Creating Knowledge Graphs Subsets using Shape Expressions

Jose Emilio Labra Gayo[1][0000−0001−8907−5348]

University of Oviedo, Oviedo, Spain
`labra@uniovi.es`

Abstract. The initial adoption of knowledge graphs by Google and later by big companies has increased their adoption and popularity. In this paper we present a formal model for three different types of knowledge graphs which we call RDF-based graphs, property graphs and wikibase graphs.

In order to increase the quality of Knowledge Graphs, several approaches have appeared to describe and validate their contents. Shape Expressions (ShEx) has been proposed as concise language for RDF validation. We give a brief introduction to ShEx and present two extensions that can also be used to describe and validate property graphs (PShEx) and wikibase graphs (WShEx).

One problem of knowledge graphs is the large amount of data they contain, which jeopardizes their practical application. In order to palliate this problem, one approach is to create subsets of those knowledge graphs for some domains. We propose the following approaches to generate those subsets: Entity-matching, simple matching, ShEx matching, ShEx plus Slurp and ShEx plus Pregel which are based on declaratively defining the subsets by either matching some content or by Shape Expressions. The last approach is based on a novel validation algorithm for ShEx based on the Pregel algorithm that can handle big data graphs and has been implemented on Apache Spark GraphX.

Keywords: RDF · Shape Expressions · Knowledge graphs · Data Modelling · ShEx · Subsetting

Table of Contents

1   Introduction

The concept of Knowledge Graphs was popularized by Google in 2012 [71] as a tool that collects information about real world entities and makes relationships between them with the goal of improving search results, understand relationships better and make unexpected discoveries. Since them, there has been a tremendous interest and adoption about Knowledge Graphs, with open, general purpose ones as well as closed, proprietary ones like those employed by some big companies. In the former case we can mention DBpedia [4], YAGO [73] or Wikidata [77]. In the latter, some example companies that have announced their use are Airbnb [11], Amazon [35], eBay [61], Facebook [56], IBM [16], LinkedIn [27], Microsoft [70], etc.

There are different models associated with Knowledge Graphs like RDF-based graphs, property graphs and wikibase graphs:

- RDF-based graphs is one of the most well-known data models given that RDF was proposed as a W3C recommendation already in 1999 [57] and a large ecosystem of tools have been created around it. An important aspect of RDF is the use of URIs, which facilitates interoperability and was the basis semantic web and linked data. Around RDF, a whole ecosystem of technologies have appeared, like the SPARQL query language and protocol [26], which enables the creation of public endpoints.
- Graph databases like Neo4j [53] have also been employed to represent knowledge graphs. They have a data model which allows to annotate both nodes and edges with pairs of property-values which has become to be known as property graphs [65].
- Wikidata started in 2012 as a support project for Wikipedia but has been evolving and acquiring more and more importance as a hub of public knwowledge. The data model emplyed by Wikidata combined several aspects from RDF following linked data principles and from property graphs, allowing the annotation of statements by property-values using qualifiers and references. The software suite that implements Wikidata is known as Wikibase and can be used to represent other knowledge graphs with the same data model, we will refer to these kind of knowledge graphs as wikibase graphs. Wikidata also offers an RDF serialization format which can be accessed through a public SPARQL endpoint.

One of the key factors of knowledge graph models is their flexibility which enables easy addition of content. This flexibility also comes with a price for the applications and users that want to consume the data, which are required to use defensive programming techniques to handle lack of some mandatory properties, errors in values, duplicates, etc. It is also difficult for the producers who want to add data. Although they usually have some schema (explicit or implicit) about that represents the structure of the data, it is also difficult to document that the added data conforms to that schema.

In the case of RDF graphs, Shape Expressions (ShEx) were developed in 2014 to describe and validate the topology of RDF graphs [62]. Afterwards, ShEx was

adopted in 2019 by Wikidata to describe the RDF serialization of Wikidata content in a new namespace called entity schemas[1].

The success of Knowledge Graphs has implied that the size of their contents also increases dramatically. As an example, the size of compressed Wikidata dumps has been almost doubling every year, from 3.3Gb (31,3Gb uncompressed) in 2014 to 70.5Gb (1.256Gb uncompressed) in 2021 (see figure 1). A consequence of these huge sizes is that it is not possible to easily process all the amount of available data by conventional tools, preventing the consumers to analyze and process the content and threatens these technologies to be victims of their own success.

An example of this situation happens in Scholia [55] a web application that leverages on Wikidata to represent information about scholars and their works. The application also contains nice visualizations and comparisons which are based on queries over Wikidata endpoint. Although the project provides a lot of interesting information, the more complex visualizations are not possible to obtain because the huge amounts of data generate timeouts.

In order to address these issues, a possible approach is to create subsets of the Knowledge Graphs for some domains. These subsets can capture snapshots of the content at some specific moment and be used to improve the performance of the applications that consume that data, facilitating research work over Knowledge Graphs contents.
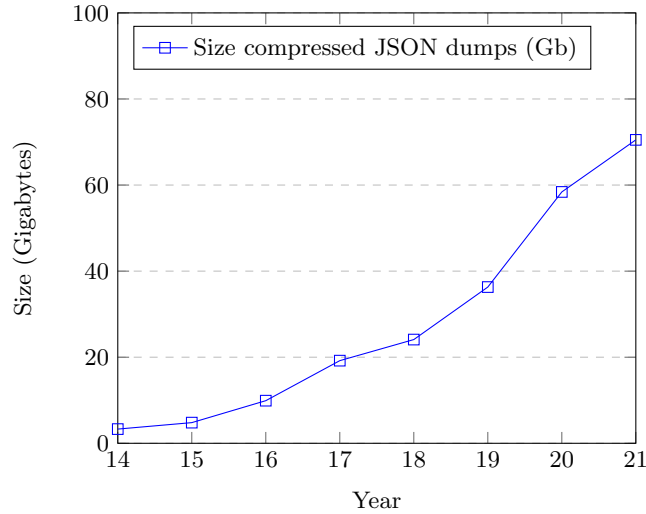

Fig. 1. Size compressed wikidata Json Dumps between 2014-2021. Source: http://archive.org

---

[1] https://www.wikidata.org/wiki/Wikidata:Schemas

In this paper we review different approaches to generate knowledge graphs subsets. Given that the first step to create a subset is to describe the intended content, ShEx seems a natural choice to be used for those descriptions. In this way, some approaches are based on ShEx schemas which describe the intended content of the subsets. To that end, we define two extensions of ShEx: PShEx to describe property graphs and WShEx to describe wikibase graphs.

We define the following approaches to generate subsets for Wikibase graphs which could also be applied to RDF graphs and property graphs:

– *Entity-matching* defines a subset by identifying some target entities. The subset contains information related with those entities and their neighborhood.
– *Simple-matching* defines a subset by a set of matching patterns, for example, the triples that have a given property, that satisfy some condition, etc.
– *ShEx-based matching* uses ShEx shapes without taking into account shape references to check which nodes conform to them. This approach only requires to take into account the neighborhood of a node and can be used to sequentially process the dumps without requiring graph traversal.
– *ShEx+Slurp* consists on validating the graph contents using ShEx and collect the nodes and triples that are being visited during the validation. This approach can refine the obtained subsets but requires graph traversal, which can be difficult when sequentially processing the dumps. If it is used against an endpoint, it can exceed the limit of allowed requests by client.
– *ShEx+Pregel* proposes to validate the graph using an adaptation of the Pregel algorithm [42] for ShEx validation. This approach can process and validate big knowledge graphs. It has the advantage of scalability and in principle, it can handle graph traversal, but it also consumes a large amount of resources.

The main contributions of this paper are:

– We created a formal model for Wikibase graphs which can be compared with the formal model of RDF-graphs and property graphs.
– We created two extensions of ShEx: PShEx for property graphs and WShEx for wikibase graphs.
– We identify and formally describe five approaches to generate knowledge graphs subsets. Some of them, like *Simple matching* and *ShEx+Slurp* had already been implemented but were not formally described.
– We describe and implement an algorithm for large scale validation of knowledge graphs based on Pregel.

The structure of the paper is as follows: Section 2 presents some preliminary definitions about sets and graphs. Section 3 introduces knowledge graphs and presents 3 main types of knowledge graphs: RDF-based, Property graphs and Wikibase graphs. Section 4 presents techniques to describe knowledge graphs: ShEx for RDF graphs, PShEx for property graphs and WShEx for wikibase graphs. For each of them, we define the abstract syntax and the semantics using

inference rules. Section 5 presents the problem of creating subsets of knowledge graphs and describes several approaches to create subsets of wikibase graphs. Finally, section 7 reviews the related work and section 8 presents some conclusions. Along the paper we use a running example based on information about Tim Berners-Lee whose information was obtained from Wikidata (entity Q80).

2   Preliminaries

In this section, we provide some basic definitions that we will use in the rest of the paper.

Sets . The finite set with elements $a_1, \ldots, a_n$ is written $\{a_1, \ldots, a_n\}$, $\emptyset$ represents the empty set, $S_1 \cup S_2$ is the union of sets S_1 and S_2, $S_1 \cap S_2$ the intersection and $S_1 \times S_2$ the Cartesian product. $FinSet(S)$ represents the set of all finite subsets of S. A tuple $\langle A_1, \ldots A_n \rangle$ is the cartesian product $A_1 \times \cdots \times A_n$.

