Extraction of Validating Shapes from very large Knowledge Graphs [Extended Version]

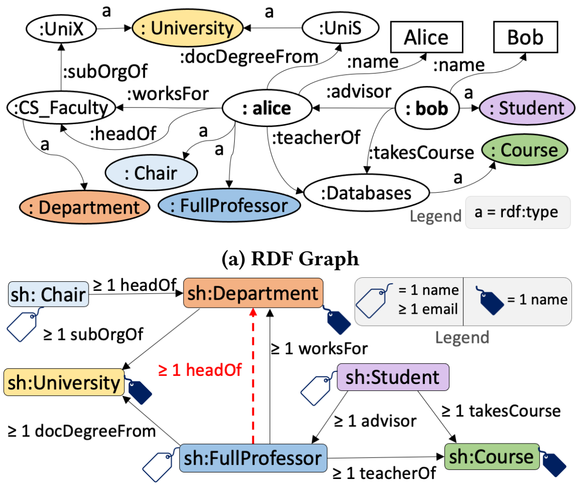
Kashif Rabbani
Aalborg University, Denmark
kashifrabbani@cs.aau.dk

Matteo Lissandrini
Aalborg University, Denmark
matteo@cs.aau.dk

Katja Hose
Aalborg University, Denmark
khose@cs.aau.dk

ABSTRACT

Knowledge Graphs (KGs) represent heterogeneous domain knowledge on the Web and within organizations. There exist shapes constraint languages to define *validating shapes* to ensure the quality of the data in KGs. Existing techniques to extract validating shapes often fail to extract complete shapes, are not scalable, and are prone to produce spurious shapes. To address these shortcomings, we propose the Quality Shapes Extraction (QSE) approach to extract validating shapes in very large graphs, for which we devise both an exact and an approximate solution. QSE provides information about the reliability of shape constraints by computing their confidence and support within a KG and in doing so allows to identify shapes that are most informative and less likely to be affected by incomplete or incorrect data. To the best of our knowledge, QSE is the first approach to extract a complete set of validating shapes from WikiData. Moreover, QSE provides a 12x reduction in extraction time compared to existing approaches, while managing to filter out up to 93% of the invalid and spurious shapes, resulting in a reduction of up to 2 orders of magnitude in the number of constraints presented to the user, e.g., from 11,916 to 809 on DBpedia.



(a) RDF Graph

(b) Validating Shapes

Figure 1: An example RDF Graph and Validating Shapes

Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/dkw-aau/qse.

1 INTRODUCTION

Knowledge Graphs (KGs), stored as collections of triples of the form $\langle subject, relation, object \rangle$ using the Resource Description Framework (RDF) [10], are in widespread use both within companies [29, 43, 44] and on the Web [46, 49]. Nonetheless, as KGs quickly accrue more data, practical applications impose further demands in terms of quality assessment and validation [34, 38, 52]. Hence, shapes constraint languages, e.g., SHACL [23], and ShEx [35], have been proposed to validate KGs by enforcing constraints represented in the form of *validating shapes*. For instance, we can express that an entity of type Student requires a name, a registration number, and should be enrolled in some courses; and that these attributes should be of type string, integer, and Course, respectively – see Figures 1a and 1b for an example KG and corresponding shapes.

Often, validating shapes are manually specified by domain experts. Yet, when trying to specify validating shapes for already-existing large-scale KGs, data scientists are in need of tools that can speed up this process [38]. Thus, various tools have been proposed to automatically [9, 12, 20, 26] or semi-automatically [5, 32, 36] produce a set of validating shapes for a target KG. Unfortunately, these methods suffer from 3 important limitations: (1) they are not able to produce complete shapes, e.g., they can identify that a student should have a property of type takesCourse but they do not extract the fact that the object should be of type Course; (2) the shapes they produce are easily affected by errors and inconsistencies in the KG, e.g., if some departments, by mistake, are attached the property hasAdvisor, a corresponding *spurious shape* is extracted; and (3) they do not scale to large KGs, e.g., they cannot process the full English WikiData, and they take days to process a subset of it. Therefore, in this work, we present the first techniques for *efficient extraction of validating shapes from very large existing KGs that also ensures robustness against the effects of spuriousness.*

Spuriousness poses important challenges to automatic shape extraction methods. For instance, in DBpedia [4], some of the entities representing musical bands are wrongly assigned to the class dbo:City. As a consequence, when shapes are extracted from its instance data using existing approaches, the resulting node shape for dbo:City specifies that cities are allowed optional properties like dbo:genre and dbo:formerBandMember. Hence, due to the effect of *spuriousness*, existing approaches generate tens of thousands of shapes (our experiments show that standard extraction processes produce more than 2 million property shapes for WikiData [49]). Thus, it becomes unmanageable for domain experts to manually identify valid shapes. SheXer [12] is the only existing approach that attempts to tackle this issue by filtering shapes based on a "trustworthiness" score. Unfortunately, this score does not directly translate into how frequently a shape is satisfied in a dataset, so it is still prone to generate spurious shapes, and it is also hard to tune. Furthermore, SheXer is not able to efficiently process large KGs.

Therefore, to tackle the issue of *spuriousness*, we study and formalize the problem of *support-based shapes extraction* and propose the Quality Shapes Extraction (QSE) approach as a solution to this problem. To tackle the issue of *scalability*, we devise two efficient algorithms, QSE-Exact and QSE-Approximate. *Hence, QSE can filter out shapes affected by spurious or erroneous data based on robust and easily understandable measures.* QSE allows shapes extraction both from KGs available as files as well as SPARQL endpoints. Moreover, our efficient approximation algorithm enables shape extraction even on a commodity machine by sampling the KG entities via a dynamic multi-tiered reservoir sampling technique.

We perform a thorough experimental evaluation using both synthetic (LUBM [18]) and real (DBpedia [4], YAGO-4 [46], WikiData [49]) KGs demonstrating the benefits of our approach. The shapes produced by our approach are of high quality and instrumental for easily finding errors in real KGs. The results show that QSE-Exact can extract shapes from the entire WikiData's 2015 dump in 16 minutes and from 2021's dump (1.9B triples) in 2.5 hours. Similarly, QSE-Approximate can extract shapes from WikiData's 2021 dump in 90 minutes on a 32GB machine while still achieving 100% precision and 95% recall in the set of shapes produced. Hence, our sampling strategy is accurate and efficient both when extracting shapes from a file as well as when using an endpoint.

2 RDF SHAPES AND THE QSE PROBLEM

In the following, we first introduce the KG data model and the concepts of validating shapes, their support, and confidence, then we define our focus: the Quality Shapes Extraction problem.

2.1 Preliminaries

The standard model for encoding KGs is the Resource Description Framework (RDF [10]), which describes data as a set of $\langle s, p, o\rangle$ triples stating that a subject s is in a relationship with an object o through predicate p. Therefore, we define an RDF graph as follows:

Definition 2.1 (RDF graph). Given pairwise disjoint sets of IRIs $\mathcal{I}$, blank nodes $\mathcal{B}$, and literals $\mathcal{L}$, an RDF Graph $\mathcal{G}{:}\langle N, E\rangle$ is a graph with a finite set of nodes $N\subset(\mathcal{I}\cup\mathcal{B}\cup\mathcal{L})$ and a finite set of edges $E\subset\{\langle s, p, o\rangle\in(\mathcal{I}\cup\mathcal{B})\times\mathcal{I}\times(\mathcal{I}\cup\mathcal{B}\cup\mathcal{L})\}$.

Moreover, we distinguish two special subsets of the IRIs $\mathcal{I}$: predicates $\mathcal{P}$ and classes $\mathcal{C}$. The set of predicates $\mathcal{P}\subset\mathcal{I}$ is the subset of IRIs that appear in the predicate position p in any $\langle s, p, o\rangle\in\mathcal{G}$. Among predicates $\mathcal{P}$, we identify the type predicate $\mathbf{a}\in\mathcal{P}$, which corresponds to IRI rdf:type [51] or wdt:P31 WikiData [49], as the predicate that connects all entities that are instances of a class to the node representing the class itself, i.e., their type. Thus, all the IRIs that are classes in $\mathcal{G}$ form the subset $\mathcal{C}{:}\{c\in\mathcal{I}\,|\,\exists s\in\mathcal{I}$ s.t. $\langle s, \mathbf{a}, c\rangle\in\mathcal{G}\}$.

Given a KG $\mathcal{G}$, a set of *validating shapes* represents integrity constraints in the form of a shape schema $\mathcal{S}$ over $\mathcal{G}$. Since the shape schema describes shapes associated with node types and their connections to other attributes and node types, we can also visualize the shape schema $\mathcal{S}$ as a particular type of graph (see Figures 1a and 1b). Therefore, in the following, we refer to two concepts: the *data graph* $\mathcal{G}$ and the *shape graph* derived from $\mathcal{S}$. The *data graph* is the RDF graph $\mathcal{G}$ to be validated, while the *shape graph* consists of constraints in the form of the shape schema $\mathcal{S}$ against which entities of the data graph are validated. These constraints are defined using node and property shapes. In the following, we adopt the previously defined syntax [42] to refer to the set $\mathcal{S}$ according to the SHACL *core constraint components* [50]. Finally, while validating shapes can also be expressed in ShEx [34], our approach can be trivially extended to output ShEx directly, or it can exploit existing SHACL to ShEx converters [53]. Thus, without loss of generality, we focus on the current standard for SHACL shapes in the following.

Definition 2.2 (Shape Schema). A SHACL shape schema consists of a set of node shapes $\mathcal{S}$, with $\langle s, \tau_s, \Phi_s\rangle\in\mathcal{S}$, where s is the shape *name*, $\tau_s\in\mathcal{C}$ is the *target class*, and Φ_s is a set of property shapes of the form $\phi_s{:}\langle\tau_{\mathsf{p}}, \mathsf{T}_{\mathsf{p}}, \mathsf{C}_{\mathsf{p}}\rangle$, where $\tau_{\mathsf{p}}\in\mathcal{P}$ is called the target property, $\mathsf{T}_{\mathsf{p}}\subset\mathcal{I}$ contains either an IRI defining a *literal type*, e.g., xsd:string, or a set of IRIs – called *class type constraint*, and C_{p} is a pair $(n, m)\in\mathbb{N}\times(\mathbb{N}\cup\{\infty\})$. $n\leq m$ – called min and max *cardinality constraints*.

Therefore, given a node shape $s\in\mathcal{S}$ for the *target class* $\tau_s\in\mathcal{C}$, Φ_s defines which properties each instance of τ_s can or should be associated with. For instance, the shape $\langle$sh:Student, :Student, $\{\phi_{s_1}, \phi_{s_2}\}\rangle$ from Figure 1b, contains a node shape for target class :Student and enforces two property shapes ϕ_1 and ϕ_2. The property shape ϕ_1 has a target property τ_{p}= :name, a literal type constraint T_{p}= xsd:string, and the cardinality constraints C_{p}=$(1, 1)$. Similarly, the property shape ϕ_2 has a target property τ_{p}=:takesCourse, a class type constraint T_{p}= :Course, and the cardinality constraint $\mathsf{C}_{\mathsf{p}} = (1, \infty)$.

When validating a graph $\mathcal{G}$ against a shape schema $\mathcal{S}$ having a node shape $\langle s, \tau_s, \Phi_s\rangle\in\mathcal{S}$, we verify that each entity $e\in\mathcal{G}$ that is an instance of τ_s satisfies all the constraints Φ_s. Note that we use the term entity and node interchangeably throughout the paper. Thus, we define the semantics of $\mathcal{S}$ as follows:

Definition 2.3 (Validating Shape Semantics). Given a node shape $\langle s, \tau_s, \Phi_s\rangle\in\mathcal{S}$, a graph $\mathcal{G}$, and an entity e s.t. $\langle e, \mathbf{a}, \tau_s\rangle\in\mathcal{G}$, we have that s validates e, and we write $e\models_{\mathcal{G}}\phi$, if for every property shape $\phi_s{:}\langle\tau_{\mathsf{p}}, \mathsf{T}_{\mathsf{p}}, \mathsf{C}_{\mathsf{p}}\rangle\in\Phi_s$ the following conditions hold:

- If T_{p} is a literal type constraint, then for every triple $(e, \tau_p, l)\in\mathcal{G}$, l is a literal of type T_{p}.
- If T_{p} is a set of class type constraints T_{p}=$\{t_1, t_2, ...t_n\}$, then for every triple $(e, \tau_p, o)\in\mathcal{G}$, it holds that $\forall t\in\mathsf{T}_{\mathsf{p}}$, o is an instance of t (or of a subclass of t) and if $\exists S_t\in\mathcal{S}$, $o\models_{\mathcal{G}}S_t$.
- $n\leq|\{(s, p, o)\in\mathcal{G} : s = e\wedge p = \tau_p\}|\leq m$, where C_{p}=(n, m).

Here we study the case where $\mathcal{G}$ is given, and we want to extract the set of validating shapes $\mathcal{S}$ that validates every class in $\mathcal{C}$ from $\mathcal{G}$. This is the *shapes extraction* problem. In this case, existing automatic approaches [38] assume the graph to be correct, then iterate over all entities in it, and extract for each entity e all necessary shapes that validate e. The union of all such shapes is assumed to be the final schema $\mathcal{S}$. This is useful when we want to validate new data that will be added *in the future* to the KG so that it will conform to the data already in the graph. Unfortunately, this approach will produce spurious shapes. For instance, in Figure 1, since :alice has both type Full Professor and Chair, when parsing the triple (:alice, :headOf, :CS_Faculty), the property shape headOf (the red dotted arrow in Figure 1b) is assigned to both node shapes, instead of assigning it to the Chair node shape only.

2.2 Shapes Support and Confidence

To contrast the effect of spuriousness, we want to exploit statistics on how often properties are applied to entities of a given type. Therefore, we introduce the notion of *support* and *confidence* for shape constraints to study the reliability of extracted shapes. These concepts are inspired by the well-known theory developed for the task of frequent patterns mining [19] and the concept of MNI support for graph patterns [7]. The MNI support of a graph pattern is the minimum cardinality of the *set of all nodes of $\mathcal{G}$ that are mapped to a specific pattern node by some isomorphism* across all the nodes of the pattern. In our approach, a property shape corresponds to a node- and edge-labeled graph pattern. Thus, given the shape $s{:}\langle s, \tau_s, \Phi_s\rangle{\in}\mathcal{S}$ its support is the number of entities that are of type τ_s, while the support of a property shape $\phi_s{:}\langle \tau_\text{p}, \text{T}_\text{p}, \text{C}_\text{p}\rangle{\in}\Phi_s$ is the cardinality of entities conforming to it.

