wolff, B.G.J., Fletcher, G.H.L., Lu, J.J.: An extensible framework for query optimization on triplet-based RDF stores. In: Fischer, P.M., Alonso, G., Arenas, M., Geerts, F. (eds.) Proceedings of the Workshops of the EDBT/ICDT 2015 Joint Conference (EDBT/ICDT), Brussels, Belgium, March 27th, 2015. CEUR Workshop Proceedings, vol. 1330, pp. 190–196. CEUR-WS.org (2015). https://ceur-ws.org/Vol-1330/paper-31.pdf
44. Sagi, T., Lissandrini, M., Pedersen, T.B., Hose, K.: A Design Space for RDF Data Representations. https://doi.org/10.1007/s00778-021-00725-x
45. Rabbani, K., Lissandrini, M., Hose, K.: Extraction of Validating Shapes from Very Large Knowledge Graphs. https://doi.org/10.14778/3579075.3579078
46. Hellings, J., Gyssens, M., Paredaens, J., Wu, Y.: Implication and axiomatization of functional constraints on patterns with an application to the RDF data model. https://doi.org/10.1007/978-3-319-04939-7_12