Given a set S, its set of partitions is defined as $part(s) = \{(s_1, s_2) \mid s_1 \cup s_2 = s \wedge s_1 \cap s_2 = \emptyset\}$.

Definition 1 (Graph). *A* graph *is a tuple* $G = \langle \mathcal{V}, \mathcal{E} \rangle$*, where* $\mathcal{V}$ *is a set of nodes, and* $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{E} \times \mathcal{V}$ *is a set of edges.*

A multigraph is a graph where it is possible to have more than one edge between the same two nodes.

Definition 2 (Directed edge-labelled graph). *A* directed edge-labelled graph *is a tuple* $\mathcal{G} = \langle \mathcal{V}, \mathcal{E}, \mathcal{P} \rangle$*, where* $\mathcal{V}$ *is a set of nodes,* $\mathcal{P}$ *is a set of labels also called predicates or properties, and* $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{L} \times \mathcal{V}$ *is a set of edges.   Each element* $(x, p, y) \in \mathcal{E}$ *is called a triple, where x is the subject, p is the predicate or property and y is the object.*

Definition 3 (Triple-based graphs). *A* triple-based graphs *is a directed edge-labelled graph* $\mathcal{G} = \langle \mathcal{S}, \mathcal{P}, \mathcal{O}, \rho \rangle$ *where* $\mathcal{S}$ *is a set of subjects,* $\mathcal{P}$ *is a set of predicates or properties and* $\mathcal{O}$ *is a set of objects or values, and* $\rho \subseteq \mathcal{S} \times \mathcal{P} \times \mathcal{O}$*. Those sets don't need to be disjoint, and usually* $\mathcal{P} \subseteq \mathcal{S} \subseteq \mathcal{O}$*.*

Definition 4 (Hypergraph). *A* hypergraph *is a tuple* $G = \langle \mathcal{V}, \mathcal{E} \rangle$ *where* $\mathcal{V}$ *is a set of nodes and* $\mathcal{E} \subseteq FinSet(\mathcal{V})$ *is a set of edges. Notice that* $\mathcal{E}$ *is a family of subsets of vertices.*

Definition 5 (Bag [72]).   *Given a set of symbols* Δ*, a* bag *w (also called* multiset*) can be seen as a set whose elements can be repeated and it is defined as a function* $w : \Delta \mapsto \mathbb{N}$ *that maps a symbol to the number of its occurrences. The set of all bags over* Δ *is denoted as* $Bag[\Delta]$*. The empty bag* ϵ *has 0 occurrences for every symbol* $a \in \Delta$*. A bag is usually represented as* $\{|0|\}a \ldots$ *with elements that can be repeated. The union of two bags* w_1 *and* w_2 *is defined as* $w_1 \uplus w_2(a) = w_1(a) + w_2(a)$*. A set of bags is a* bag language*. The bag union of two languages* L_1 *and* L_2 *is defined as* $L_1 \uplus L_2 = \{w_1 \uplus w_2 \mid w_1 \in L_1, w_2 \in L_2\}$*.*

Definition 6 (Regular bag expression [72]). *A regular bag expression over a set of symbols Δ is defined by the following grammar: $E ::= \varepsilon \big| a \big| E \mid E \big| E; E \big| E*$, where $a \in \Delta$.*

A regular bag expression e defines a bag language $[\![e]\!]$ whose semantics is: $[\![\varepsilon]\!] = \{\epsilon\}, [\![a]\!] = \{\{|0|\}a\}, [\![e_1 \mid e_2]\!] = [\![e_1]\!] \cup [\![e_2]\!], [\![e_1; e_2]\!] = [\![e_1]\!] \uplus [\![e_2]\!], [\![e*]\!] = \bigcup_{i \geq 0} [\![e]\!]^i$. A bag b matches a regular bag expression e, denoted as $b \cong e$ if $b \in [\![e]\!]$.

Definition 7 (Shape assignment). *Given a graph $\mathcal{G}$ with vertex set $\mathcal{V}$ and a finite set of labels $\mathcal{L}$, a shape assignment over $\mathcal{G}$ and $\mathcal{L}$ is a subset of $\mathcal{V} \times \mathcal{L}$. We use τ to denote shape assignments, and we write $n@l$ instead of (n, l) for elements of shape assignments. Note that shape assignments correspond to shape maps in [63] and typings in [6, 72].*

3 Knowledge graphs models

Although the term *knowledge graphs* was already in use in the 1970s [66], the current notion of knowledge graphs was popularized by Google in 2012 [71]. We adopt an informal definition of knowledge graphs which has been inspired by Hogan et al [30]:

Definition 8 (Knowledge graph). *A Knowledge graph is graph of data intended to represent knowledge of some real world domain, whose nodes represent entities of interest and whose edges represent relations between these entities.*

The previous definition is deliberately open. The main feature of a knowledge graph is that it is intended to represent information about entities of some real world domain using a graph-based data structure.

Knowledge graphs are usually classified by:

- Licence/proprietor: There are public and open knowledge graphs like Yago [73], DBpedia [40] or Wikidata [77] as well as enterprise-based and proprietary knowledge graphs [56] like Google, Amazon, etc.
- Scope: there are general-purpose knowledge graphs which contain information about almost all domains like Wikidata as well as domain specific knowledge graphs which contain information from some specific domains like healthcare, education, chemistry, biology, cybersecurity, etc. [1]

Knowledge graphs can be represented using multiple technologies and in fact, the information about how Google's knowledge graph is implemented is not public. Nevertheless, in this paper, we will focus on three main technologies:

- **RDF based knowledge graphs** represent information using directed graphs whose edges are labels.
- **Property graphs** allow property–value pairs and a label to be associated with nodes and edges. Property graphs have been implemented by several popular graph databases like Neo4j [3].

– **Attributed graphs** allow property-value pairs associated with edges to add meta-data about the relationship represented by the edge and the values of those properties can themselves be nodes in the graph. The main example in this category is Wikidata where property-value pairs encode qualifiers and references.

3.1   RDF based knowledge graphs

Resource Description Framework (RDF) [14], is a W3C recommendation which is based on directed edge-labelled graphs.

The RDF data model defines different types of nodes, including *Internationalized Resource Identifiers* (*IRIs*) [17] which can be used to globally identify entities on the Web; literals, which allow for representing strings (with or without language tags) and values from other datatypes (integers, decimals, dates, etc.); and *blank nodes*. Blank nodes can be considered as existential variables that denote the existence of some resource for which an IRI or literal is not known or provided. They are locally scoped to the file or RDF store, and are not persistent or have portable identifiers [29].

Definition 9 (RDF Graph). *Given a set of IRIs $\mathcal{I}$, a set of blank nodes $\mathcal{B}$ and a set of literals Lit, an* RDF *graph is a triple based graph $\mathcal{G} = \langle \mathcal{S}, \mathcal{P}, \mathcal{O}, \rho \rangle$ where $\mathcal{S} = \mathcal{I} \cup \mathcal{B}$, $\mathcal{P} = \mathcal{I}$, $\mathcal{O} = \mathcal{I} \cup \mathcal{B} \cup Lit$ and $\rho \subseteq \mathcal{S} \times \mathcal{P} \times \mathcal{O}$*

There are several syntaxes for RDF graphs like Turtle, N3, RDF/XML, etc. In this document, we will use Turtle.

Example 1. As a running example, we will represent information about Tim Berners-Lee declaring that he was born in London, on 1955, and was employed by CERN and London's country is UK. That information can be encoded in Turtle as:

```
prefix :      <http://example.org/>
prefix xsd:   <http://www.w3.org/2001/XMLSchema#>

:timbl    :birthPlace     :London ;
          :birthDate      "1955-08-06"^^xsd:date ;
          :employer       :CERN .
:London   :country        :UK .
```

Figure 2 shows a possible visualization of that RDF graph using RDFShape, a tool developed by the authors of this paper which allows to play with RDF graphs[2] [36]:

The neighbors of a node $n \in \mathcal{V}$ in an RDF graph $\mathcal{G}$ are defined as $neighs(n, \mathcal{G}) = \{(n, p, y) \mid (n, p, y) \in \mathcal{G}\}$.

---

[2] It is possible to interactively play with the example following this permalink: https://rdfshape.weso.es/link/16344135752
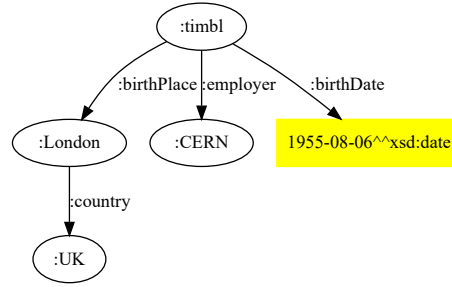
Fig. 2. Example graph representation of RDF data

RDF can be considered the basic element of the semantic web technology stack, forming a simple knowledge representation language on top of which several technologies have been developed like SPARQL for querying RDF data as well as RDFS and OWL to describe vocabularies and ontologies.