Definition 2.4 (Support of ϕ_s). Given a shape $\langle s, \tau_s, \Phi_s\rangle{\in}\mathcal{S}$ with shape constraint $\phi_s{:}\langle \tau_\text{p}, \text{T}_\text{p}, \text{C}_\text{p}\rangle{\in}\Phi_s$, the support of ϕ_s is defined as the number of entities e satisfying ϕ_s, denoted as $e{\models}\phi_s$, hence:

$$\text{supp}(\phi_s){=}|\{e{\in}\mathcal{I} \mid e{\models}\phi_s\}| \tag{1}$$

Finally, the confidence of a constraint ϕ_s measures the ratio between how many entities conform to ϕ_s and the total number of entities that are instances of the target class of the shape s.

Definition 2.5 (Confidence of ϕ_s). Given a shape $\langle s, \tau_s, \Phi_s\rangle{\in}\mathcal{S}$ having shape constraint $\phi_s{:}\langle \tau_\text{p}, \text{T}_\text{p}, \text{C}_\text{p}\rangle{\in}\Phi_s$, the confidence of ϕ_s is defined as the proportion of entities for which $e{\models}\phi_s$ among the entities that are instances of the target class τ_s of $s{\in}\mathcal{S}$, hence:

$$\text{conf}(\phi_s) = \frac{\text{supp}(\phi_s)}{|\{e|(e, type, \tau_s) \in \mathcal{G}\}|} \tag{2}$$

As it happens in the case of frequent pattern mining [19], when extracting validating shapes, the support provides insights on how frequently a constraint is matched in the graph, i.e., the number of entities e satisfying a constraint ϕ_s. While similar to the task of itemset mining [6], the confidence can tell us how strong is the association between a node type and a specific constraint, i.e., the proportion of entities e satisfying a constraint ϕ_s among all the entities that are instances of the node type τ_s of $s{\in}\mathcal{S}$. For instance, the confidence for property shape headOf (Figure 1b) in our snapshot of LUBM is 10% for the Full Professor node shape and 100% for Chair, which indicates a strong association of the headOf property shape to latter and a weak association to the former.

2.3 The Quality Shapes Extraction Problem

Given the need to extract shapes from a large existing graph $\mathcal{G}$ while limiting the effect of spuriousness, we formally define the problem of extracting high-quality shapes from KGs as follows:

Problem 1 (Quality Shapes Extraction). *Given an RDF graph $\mathcal{G}$, a threshold ω for support, and ε for confidence, the problem of quality shapes extraction over $\mathcal{G}$ is to find the set of shapes $\mathcal{S}$ such that for all node shapes $\langle s, \tau_s, \Phi_s\rangle{\in}\mathcal{S}$ it holds that* $\text{supp}(s){>}\omega$ *and for all property shapes $\phi_s{:}\langle \tau_p, T_p, C_p\rangle{\in}\Phi_s$,* $\text{supp}(\phi_s){>}\omega$ *and* $\text{conf}(\phi_s){>}\varepsilon$*.*

In the following, we provide both, an exact and an approximate solution to the problem of quality shape extraction.

3 QSE-EXACT

Extracting shapes $\mathcal{S}$ from an RDF graph $\mathcal{G}$ requires processing its triples and analyzing the types of nodes involved both as subjects and objects in those triples. At a high level, we need to know for each entity all its types, these will become node shapes, and then for each entity type, identify property shapes, which requires, in turn, knowing the types of the objects as well. Furthermore, we need to keep frequency counts to know how often a specific property connects nodes of two given types compared to how many entities exist of those types. In our solution, this is done in four steps: (1) entity extraction, (2) entity constraints extraction, (3) support and Confidence computation, and (4) shapes extraction. Here we first consider the case where the graph is stored as a complete dump on a single file. Later, we also consider the case for a graph stored within a triplestore [41] for which the KG is not available as a file.

QSE-Exact (file-based). One of the most common ways to store an RDF graph $\mathcal{G}$ on a file **F** is to represent it as a sequence of triples. Therefore, QSE reads **F** line by line and processes it as a stream of $\langle s, p, o\rangle$ triples. Algorithm 1 and Figure 2 present the four main steps of QSE to extract shapes for graph $\mathcal{G}$ stored in **F**. In the entity extraction phase, the algorithm parses each $\langle s, p, o\rangle$ triple containing a type declaration (e.g., rdf:type or wdt:P31 – this can be configured) and for each entity, it stores the set of its entity types and the global count of their frequencies, i.e., the number of instances for each class (Lines 4-8) in maps Ψ_ETD (Entity-to-Data) and Ψ_CEC (Class-to-Entity-Count), respectively. For example, Figure 2 (phase 1) presents two example entities :bob and :alice (from the example graph of Figure 1a) having entity types :Student, :FullProfessor, and :Chair, respectively. Figure 2 also presents the structure of the Entity-to-Data Ψ_ETD dictionary map to help understand the captured entities and their information. In the second phase, i.e., entity constraints extraction, the algorithm performs a second pass over **F** (Lines 9-19) to collect the constraints and the meta-data required to compute support and confidence of each candidate property shape. Specifically, it parses all triples except triples containing type declarations (which can be skipped now) to obtain for each predicate the subject and object types from the map Ψ_ETD that was populated in the previous step. The type of a literal object is inferred from the value, and for a non-literal object is obtained from Ψ_ETD (Lines 11-16). For example, Ψ_ETD records that the types of :alice are :FullProfessor and :Chair. Then, the Entity-to-Property-Data map Ψ_ETPD is updated to add the candidate property constraints associated with each subject entity (Line 17). Figure 2 (phase 2) shows the meta-data captured for the properties of :bob and :alice.

In the third phase, i.e., for support and confidence computation, the constraints' information stored in maps (Ψ_ETD, Ψ_CEC) is used to compute support and confidence for specific constraints. The algorithm iterates over the map Ψ_ETD to get the inner map Ψ_ETPD mapping entities to candidate property shapes $\phi_s{:}\langle \tau_\text{p}, \text{T}_\text{p}, \text{C}_\text{p}\rangle{\in}\Phi_s$, and retrieves the type of each entity using types information stored in Ψ_ETD to build triplets of the form $\langle \tau_e, \tau_p, \tau_{p_o}\rangle$ and compute their support and confidence (Line 25). Figure 2 (phase 3) highlights some of these triplets for τ_e =:Student. The value of support and confidence for each distinct triplet is incremented in each iteration and stored in Ψ_Supp and Ψ_Conf maps. Additionally, a map Ψ_PTT (Property to Types) is populated with distinct properties' frequencies and

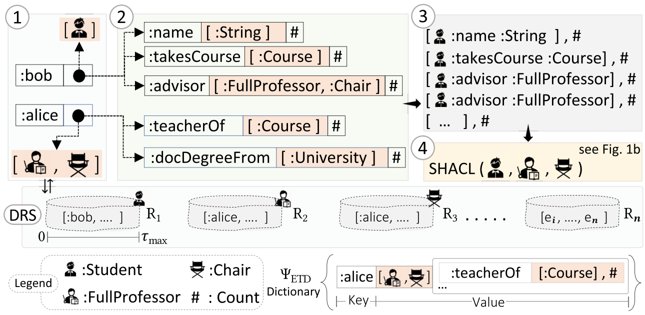


Figure 2: Overview of the four phases of QSE: ① entity extraction, ② entity constraints extraction, ③ support and confidence computation, and ④ shapes extraction. QSE-Approximate uses Dynamic Reservoir Sampling (DRS) in ①.

their object types in order to, later on, establish the corresponding min/max cardinality constraints (Line 26).

Finally, in the shapes extraction phase, the algorithm iterates over the values of the Ψ_{CTP} map and defines the *shape name* of s, the *shape's target definition* τ_s, and the set of *shape constraints* ϕ_s for each candidate class (Lines 27-29). The set of property shapes P for a given *Node Shape* are then extracted from the map $\textsc{Map}\langle\textsc{Property}, \textsc{Set}\rangle$ (Lines 30-36). An example shapes graph for our running example is shown in Figure 1. The $\mathrm{C_p}$ constraint can possibly have three types of values: sh:Literal, sh:IRI, and sh:BlankNode. In the case of literal types, the literal object types such as xsd:string, xsd:integer, or xsd:date are used. However, in the case of non-literal object types, the constraint sh:class is used to declare the type of object to define the type of value for the candidate property. It is possible to have more than one value for the sh:class and sh:datatype constraints of a candidate property shape, e.g., to state that a property can accept both integers and floats as values, in such cases, we use sh:or constraint to encapsulate multiple values. A detailed explanation of each phase is available in the extended version of the paper[1].

QSE-Exact (query-based). To support shapes extraction from a triplestore, we propose QSE-Exact query-based that uses a set of SPARQL queries [40] to extract all the necessary information that we collect across the four phases. In practice, we pose queries to extract all the distinct classes C, then, for each class $c{\in}C$, its properties $p{\in}\mathcal{P}$ along with object types are extracted as triplets, and support is computed for each triplet by a count query. This method is based on the standard procedure also implemented in other existing, query-based tools [12, 20].

Cardinality Constraints. QSE supports assigning cardinality constraints (sh:minCount and sh:maxCount) to $\mathrm{C_p}$ to each property shape constraint $\phi_s{:}\langle\tau_{\mathrm{p}}, \mathrm{T_p}, \mathrm{C_p}\rangle$. Following the open-world assumption, all shape constraints are initially assigned a minimum cardinality of 0, making them optional. However, there are cases where we can infer that some properties are mandatory (i.e., should be assigned a min count of 1), and some other properties should appear exactly once for each entity (i.e., should be assigned both a min and a max count equal to 1). Trivially one can assign minimum cardinality 1 to property shapes having confidence 100%, i.e., for those cases in which all entities have that property. In case of incomplete KGs, QSE allows users to provide a different confidence threshold value for adding the min cardinality constraints. To achieve this, we extend the fourth phase and add a min cardinality constraint in property shapes on line 35 based on the min-confidence provided by the user. QSE also keeps track of properties having maximum cardinality equal to 1 in a second phase and assigns sh:maxCount=1 to those property shapes in the fourth phase of shapes extraction.

Complexity Analysis. The time complexity of QSE-Exact (Algorithm 1) is $O(2{\cdot}|F|+|E|{\cdot}|\Phi_s|+|S|{\cdot}|\Phi_s|)$. Where $2*|F|$ refers to the first and second phases having to parse all the triples twice, E is the set of entities (i.e., the set of distinct IRIs that appear as a subject for some triple), S is the set of Node Shapes, and lastly, Φ_s represents a set of all property shape constraints, i.e., $\Phi_s = \{\phi_1, \phi_2, ..., \phi_n\}$. Therefore, our algorithm scales linearly in the number of edges and nodes in the graph and in the size of the final set of shapes.

4 QSE-APPROXIMATE

QSE-Exact keeps type and property information for each entity in memory while extracting shapes. As a result, its memory requirements are prohibitively large when dealing with large KGs. Therefore, we propose QSE-Approximate to enable shape extraction from very large KGs with reduced memory requirements. Our goal is to *solve the scalability issue in shapes extraction approaches*

Algorithm 1 Shapes Extraction

```
    Input: Graph $\mathcal{G}$ from File F, $\omega$: min-support, $\varepsilon$: min-confidence
    Output: Output: $\mathcal{S}\langle s, \tau_s, \Phi_s\rangle$
 1: $E_{data} \leftarrow \{$T: $\textsc{Set}_{\text{Types}}$ , $\Psi_{\text{ETPD}} = \text{Map}\langle\textsc{Iri}, P_{data}\rangle\}$
 2: $P_{data} \leftarrow \{$T$'$ : $\textsc{Set}_{\text{ObjTypes}}$ , Count :$\textsc{Int}\}$
 3: $\Psi_{\text{ETD}} = \text{Map}\langle\textsc{Iri}, E_{data}\rangle$ , $\Psi_{\text{CEC}} = \text{Map}\langle\textsc{Iri}, \textsc{Int}\rangle$ , $\Psi_{\text{CTP}} = \text{Map}\langle\textsc{Iri}, \textsc{Map}\langle\textsc{Iri}, \textsc{Set}\rangle\rangle$
 4: for $t \in \mathcal{G} \wedge$ t.p = Type Predicate do                  ▷ ① Entity extraction
 5:     entity $e$ : t.s ; entityType $e_t$ = t.o                  ▷ s: subject, o: object
 6:     if $e \notin \Psi_{\text{ETD}}$ then $\Psi_{\text{ETD}}$.insert($e$, ... ))
 7:     $\Psi_{\text{ETD}}$.insert($e$, $\Psi_{\text{ETD}}$.get($e$).T.add($e_t$))                  ▷ T : entity types
 8:     increment entity count for current $e_t$ in $\Psi_{\text{CEC}}$
 9: for $t \in \mathcal{G} \wedge$ t.p != Type Predicate do      ▷ ② Entity constraints extraction
10:     $\text{Set}_{\text{ObjTypes}} \leftarrow \emptyset$ , $\text{Set}_{\textsc{Tuple}} \leftarrow \emptyset$      ▷ init a type and property to type tuple set
11:     if object $t.o$ is Literal then
12:         $\text{Set}_{\text{ObjTypes}}$.add(getLiteralType(t.o))
13:         $\text{Set}_{\textsc{Tuple}}$.add(new Tuple$\langle$ $t.p$, getLiteralType($t.o$)$\rangle$)
14:     else                  ▷ for non-literal objects
15:         for $\text{obj}_{\text{type}} \in \Psi_{\text{ETD}}$.get($t.o$).T do
16:             $\text{Set}_{\text{ObjTypes}}$.add($\text{obj}_{\text{type}}$) ; $\text{Set}_{\textsc{Tuple}}$.add(new Tuple$\langle t.p, \text{obj}_{\text{type}}\rangle$)
17:     addPropertyConstraints($t.s$, $\text{Set}_{\textsc{Tuple}}$, $\Psi_{\text{ETD}}$)
18:     for $\textsc{Iri} \in \Psi_{\text{ETD}}$.get($t.s$.T) do                  ▷ if $t.s \in \Psi_{\text{ETD}}$
19:         update $\Psi_{\text{CTP}}$ with class $\textsc{Iri}$, $t.p$, and object types using $\text{Set}_{\text{ObjTypes}}$
                                          ▷ ③ Support and Confidence computation
20: $\Psi_{\text{Supp}} = \textsc{Map}\langle\textsc{Tuple}_3, \textsc{Int}\rangle$ , $\Psi_{\text{Conf}} = \textsc{Map}\langle\textsc{Tuple}_3, \textsc{Int}\rangle$ , $\Psi_{\text{PTT}}$
21: for $(e, E_{data}) \in \Psi_{\text{ETD}}$ do
22:     for $(T, \Psi_{\text{ETPD}}) \in E_{data}$ do
23:         for $e_t \in T \wedge (p, p_o, c) \in P_{data}$ do
24:             $\chi \leftarrow$ createTriplets($\langle\tau_e, \tau_p, \tau_{p_o}\rangle$)
25:             computeSupportAndConfidence ( $\Psi_{\text{Supp}}$, $\chi$, $\Psi_{\text{CEC}}$)
26:             computeMaxCardinality ($\Psi_{\text{PTT}}$, $p$ , $c$)
27: for (class, $\textsc{Map}\langle\text{Property}, \textsc{Set}_{\text{ObjTypes}}\rangle$) $\in \Psi_{\text{CTP}}$ do      ▷ ④ Shapes extraction
28:     $\Phi_s \leftarrow \emptyset$   ▷ Property shapes $\Phi_s = \{\phi_{s_1}, \phi_{s_2}, ..., \phi_{s_n}\}$ where $\phi_s{:}\langle\tau_{\mathrm{p}}, \mathrm{T_p}, \mathrm{C_p}\rangle$
29:     $s$ = class.buildShapeName() , $\tau_s$ = class
30:     for ($p$, $\textsc{Set}_{\text{ObjTypes}}$) $\in \textsc{Map}\langle\text{Property}, \textsc{Set}\rangle$ do $\phi_s.\tau_{\mathrm{p}} = p$
31:         $p.\omega = \Psi_{\text{Supp}}.\text{get}(p, \textsc{Set}_{\text{ObjTypes}})$ , $p.\varepsilon = \Psi_{\text{Conf}}.\text{get}(p, \textsc{Set}_{\text{ObjTypes}})$
32:         if $p.\omega > \omega \wedge p.\varepsilon > \varepsilon$ then
33:             build(sh:nodeKind, sh:maxCount, $\Psi_{\text{PTT}}$)
34:         $\phi_s.\mathrm{C_p}.add$(sh:minCount : 1)                  ▷ $if p.\varepsilon > \varepsilon'$
35:         $\Phi_s.add(\phi_s)$
36:     $\mathcal{S}.add(s, \tau_s, \Phi_s)$                  ▷ if s.$\omega > \omega \wedge \phi_s!\emptyset$
```

[1] https://relweb.cs.aau.dk/qse/

by using only the resources available to a commodity machine. QSE-Approximate is based on a multi-tiered dynamic reservoir-sampling (DRS) algorithm that we introduce. We maintain as many reservoirs as types in the graph, and we dynamically resize each reservoir as new triples are parsed. Moreover, the replacement of nodes in the reservoir is performed based on the number of node types across reservoirs. The resulting algorithm replaces the first phase of QSE. After sampling, the information about the sampled entities is used in the same way as before in the remaining phases of Algorithm 1. Hence, we maintain information only for a small representative sample of entities in memory but enough to detect all shapes.

Algorithm 2 QSE-Approximate Reservoir Sampling

```
    Input: Graph $\mathcal{G}$ from File F, maximum entity threshold $\tau_{\text{max}}$, Sampling$_{\%}$
    Output: $\Psi_{\text{ETD}}$ , $\Psi_{\text{CEC}}$
 1: init maps $\Psi_{\text{ETD}}$ , $\Psi_{\text{SEPC}}$, $\Psi_{\text{RCPC}}$ , $\Psi_{\text{CEC}}$, $\Psi_{\text{PC}}$
 2: $\tau_{\text{min}}$ = 1 (minimum entity threshold) ; lineCounter = 0
 3: for t ∈ $\mathcal{G}$ do                                          ▹ parse s,p,o of the triple $t$
 4:     if t.p = Type Predicate then
 5:         entity $e$ : t.s ; entityType $e_t$ = t.o                   ▹ s: subject, o: object
 6:         $\Psi_{\text{SEPC}}$.putIfAbsent($e_t$, [ ] )                        ▹ if $e_t \notin \Psi_{\text{SEPC}}$
 7:         $\Psi_{\text{RCPC}}$.putIfAbsent($e_t$, $\tau_{\text{min}}$)                        ▹ if $e_t \notin \Psi_{\text{RCPC}}$
 8:         if | $\Psi_{\text{SEPC}}$.get ($e_t$) | < $\Psi_{\text{RCPC}}$.get ($e_t$) then       ▹ Add entity $e$ in reservoir
 9:             if $\Psi_{\text{ETD}}$.get($e$).T is $\emptyset$ then $\Psi_{\text{ETD}}$.insert($e$, ... ))      ▹ T : entity types
10:             $\Psi_{\text{ETD}}$.insert($e$, $\Psi_{\text{ETD}}$.get($e$).T.add(t.$o$))
11:             $\Psi_{\text{SEPC}}$.get($e_t$).insert($e$)
12:         else              ▹ Replace random entity in reservoir with current entity $e$
13:             $r$ = generateRandomNumber(0, lineCounter)
14:             if $r$ < |$\Psi_{\text{SEPC}}$.get($e_t$) | then
15:                 $\overleftarrow{n}, \widehat{n}, \vec{n}$ = $\Psi_{\text{SEPC}}$.get($e_t$).nodeAtIndex($r - 1, r, r + 1$))
16:                 n = getNodeWithMinimumScope($\overleftarrow{n}, \widehat{n}, \vec{n}$)
17:                 replace node at index $n$ with current $e$ & $e_t$ in $\Psi_{\text{ETD}}$
18:                 $\Psi_{\text{SEPC}}$.get($e_t$).add ($e$)
19:         increment entity count for current $e_t$ in $\Psi_{\text{CEC}}$
20:                                                                  ▹ Resize reservoir
21:         ratio = ($\Psi_{\text{SEPC}}.get(e_t).size()/\Psi_{\text{CEC}}.get(e_t)$) × 100
22:         capacity = Sampling$_{\%}$ × $\Psi_{\text{SEPC}}$.get($e_t$).size()
23:         if capacity < $\tau_{max}$ ∧ ratio ≤ Sampling$_{\%}$ then $\Psi_{\text{RCPC}}$.insert($e_t$ , capacity)
24:     else→ increment property count for current $t.p$ in $\Psi_{\text{PC}}$
25:     lineCounter + +
```

Algorithm 2 receives as input a graph file **F**, sampling percentage (Sampling$_{\%}$), and maximum size of the reservoir per class (τ_{max}). After initialization, triples t of **F** are parsed (Line 3) and filtered based on whether they contain a type declaration. From these, we extract the entities to populate the Entity-to-Data map Ψ_{ETD} (Lines 4-24), while non-type triples are parsed on Line 24 to keep count of distinct properties in the Property-Count map Ψ_{PC}. For instance, :alice is an entity of type :FullProfessor and :Chair in Ψ_{ETD} shown in Figure 2. QSE-Approximate maintains a reservoir for each distinct entity type e_t, e.g., maintaining a distinct reservoir of entities of type :Student (R_1), :FullProfessor (R_2), and :Chair (R_3) shown in Figure 2, using a map of sampled entities per class (Ψ_{SEPC}). The reservoir capacity map (Ψ_{RCPC}) stores the current max capacities for the reservoir for each e_t. If e_t does not exist in Ψ_{SEPC} and Ψ_{RCPC}, i.e., if it has not a reservoir, one is created (lines 6-7). Then, e is inserted in the reservoir for e_t (Lines 8-11), e.g., :alice is inserted into both reservoirs R_2 and R_3 shown in Figure 2. If the reservoir has reached its current capacity limit, we may have to replace an entity in the reservoir with the current one. Hence, neighbor-based dynamic reservoir sampling is performed (Lines 13-18), i.e., a random number r is generated between zero and the current number of type declarations read from **F**. If r falls within the reservoir size, then a node in the reservoir is replaced with e. To select which node to replace, we identify as $\widehat{n}$ the target node at index r, and with $\overleftarrow{n}$ and $\vec{n}$ its neighbors at indexes r−1 and r+1, respectively. Among these, the node having minimum scope (i.e., the minimum number of types that are known at this point in time) is selected to be replaced by the current e (Line 17). Additionally, the algorithm keeps track of actual Class-to-Entity-Count in Ψ_{CEC} (Line 19), i.e., the exact count of how many entities of each type we have seen. Once the reservoir for e_t is updated, the sampling ratio for this type is computed, i.e., the proportion of entities kept so far with type e_t over the total number of entities of that type seen up to now. Given the current and target sampling ratio (Sampling$_{\%}$) provided as input, the algorithm evaluates whether to resize the reservoir for e_t, if it has not already reached the limit τ_{max} (Lines 21-23).

While performing shapes pruning using counts over sampled entities, QSE-Approximate requires to estimate actual support $\overline{\omega}_{\phi}$ and confidence $\overline{\varepsilon}_{\phi}$ of a property shape ϕ from the current values ω and ε computed from the sampled data. Thus, it estimates with $\overline{\omega}_{\phi}$=$\omega_{\phi}/min(|P_r^*|/|P|, |T_r|/|T|)$ the effective support for a property shape ϕ, where ω_{ϕ} is the support computed for ϕ in the current sample, P represents all triples in $\mathcal{G}$ having property τ_p, P_r^* represents triples having property τ_p across all entities in all reservoirs, T represents all entities of type e_t in $\mathcal{G}$, and T_r represents all entities of type e_t in the reservoir. Similarly, the confidence $\overline{\varepsilon}_{\phi}$ of a property shape is estimated by replacing denominator in eq. (2) with $|T_r|$.

QSE-Approximate (query-based). We apply the same sampling technique in the query-based shapes extraction approach where in Algorithm 2 entities and their meta-data are retrieved via SPARQL queries, resulting in *query-based* QSE-Approximate.

Space Analysis. The space requirement of QSE-Approximate depends on the values of target Sampling$_{\%}$, the maximum reservoir size τ_{max}, and the number of entity types |T| in $\mathcal{G}$. In the worst case, it requires $O(2{\cdot}|T|{\cdot}\tau_{max})$, therefore while $\mathcal{G}$ can contain hundreds of millions of entities, we can still easily estimate how many distinct types are in the graph and select τ_{max} to fit the available memory.

5 EVALUATION

In the following, we evaluate our QSE solutions and their effectiveness in tackling the problem of *spuriousness* along with a comparison to existing state-of-the-art approaches.

Datasets. We selected a synthetic dataset, LUBM-500 [18], and three real-world datasets: DBpedia [4] downloaded on 01.10.2020; YAGO-4 [46], for which we use the subset containing instances from the English Wikipedia, downloaded on 01.12.2020; and WikiData [49], in two variants, i.e., a dump from 2015 [54] (Wdt15), used in the original evaluation of SheXer [12], and the truthy dump from September 2021 (Wdt21) filtered by removing non-English strings. Table 1 provides a comparison of their contents.

Experimental Setup. We have implemented QSE algorithms in JAVA-11. All experiments are performed on a single machine with Ubuntu 18.04, having 16 cores and 256 GB RAM. We have used GraphDB [16] 9.9.0 to experiment with *query-based* variants of QSE with a maximum memory usage limit of 16 GB. The source code of QSE is available as open-source [37] along with experimental settings and datasets. We have also published the extracted SHACL shapes of all our datasets on Zenodo [39]. For SheXer, we cloned its original code from GitHub and used the same settings as the original paper, i.e., default tuned parameters for the sheXing process and customized tuned parameters to output shapes equivalent to QSE.

Metrics. We measure the *running time*, and maximum *memory usage* (defined using Java `-Xmx`) during QSE shapes extraction process, and *Shape Statistics* of the output shapes.

QSE-Exact. We use QSE-Exact to extract shapes from LUBM (L), DBpedia (D), YAGO-4 (Y), and WikiData (W). The statistics of the shapes extracted from these datasets using QSE-Exact (file-based) are shown in Table 2. It shows the count of Node Shapes (NS), Property Shapes (PS), and Property Shape Constraints (PSc), i.e., literal and non-literal node types constraints. We refer to these statistics as *default shape statistics.* We initially considered SheXer [12], ShapeDesigner [5], and SHACLGEN [20] as state-of-the-art approaches [38] to compare against QSE. Among these, both ShapeDesigner and SHACLGEN load the whole graph into a triplestore similar to our QSE-Exact (query-based). Yet, their current implementations cannot handle large KGs with more than a few million triples and do not manage to extract shapes of KGs having more than some hundreds of classes. In our experiments, either they crashed because they tried to load the graph into an in-memory triplestore or required multiple hours to generate shapes for large KGs such as YAGO-4 (with 8,897 classes). Therefore, in the following, we focus our comparison on SheXer, which supports both the file-based and the query-based methods. Table 3 shows the running time and memory consumption to extract shapes for all datasets using File (**F**) and Query-based (**Q**) variants of SheXer, QSE-Exact, and QSE-Approximate. Among the *file-based* approaches, QSE-Exact is 1 order of magnitude faster than SheXer for all datasets. It consumes up to 50% less memory than SheXer to extract shapes from D, L, Y, and Wdt15, whereas SheXer goes out of memory (Out_M) for Wdt21. Similarly, among the *query-based* approaches, QSE-Exact is 1 order of magnitude faster and consumes less than 50% memory to extract shapes from D, Y, L, and Wdt15. SheXer timed out (Out_T – 24 hours) for Y and Wdt21, while QSE-Exact timed out for Wdt21 only.

Table 1: Size and characteristics of the datasets

	DBpedia	**LUBM**	**YAGO-4**	**Wdt15**	**Wdt21**
# of triples	52 M	91 M	210 M	290 M	1.926 B
# of objects	19 M	12 M	126 M	64 M	617 M
# of subjects	15 M	10 M	5 M	40 M	196 M
# of literals	15 M	5.5 M	111 M	40 M	904 M
# of instances	5 M	1 M	17 M	3 M	91 M
# of classes	427	22	8,902	13,227	82,693
# of properties	1,323	20	153	4,906	9,017
Size in GBs	6.6	15.66	28.59	42	234

Table 2: Shapes Statistics using QSE-Exact.

	NS	**PS**	**Non-Literal PSc**	**Literal PSc**
	COUNT	**COUNT/AVG**	**COUNT/AVG**	**COUNT/AVG**
LUBM	23	164 / 7.1	323 / 3.0	57 /1.0
DBpedia	426	11,916 / 27.9	38,454 / 6.9	5,335 /1.0
YAGO-4	8,897	76,765 / 8.6	315,413 / 14.5	50,708 / 1.0
Wdt15	13,227	202,085 / 15.2	114,890 / 3.0	106,599 / 1.0
Wdt21	82,651	2,051,538 / 24.8	3,765,953 / 5.6	1,113,856 / 1.0

Table 3: Running Time (T) in minutes (m) and hours (h) along with Memory (M) consumption in GB.