RDF reification and RDF-* An important aspect of RDF as a knowledge representation formalism is to be able to represent information about RDF triples themselves, which is called reification. In this section we will present some of the possible approaches for reification using a simple example to help understand the approach used by Wikibase to serialize its data model to RDF [28]. We also present the RDF-* approach which has become popular in the RDF-ecosystem with its support by several RDF stores like GraphDB [3].

Example 2. As an example, we may want to qualify the statement that Tim Berners-Lee was employed by CERN, declaring that he was employed at two different points in time: in 1980 and between 1984 and 1994.

The following approaches have been proposed for RDF reification:

– Standard RDF reification was introduced in RDF 1.0 [45]. It consists of using the predicates `rdf:subject`, `rdf:predicate` and `rdf:object` as well as the class `rdf:Statement` to explicitly declare statements.

```
_:s1 rdf:type        rdf:Statement ;
     rdf:subject     :timbl ;
     rdf:predicate   :employer ;
     rdf:object      :CERN ;
     :start          "1980"^^xsd:gYear ;
     :end            "1980"^^xsd:gYear .
_:s2 rdf:type        rdf:Statement ;
     rdf:subject     :timbl ;
     rdf:predicate   :employer ;
     rdf:object      :CERN ;
     :start          "1984"^^xsd:gYear ;
     :end            "1994"^^xsd:gYear .
```

_____

[3] https://www.ontotext.com/knowledgehub/fundamentals/what-is-rdf-star/

- Create a statement that models the *n*-ary relation [19]. For example, we can create two nodes :s1 and :s2 to represent the the 2 employments of Tim-Berners-Lee at CERN.

```
:timbl :employer :s1, :s2.
:s1  :employerV :CERN;
     :start      "1980"^^xsd:gYear ;
     :end        "1980"^^xsd:gYear .
:s2  :employerV :CERN;
     :start      "1984"^^xsd:gYear ;
     :end        "1994"^^xsd:gYear .
```

- Create *singleton properties* for each statement and link those properties with a specific predicate to the real property [54].

```
:timbl :employer1 :CERN ;
       :employer2 :CERN .

:employer1 :singletonPropertyOf :employer ;
 :start "1980"^^xsd:date ;
 :end   "1980"^^xsd:date .

:employer2 :singletonPropertyOf :employer ;
 :start "1984"^^xsd:date ;
 :end   "1994"^^xsd:date .
```

- RDF1.1 [14] included the concept of named graphs, which can be used to associate each triple with a different graph.

```
:g1 :timbl    :employer :CERN .
:g1 :employed :start     "1980"^^xsd:date  .
:g1 :employed :end       "1980"^^xsd:date  .
:g2 :timbl    :employer :CERN .
:g2 :employed :start     "1984"^^xsd:date  .
:g2 :employed :end       "1994"^^xsd:date  .
```

- RDF-* [15] has been recently introduced as an extension of RDF that includes RDF graphs as either the subjects or objects of a statement.

```
<<:timbl :employer :CERN>> :start "1980"^^xsd:gYear ;
                           :end   "1980"^^xsd:gYear .
<<:timbl :employer :CERN>> :start "1984"^^xsd:gYear ;
                           :end   "1994"^^xsd:gYear .
```

- Wikidata's RDF serialization follows a hybrid approach using a direct link to capture the preferred value and singleton nodes that represent the statements capturing the *n*-ary relationship [19]. It also follows a convention that employs the same local name of the property preceded by different namespaces: wdt: for the direct link, p for the link between the node and the singleton statements, ps: for the link between the singleton statements and the values, and pq: for the link between the singleton statements and the

qualified values. The previous example using Wikidata RDF serialization could be [4]:

```
:timbl wdt:employer :CERN .
:timbl p:employer :s1 .
:timbl p:employer :s2 .
:s1    ps:employer :CERN ;
       pq:start "1980"^^xsd:gYear ;
       pq:end   "1980"^^xsd:gYear .
:s2    ps:employer :CERN ;
       pq:start "1984"^^xsd:gYear ;
       pq:end   "1994"^^xsd:gYear .
```

3.2 Property graphs

Property graphs have become popular thanks to several commercial graph databases like Neo4j [5], JanusGraph [6] or Sparksee [7]. A property graph has unique identifiers for each node/edge and allows to add property-value annotations to each node/edge in the arc as well as type annotations.

The following definition of a property graph follows [68].

Definition 10 (Property graph). *Given a set of types $\mathcal{T}$, a set of properties $\mathcal{P}$, and a set of values $\mathcal{V}$, a* property graph *$\mathcal{G}$ is a tuple $\langle \mathcal{N}, \mathcal{E}, \rho, \lambda_n, \lambda_e, \sigma \rangle$ where $\mathcal{N} \cap \mathcal{E} = \emptyset$, $\rho : \mathcal{E} \mapsto \mathcal{N} \times \mathcal{N}$ is a total function, $\lambda_n : \mathcal{N} \mapsto FinSet(\mathcal{T})$, $\lambda_e : \mathcal{E} \mapsto \mathcal{T}$, and $\sigma : \mathcal{N} \cup \mathcal{E} \times \mathcal{P} \mapsto FinSet(\mathcal{V})$.*

A property graph is formed by a set of node identifiers $\mathcal{N}$ and a set of edges $\mathcal{E}$ where ρ associates a pair of nodes (n_1, n_2) to every $e \in \mathcal{E}$ where n_1 is the subject and n_2 is the object, λ_n associates a set of types for node identifiers (notice that property graphs allow nodes to have more than one type), λ_e associates a types for each edge identifier, and σ associates a set of values to pairs (i, p) such that $i \in \mathcal{N} \cup \mathcal{E}$ is a node or edge and $p \in \mathcal{P}$ is a property.

Example 3. As an example, we will represent information about Tim Berners-Lee in a property graph encoding his birth place with a relation to a node that represents London, and his birth date with a value for that property in the same node. We can also represent that its employer has been CERN in two times, one in 1980, and another between 1984 and 1994.

---

[4] We omit the representation of values and use English names instead of numbers for clarity

[5] https://neo4j.com/

[6] https://janusgraph.org/

[7] https://www.sparsity-technologies.com/#sparksee

$\mathcal{T} = \{\text{Human}, \text{City}, \text{Metropolis}, \text{Country}, \text{Organization}, \text{birthPlace}, \text{country}, \text{employer}\}$

$\mathcal{P} = \{\text{label}, \text{birthDate}, \text{start}, \text{end}\}$

$\mathcal{V} = \{\text{Tim Berners-Lee}, 1955, 1980, 1984, 1994, \text{London}, \text{UK}\}$

$\mathcal{N} = \{n_1, n_2, n_3, n_4\} \qquad \mathcal{E} = \{r_1, r_2, r_3, r_4\}$

$\rho = r_1 \mapsto (n_1, n_2), r_2 \mapsto (n_2, n_3), r_3 \mapsto (n_1, n_4), r_4 \mapsto (n_1, n_4)$

$\lambda_n = n_1 \mapsto \{\text{Human}\}, n_2 \mapsto \{\text{City,Metropolis}\}, n_3 \mapsto \{\text{Country}\}, n_4 \mapsto \{\text{Organization}\}$

$\lambda_e = r_1 \mapsto \text{birthPlace}, r_2 \mapsto \text{country}, r_3 \mapsto \text{employer}, r_4 \mapsto \text{employer}$

$\sigma = (n_1, label) \mapsto \text{Tim Berners-Lee}, (n_1, birthDate) \mapsto 1955$

$\quad (n_2, \text{label}) \mapsto \text{London}\}, (n_3, \text{label}) \mapsto UK, (n_4, \text{label}) \mapsto \text{CERN}$

$\quad (r_3, \text{start}) \mapsto 1980, (r_3, \text{end}) \mapsto 1980, (r_4, \text{start}) \mapsto 1984, (r_4, \text{end}) \mapsto 1994$

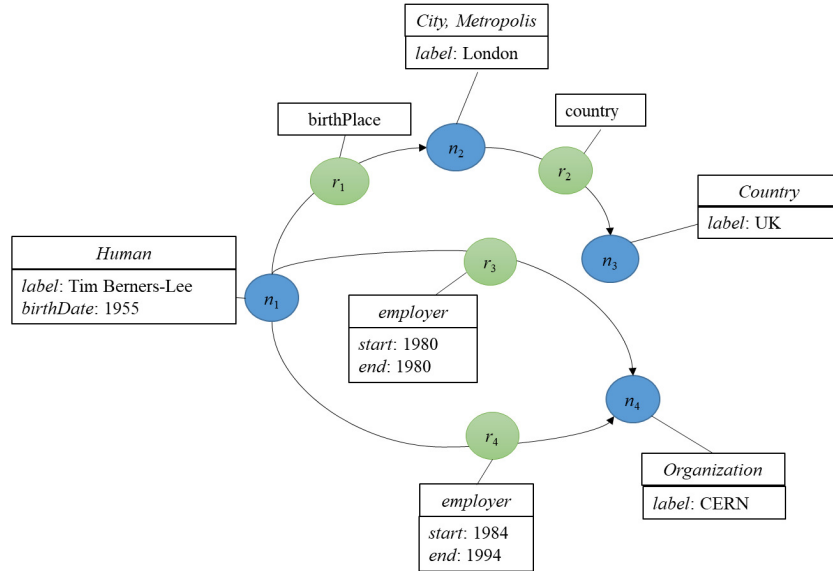Figure 3 presents a possible visualization of a property graph.

Fig. 3. Example graph visualization of a property graph

Cypher is a property graph query language that was initially developed for Neo4j [21]. Figure 4 presents an example Cypher script that can can generate the property graph represented in figure 3.

Notice that it is possible to have more than one edge between nodes in property graphs, so they can be considered multigraphs.

```
CREATE (n1:Human {label:'Tim Berners-Lee', birthDate:1955})
CREATE (n2:City:Metropolis {label:'London'})
CREATE (n3:Country {label:'UK'})
CREATE (n4:Organization {label:'CERN'})
CREATE
  (n1)-[:birthPlace]->(n2),
  (n2)-[:country]->(n3),
  (n1)-[:employer {start:[1980], end: [1980]}]->(n4),
  (n1)-[:employer {start:[1984], end:[1994]}]->(n4),
```

Fig. 4. Cypher code to generate a sample property graph

3.3   Wikibase graphs

Wikidata[8] started in 2012 to support Wikipedia [77]. It has become one of the biggest human knowledge bases, maintained both by humans collaboratively as by bots, which update the contents from external services or databases. Several organizations are donating their data to Wikidata and collaborate in its maintenance providing resources. A remarkable case is Google, which migrated its previous knowledge graph Freebase to Wikidata in 2017 [74].

Apart of Wikipedia, Wikidata has been reported to be used by external applications like Apple's Siri [9] and it has been adopted as the central hub for knowledge in several domains like life sciences [9], libraries [67] or social science [13]. As of August, 2021, it contains information about more than 94 millions of entities [10] and since its launch there have been more than 1,400 millions of edits.

Wikibase [11] is a set of open source tools which run Wikidata. With Wikibase it is possible to create Knowledge graphs that follow the same data model as Wikidata but that represent information from other domains. The projects that are using Wikibase are called Wikibase instances, some examples of wikibase instances are Rhizome [12] or Enslaved [13]. Given that Wikidata was the first and most common Wikibase instance the terms are sometimes used indistinctly.

Wikibase was initially created from MediaWiki software which ensured adoption by the Wikimedia community. Internally, Wikidata content is managed by a relational database (MariaDB) which consists of strings stored and versioned as character blobs [44]. but was not suitable for advanced data analysis and querying. With the goal of facilitating those tasks and integrate Wikibase within the semantic web ecosystem, the Wikimedia Foundation adopted BlazeGraph [14] as a complementary triplestore and graph database. In this way, there are 2 main

---

[8] http://wikidata.org/

[9] https://lists.wikimedia.org/pipermail/wikidata/2017-July/010919.html

[10] https://www.wikidata.org/wiki/Wikidata:Statistics

[11] https://wikiba.se/

[12] https://rhizome.org/about/

[13] https://enslaved.org/

[14] https://blazegraph.com/

data models that coexist in Wikibase: a document-centric model based on Me-
diaWiki and an RDF-based model based on RDF which can be used to do
SPARQL queries through the Query Service.

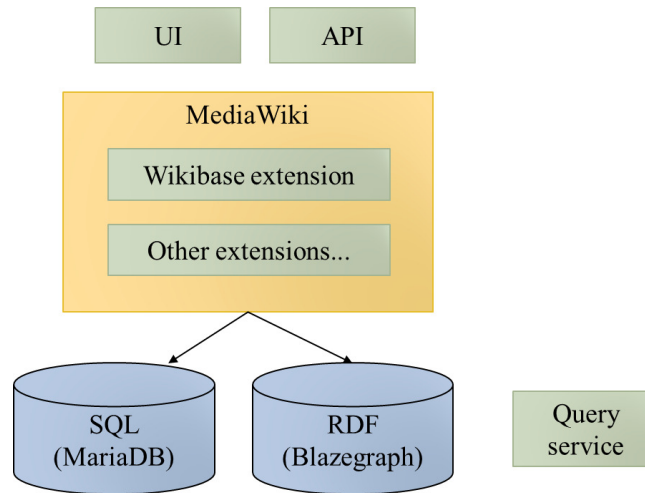A simplified view of Wikibase architecture is depicted in figure 5 [15].

Fig. 5. Simplified architecture of Wikibase

Wikibase data model: informal introduction The Wikibase data model [16]
is defined as an abstract data model that can have different serializations like
JSON and RDF. It is defined using UML data structures and a notation called
Wikidata Object Notation.

Informally, the Wikibase data model is formed from entities and statements
about those entities. An entity can either be an item or a property. An item
is usually represented using a Q followed by a number and can represent any
thing like an abstract of concrete concept. For example, Q80 represents Tim
Berners-Lee in Wikidata.

A property is usually represented by a P followed by a number and represents
a relationship between an item and a value. For example, P19 represents the
property *place of birth* in Wikidata.

The values that can be associated to a property are constrained to belong
to some specific datatype. There can be compound datatypes like geographical
coordinates.

---

[15] A more in-depth view of Wikibase architecture can be found at https://addshore.
com/2018/12/wikidata-architecture-overview-diagrams/
[16] https://www.mediawiki.org/wiki/Wikibase/DataModel

Some of Wikibase datatypes are: quantities, dates and times, geographic locations and shapes, monolingual and multilingual texts, etc.

A statement consists of:

- A property which is usually denoted using a `P` followed by a number.
- A declaration about the possible value (in wikibase terms, it is called a *snak*) which can be a specific value, `no value` declaration or a `some value` declaration.
- A rank declaration which can be either `preferred`, `normal` or `deprecated`.
- Zero or more qualifiers which consist of a list of property-value pairs
- Zero or more references which consist of a list of property-value pairs.

Wikibase data model: formal definition We define a formal model for Wikibase which is inspired from Multi-Attributed Relational Structures (MARS) [51]. For brevity, we model both qualifiers and references as attributes and don't handle the no-value and some-value snaks.

Definition 11 (Wikibase graphs). *Given a mutually disjoint set of items $\mathcal{Q}$, a set of properties $\mathcal{P}$ and a set of data values $\mathcal{D}$, a* Wikibase graph *is a tuple $\langle \mathcal{Q}, \mathcal{P}, \mathcal{D}, \rho \rangle$ such that $\rho \subseteq \mathcal{E} \times \mathcal{P} \times \mathcal{V} \times FinSet(\mathcal{P} \times \mathcal{V})$ where $\mathcal{E} = \mathcal{Q} \cup \mathcal{P}$ is the set of entities which can be subjects of a statement and $\mathcal{V} = \mathcal{E} \cup \mathcal{D}$ is the set of possible values of a property.*

In practice, Wikibase graphs also add the constraint that every item $q \in \mathcal{Q}$ (or property $p \in \mathcal{P}$) has a unique integer identifier $q^i \in \mathbb{N}$ ($p^i \in \mathbb{N}$).

In the Wikibase data model, statements contain a list of property-values and the values can themselves be nodes from the graph. This is different from property graphs, where the set of vertices and the set of values are disjoint.

Example 4 (Running example as a Wikibase graph). We continue with the running example about Tim Berners-lee, but extend it with more information about awards. More concretely, we add the information that Tim Berners-Lee was awarded with the *Princess of Asturias* (PA) award together with Vinton Cerf ($vintCerf$) [17], and that the country of that award is Spain:

---

[17] The award was really obtained by Tim Berners-Lee, Vinton Cerf, Robert Kahn and Lawrence Roberts, we included here only the first two for simplicity

$\mathcal{Q} = \{ \ timBl, \ vintCerf, \ London, \ CERN, \ UK, \ Spain, \ PA, \ Human\}$
$\mathcal{P} = \{ \ birthDate, \ birthPlace, \ country, \ employer, \ awarded,$
$\qquad start, \ end, \ pointTime, \ togetherWith, \ instanceOf\}$
$\mathcal{D} = \{ \ 1984,1994,1980,1955\}$
$\rho = \{ \ (timBl, \ instanceOf, \ Human, \ \{\}),$
$\qquad (timBl, \ birthDate, \ 1955, \ \{\}),$
$\qquad (timBl, \ birthPlace, \ London, \ \{\}),$
$\qquad (timBl, \ employer, \ CERN, \ \{ \ start{:}1980, \ end{:}1980 \ \}),$
$\qquad (timBl, \ employer, \ CERN, \ \{ \ start{:}1984, \ end{:}1994 \ \}),$
$\qquad (timBl, \ awarded, \ PA, \ \{pointTime{:} \ 2002, \ togetherWith{:}vintCerf\}),$
$\qquad (London, \ country, \ UK, \ \{\}),$
$\qquad (vintCerf, \ instanceOf, \ Human, \ \{\})$
$\qquad (vintCerf, \ birthPlace, \ NewHaven, \ \{\})$
$\qquad (CERN, \ awarded, \ PA, \ \{ \ pointTime{:} \ 2013 \ \})$
$\qquad (PA, \ country, \ Spain, \ \{ \ \}) \ \}$