		DBpedia		**LUBM**		**YAGO-4**		**Wdt15**		**Wdt21**	
		T	M	T	M	T	M	T	M	T	M
	SheXer	26 m	18	58 m	33	1.9 h	24	3.2 h	59	-	Out_M
F	QSE-Exact	3 m	16	8 m	16	23 m	16	16 m	50	2.5 h	235
	QSE-Approx	**1 m**	10	**2 m**	10	**13 m**	10	**13 m**	16	**1.3 h**	32
	SheXer	9 h	65	15 h	140	Out_T	-	13 h	180	Out_T	-
Q	QSE-Exact	34 m	16	47 m	16	2.4 h	16	1.2 h	16	Out_T	-
	QSE-Approx	**16 m**	6	**3 m**	7	**39 m**	16	**49 m**	16	**5.7 h**	64

Taming spuriousness. To deal with the issue of spuriousness, we analyze the shapes extracted and kept after pruning. QSE performs *support-based shapes extraction* by producing only the shapes with support and confidence greater than or equal to a threshold specified by the user. For instance, given a minimum support threshold of 100 and minimum confidence value 25%, for every PS, QSE prunes all the PSc that do not appear with at least 100 entities or if not at least for 25% of entities for that type. We remind that the pruning of PSc has a cascading effect that also affects the pruning of PS, and the pruning of PS can, in turn, cause the pruning of NS. To study the impact of various confidence and support thresholds on the number of PSc, PS, and NS, we analyze the effect of pruning by specifying various values for confidence and support. Figure 3 shows the result of pruning PSc (3a,b), PS and NS (3c,d) for confidence >(25, 50, 75,90)% and support (≥1, >100) on DBpedia and Wdt21. Experimental results on LUBM, YAGO-4, and Wdt15 are comparable to the results presented for DBpedia and Wdt21, and are reported in the extended version of the paper[1]. In general, as expected, the results show that the higher we set the threshold for support and confidence, the higher the percentage of PSc and PS to be pruned. Precisely, DBpedia contains $11K$ PS, $38K$ non-literal, and $5K$ literal PSc (Table 2), when QSE performs pruning with confidence >25% and support ≥ 1, it prunes out 99% PSc and PS (Figure 3a,b). Similarly for Wdt21, QSE prunes 85% non-literal and 97% literal constraints, and 66% PS for confidence >25% and support ≥1 (Figure 3b). In comparison to the default shape statistics (Table 2), increasing confidence to >50%, 75%, and 90%, pruning resulted in a drastic decrease in the number of PSc and PS. In DBpedia, the majority of non-literal PSc are pruned out, and in Wdt21, the majority of literal constraints are pruned out. Pruning of NS is lower compared to PS and PSc for all combinations of support and

Table 4: QSE-Approximate: Effect of Sampling$_{\%}$ (S%) and reservoir size (τ_{max}) on Precision (P), Recall (R), and Relative Error (Δ) with min. support 1 and confidence 25% on Wdt21

S%	τ_{max}	**Property Shapes (PS)**				**Time (Min)**	**Mem (GB)**
		Real	**Sample**	**P / R**	Δ		
10%	20	698,825	470,562	1.00 / 0.61	228,263	81	16
	200	698,825	497,035	0.92 / 0.65	201,790	81	16
50%	500	698,825	548,381	0.96 / 0.79	150,444	82	24
	5000	698,825	605,785	0.96 / 0.83	93,040	95	24
100%	500	698,825	617,349	1.00 / 0.88	81,476	87	32
	5000	698,825	645,810	1.00 / 0.92	53,015	98	32

confidence, showing that almost all types are associated at least with some very common PSc, e.g., the fact to have a :name.

QSE-Approximate. The QSE-Approximate approach reduces the memory requirements of the exact approach by allowing users to specify the *sampling percentage* (Sampling$_{\%}$, S% for short) and maximum limit of the *reservoir size* (τ_{max}), i.e., the maximum number of entities to be sampled per class, to reduce the number of entities to keep in memory. Table 3 shows that among the *file-based* approaches, QSE-Approximate is the most efficient approach compared to QSE-Exact and SheXer. For example, to extract shapes from Wdt21, QSE-Approximate (with τ_{max} = 1000 and S%=100%) required almost half the time with 1 order of magnitude less memory than QSE-Exact, while SheXer could not complete the computation. Similarly, among *query-based* approaches, QSE-Approximate proved to be the only approach to extract shapes from the Wdt21 endpoint in 5.7 hours with 64 GB memory consumption. In contrast, QSE-Exact and SheXer timed out (24 hours). Analogously to Wdt21, QSE-Approximate remains 1 order of magnitude faster with 50% less memory consumption than SheXer (for both query and file-based variants) to extract shapes from D, L, Y, and Wdt15. Overall, these results show that our proposals have solved scalability issues in shape extraction approaches regardless of the type of input data source (file or endpoint). The choice of using a query-based or file-based version depends on the given setting. For instance, querying an endpoint to extract shapes can impose excessive stress on a production DBMS serving other applications. On the other hand, the file-based approach is less resource-intensive and can be used if the user can afford the cost of dumping the graph into a file.

QSE Sampling Parameters. We further evaluate the quality of the output of QSE-Approximate using multiple combinations of values for S% and τ_{max} on Wdt21 with a fixed confidence and support threshold. This analysis helps the user to choose the best values for S% and τ_{max} parameters given some memory constraints. We show the results in Table 4, where the values shown in columns *Real* and *Sampled* are extracted by QSE-Exact and QSE-Approximate, respectively. Here we skip listing values for NS as they are not affected by the values of S%, τ_{max}, confidence, and support. The results show that S%=10 and τ_{max} up to 200 provide a 92% precision for PS extracted using QSE-Approximate and pruned with support ≥1 and confidence >25%. This requires only 16 GB RAM and 81 minutes. If a machine up to 24 GB RAM is available, then S%=50% and τ_{max}=5K provide 96% precision with Δ = 93K in 95 minutes. Similarly, on a machine having 32 GB RAM, S%=100% and τ_{max}=5K provide 100% precision with Δ = 53K in 98 minutes. The non-perfect precision translates into some shapes being produced despite their support and confidence is slightly lower than required. We also see that, for very small values of τ_{max} we achieve a lower recall, meaning that some shapes that should have been produced are instead wrongly pruned. We note though that min support 1 and confidence 25% are still quite low values and the shapes produced are thus more affected by spuriousness. Nonetheless, on a standard commodity machine with 32GB we see we can easily achieve perfect precision (100%) and very high recall (92%).

We further study the effect of pruning on shapes extracted from Wdt21 using QSE-Approximate with confidence >25% and >75% having support 1, 10, and 100 (shown in Table 5). We see that with support ≥1 and confidence >25%, QSE-approximate is able to get almost all the PS extracted by QSE-Exact for Wdt21 (Figure 3d) having 89% recall and 100% precision. Additionally, upon increasing the support to 10 and 100, we notice a constant recall of around 88-99% and a slight reduction in precision, i.e., 98% and 96% with decreasing relative error (i.e., Δ). Similarly, we notice the same trend with confidence=75%. Therefore, while we very rarely overestimate the support and confidence of the shapes produced, we underestimate some of these values, although still in a few cases only.

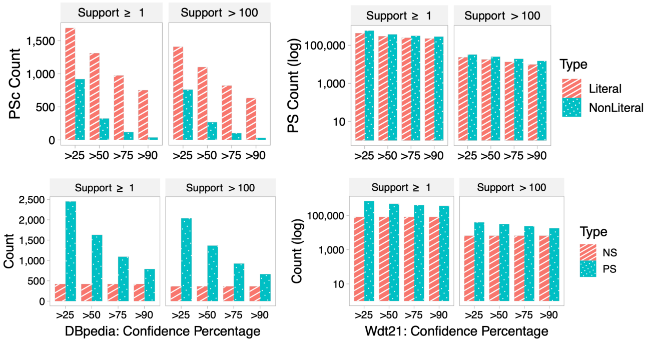


Figure 3: QSE-Exact on DBpedia and Wdt21

Practical Implications of QSE. We show the practical utility of QSE by evaluating the correctness of extracted shapes and their effect when used to validate the KG. We extracted shapes from DBpedia using QSE with confidence >25% and support >100. Then, we randomly selected 10 shapes and manually inspected them to evaluate their correctness, i.e., whether these shapes describe valid constraints. This allows us to measure precision and recall based on the pruning parameters. The results of this analysis showed that QSE extracts shapes with 100% precision in terms of correct shapes constraints that should be part of the final set of shapes (qualified as quality shapes) by removing spurious shape constraints. Further, we used these 10 shapes, extracted by QSE, to validate DBpedia using a SHACL validator and found 20,916 missing triples and 155 erroneous triples. The detailed results of this analysis are contained in the extended version[1]. Overall, this experiment shows that by using our technique the user is provided with a refined set of valid shapes that can effectively identify errors in the KG.

Constraints Coverage. Comparing the constraints supported by QSE and existing approaches (i.e., SheXer [12], SHACLGEN [20], and ShapeDesigner [5]), we report that QSE is able to extract the

Table 5: Output quality of QSE-Approximate on Wdt21 with S% = 100% and τ_{max} = 500 as # of real and sampled NS, PS, and corresponding Precision (P), Recall (R), and Relative Error Δ.

Conf	Supp	Node Shapes (NS)				Property Shapes (PS)			
		Real	Sample	P / R	Δ	Real	Sample	P / R	Δ
> 25 %	≥ 1	82,651	82,651	1.0 / 1.0	0	698,825	620,622	1.00 / 0.89	78,203
	10	23,640	23,640	1.0 / 1.0	0	158,283	141,040	0.99 / 0.88	17,243
	100	6,596	6,596	1.0 / 1.0	0	39,877	36,362	0.96 / 0.88	3,515
> 75 %	≥ 1	82,651	82,651	1.0 / 1.0	0	405,344	362,717	1.00 / 0.89	42,627
	10	23,640	23,640	1.0 / 1.0	0	91,947	83,329	0.99 / 0.90	8,618
	100	6,596	6,596	1.0 / 1.0	0	23,944	22,193	0.97 / 0.90	1,751

widest range of constraints (i.e., 15 out of 16 specific core constraints). Amongst those that are usually not supported, we support sh:in, sh:Literal, sh:class, sh:not, and sh:node. We currently do not support sh:inverse but we plan to support it in the future. More details are available in the extended version of our paper[1].

Optimal Pruning Thresholds. For each class in the KG, QSE computes its frequency. Thus, this information can be used as a reference for the support and confidence thresholds. Further, QSE also supports the extraction of shapes for specific classes only. Therefore, the user can make use of frequency information and set class-specific pruning thresholds.

6 RELATED WORK

KG Data Validation. Integrity constraints for KGs were initially defined with the RDF schema vocabulary [11] and then with the OWL language [27, 28, 47]. Later, the SPARQL Inferencing Notation (SPIN) [22] was proposed. SHACL [23] (a W3C standard since 2017) is known as the next generation of SPIN. Similar to SHACL, ShEX [35] is a constraint language that is built on regular bag expressions inspired by schema languages for XML. While ShEx is not a standard, it is used within the WikiData project [48]. Even though SHACL and ShEx are not completely equivalent [15], their core mechanism revolves around the same concept of enforcing for each node to satisfy specific constraints on the combination of its types and predicates [12]. In this work, we support the extraction of validating shapes that can be represented in both languages.

Shape Extraction. Given the abundance of large-scale KGs, various applications have been created to assist the process of extracting information about its implicit or explicit schema [21]. Among these, shapes construction or extraction approaches, i.e., to generate a set of shapes given information from an existing KG, are used in order to obtain validating schema to ensure the quality of a KG's content. We have classified existing approaches in Table 6 based on their features, i.e., support for shapes extraction from data or ontologies, support for automatic extraction of shapes, support for shapes extraction from a SPARQL triplestore, and whether they extract SHACL, ShEx, or both types of validating shapes. In our recent community survey [38] on extraction and adoption of validating shapes, we show that *there is a growing need among practitioners for techniques for efficient extraction of validating shapes from very large existing KGs.* Note that there exist approaches for schema extraction from property graphs as well [24]. Such approaches are not directly applicable to RDF KGs since their schema is more complex, moreover they focus on identifying sub-types based on node labels (which do not exist in RDF data, since types are nodes in the graph), and finally are not designed to handle the issue of spuriousness. Once shapes are extracted, they can be used to validate KGs using validation approaches like MagicShapes [2] and Trav-SHACL [13].

Table 6: State-of-the-art to extract validating shapes [38]

Approach	*Extracted from*		*Auto-*	*Triple-*	*Type*
	data	*ontology*	*matic*	*store*	
Shape Induction [26]	✓	✗	✓	✓	SHACL,ShEx
SheXer [12]	✓	✗	✓	✓	SHACL,ShEx
Spahiu et al. [45]	✓	✗	✓	✓	SHACL
ShapeDesigner. [5]	✓	✗	✓	✓	SHACL,ShEx
SHACLGEN [20]	✓	✓	✓	✓	SHACL
TopBraid [36]	✓	✓	✓	✓	SHACL
Pandit et al. [32]	✗	✓	✗	✓	SHACL
Astrea [9]	✗	✓	✓	✗	SHACL
SHACLearner [30]	✓	✗	✓	✗	SHACL
Groz et al. [17]	✓	✗	✓	✗	ShEx

Rules, Patterns, and Summaries. There exist various approaches for rule discovery in graphs [25]. These systems [1, 14, 31] derive rules from large KGs using structural information by exploring the frequently occurring graph patterns. In contrast to validating shapes, rules are mainly used to derive new facts from an incomplete KG or identify specific sets of wrong connections. Frequent subgraph mining (FPM) approaches, instead, are designed to find frequently recurring structures in a large graph. In FPM, the occurrence of subgraphs (the number of times a subgraph appears) cannot be taken as the support of subgraphs since it does not satisfy the non-monotonic property [7]. The most practical measurement for measuring this support is, instead, the minimum image-based support (MNI [7]). Our proposed definition of support for shape constraints is inspired by the concept of MNI support and its use in FPM [19]. Yet, different than FPM, we do not extract patterns of arbitrary shape and size, thus we are able to provide better performance guarantees as we solve a simpler problem. Finally, our approach is also related to the techniques of graph summarization [8] and can be seen as a special form of structural summarization [33]. Additionally, QSE provides a scalable solution for understanding the content of large KGs (by extracting their shapes) like ABSTAT-HD [3], which is based on exploring semantic profiles of large KGs.