Figure 6 presents a possible visualization of a wikibase graph.
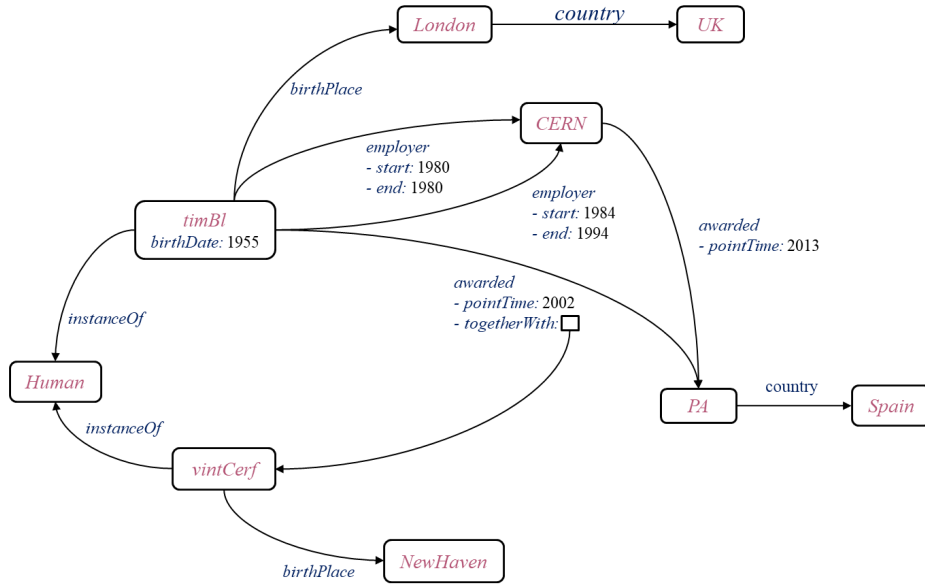

Fig. 6. Visualization of example wikibase graph

The Wikibase data model supports 2 main export formats: JSON and RDF. The JSON one directly follows the structure of the Wikibase data model and is employed by the JSON Dumps while the RDF serialization follows semantic web and linked data principles.

Wikibase JSON serialization The JSON serialization follows the Wikibase data model. It basically consists of an array of entities where each entity is a JSON object that captures all the local information about the entity: the labels, descriptions, aliases, sitelinks and statements that have the entity as subject. Each JSON object is represented in a single line. A remarkable feature of this encoding is that it captures the output neighborhood of every entity in a single line making it amenable to processing models that focus on local neighborhoods because the whole graph can be processed in a single pass.

```
[
 { "type": "item", "id": "Q42", "claims": { "P31": [...
 { "type": "item", "id": "Q80", "claims": { "P108": [...
 { "type": "property", "id": "P108", "claims": { ...
 ...
]
```

Wikibase RDF serialization The RDF serialization[18] of Wikidata was designed with the goal of being able to represent all the structures of the Wikibase data model in RDF, maintaining compatibility with semantic web vocabularies like RDFS and OWL and avoiding the use of blank nodes [19].

Example 5 (RDF serialization of a node). As an example, the information about Tim Berners-Lee (Q80) declaring that he was as employer (P108) of CERN (Q42944) between 1984 and 1994 is represented as [19]:

```
wd:Q80 rdf:type wikibase:Item ;
 wdt:P108 wd:Q42944      ;
 p:P108 :Q80-4fe7940f    .

:Q80-4fe7940f rdf:type wikibase:Statement ;
 wikibase:rank  wikibase:NormalRank ;
 ps:P108        wd:Q42944 ;
 pq:P580        "1984-01-01T00:00:00Z"^^xsd:dateTime ;
 pq:P582        "1994-01-01T00:00:00Z"^^xsd:dateTime .
```

The RDF serialization uses a direct arc to represent the preferred statement represented by prefix alias `wdt:` leaving the rest of the values of a property accessible through the namespaces `p:`, `ps:` and `pq:`.

The reification model employed by Wikidata creates auxiliary nodes that represent each statement. In the previous example, the node `:Q80-4fe7940f` represents the statement which can be qualified with the start and end time.

Apart of the the dumps, RDF serialization is also employed by the Wikidata Query Service [5, 43] and users of Wikidata are required to use and understand the singleton statement approach and namespace conventions employed.

---

[18] https://www.mediawiki.org/wiki/Wikibase/Indexing/RDF_Dump_Format

[19] The full Turtle serialization can be obtained at: https://www.wikidata.org/wiki/Special:EntityData/Q80.ttl

4   Describing Knowledge Graphs

4.1   Describing and validating RDF

At the end of 2013, an *RDF Validation Workshop* [20] was organized by W3C/MIT to discuss use cases and requirements related with the quality of RDF data. One of the conclusions of the workshop was that there was a need for a high-level language that could describe and validate RDF data.

Shape Expressions (ShEx) were proposed as such a language in 2014 [62]. It was designed as a high-level and concise domain-specific language to describe RDF. The syntax of ShEx is inspired by Turtle and SPARQL, while the semantics is inspired by RelaxNG and XML Schema.

In this section we describe a simplified abstract syntax of ShEx following [7][21].

Definition 12 (ShEx schema). *A* ShEx Schema *is defined as a tuple* $\langle \mathcal{L}, \delta \rangle$ *where* $\mathcal{L}$ *set of shape labels, and* $\delta : \mathcal{L} \to \mathcal{S}$ *is a total function from labels to shape expressions.*

The set of shape expressions $se \in \mathcal{S}$ *is defined using the following abstract syntax:*

$$
\begin{array}{lll}
se ::= & cond & \text{\textit{Basic boolean condition on nodes (node constraint)}} \\
\ \ \ \ | & s & \text{\textit{Shape}} \\
\ \ \ \ | & se_1 \ \texttt{AND} \ se_2 & \text{\textit{Conjunction}} \\
\ \ \ \ | & @l & \text{\textit{Shape label reference for }} l \in \mathcal{L} \\
s \ ::= & \texttt{CLOSED} \ \{te\} & \text{\textit{Closed shape}} \\
\ \ \ \ | & \{te\} & \text{\textit{Open shape}} \\
te ::= & te_1 ; te_2 & \text{\textit{Each of }} te_1 \text{\textit{ and }} te_2 \\
\ \ \ \ | & te_1 \mid te_2 & \text{\textit{Some of }} te_1 \text{\textit{ or }} te_2 \\
\ \ \ \ | & te* & \text{\textit{Zero or more te}} \\
\ \ \ \ | & \epsilon & \text{\textit{Empty triple expression}} \\
\ \ \ \ | & \ \_ \xrightarrow{p} @l & \text{\textit{Triple constraint with predicate }} p
\end{array}
$$

Intuitively, shape expressions define conditions about nodes while triple expressions define conditions about the neighborhood of nodes, and shapes qualify those neighborhoods by disallowing triples with other predicates in the case of closed shapes or allowing them in the case of open shapes.

In this paper we omit the negation and disjunction operator to facilitate the presentation of the subsetting semantics. Adding those operators increases the expressiveness of ShEx to validate RDF graphs but we consider that their use to create subsets is not yet clear so we decided to leave them for further research.

The restrictions imposed on shape expressions schemas in [63] also apply here. Namely, in a schema $(\mathcal{L}, \delta, \mathcal{S})$

- The shape label references used by the definition function δ are themselves defined, i.e. if $@l$ appears in some shape definition, then l belongs to $\mathcal{L}$;

– No definition $\delta(l)$ uses a reference $@l$ to itself, neither directly nor transitively, except while traversing a shape. For instance, $\delta(l) = @l$ `AND` se is forbidden, but $\delta(l) = \{\_ \xrightarrow{p} @l\}$ is allowed.

Example 6 (Example of ShEx schema).

A ShEx schema that describes the RDF graph presented in example 1 can be defined as:

$$
\begin{aligned}
\mathcal{L} &= \{\ Person, Place, Country, Organization, Date\} \\
\delta(Person) &= \{\ \_ \xrightarrow{birthDate} @Date;\ \_ \xrightarrow{birthPlace} @Place; \\
&\qquad \_ \xrightarrow{employer} @Organization * \} \\
\delta(Place) &= \{\ \_ \xrightarrow{country} @Country\} \\
\delta(Country) &= \{\ \} \\
\delta(Organization) &= \{\ \} \\
\delta(Date) &=\ \ \text{xsd:Date}
\end{aligned}
$$

ShEx has several concrete syntaxes like a compact syntax (ShExC) and an RDF syntax defined based on JSON-LD (ShExJ) [22].

Example 7 (Example of ShEx in ShExC compact syntax).

The previous ShEx schema can be defined using the compact syntax as:

```
:Person {
  :birthPlace  @:Place  ;
  :birthDate   @:Date  ;
  :employer    @:Organization  ;
}
:Place {
  :country     @:Country
}
:Country      {}
:Organization {}
:Date         {}
```

In general, it is possible to visualize ShEx schemas using UML-like class diagrams. Figure 7 presents a visualization of the previous schema using RDF-Shape [23]

Apart of describing RDF data, Shape Expressions have been designed to enable validation and checking if an RDF node conforms to some shape.