7 CONCLUSION

In this paper, we propose an automatic shape extraction approach that addresses the two common limitations in other existing techniques, i.e., scalability and spuriousness. We addressed these limitations by introducing the Quality Shapes Extraction (QSE) problem. We devised an exact and approximate solution for QSE to enable the efficient extraction of shapes on commodity machines. Our method is based on the well-understood concepts of support and confidence, hence it allows a data scientist to focus on the shapes providing the highest reliability first when addressing issues of data quality. By setting even low pruning thresholds, QSE can prune up to 93% of the shapes that a trivial extraction would produce (i.e., a reduction of 2 orders of magnitude), shapes that hence have little support from the data and are thus likely spurious. Furthermore, we show that our approximate technique introduces only negligible loss in the quality and completeness of the produced shapes. In the future, we will extend the scope of constraints covered by QSE and a solution to automatically learn the optimal configurations for pruning thresholds for QSE.

ACKNOWLEDGMENTS

This research was partially funded by the Danish Council for Independent Research (DFF) under grant agreement no. DFF-8048-00051B, the EU's H2020 research and innovation programme under grant agreement No 838216, and the Poul Due Jensen Foundation.

REFERENCES

[1] Naser Ahmadi, Thi-Thuy-Duyen Truong, Le-Hong-Mai Dao, Stefano Ortona, and Paolo Papotti. 2020. Rulehub: A public corpus of rules for knowledge graphs. *Journal of Data and Information Quality (JDIQ)* 12, 4 (2020), 1–22.

[2] Shqiponja Ahmetaj, Bianca Löhnert, Magdalena Ortiz, and Mantas Šimkus. 2022. Magic shapes for SHACL validation. *Proceedings of the VLDB Endowment* 15, 10 (2022), 2284–2296.

[3] Renzo Arturo Alva Principe, Andrea Maurino, Matteo Palmonari, Michele Ciavotta, and Blerina Spahiu. 2022. ABSTAT-HD: a scalable tool for profiling very large knowledge graphs. *The VLDB Journal* 31, 5 (2022), 851–876.

[4] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary G. Ives. 2007. DBpedia: A Nucleus for a Web of Open Data. In *The Semantic Web, 6th International Semantic Web Conference (Lecture Notes in Computer Science)*, Vol. 4825. Springer, Busan, Korea, 722–735.

[5] Iovka Boneva, Jérémie Dusart, Daniel Fernández-Álvarez, and José Emilio Labra Gayo. 2019. Shape Designer for ShEx and SHACL constraints. In *Proceedings of the ISWC 2019 Satellite Tracks (CEUR Workshop Proceedings)*, Vol. 2456. CEUR-WS.org, Auckland, New Zealand, 269–272.

[6] Christian Borgelt. 2012. Frequent item set mining. *Wiley interdisciplinary reviews: data mining and knowledge discovery* 2, 6 (2012), 437–456.

[7] Björn Bringmann and Siegfried Nijssen. 2008. What Is Frequent in a Single Graph?. In *PAKDD (Lecture Notes in Computer Science)*, Vol. 5012. Springer, Osaka, Japan, 858–863.

[8] Šejla Čebirić, François Goasdoué, Haridimos Kondylakis, Dimitris Kotzinos, Ioana Manolescu, Georgia Troullinou, and Mussab Zneika. 2019. Summarizing semantic graphs: a survey. *The VLDB journal* 28, 3 (2019), 295–327.

[9] Andrea Cimmino, Alba Fernández-Izquierdo, and Raúl García-Castro. 2020. Astrea: Automatic Generation of SHACL Shapes from Ontologies. In *ESWC (Lecture Notes in Computer Science)*, Vol. 12123. Springer, Heraklion, Crete, Greece, 497.

[10] WWW Consortium. 2014. RDF 1.1. https://w3.org/RDF/. Accessed 17th January, 2023.

[11] WWW Consortium. 2014. RDF Schema 1.1. https://www.w3.org/TR/rdf-schema/. Accessed 17th January, 2023.

[12] Daniel Fernandez-Álvarez, Jose Emilio Labra-Gayo, and Daniel Gayo-Avello. 2022. Automatic extraction of shapes using sheXer. *Knowledge-Based Systems* 238 (2022), 107975.

[13] Mónica Figuera, Philipp D. Rohde, and Maria-Esther Vidal. 2021. Trav-SHACL: Efficiently Validating Networks of SHACL Constraints. In *WWW '21: The Web Conference 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021.* ACM / IW3C2, Ljubljana, Slovenia, 3337–3348. https://doi.org/10.1145/3442381.3449877

[14] Luis Galárraga, Christina Teflioudi, Katja Hose, and Fabian M Suchanek. 2015. Fast rule mining in ontological knowledge bases with AMIE+. *The VLDB Journal* 24, 6 (2015), 707–730.

[15] Jose Emilio Labra Gayo, Eric Prud'Hommeaux, Iovka Boneva, and Dimitris Kontokostas. 2017. Validating RDF data. *Synthesis Lectures on Semantic Web: Theory and Technology* 7, 1 (2017), 1–328.

[16] GraphDB. 2023. GraphDB. https://graphdb.ontotext.com. Accessed 17th January, 2023.

[17] Benoît Groz, Aurélien Lemay, Slawek Staworko, and Piotr Wieczorek. 2022. Inference of Shape Graphs for Graph Databases. In *25th International Conference on Database Theory, ICDT 2022, March 29 to April 1, 2022, Edinburgh, UK (Virtual Conference) (LIPIcs)*, Vol. 220. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Edinburgh, UK, 14:1–14:20. https://doi.org/10.4230/LIPIcs.ICDT.2022.14

[18] Y. Guo, Z. Pan, and J. Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics* 3 (2005), 158–182.

[19] Jiawei Han, Hong Cheng, Dong Xin, and Xifeng Yan. 2007. Frequent pattern mining: current status and future directions. *Data mining and knowledge discovery* 15, 1 (2007), 55–86.

[20] Alexis Keely. 2023. SHACLGEN. https://pypi.org/project/shaclgen/. Accessed 17th January, 2023.

[21] Kenza Kellou-Menouer, Nikolaos Kardoulakis, Georgia Troullinou, Zoubida Kedad, Dimitris Plexousakis, and Haridimos Kondylakis. 2022. A survey on semantic schema discovery. *VLDB J.* 31, 4 (2022), 675–710. https://doi.org/10.1007/s00778-021-00717-x

[22] Holger Knublauch, James A Hendler, and Kingsley Idehen. 2011. SPIN-overview and motivation. *W3C Member Submission* 22 (2011), W3C.

[23] Holger Knublauch and Dimitris Kontokostas. 2017. Shapes constraint language (SHACL). *W3C Candidate Recommendation* 11, 8 (2017).

[24] Hanâ Lbath, Angela Bonifati, and Russ Harmer. 2021. Schema Inference for Property Graphs. In *Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021.* OpenProceedings.org, Nicosia, Cyprus, 499–504. https://doi.org/10.5441/002/edbt.2021.58

[25] Michael Loster, Davide Mottin, Paolo Papotti, Jan Ehmüller, Benjamin Feldmann, and Felix Naumann. 2021. Few-Shot Knowledge Validation using Rules. In *WWW '21: The Web Conference 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021.* ACM / IW3C2, Ljubljana, Slovenia, 3314–3324. https://doi.org/10.1145/3442381.3450040

[26] Nandana Mihindukulasooriya, Mohammad Rifat Ahmmad Rashid, Giuseppe Rizzo, Raúl García-Castro, Óscar Corcho, and Marco Torchiano. 2018. RDF shape induction using knowledge base profiling. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC.* ACM, Pau, France, 1952–1959.

[27] Boris Motik, Ian Horrocks, and Ulrike Sattler. 2007. Adding Integrity Constraints to OWL. In *Proceedings of the OWLED 2007 Workshop on OWL: Experiences and Directions (CEUR Workshop Proceedings)*, Vol. 258. CEUR-WS.org, Austria.

[28] Boris Motik, Ian Horrocks, and Ulrike Sattler. 2009. Bridging the gap between OWL and relational databases. *Journal of Web Semantics* 7, 2 (2009), 74–89.

[29] Natasha Noy, Yuqing Gao, Anshu Jain, Anant Narayanan, Alan Patterson, and Jamie Taylor. 2019. Industry-scale knowledge graphs: lessons and challenges. *Commun. ACM* 62, 8 (2019), 36–43.

[30] Pouya Ghiasnezhad Omran, Kerry Taylor, Sergio José Rodríguez Méndez, and Armin Haller. 2020. Towards SHACL Learning from Knowledge Graphs. In *Proceedings of the ISWC 2020 Demos and Industry Tracks (CEUR Workshop Proceedings)*, Vol. 2721. CEUR-WS.org, Globally online, 94–99.

[31] Stefano Ortona, Venkata Vamsikrishna Meduri, and Paolo Papotti. 2018. Robust Discovery of Positive and Negative Rules in Knowledge Bases. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018.* IEEE Computer Society, Paris, France, 1168–1179. https://doi.org/10.1109/ICDE.2018.00108

[32] Harshvardhan J. Pandit, Declan O'Sullivan, and Dave Lewis. 2018. Using Ontology Design Patterns To Define SHACL Shapes. In *Proceedings of the 9th Workshop on Ontology Design and Patterns (CEUR Workshop Proceedings)*, Vol. 2195. CEUR-WS.org, Monterey, USA, 67–71.

[33] Minh-Duc Pham, Linnea Passing, Orri Erling, and Peter A. Boncz. 2015. Deriving an Emergent Relational Schema from RDF Data. In *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015.* ACM, Florence, Italy, 864–874. https://doi.org/10.1145/2736277.2741121

[34] Eric Prud'hommeaux, José Emilio Labra Gayo, and Harold R. Solbrig. 2014. Shape expressions: an RDF validation and transformation language. In *Proceedings of the 10th International Conference on Semantic Systems, SEMANTiCS 2014, Leipzig, Germany, September 4-5, 2014.* ACM, Leipzig, Germany, 32–40. https://doi.org/10.1145/2660517.2660523

[35] Eric Prud'hommeaux, José Emilio Labra Gayo, and Harold R. Solbrig. 2014. Shape expressions: an RDF validation and transformation language. In *Proceedings of the 10th International Conference on Semantic Systems, SEMANTiCS.* ACM, Leipzig, Germany, 32–40.

[36] Top Quadrant. 2023. TopBraid. https://www.topquadrant.com/products/topbraid-composer/. Accessed 17th January, 2023.

[37] Kashif Rabbani. 2023. Quality Shape Extraction - resources and source code. https://github.com/dkw-aau/qse. Accessed 17th January, 2023.

[38] Kashif Rabbani, Matteo Lissandrini, and Katja Hose. 2022. SHACL and ShEx in the Wild: A Community Survey on Validating Shapes Generation and Adoption. In *Proceedings of the ACM Web Conference 2022.* ACM, Online, Lyon, France, 260–263. https://www2022.thewebconf.org/PaperFiles/65.pdf

[39] Kashif Rabbani, Matteo Lissandrini, and Katja Hose. 2022. SHACL shapes for DBpedia, LUBM, YAGO-4, and WikiData published on Zenodo. https://doi.org/10.5281/zenodo.5958985.

[40] Kashif Rabbani, Matteo Lissandrini, and Katja Hose. 2023. SPARQL queries used for shape extraction in QSE-Exact (query-based) approach. https://github.com/dkw-aau/qse/tree/main/src/main/resources/queries. Accessed 17th January, 2023.

[41] Tomer Sagi, Matteo Lissandrini, Torben Bach Pedersen, and Katja Hose. 2022. A design space for RDF data representations. *VLDB J.* 31, 2 (2022), 347–373. https://doi.org/10.1007/s00778-021-00725-x

[42] Ognjen Savkovic, Evgeny Kharlamov, and Steffen Lamparter. 2019. Validation of SHACL Constraints over KGs with OWL 2 QL Ontologies via Rewriting. In *The Semantic Web - 16th International Conference, ESWC 2019, Portorož (Lecture Notes in Computer Science)*, Vol. 11503. Springer, Slovenia, 314–329.

[43] Stefan Schmid, Cory Henson, and Tuan Tran. 2019. Using Knowledge Graphs to Search an Enterprise Data Lake. In *The Semantic Web: ESWC 2019 Satellite Events - ESWC (Lecture Notes in Computer Science)*, Vol. 11762. Springer, Portorož, Slovenia, 262–266. https://doi.org/10.1007/978-3-030-32327-1_46

[44] Juan Sequeda and Ora Lassila. 2021. Designing and Building Enterprise Knowledge Graphs. *Synthesis Lectures on Data, Semantics, and Knowledge* 11, 1 (2021), 1–165.

[45] Blerina Spahiu, Andrea Maurino, and Matteo Palmonari. 2018. Towards Improving the Quality of Knowledge Graphs with Data-driven Ontology Patterns and SHACL. In *Emerging Topics in Semantic Technologies - ISWC 2018 Satellite Events (Studies on the Semantic Web)*, Vol. 36. IOS Press, Satellite, USA, 103–117.

[46] Thomas Pellissier Tanon, Gerhard Weikum, and Fabian M. Suchanek. 2020. YAGO 4: A Reason-able Knowledge Base. In *The Semantic Web - 17th International Conference, ESWC (Lecture Notes in Computer Science)*, Vol. 12123. Springer, Heraklion, Crete, Greece, 583–596.

[47] Jiao Tao, Evren Sirin, Jie Bao, and Deborah L. McGuinness. 2010. Integrity Constraints in OWL. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI.* AAAI Press, Atlanta, Georgia, USA.

[48] Katherine Thornton, Harold Solbrig, Gregory S Stupp, Jose Emilio Labra Gayo, Daniel Mietchen, Eric Prud'Hommeaux, and Andra Waagmeester. 2019. Using shape expressions (ShEx) to share RDF data models and to guide curation with rigorous validation. In *ESWC.* Springer, Cham, 606–620.

[49] Denny Vrandečić and Markus Krötzsch. 2014. Wikidata: a free collaborative knowledgebase. *Commun. ACM* 57, 10 (2014), 78–85.
[50] W3C. 2023. SHACL- core constraint components. https://www.w3.org/TR/shacl/#core-components. Accessed 17th January, 2023.
[51] W3C. 2023. W3C: RDF Type. http://www.w3.org/1999/02/22-rdf-syntax-ns#type. Accessed 17th January, 2023.
[52] WESO. 2023. RDFShape. http://rdfshape.weso.es. Accessed 17th January, 2023.
[53] WesoShaclConvert. 2023. SHACL to ShEx converter. https://rdfshape.weso.es/shaclConvert. Accessed 17th January, 2023.
[54] WikiData. 2023. WikiData-2015. https://archive.org/details/wikidata-json-20150518. Accessed 17th January, 2023.

A APPENDIX

This section presents a syntactical example of SHACL shapes and detailed algorithms for shape extraction using QSE (both exact and approximate).

A.1 SHACL Syntax and Constraints Coverage

In section 1, we presented a figure 1 to show an example RDF graph (1a) with its validating shapes (1b). Here, we provide a syntactical example of SHACL shapes for a :Student node shape and :takesCourse property shape in Listing 1. There exist two different types of restrictions in property shapes constraints, namely: *existential restrictions* and *property type* restrictions. The former are defined by *cardinality constraints* using sh:minCount and sh:maxCount properties, while the latter define the target type for the value node of a specific property using sh:class property.

```
sh:StudentShape   a     sh:NodeShape ;
sh:targetClass    :Student
sh:property [ a  sh:PropertyShape  ;
                     sh:path      :name  ;
                     sh:nodeKind  sh:Literal  ;
                     sh:datatype  xsd:String  ;
                     sh:minCount  1  ;
                     sh:maxCount  1  ;    ].
sh:property [ a  sh:PropertyShape  ;
                     sh:path      :takesCourse  ;
                     sh:nodeKind  sh:IRI  ;
                     sh:minCount  1  ;
                     sh:class     :Course  ;    ].
```

Listing 1: Student Node Shape having two Property Shapes

In practice, the SHACL core constraint components [50] provide a large variety of constraints to validate knowledge graphs (KGs). Our method satisfies at least the same types of shapes supported by the most extensive automatic shape extraction tools. QSE is able to generate all the constraints that are usually judged essential for finding errors in KGs, since they are implemented by all tools. Then, we also support a series of additional constraints (see Table 7). Specifically, QSE is able to extract more constraints than existing approaches like SheXer [12], SHACLGEN [20], and ShapeDesigner [5].

The only exception for QSE is the sh:inversePath constraint, this constraints applies to objects instead of subjects, e.g., marking A-hasCapital-B, will constrain nodes of type :Capital to be object of the :hasCapital. Our approach can be extended to support this type of relationships, but it requires either twice the amount of memory or to execute phase 2 an additional time. Therefore, we have not implemented it yet.

Automatic shapes extraction approaches do not support extraction of all the SHACL core constraints components [50], e.g., constraints like sh:qualifiedValueShapesDisjoint, sh:zeroOr MorePath, and sh:qualifiedValueShape require a domain expert to accurately impose these restrictions. Some constraints can be extracted from an ontology (if available) using existing approaches like Astrea [9]. We plan to extend the QSE algorithm in the future to support automatic extraction of more constraints to specify min/max length or pattern of values and min/max exclusivity and inclusivity by extending the constraints' statistics captured by the QSE algorithm.

Table 7: SHACL constraints supported by QSE, SheXer [12], SHACLGEN [20], and ShapeDesigner [5]. Supported constraints are marked (✔) and not supported constraint as (✗).

Constraint	QSE	SheXer	SHACLGEN	ShapeDesigner
sh:NodeShape	✔	✔	✔	✔
sh:PropertyShape	✔	✔	✔	✔
sh:nodeKind	✔	✔	✔	✔
sh:targetClass	✔	✔	✔	✔
sh:minCount	✔	✔	✔	✔
sh:maxCount	✔	✔	✔	✔
sh:path	✔	✔	✔	✔
sh:datatype	✔	✔	✔	✔
sh:in	✔	✔	✗	✗
sh:property	✔	✔	✔	✔
sh:IRI	✔	✔	✔	✔
sh:Literal	✔	✗	✗	✔
sh:or	✔	✗	✔	✗
sh:class	✔	✗	✔	✗
sh:not	✔	✗	✗	✗
sh:node	✔	✔	✗	✗
sh:inversePath	✗	✔	✗	✗

A.2 QSE-Exact

We presented a concise version of QSE-Exact (file-based) approach in section 3 where the algorithm consists of four phases of shape extraction. Here we present detailed algorithms for all four phases of shape extraction. The core steps of phase one and two are described in Algorithm 3. It starts by reading the file **F** line by line and processes it as a stream of $\langle s, p, o \rangle$ triples. In the first pass, the algorithm parses each $\langle s, p, o \rangle$ triple containing a type declaration and for each entity it stores the set of its entity types and their frequency, i.e., the number of instances for each class (Lines 6-10) in maps Ψ_{ETD} (Entity to Data) and Ψ_{CEC} (Class to Entity Count), respectively. Once the types of all entities and their counts are computed, in the second phase, the algorithm performs a second pass by streaming over **F** again (Lines 11-28) to collect the constraints and the meta-data required to compute support and confidence of each candidate shape. Specifically, it parses the triples to obtain subject and object types of all triples except for the type defined triples as they are already processed in the first pass. Thus, while processing each triple $\langle s, p, o \rangle$, the algorithm ignores triples specifying type declarations and obtains both the subject's type and the object of all the other triples. Here, the map Ψ_{ETD}, obtained in the first phase, is used to identify the types of non-literal objects, while the map Ψ_{ETPD} (Entity to Property Data) is updated to contain the candidate property constraints associated with each entity (Lines 13-21). This

Algorithm 3 Extract Constraints (Phase 1 and 2)

```
    Input: Graph $\mathcal{G}$ from File F
    Output: $\Psi_{ETD}$ , $\Psi_{CEC}$ , $\Psi_{CTP}$
 1: EntityData ← {T: Set_Types , $\Psi_{ETPD}$ = Map⟨Iri, PropertyData⟩}
 2: PropertyData ← {T′ : Set_ObjTypes , Count :Int}
 3: $\Psi_{ETD}$ = Map⟨Iri, EntityData⟩                      ▷ Entity to entity's data
 4: $\Psi_{CEC}$ = Map⟨Iri, Int⟩                             ▷ Class to entity count
 5: $\Psi_{CTP}$ = Map⟨Iri, Map⟨Iri, Set⟩⟩               ▷ Class to props with object types
 6: for $t \in \mathcal{G}$ ∧ t.p = Type Predicate do              ▷ ① Entity extraction
 7:     if t.s $\notin \Psi_{ETD}$ then
 8:         $\Psi_{ETD}$.insert($t.s$, new EntityData (Set.init($t.o$) ))
 9:     $\Psi_{ETD}$.insert($t.s$, $\Psi_{ETD}$.get($t.s$).T.add($t.o$))
10:     $\Psi_{CEC}$.putIfAbsent($t.o$, Set.init(0))
11:      $\Psi_{CEC}$.insert($t.o$, Set.add($\Psi_{CEC}$.get($t.o$)+1))
12: for $t \in \mathcal{G}$ ∧ t.p != Type Predicate do    ▷ ② Entity constraints extraction
13:     // Initialize a object type and property to object type tuple set
14:     Set_objTypes ← ∅ , Set_Tuple ← ∅
15:     if $t.o$ is Literal then
16:         Set_objTypes.add(getType(t.o))
17:         Set_Tuple.add(Tuple ⟨ t.p, getType(t.o) ⟩)
18:     else
19:         for Obj_type ∈ $\Psi_{ETD}$.get($t.o$) do
20:             Set_objTypes.add(Obj_type)
21:             Set_Tuple.add(Tuple ⟨t.p, Obj_type⟩)
22:     addPropertyConstraints( t.s, Set_Tuple , $\Psi_{ETD}$)
23:     for Iri ∈ $\Psi_{ETD}$.get($t.s$.T) do                           ▷ if $t.s \in \Psi_{ETD}$
24:         if $\Psi_{CTP}$.get(Iri) is ∅ then
25:             $\Psi_{CTP}$.insert(Iri, $\psi$.init($t.p$, Set_objTypes))
26:         if $\Psi_{CTP}$.get(Iri).containsKey($t.p$) then
27:             $\Psi_{CTP}$.get(Iri).get($t.p$).add(Set_objTypes)
28:         else$\Psi_{CTP}$.get(Iri).insert($t.p$, Set_objTypes)
29: function addPropertyConstraints (Iri, Set_Tuple , $\Psi_{ETD}$ )
30:     entityData = $\Psi_{ETD}$.get(Iri)
31:     for tuple t ∈ Set_Tuple do
32:         propertyData = entityData.$\Psi_{ETPD}$.get(t.$_1$)
33:         if propertyData is NULL then
34:             propertyData = new PropertyData()
35:             entityData.$\Psi_{ETPD}$.insert(t.$_1$, propertyData)
36:         propertyData.$T'$.add(t.$_2$) ; propertyData.Count += 1
37:     $\Psi_{ETD}$.insert( Iri, entityData )
```

information about the entity types and their count is stored in maps Ψ_{ETC} and Ψ_{CEC}, respectively. Additionally, the types of objects for each particular property of a class are extracted for all classes and stored in a map Ψ_{ETP}.

The constraints' information extracted in the form of maps (Ψ_{ETD}, Ψ_{CEC}) in Algorithm 3 is used as input by the Algorithm 4 in the third phase to compute support and confidence for specific constraints. In this phase, the algorithm iterates over the map Ψ_{ETD} to extract the map Ψ_{ETPD} mapping entities to candidate property shapes ϕ_s:$\langle \tau_{\text{p}}, \text{T}_{\text{p}}, \text{C}_{\text{p}} \rangle \in \Phi_s$, and retrieves the type of each entity using types information stored in Ψ_{ETD} to build triplets for each distinct triplet of the form $\langle \tau_s, \tau_p, \tau_o \rangle$ (Lines 3-10). The value of support for each distinct triplet is incremented in each iteration and stored in the map Ψ_{SUPP}. Additionally, a map Ψ_{PTT} (Property to Types) is populated with properties and their object types for all the properties having max count equal to one (Lines 11-14). Once the support for each distinct triplet is computed, the value of confidence for each triplet is computed using Equation 2 and stored in the map Ψ_{CONF} (Lines 15-16).

We now use the constraints' information extracted in the first phase and the support/confidence computed in the third phase to construct the final set of shapes in phase four (presented in Algorithm 5). The algorithm iterates over the values of the Ψ_{CTP} map and defines the *shape name* of s, the *shape's target definition* τ_s, and the set of *shape constraints* ϕ_s for each candidate class (Lines 1-3). The map value Map⟨Property, Set⟩ for each candidate class is used to extract properties and object types to define property constraints P for each *Node Shape* (Lines 4-22). The property constraints specification for P includes sh:path, and depending on sh:nodeKind either sh:class, or sh:datatype. As a matter of fact, the sh:nodeKind constraint can possibly have three types of values: sh:Literal, sh:IRI, and sh:BlankNode. In case of Literal types, the literal object type is used, e.g., xsd:string, xsd:integer, or xsd:date. However, in the case of non-literal object types, the constraint sh:class is used to declare the type of object associated with the candidate property. It is possible to have more than one value for the sh:class and sh:datatype constraints of a candidate property shape, in such cases, we use sh:or constraint to encapsulate multiple values.

Algorithm 4 Support And Confidence Computation (Phase 3)

```
    Input: $\Psi_{ETD}$ , $\Psi_{CEC}$
    Output: $\Psi_{Supp}$ , $\Psi_{Conf}$ , $\Psi_{PTT}$
 1: $\Psi_{Supp}$ = Map⟨Tuple$_3$, Int⟩ , $\Psi_{Conf}$ = Map⟨Tuple$_3$, Int⟩
 2: $\Psi_{PTT}$ = Map⟨Iri, Set_Types⟩                ▷ Property to types having max count one
 3: for ⟨Iri , E_Data⟩ ∈ $\Psi_{ETD}$ do                       ▷ $\Psi_{ETD}$ : Map⟨Iri, EntityData⟩
 4:     for $\tau_s$ ∈ E_Data.get( T: Set_Types ) do                    ▷ To compute support
 5:         PC: Set_$Tuple2$ = constructTupleSet( E_Data.get($\Psi_{ETPD}$) )
 6:         for $T(\tau_p, \tau_o)$ ∈ PC: do               ▷ PC: Property Constraints Tuple$_2$ Set
 7:             Tuple$_3$ ← $Tuple_3\langle \tau_s, \tau_p, \tau_o \rangle$
 8:             if $\Psi_{Supp}$.containsKey(Tuple$_3$) then
 9:                 $v$ = $\Psi_{Supp}$.get(Tuple$_3$) , $\Psi_{Supp}$.insert( Tuple$_3$, $v + 1$ )
10:             else $\Psi_{Supp}$.insert( Tuple$_3$, 1 )
11:     for ⟨Iri , P_Data⟩ ∈ E_Data.get($\Psi_{ETPD}$) do     ▷ To track max count cardinality
12:         if P_Data.get(Count) = 1 then
13:             $\Psi_{PTT}$.putIfAbsent(Iri, Set.init() )
14:             $\Psi_{PTT}$.get(Iri).insert(E_Data.get( T: Set_Types) )
15: for (Tuple$_3$, $supp$) ∈ $\Psi_{Supp}$ do
16:     $c$ = $\Psi_{CEC}$.get(Tuple3$_1$) , $conf = \frac{supp}{c}$ , $\Psi_{Conf}$.insert(Tuple$_3$, $conf$)
```

Algorithm 5 Shapes Extraction (Phase 4)

```
    Input: $\Psi_{CTP}$ $\Psi_{Conf}$, $\Psi_{Supp}$, $\Psi_{PTT}$ , $\omega$: min-support, $\varepsilon$: min-confidence, $\varepsilon'$: min-
    confidence for assigning cardinality constraint
    Output: $\mathcal{S}\langle s, \tau_s, \Phi_s \rangle$
 1: for (class, Map⟨Property, Set⟩) ∈ $\Psi_{CTP}$ do
 2:     $\Phi_s \leftarrow \emptyset$                ▷ $\Phi_s = \{\phi_{s_1}, \phi_{s_2}, ..., \phi_{s_n}\}$ where $\phi_s$:$\langle \tau_p, T_p, C_p \rangle$
 3:     $s$ = class.buildShapeName() , $\tau_s$ = class
 4:     for ($p$, Set_Object Types) ∈ Map⟨Property, Set⟩ do
 5:         $\phi_s.\tau_p = p$
 6:         $p.\omega$ = $\Psi_{Supp}$.get($p$, Set_Object Types)                ▷ Support of property $p$
 7:         $p.\varepsilon$ = $\Psi_{Conf}$.get($p$, Set_Object Types)             ▷ Confidence of property $p$
 8:         if $p.\omega > \omega \wedge p.\varepsilon > \varepsilon$ then
 9:             if $p$ is Literal then for each (objType ∈ Set_Object Types)
10:                 if |Set_Object Types| > 1 then encapsulate all $\phi_s$.T_p in sh:or
11:                 $\phi_s$.T_p.$add$( sh:nodeKind : Literal)
12:                 $\phi_s$.T_p.$add$( sh:datatype :objType)
13:                 if $\Psi_{PTT}$.containsKey($p$) ∧ $\Psi_{PTT}$.get($p$).exists(objType) then
14:                     $\phi_s$.C_p.$add$(sh:maxCount : 1)
15:             else for each (objType ∈ Set_Object Types)
16:                 if |Set_Object Types| > 1 then encapsulate all $\phi_s$.T_p in sh:or
17:                 $\phi_s$.T_p.$add$(sh:nodeKind : IRI)
18:                 $\phi_s$.T_p.$add$( sh:class :objType)
19:                 if $\Psi_{PTT}$.containsKey($p$) ∧ $\Psi_{PTT}$.get($p$).exists(objType) then
20:                     $\phi_s$.C_p.$add$(sh:maxCount : 1)
21:         $\phi_s$.C_p.$add$(sh:minCount : 1)                                    ▷ $if p.\varepsilon > \varepsilon'$
22:         $\Phi_s.add(\phi_s)$
23:     $\mathcal{S}.add(s, \tau_s, \Phi_s)$                                  ▷ if s.$\omega > \omega \wedge \phi_s$!∅
```