The semantics of Shape Expression validation can be defined with a relation between an RDF node, an RDF graph, a ShEx schema and a shape assignment.

As an example of validation, we have implemented the ShEx-s library which is used by RDFShape [24].

---

[22] See [63] for details.

[23] This visualization can be interactively generated following: https://rdfshape.weso.es/link/16344153229

[24] It is possible to see the results of validating the previous example in RDFShape following this link: https://rdfshape.weso.es/link/16275436158
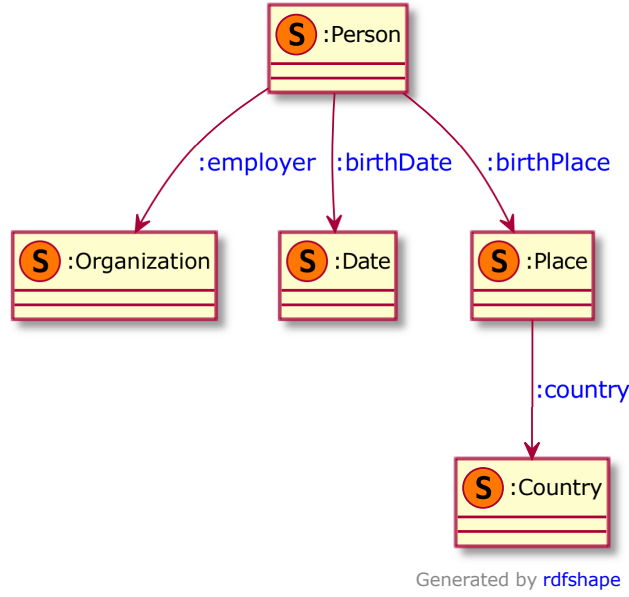
Generated by rdfshape

Fig. 7. ShEx schema visualization as UML-like diagrams

The semantics of ShEx schemas is based on a conformance relation parameterized by a shape assignment: we say that node n in graph $\mathcal{G}$ conforms to shape expression se with shape assignment τ, and we write $\mathcal{G}, n, \tau \vDash se$.

The following rules are defined similar to [6], where it is shown that there exists a unique maximal shape assignment $\tau_{\max}$ that allows to define conformance independently on the shape assignment.

The conformance relation is defined recursively on the structure of se by the set of inference rules presented in table 1 where $preds(te)$ is the set of predicates that appear in a triple expression te and can be defined as:

$$Cond \frac{cond(n) = true}{\mathcal{G}, n, \tau \vDash cond} \qquad AND \frac{\mathcal{G}, n, \tau \vDash se_1 \quad \mathcal{G}, n, \tau \vDash se_2}{\mathcal{G}, n, \tau \vDash se_1 \text{ AND } se_2}$$

$$ClosedShape \frac{neighs(n, \mathcal{G}) = ts \quad \mathcal{G}, ts, \tau \Vdash te}{\mathcal{G}, n, \tau \vDash \text{CLOSED } \{te\}}$$

$$OpenShape \frac{ts = \{\langle x, p, y \rangle \in neighs(n, \mathcal{G}) \mid p \in preds(te)\} \quad \mathcal{G}, ts, \tau \Vdash te}{\mathcal{G}, n, \tau \vDash \{te\}}$$

Table 1. Inference rules for ShEx shape expressions

$$preds(te_1; te_2) \ = preds(te_1) \cup preds(te_2)$$
$$preds(te_1 \mid te_2) = preds(te_1) \cup preds(te_2)$$
$$preds(\_ \xrightarrow{p} te) \ = \{p\}$$
$$preds(te*) \ \quad = preds(te)$$
$$preds(\epsilon) \ \qquad = \emptyset$$

The rules for node constraint (*Cond*) and conjunction are as expected. A node n conforms to an open shape with triple expression te if its neighborhood restricted to the triples with predicates from te conform, meaning that triples whose predicates are not mentioned in te are not constrained by the shape (rule *OpenShape*). Conformance to a closed shape requires to consider the whole neighborhood of the node (rule *ClosedShape*).

Conformance to a triple expression uses a second conformance relation defined on sets on neighborhood triples ts instead of nodes n. The set of neighborhood nodes ts of a graph $\mathcal{G}$ conforms to a triple expression te with shape assignment τ, written as $\mathcal{G}, ts, \tau \Vdash te$, as defined by the inference rules in table 2.

$$EachOf \ \frac{(ts_1, ts_2) \in part(ts) \quad \mathcal{G}, ts_1, \tau \Vdash te_1 \quad \mathcal{G}, ts_2, \tau \Vdash te_2}{\mathcal{G}, ts, \tau \Vdash te_1; te_2}$$

$$OneOf_1 \ \frac{\mathcal{G}, ts, \tau \Vdash te_1}{\mathcal{G}, ts, \tau \Vdash te_1 \mid te_2} \qquad OneOf_2 \ \frac{\mathcal{G}, ts, \tau \Vdash te_2}{\mathcal{G}, ts, \tau \Vdash te_1 \mid te_2}$$

$$TripleConstraint \ \frac{ts = \{\langle x, p, y \rangle\} \quad \mathcal{G}, y, \tau \vDash @l}{\mathcal{G}, ts, \tau \Vdash \_ \xrightarrow{p} @l} \qquad Star_1 \ \frac{}{\mathcal{G}, \emptyset, \tau \Vdash te*}$$

$$Star_2 \ \frac{(ts_1, ts_2) \in part(ts) \quad \mathcal{G}, ts_1, \tau \Vdash te \quad \mathcal{G}, ts_2, \tau \Vdash te*}{\mathcal{G}, ts, \tau \Vdash te*}$$

Table 2. Inference rules for ShEx triple expressions

The semantics of ShEx schema can be defined independently of shape assignments. A shape assignment τ for graph $\mathcal{G}$ and $\mathcal{S}$ is called *valid* if for every node n in $\mathcal{G}$ and every shape expression label l defined in $\mathcal{S}$, if $n@l \in \tau$, then $\mathcal{G}, n, \tau \vDash @l$.

Lemma 1 (Boneva et al [7]). *For every graph $\mathcal{G}$, there exists a unique maximal valid shape shape assignment τ_{max} such that if τ is a valid shape assignment for $\mathcal{G}$ and $\mathcal{S}$, then $\tau \subseteq \tau_{max}$.*

4.2   Describing and validating Property graphs

In this section we define a ShEx extension called PShEx that can be used to describe and validate Property graphs. According to the definition of property

graphs given in section 3.2, nodes and edges can have associated labels as well as a set of property/values. In this way, it is necessary to adapt the definition of ShEx to describe pairs or property/values that we will call qualifiers.

The language PShEx is composed of three main categories: shape expressions (se) that describe the shape of nodes, triple expressions (te) that describe the shape of edge relationships and qualifier expressions (qs) that describe qualifiers sets of property/values associated with node/edge identifiers.

Definition 13 (PShEx schema). *A* PShEx Schema *is a tuple* $\langle \mathcal{L}, \delta \rangle$ *where* $\mathcal{L}$ *set of shape labels, and* $\delta : \mathcal{L} \rightarrow \mathcal{S}$ *is a total function from labels to shape expressions* $se \in \mathcal{S}$ *defined using the abstract syntax:*

$$
\begin{aligned}
se ::= \ & cond_{t_s} && \textit{Basic boolean condition on set of types } t_s \subseteq \mathcal{T} \\
| \ & s && \textit{Shape} \\
| \ & se_1 \ \texttt{AND} \ se_2 && \textit{Conjunction} \\
| \ & @l && \textit{Shape label reference for } l \in \mathcal{L} \\
| \ & qs && \textit{Qualifiers of that node} \\
s ::= \ & \texttt{CLOSED} \ \{te\} && \textit{Closed shape} \\
| \ & \{te\} && \textit{Open shape} \\
te ::= \ & te_1; te_2 && \textit{Each of } te_1 \textit{ and } te_2 \\
| \ & te_1 \mid te_2 && \textit{Some of } te_1 \textit{ or } te_2 \\
| \ & te* && \textit{Zero or more te} \\
| \ & \_ \xrightarrow{p} @l \ qs && \textit{Triple constraint with property type } p \\
& && \textit{whose nodes satisfy the shape l and qualifiers qs} \\
qs ::= \ & \lfloor ps \rfloor && \textit{Open qualifier specifiers ps} \\
| \ & \lceil ps \rceil && \textit{Closed qualifier specifiers ps} \\
ps ::= \ & ps_1 \, , \, ps_2 && \textit{Each of } ps_1 \textit{ and } ps_2 \\
| \ & ps_1 \mid ps_2 && \textit{OneOf of } ps_1 \textit{ or } ps_2 \\
| \ & ps* && \textit{zero of more ps} \\
| \ & p : cond_v && \textit{Property p with value conforming to } cond_v \\
& && cond_{v_s} \textit{ is a boolean condition on sets of values } v_s \subseteq \mathcal{V}
\end{aligned}
$$

We will omit the list of qualifiers when it is empty.

Example 8. As an example, we can define a PShEx schema that describes the property graph from example 3 where $hasType_t$ is a condition that is satisfied when the set of types of a node contains the type t, i.e. $hasType_t(vs) = $ `true` if $t \in vs$ and $String, Date$ are conditions on the values that are satisfied when the values have the corresponding type.

$$\begin{aligned}
\mathcal{L} \quad &= \{\ Person, Place, Country, Org\} \\
\delta(\text{Person}) \ &= \ hasType_{Human} \ \texttt{AND} \ \lfloor label : String, birthDate : Date \rfloor \ \texttt{AND} \ \{ \\
&\qquad \ _{\textvisiblespace} \xrightarrow{birthPlace} @Place; \\
&\qquad \ _{\textvisiblespace} \xrightarrow{employer} @Org \ \lfloor start : Date, end : Date \rfloor * \\
&\ \} \\
\delta(\text{Place}) \ &= \ \lfloor label : String \rfloor \ \texttt{AND} \ \{ \\
&\qquad \ _{\textvisiblespace} \xrightarrow{country} @Country \\
&\ \} \\
\delta(\text{Country}) &= \ hasType_{Country} \ \texttt{AND} \lfloor label : String \rfloor \ \{\} \\
\delta(\text{Org}) \quad &= \ hasType_{Organization} \ \texttt{AND} \lfloor label : String \rfloor \ \{\}
\end{aligned}$$

In order to define the semantic specification of PShEx we will need to define the neighborhood of a node in a property graph.

Definition 14 (Neighborhood of node in property graph). *The neighbors of a node $n \in \mathcal{N}$ in a property graph $\mathcal{G} = \langle \mathcal{N}, \mathcal{E}, \rho, \lambda_n, \lambda_e, \sigma \rangle$ are defined as $neighs(n) = \{(n, p, y, vs) \mid \exists v \in \mathcal{E} \text{ such that } \rho(v) = (n, y) \wedge \lambda_e(v) = p \wedge vs = \{(k, v) \mid \sigma(k, v) = ws \wedge v \in ws\}\}$*

Example 9. The neighbors of node n_1 in property graph 3 are:
$$\begin{aligned}
neighs(n_1) = \{\ &(n_1, birthPlace, n_2, \{\}), \\
&(n_1, employer, n_4, \{(start, 1980), (end, 1980)\}), \\
&(n_1, employer, n_4, \{(start, 1984), (end, 1994)\})\}
\end{aligned}$$

The semantic specification of PShEx can defined in a similar way to the ShEx one. Given a property graph $\mathcal{G}$, and a shape assignment τ, a node identifier $n \in \mathcal{N}$ conforms with a shape expression *se*, which is represented as $\mathcal{G}, n, \tau \vDash se$ and follows the rules presented in 3 where $preds(te)$ is the set of edge labels (or predicates) that appear in a triple expression *te* and can be defined as:

$$Cond_{ts} \frac{\lambda_n(n) = vs \quad cond_{ts}(vs) = true}{\mathcal{G}, n, \tau \vDash cond_{ts}} \qquad AND \frac{\mathcal{G}, n, \tau \vDash se_1 \quad \mathcal{G}, n, \tau \vDash se_2}{\mathcal{G}, n, \tau \vDash se_1 \ \texttt{AND} \ se_2}$$

$$ClosedShape \frac{neighs(n, \mathcal{G}) = ts \quad \mathcal{G}, ts, \tau \Vdash s'}{\mathcal{G}, n, \tau \vDash \texttt{CLOSED} \ \{te\}}$$

$$OpenShape \frac{ts = \{\langle x, p, y \rangle \in neighs(n, \mathcal{G}) \mid p \in preds(te)\} \quad \mathcal{G}, ts, \tau \Vdash te}{\mathcal{G}, n, \tau \vDash \{te\}}$$

Table 3. Rules for PShEx shape expressions

$$\begin{aligned}
preds(te_1; te_2) \ &= preds(te_1) \cup preds(te_2) \\
preds(te_1 \mid te_2) &= preds(te_1) \cup preds(te_2) \\
preds(_{\textvisiblespace} \xrightarrow{p} te) \ &= \{p\} \\
preds(te*) \qquad &= preds(te) \\
preds(\epsilon) \qquad &= \emptyset
\end{aligned}$$

As in the case of ShEx, the previous definition uses a second conformance relation defined on sets of triples ts instead of nodes n. The set of neighborhood nodes ts from a property graph $\mathcal{G}$ conforms to a triple expression te with shape assignment τ, written $\mathcal{G}, ts, \tau \Vdash s$, as defined by the inference rules represented in table 4.

$$EachOf \frac{(ts_1, ts_2) \in part(ts) \quad \mathcal{G}, ts_1, \tau \Vdash te_1 \quad \mathcal{G}, ts_2, \tau \Vdash te_2}{\mathcal{G}, ts, \tau \Vdash te_1; te_2}$$

$$OneOf_1 \frac{\mathcal{G}, ts, \tau \Vdash te_1}{\mathcal{G}, ts, \tau \Vdash te_1 \mid te_2} \qquad OneOf_2 \frac{\mathcal{G}, ts, \tau \Vdash te_2}{\mathcal{G}, ts, \tau \Vdash te_1 \mid te_2}$$

$$TripleConstraint \frac{ts = \{\langle x, p, y, s \rangle\} \quad \mathcal{G}, y, \tau \vDash @l \quad \mathcal{G}, s, \tau \vdash qs}{\mathcal{G}, ts, \tau \Vdash \_ \xrightarrow{p} @l \ qs}$$

$$Star_1 \frac{}{\mathcal{G}, \emptyset, \tau \Vdash te*}$$

$$Star_2 \frac{(ts_1, ts_2) \in part(ts) \quad \mathcal{G}, ts_1, \tau \Vdash te \quad \mathcal{G}, ts_2, \tau \Vdash te*}{\mathcal{G}, ts, \tau \Vdash te*}$$

Table 4. Rules for PShEx triple expressions

In the case of PShEx we declare a new conformance relationship $\mathcal{G}, s, \tau \vdash qs$ between a graph $\mathcal{G}$ a set $s \in P \times V$ of property-value elements, a shape assignment τ and a qualifier specifier qs whose rules are defined in table 5 where $props(ps)$ is the set of properties that appear in a property specifier ps and can be defined as:

$$OpenQs \frac{s' = \{(p, v) \in s \mid p \in props(ps)\} \quad \mathcal{G}, s', \tau \vdash ps}{\mathcal{G}, s, \tau \vdash \lfloor ps \rfloor} \qquad CloseQs \frac{\mathcal{G}, s, \tau \vdash ps}{\mathcal{G}, s, \tau \vdash \lceil ps \rceil}$$

$$EachOfQs \frac{\mathcal{G}, s, \tau \vdash ps_1 \quad \mathcal{G}, s, \tau \vdash ps_2}{\mathcal{G}, s, \tau \vdash ps_1, ps_2}$$

$$OneOfQs_1 \frac{\mathcal{G}, s, \tau \vdash ps_1}{\mathcal{G}, s, \tau \vdash ps_1 \mid ps_2} \qquad OneOfQs_2 \frac{\mathcal{G}, s, \tau \vdash ps_2}{\mathcal{G}, s, \tau \vdash ps_1 \mid ps_2}$$

$$StarQs_1 \frac{}{\mathcal{G}, \emptyset, \tau \vdash ps*} \qquad StarQs_2 \frac{(s_1, s_2) \in part(s) \quad \mathcal{G}, s_1, \tau \vdash ps \quad \mathcal{G}, s_2, \tau \vdash ps*}{\mathcal{G}, s, \tau \vdash ps*}$$

$$PropertyQs \frac{s = \{(p, w)\} \quad conv_v(w) = \texttt{true}}{\mathcal{G}, s, \tau \vdash p : cond_v}$$

Table 5. Rules for PShEx qualifiers

$$props(ps_1\,,\,ps_2)\ = props(ps_1) \cup props(ps_2)$$
$$props(ps_1 \mid ps_2)\ = props(ps_1) \cup props(ps_2)$$
$$props(ps*)\qquad = preds(ps)$$
$$props(p : cond_v) = \{p\}$$

As in the case of ShEx, the semantics of ShEx schemas can be defined independently on shape assignments. A shape assignment τ for graph $\mathcal{G}$ and $\mathcal{S}$ is called *valid* if for every node n in $\mathcal{G}$ and every shape expression label l defined in $\mathcal{S}$, if $n@l \in \tau$, then $\mathcal{G}, n, \tau \vDash @l$.

4.3   Describing and validating Wikibase graphs

Wikidata adopted ShEx in 2019 as the language to define entity schemas which can be used to validate entities. Nevertheless, they describe the RDF serialization of Wikibase entities instead of the Wikibase datamodel. This requires users to be aware of how qualifiers and references are serialized in Wikibase which can lead to duplicated properties. Another problem of ShEx schemas is that they cannot be used to directly describe the contents of Wikidata dumps in JSON which are closer to the Wikibase data model.