Algorithm 6 QSE Approximate - Reservoir Sampling

```
    Input: Graph $\mathcal{G}$ from File F, maximum entity threshold $\tau_{\text{max}}$, $\text{Sampling}_{\text{percentage}}$
    Output: $\Psi_{\text{ETD}}$ , $\Psi_{\text{CEC}}$ , $\Psi_{\text{CTP}}$
 1: $\Psi_{\text{ETD}}$ = Map⟨Iri, EntityData⟩        ▷ Entity to entity's data : Algorithm 3 line 1-2
 2: $\Psi_{\text{SEPC}}$ = Map⟨Iri, List<Iri>⟩                    ▷ Sampled entities per class
 3: $\Psi_{\text{RCPC}}$ = Map⟨Iri, Int⟩                          ▷ Reservoir capacity per class
 4: $\Psi_{\text{CEC}}$ = Map⟨Iri, Int⟩                               ▷ Class to entity count
 5: $\Psi_{\text{PC}}$ = Map⟨Int, Int⟩                                      ▷ Property count
 6: $\tau_{\text{min}}$ = 1 (minimum entity threshold) ; lineCounter = 0
 7: for t ∈ $\mathcal{G}$ do                                   ▷ $t$ denotes triple $\langle s, p, o \rangle$
 8:     if t.p = Type Predicate then
 9:         $\Psi_{\text{SEPC}}$.putIfAbsent(t.o, new List<>(size: $\tau_{\text{max}}$))
10:         $\Psi_{\text{RCPC}}$.putIfAbsent(t.o, $\tau_{\text{min}}$)
11:         if $\Psi_{\text{SEPC}}$.get (t.o).size() < $\Psi_{\text{RCPC}}$.get (t.o) then        ▷ Fill the reservoir
12:             if $\Psi_{\text{ETD}}$.get(t.$s$).T is $\emptyset$ then
13:                 $\Psi_{\text{ETD}}$.insert(t.$s$, new EntityData (Set.init(t.$o$) ))
14:             $\Psi_{\text{ETD}}$.insert(t.$s$, $\Psi_{\text{ETD}}$.get(t.$s$).T.add(t.$o$))
15:             $\Psi_{\text{SEPC}}$.get(t.o).insert(t.s)
16:         else                                   ▷ Replace random node in the reservoir
17:             $r$ = generateRandomNumberBetween(0, lineCounter)
18:             if $r$ < $\Psi_{\text{SEPC}}$.get(t.o).size() then
19:                 $\vec{n}$ = NULL ; $\overleftarrow{n}$ = NULL ; $\vec{n}$.SCOPE = ∞ ; $\overleftarrow{n}$.SCOPE = ∞
20:                 $\widehat{n}$ = $\Psi_{\text{SEPC}}$.get (t.o).valueAtIndex($r$))
21:                 if $r$ ! = 0 then                         ▷ Avoid first item in reservoir
22:                     $\overleftarrow{n}$ = $\Psi_{\text{SEPC}}$.get(t.o).valueAtIndex($r - 1$))
23:                     if ∃ $\overleftarrow{n}$ ∈ $\Psi_{\text{ETD}}$ then $\overleftarrow{n}$.SCOPE = $\Psi_{\text{ETD}}$.get( $\overleftarrow{n}$ ).T.size()
24:                 if $r$ ! = $\Psi_{\text{SEPC}}$.get(t.o).size() −1 then ▷ Avoid last item in reservoir
25:                     $\vec{n}$ = $\Psi_{\text{SEPC}}$.get(t.o).valueAtIndex($r + 1$))
26:                     if ∃ $\vec{n}$ ∈ $\Psi_{\text{ETD}}$ then $\vec{n}$.SCOPE = $\Psi_{\text{ETD}}$.get( $\vec{n}$ ).T.size()
27:                 $\widehat{n}$.SCOPE = $\Psi_{\text{ETD}}$.get( $\widehat{n}$ ).T.size()
28:                 $n$ = findNodeWithMinimumScope($\overleftarrow{n}, \widehat{n}, \vec{n}$)
29:                 ∀ $\Psi_{\text{ETD}}$.T → if ∃ t.o ∈ $\Psi_{\text{SEPC}}$ then $\Psi_{\text{SEPC}}$.get(t.o).removeAtIndex($n$)
30:                 $\Psi_{\text{ETD}}$.remove($n$) ; entityData = $\Psi_{\text{ETD}}$.get(t.o)
31:                 if entityData is NULL then entityData = new EntityData()
32:                 entityData.T.add (t.o) ; $\Psi_{\text{ETD}}$.insert(t.s, entityData)
33:                 $\Psi_{\text{SEPC}}$.get(t.o).add (t.s)
34:         $\Psi_{\text{CEC}}$.putIfAbsent(t.o, Set.init(0))
35:         $\Psi_{\text{CEC}}$.insert(t.o, Set.add($\Psi_{\text{CEC}}$.get($t.o$)+1))
36:                                                                 ▷ Resize reservoir
37:         currentRatio = ($\Psi_{\text{SEPC}}.get$(t.o)$.size()/\Psi_{\text{CEC}}.get$(t.o)) × 100
38:         newCapacity = $\text{Sampling}_{\text{percentage}}$ × $\Psi_{\text{SEPC}}$.get(t.o).size()
39:         if newCapacity < $\tau_{max}$ ∧ currentRatio ≤ $\text{Sampling}_{\text{percentage}}$ then
40:             $\Psi_{\text{RCPC}}$.insert(t.o , newCapacity)
41:     else $\Psi_{\text{PC}}$.putIfAbsent(t.p, 0) ; $\Psi_{\text{PC}}$.get(t.p).incrementByOne()
42:     lineCounter + +
```

A.3 QSE-Approximate

In section 4, we presented QSE-approximate approach with an abstract level of its neighbor-based dynamic reservoir sampling (NbDRS) algorithm. Here, we present a detailed version of NbDRS algorithm. The algorithm 6 presents a detailed pseudocode for the first step where graph $\mathcal{G}$ is sampled using NbDRS. It requires graph $\mathcal{G}$ (from a file **F**), sampling percentage ($\text{Sampling}_{\text{percentage}}$), and threshold for maximum number of entities to be sampled per reservoir, i.e., τ_{max}. It starts by declaring hash maps (Lines 1-5) required for storing extracted information while parsing each $\langle s, p, o \rangle$ triple t from **F** in the loop starting from Line 7. Triples are filtered based on their type predicate (such as rdf:type and wdt:P31) to extract entities' information (Line 8-40), while non-type triples are filtered on Line 41 to keep count of distinct properties using Ψ_{PC} map.

In NbDRS approach, a reservoir is maintained for each distinct entity type (e.g., reservoir for :Student and :FullProfessor in Figure 1) using a map called Ψ_{SEPC} (Sampled Entities per class). The map Ψ_{RCPC} is designated to store reservoir's capacity for each class and allows to increase its capacity dynamically. Both these maps are initialized (Line 9-10) and if the reservoir for the current type has capacity, it is filled with its entity information using Ψ_{ETD} map (Lines 11-15); otherwise a random number r is generated between zero and the current line number of **F** (Line 17). Here we use the term entity and node interchangeably. The node at index r is called focus node $\widehat{n}$, its left neighbor node $\overleftarrow{n}$ and right neighbor node $\vec{n}$ are identified and their scope is computed (Lines 19-27). The scope of a node represents the number of types of a specific entity, e.g., a person can have multiple types like Student, Teaching Assistant or a Researcher. The node with a minimum scope is marked as the node to be replaced in the reservoirs (Line 28), it is removed from the reservoirs of its type (Ψ_{SEPC}) and Ψ_{ETD} map (Lines 29-30). The new entity is added at the index of the replaced node (Line 32-33). The real entity count of each type is also computed and stored in Ψ_{CEC} (Class to Entity Count) map. This reservoir sampling approach is called dynamic as it computes the current sampling ratio (with respect to sampled entities and the total entities of the current type/class - Line 37) and new size of the reservoir using sampling percentage and count of sampled entities (Line 38). If the newly computed size is less than τ_{max} (the maximum number of entities to be sampled) and the current sampling ratio is less than the specified sampling percentage, then the reservoir capacity of the current type is increased by the newly computed reservoir's capacity (Lines 39-40). Once the sampling is performed, the sampled entities of $\mathcal{G}$ are used in the 2nd phase Algorithm 3 (From line 11).

A.4 Evaluation

We evaluated QSE with DBpedia and Wdt21 in Evaluation (Section 5). We could not present the results on LUBM, YAGO-4, and Wdt15 datasets in that section due to space limitations. Here we discuss the results of using QSE on these datasets.

A.4.1 ***YAGO-4*** . Figure 5 shows the result of PSc (5b), PS and NS (5e) pruning on YAGO-4. Our results show that applying pruning with confidence $>$ 25% and support $\geq$ 1 on YAGO-4 results in the pruning of the PSc (non-literal) up to 10x compared to default shape statistics (Table 2). Increasing the support threshold to $>$ 100 with confidence $>$ 25% shows an interesting sudden decrease in the number of PSc of two orders of magnitude. This shows an interesting fact about entities in YAGO-4, i.e., a large portion of its taxonomy contains fewer than 100 entities. More interestingly, our results show a consistent decrease in the number of constraints when increasing the confidence threshold for all support values. Analogously to these results, Figure 5e shows the same trend for pruning of PS. In comparison to DBpedia, we noticed an interesting fact about YAGO-4 and significant reduction in the number of PSc and PS by performing shapes pruning. Analogous to these results, we notice the same decreasing trend of pruning on PS and PSc by increasing the confidence percentage to 50, 75, and 90.

A.4.2 ***Wdt15*** . We performed qualitative analysis on Wdt15 using confidence values $>$ $(25, 50, 75, 90)$ and support $\geq$ $1, 100$. Figure 5c shows the results of pruning performing Property Shape constraints (PSc), i.e., literal and non-literal constraints in property shapes. Given confidence 25% and support $\geq$ 1, QSE prunes 71% of non-literal and 22% of literal constraints. Upon increasing support to $>$ 100, it further prunes out PSc by 2 orders of magnitude. Similarly, fig. 5f shows the results of pruning Node Shapes (NS) and Property Shapes (PS). The results show that QSE prunes 51% PS

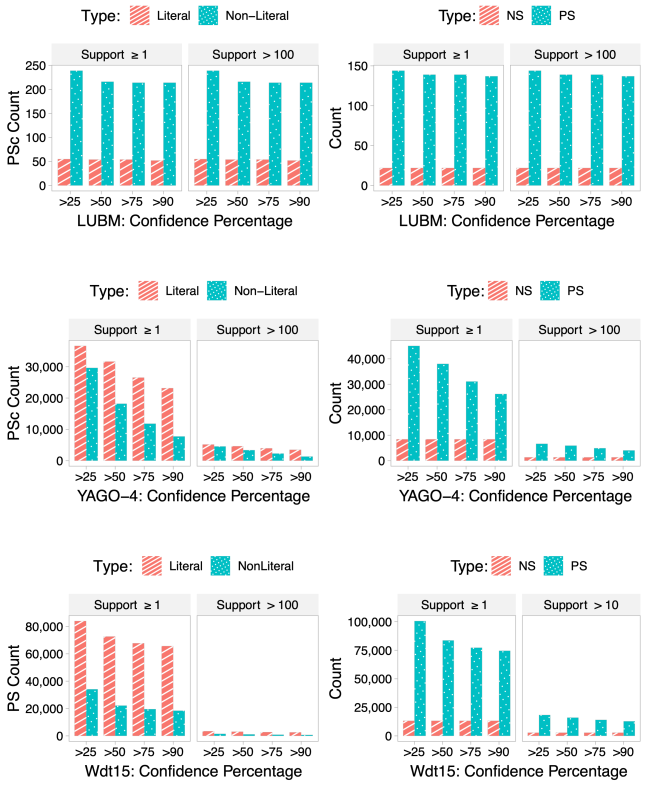


Figure 5: Pruning analysis of Node Shapes (NS), Property Shapes (PS), and Property Shapes constraints (PSc) on LUBM, YAGO-4, and Wdt15 KGs using QSE-Exact.

with confidence 25% and support ≥ 1 and 99% of property shapes with support > 100.

A.4.3 ***LUBM.*** The qualitative analysis on LUBM dataset is performed using confidence values $>$ (25, 50, 75, 90) and support ≥ 1, 100. LUBM is a synthetic dataset, it contains 23 node shapes, 164 property shapes, 323 non-literal, and 57 literal constraints. Figure 5a shows the results of pruning performing Property Shape constraints (PSc), i.e., literal and non-literal constraints in property shapes. Similarly, fig. 5d shows the results of pruning Node Shapes (NS) and Property Shapes (PS). Due to synthetic nature of LUBM knowledge graph, pruning of PSc and PS is not significant compared to DBpedia, YAGO-4, and WikiData for defined values of support and confidence.

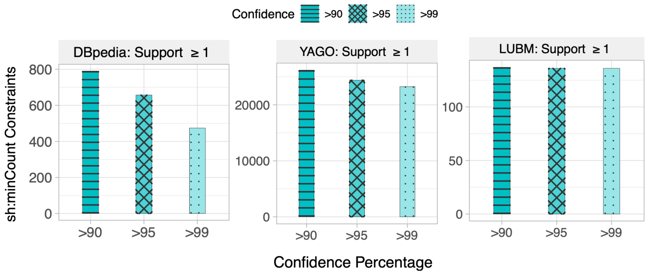


Figure 4: Cardinality constraint (sh:minCount) analysis on DBpedia, YAGO-4, and LUBM KGs.

A.4.4 ***Cardinality Constraints Analysis.*** As discussed in Section 3, QSE makes use of support and confidence computed for PSc to add a min cardinality constraint (sh:min Count) in PS. As a baseline, it assigns min cardinality to PS having a support equivalent to the number of entities of its node shape target class (Table 2). QSE also supports assigning min cardinality constraints based on the user's provided support and confidence. Figure 4 shows the number of min cardinality constraints created for PS on each dataset. Results show that the greater the value for confidence and support, the lower the number of min cardinality constraints assigned to PS. QSE supports assigning max cardinality constraints in the same way.

A.4.5 ***Computational Analysis.*** The computational complexity of SheXer is computed as $O(2{\cdot}|T| + |E|{\cdot}\frac{|\Phi_s|^2}{|S|})$. Where $2 * |T|$ refers to the parsing of all the triples twice, E is the set of entities, S is the set of Node Shapes, and Φ_s represents a set of all property shape constraints. To compare SheXer's computational complexity against QSE (section 3), we consider Wdt21 and use statistics from Tables 1 and 2 to compute complexities. To keep it simple, we ignore the number of triples and consider number of entities $E = 91M$, node shapes $S = 82,651$, and property shapes $\Phi_s = 2M$ having literal and non literal constraints $4.8M$. The results show that the complexity of QSE-Exact is lower than SheXer by one order of magnitude. QSE-approximate allows to further reduce this complexity by choosing an appropriate values for Sampling$_{\%}$ and the τ_{max}. Although, QSE is implemented in Java and SheXer is implemented in Python, both approaches have different algorithms and complexities.

A.4.6 ***QSE-Approximate vs. Standard Reservoir Sampling.*** We studied the difference between our multi-tiered dynamic reservoir sampling in QSE-Approximate with and without the option to replace neighbors of the node. We studied the relative error for various values of sampling percentage S% (i.e., 25, 50, 75, 100) and maximum size of reservoir τ_{max} (i.e., 500, 1000, 1500). The results showed that QSE-Approximate without neighbor-based sampling approach extracts between 4K to 25K less property shapes, that is, by enabling neighbor-based sampling, we reduce by 50% the relative error between the approximate and the exact method.

A.5 Practical Implication of QSE

We have performed an experiment analyzing the impact and applicability of QSE on real-world dataset. These results comply with what we believe are general expectations in this type of application, so in the main paper we only provide a short summary so that the paper is self contained. In the following we provide more extensive details.

Assume that an enterprise application is using an RDF knowledge graph in deployment and it is growing day by day. The enterprise development team would like to define or extract constraints for the knowledge graph to validate its existing data and ensure valid future insertions and updates. This is the case with WikiData or DBpedia, for instance. At this point, the team has two challenges: first, to clean the graph by finding and removing erroneous triples, and second, define shapes with minimum manual effort and resources in a reasonable amount of time. Usually, the common approach is "not to try to boil the ocean", and work progressively. We show that *QSE can help overcoming both these challenges.*

Without our approach a user can only produce all possible shapes and then has to manually inspect them. Using support and confidence allows to focus first on those shapes that are most important or, by looking at those with the lower support and confidence, find candidate spurious triples that are likely to be erroneous. In practice, QSE takes the RDF Graph $\mathcal{G}$ (as in Definition 2.1) provided as a file or a SPARQL endpoint along with pruning thresholds (*support* and *confidence*) as inputs and extracts shapes $\mathcal{S}$ (as in Definition 2.2) by removing spurious node and property shape constraints. These shapes $\mathcal{S}$ can be used for two purposes: to validate future insertions and updates in $\mathcal{G}$ (this is an important goal by itself for which having reliable shapes is fundamental!), and validate existing data in $\mathcal{G}$ using any validator to generate a validation report[2], which identifies constraints violations. We parse the validation report generated using our produced shapes and show that shapes generated by QSE help to specifically find errors of two types:

(i) **Missing information**: entities for which a specific type is not declared when it should have been, for example, :bob is a :Student who :takesCourse "Web Engineering". However, "Web Engineering" has never been defined as being of type :Course in $\mathcal{G}$.

(ii) **Spurious information**: an erroneous triple $t \in \mathcal{G}$ responsible for generating spurious shape constraints. For example, an entity e of type :City which was mistaken for an entity with similar name but of type :MusicalBand now and has been subject in triples stating that e has properties like :genre, :artist, or :bandMember, which will ultimately result in extraction of spurious shape constraints (with low support and confidence) for :City node shape.

Next, we explain this use case using an example real-world dataset.

Example: We used DBpedia and followed the reviewer's suggested protocol. At first, we randomly selected 10 classes from the DBpedia data graph $\mathcal{G}$ and extracted all possible shapes constraints $\mathcal{S}_{\mathsf{ALL}}$ (without filtering based on support and confidence), these shapes are those that would be generated by existing shapes extraction approaches (most likely to contain spurious constraints). Note, for each class multiple property constraints (PS) are generated, and we consider each set in isolation. Second, we manually inspected shape constraints from $\mathcal{S}_{\mathsf{ALL}}$ to identify $\underline{\text{C}}$orrect $\mathcal{S}_{\mathsf{C}}$ and $\underline{\text{W}}$rong $\mathcal{S}_{\mathsf{W}}$ shape constraints. We use these labels as our ground truth, thus we labelled a total of **749** property shapes. Third, we used QSE to produce only the subset with confidence >25% and support >100 to obtain a pruned set of shapes $\mathcal{S}_{\mathsf{QSE}} \subset \mathcal{S}_{\mathsf{ALL}}$. Finally, we compared $\mathcal{S}_{\mathsf{QSE}}$ (shapes produced by QSE) against the ground truth labels to compute precision and recall.

Shapes Evaluation: We show the results of our evaluation in Table 8 with statistics, precision, and recall of nodes and property shapes for $\mathcal{S}_{\mathsf{ALL}}$, $\mathcal{S}_{\mathsf{C}}$, $\mathcal{S}_{\mathsf{W}}$, and $\mathcal{S}_{\mathsf{QSE}}$. The first column in Table 8 refers to target class τ_s of node shape $\langle s, \tau_s, \Phi_s \rangle$ for each of the 10 randomly selected classes from DBpedia, the second column shows the total number of entities for a particular class value τ_s of a node shape s, while third, fourth, fifth, and sixth columns contain the number of property shapes Φ_s for $s \in \{\mathcal{S}_{\mathsf{ALL}}, \mathcal{S}_{\mathsf{C}}, \mathcal{S}_{\mathsf{W}}, \mathcal{S}_{\mathsf{QSE}}\}$, respectively. And the last four columns show true negatives, false negatives, precision, and recall for Φ_s of s. We compute the precision and recall as explained in Section 5 of the paper, for more convenience, we have also placed the detailed formulas in Appendix A.6. For instance, the first row represents analysis for :village node shape $\langle s\text{:village}, \tau_s\text{:dbo:village}^3, \Phi_{\text{:village}} \rangle$, i.e., out of 218 property shapes Φ_s of :village node shape, only 35 are relevant, and QSE is able to prune out 100% of spurious property shapes, with 0% false positives (precision 1.0) and only 11% false negatives. Thus, we can see that for all inspected classes, shapes extracted by QSE ($\mathcal{S}_{\mathsf{QSE}}$) never contain spurious constraints (i.e., they have precision 1.0 for all node and property shapes) and thus are always useful to identify errors in $\mathcal{G}$. As it is common in these situations, to obtain a higher recall the user would need to fine-tune the values of pruning thresholds (support and confidence), for example by using the frequencies of each class as a reference to find the most optimal values and improve the recall by reducing the number of false negatives. Nonetheless, this can be an iterative process, where first high quality shapes are produced and deployed without manual effort, and then the user can focus on shapes with slightly lower confidence or support. This kind of workflow is more efficient than having to inspect shapes without any insights on how reliable they can be. While tuning these parameters is left for future work, we see that our proposed values of support 100 and confidence 25 are good starting values across various KGs, so they can be used as defaults.

Errors Analysis: As mentioned, we help users identifying errors of two types, i.e., missing information and spurious information. Here, we validated DBpedia's data graph $\mathcal{G}$ using $\mathcal{S}_{\mathsf{QSE}}$ with the help of the Jena SHACL validator[4], which generates a validation report. We randomly picked one property shape for each node shape (generated for 10 randomly selected classes). The report pointed to a first set of triples for which a given type triple was not defined in $\mathcal{G}$. We show the results in Table 9 along with 'path' of the randomly picked property shape, its 'confidence' and 'support', 'all triples (T_A)', 'number of missing type triples (T_{MT})', 'number of triples not missing type triple among all', i.e., difference of T_A and T_{MT}, and their 'precision' (T_{MT}/T_A). We can see that $\mathcal{S}_{\mathsf{QSE}}$ helps user find out such missing triples with 100% precision, i.e., each error actually corresponds to a missing triple. Similarly, the report also pointed to spurious information, summarized in Table 10, where we show the range of support and confidence of property shapes for which spurious/erroneous triples were identified along with statistics such as total number of detected erroneous triples in that range, actual erroneous (via manual analysis), and not erroneous along with precision. We can see that $\mathcal{S}_{\mathsf{QSE}}$ helps users finding out such spurious information with 100% precision, i.e., behind every error in the report we found a corresponding real error in the data that needed to be fixed.

A.6 Precision and Recall

We compute the precision and recall in terms of set of retrieved shape constraints (e.g., the list of shapes extracted by a shapes

[2] https://www.w3.org/TR/shacl/#results-validation-report

[4] https://jena.apache.org/documentation/shacl

Table 8: Analysis of 10 randomly selected SHACL shapes extracted with confidence >25% and support >100.

NS Target	# Entities	# PS_ALL	# PS_Correct	# PS_Wrong	# PS_QSE	True	False	Precision	Recall
$\tau_s \in \langle s, \tau_s, \Phi_s \rangle$	$\lvert(e, type, \tau_s)\rvert$	$\lvert\Phi_s\rvert \in \mathcal{S}_{\text{ALL}}$	$\lvert\Phi_s\rvert \in \mathcal{S}_{\text{C}}$	$\lvert\Phi_s\rvert \in \mathcal{S}_{\text{W}}$	$\lvert\Phi_s\rvert \in \mathcal{S}_{\text{QSE}}$	Negatives	Negatives		
dbo:Village	214,373	218	35	183	10	183	25	1.0	0.29
dbo:Film	85,917	93	26	67	9	67	17	1.0	0.35
dbo:Book	31,296	58	29	29	10	29	19	1.0	0.34
dbo:Song	29,990	52	20	32	13	32	7	1.0	0.65
dbo:Organisation	25,826	60	24	36	7	36	17	1.0	0.29
dbo:City	20,457	107	44	63	18	63	26	1.0	0.41
dbo:Cricketer	19,862	36	18	18	9	18	9	1.0	0.50
dbo:Software	10,533	45	23	22	10	22	13	1.0	0.43
dbo:Automobile	10,276	51	39	12	19	12	20	1.0	0.49
dbo:Drug	6,845	29	13	16	9	16	4	1.0	0.69

Table 9: Analysis of missing information for randomly chosen one property shape (PS) of every Node Shape (NS).

NS Target	PS Path	PS	PS	All Triples	Missing Type	Diff (T_A, T_{MT})	Precision
		Confidence	Support	T_A	Triples T_{MT}		
dbo:Village	dbo:country	0.89	190,577	99	99	0	1.0
dbo:Film	dbo:distributor	0.25	21,712	3,680	3,680	0	1.0
dbo:Book	dbo:author	0.46	14,450	9,565	9,565	0	1.0
dbo:Song	dbo:artist	0.31	9,280	4,379	4,379	0	1.0
dbo:Organisation	N/A	N/A	N/A	N/A	N/A	N/A	N/A
dbo:City	dbo:country	0.59	12,014	67	67	0	1.0
dbo:Cricketer	dbo:birthPlace	0.4	8,000	2,034	2,034	0	1.0
dbo:Software	N/A	N/A	N/A	N/A	N/A	N/A	N/A
dbo:Automobile	dbo:relatedMeanOfTransportation	0.35	3,633	1,092	1,092	0	1.0
dbo:Drug	N/A	N/A	N/A	N/A	N/A	N/A	N/A

N/A : No property shape (PS) is identified having missing information for a given node shape (NS).

Table 10: Analysis of spurious information for all property shapes (PS) of every Node Shape (NS).

NS Target	Support Range	Confidence Range	Total No. of Errors	Erroneous	Not Erroneous	Precision
dbo:Village	[1 - 10]	[4.66E-06 - 4.66E-05]	37	37	0	1.0
dbo:Film	[1 - 9]	[1.16E-05 - 1.05E-04]	33	33	0	1.0
dbo:Book	[1 - 13]	[3.20E-05 - 4.15E-04]	18	18	0	1.0
dbo:Song	[1 - 5]	[3.33E-05 - 1.67E-04]	16	16	0	1.0
dbo:Organisation	[1 - 6]	[3.87E-05 - 2.32E-04]	12	12	0	1.0
dbo:City	[1 - 3]	[4.89E-05 - 1.47E-04]	20	20	0	1.0
dbo:Cricketer	[1]	[5.03E-05]	1	1	0	1.0
dbo:Automobile	[1]	[9.73E-05]	2	2	0	1.0
dbo:Software	[1 - 4]	[9.49E-05 - 3.80E-04]	12	12	0	1.0
dbo:Drug	[1 - 2]	[1.46E-04 - 2.92E-04]	4	4	0	1.0

extraction approach) and a set of relevant shape constraints (e.g., the list of all shapes extracted for a particular dataset). They are defined in the same way in the field of information retrieval[5]. For example, precision is computed using the following formula shown in Equation (3), which is the size of intersection of relevant shapes and retrieved shapes divided by size of retrieved shapes.

Similarly, recall is computed by the formula shown in Equation (4) which is the fraction of the relevant shapes that are successfully retrieved. For example, in the context of information retrieval, for a text search on a set of documents, recall is the number of correct results divided by the number of results that should have been returned. In our case, it is number of correct shape constraints extracted by QSE divided by the number of shape constraints that should have been retrieved.

$$\text{Precision} = \frac{|\text{relevant shapes} \cap \text{retrieved shapes}|}{|\text{retrieved shapes}|} \tag{3}$$

$$\text{Recall} = \frac{|\text{relevant shapes} \cap \text{retrieved shapes}|}{|\text{relevant shapes}|} \tag{4}$$

[5] https://en.wikipedia.org/wiki/Precision_and_recall