To that end, we designed an extension of ShEx called WShEx that can describe the Wikibase data model and so, be used to validate Wikibase dumps in JSON without requiring them to be serialized in RDF. Figure 8 represents the relationship between ShEx and WShEx.
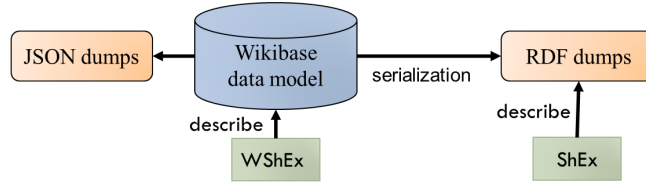

Fig. 8. Relationship between ShEx, WShEx and Wikibase data model

WShEx is presented as an extension of the ShEx language defined in section 4.1 adapted to the wikibase graphs definitions 11.

Definition 15 (WShEx schema). *A WShEx Schema is defined as a tuple $\langle \mathcal{L}, \delta \rangle$ where $\mathcal{L}$ set of shape labels, and $\delta : \mathcal{L} \to \mathcal{S}$ is a total function from labels to w-shape expressions.*

The set of shape expressions $se \in \mathcal{S}$ is defined using the abstract syntax presented in table 6. Notice that it is an extension of the abstract syntax for ShEx modifying the rule for triple constraint adding a new element for qualifier specifiers and adding the corresponding rules for qualifier specifiers.

Example 10 (Example of WShEx schema).

se ::= cond	Basic boolean condition on nodes (node constraint)
| s	Shape
| se_1 AND se_2	Conjunction
| @l	Shape label reference for $l \in \mathcal{L}$
s ::= CLOSED s'	Closed shape
| s'	Open shape
s' ::= { te }	Shape definition
te ::= $te_1; te_2$	Each of te_1 and te_2
| $te_1 \| te_2$	Some of te_1 or te_2
| $te*$	Zero or more te
| $\_ \xrightarrow{p} @l$ qs	Triple constraint with predicate p
	value conforming to l and qualifier specifier qs
| ϵ	Empty triple expression
qs ::= $\lfloor ps \rfloor$	Open property specifier
| $\lceil ps \rceil$	Closed property specifier
ps ::= ps , ps	*EachOf* property specifiers
| $ps \| ps$	*OneOf* property specifiers
| ps*	zero of more property specifiers
| ϵ	Empty property specifier
| p:@l	Property p with value conforming to shape l

Table 6. Abstract syntax of WShEx

A ShEx schema that describes the Wikibase graph presented in example 4 can be defined as:

$$\mathcal{L} = \{\ Person, Place, Country, Organization, Date,\ Award\}$$
$$\delta(Person) = \{\ \_ \xrightarrow{birthDate} @Date;\ \_ \xrightarrow{birthPlace} @Place;$$
$$\_ \xrightarrow{employer} @Organization \lfloor start : @Date\ , end : @Date \rfloor*$$
$$\_ \xrightarrow{awarded} @Award \lfloor pointTime : @Date\ , togetherWith : @Person \rfloor*$$
$$\}$$
$$\delta(Place) = \{\ \_ \xrightarrow{country} @Country\}$$
$$\delta(Country) = \{\ \}$$
$$\delta(Award) = \{\ \_ \xrightarrow{country} @Country\}$$
$$\delta(Organization) = \{\ \}$$
$$\delta(\text{Date}) = \ \in xsd : date$$

It is possible to define a compact syntax for WShEx in a similar way to ShExC adding the symbols {{...}} to declare open qualifier specifiers and [[...]] for closed ones.

Example 11 (Example of WShEx schema using the compact Syntax).

```
:Researcher {
 birthPlace      @<Place>            ;
 birthDate       @<Time>             ;
 employer        @<Organization> *
    {{ :start    @:Date ,
       :end      @:Date
    }} ;
```

```
 awarded         @<Award> *
    {{ :pointTime     @:Date,
       :togetherWith @:Person
    }}
}
:Place           { country @<Country> }
:Organization    {}
:Award           { country @<Country> }
:Country         {}
:Date            xsd:date
```

The semantics of WShEx is similar to the semantics defined for ShEx and PShEx. We define a conformance relation parameterized by a shape assignment $\mathcal{G}, n, \tau \vDash se$ with the meaning that node n in graph $\mathcal{G}$ conforms to shape expression se with shape assignment τ according to the rules 7.

$$Cond\frac{cond(n) = true}{\mathcal{G}, n, \tau \vDash cond} \qquad AND\frac{\mathcal{G}, n, \tau \vDash se_1 \quad \mathcal{G}, n, \tau \vDash se_2}{\mathcal{G}, n, \tau \vDash se_1 \text{ AND } se_2}$$

$$ClosedShape\frac{neighs(n, \mathcal{G}) = ts \quad \mathcal{G}, ts, \tau \Vdash s'}{\mathcal{G}, n, \tau \vDash \texttt{CLOSED } s'}$$

$$OpenShape\frac{ts = \{\langle x, p, y \rangle \in neighs(n, \mathcal{G}) \mid p \in preds(te)\} \quad \mathcal{G}, ts, \tau \Vdash s'}{\mathcal{G}, n, \tau \vDash s'}$$

Table 7. Inference rules for WShEx shape expressions

We also define a conformance relation $\mathcal{G}, ts, \tau \Vdash te$ which declares that the triples ts in graph $\mathcal{G}$ conform to the triple expression te with the shape assignment τ using the rules 8 which takes into account qualifier specifiers.

Finally, the conformance relationship $\mathcal{G}, s, \tau \vdash qs$ between a graph $\mathcal{G}$ a set $s \in P \times V$ of property-value elements, a shape assignment τ and a qualifier specifier qs is defined with the rules 9.

5   Knowledge Graphs Subsets

In this section we review several approaches to create knowledge graphs subsets. Although we will focus on Wikibase graphs subsets, the approaches described can also be applied to RDF-based graphs and property graphs.

5.1   Wikibase Subsets: Formal definition

The following definition of Wikibase subset is based on the wikibase graphs definition given at section 3.3.

$$EachOf\ \frac{(ts_1, ts_2) \in part(ts) \quad \mathcal{G}, ts_1, \tau \Vdash te_1 \quad \mathcal{G}, ts_2, \tau \Vdash te_2}{\mathcal{G}, ts, \tau \Vdash te_1; te_2}$$

$$OneOf_1\ \frac{\mathcal{G}, ts, \tau \Vdash te_1}{\mathcal{G}, ts, \tau \Vdash te_1 \mid te_2} \qquad OneOf_2\ \frac{\mathcal{G}, ts, \tau \Vdash te_2}{\mathcal{G}, ts, \tau \Vdash te_1 \mid te_2}$$

$$Star_1\ \frac{}{\mathcal{G}, \emptyset, \tau \Vdash te*}$$

$$Star_2\ \frac{(ts_1, ts_2) \in part(ts) \quad \mathcal{G}, ts_1, \tau \Vdash te \quad \mathcal{G}, ts_2, \tau \Vdash te*}{\mathcal{G}, ts, \tau \Vdash te*}$$

$$TripleConstraint\ \frac{ts = \{\langle x, p, y, s \rangle\} \quad \mathcal{G}, y, \tau \vDash @l \quad \mathcal{G}, s, \tau \vdash qs}{\mathcal{G}, ts, \tau \Vdash \_ \xrightarrow{p} @l \ qs}$$

Table 8. Inference rules for WShEx triple expressions

$$OpenQs\ \frac{s' = \{(p, v) \in s \mid p \in preds(ps)\} \quad \mathcal{G}, s', \tau \vdash ps}{\mathcal{G}, s, \tau \vdash \lfloor ps \rfloor} \qquad CloseQs\ \frac{\mathcal{G}, s, \tau \vdash ps}{\mathcal{G}, s, \tau \vdash \lceil ps \rceil}$$

$$EachOfQs\ \frac{\mathcal{G}, s, \tau \vdash ps_1 \quad \mathcal{G}, s, \tau \vdash ps_2}{\mathcal{G}, s, \tau \vdash ps_1 , ps_2}$$

$$OneOfQs_1\ \frac{\mathcal{G}, s, \tau \vdash ps_1}{\mathcal{G}, s, \tau \vdash ps_1 \mid ps_2} \qquad OneOfQs_2\ \frac{\mathcal{G}, s, \tau \vdash ps_2}{\mathcal{G}, s, \tau \vdash ps_1 \mid ps_2}$$

$$StarQs_1\ \frac{}{\mathcal{G}, \emptyset, \tau \vdash ps*} \qquad StarQs_2\ \frac{(s_1, s_2) \in part(s) \quad \mathcal{G}, s_1, \tau \vdash ps \quad \mathcal{G}, s_2, \tau \vdash ps*}{\mathcal{G}, s, \tau \vdash ps*}$$

$$EmptyQs\ \frac{}{\mathcal{G}, \emptyset, \tau \vdash \epsilon} \qquad PropertyQs\ \frac{s = \{(p, v)\} \quad \mathcal{G}, v, \tau \vDash @l}{\mathcal{G}, s, \tau \vdash p : @l}$$

Table 9. Inference rules for WShEx qualifier expressions

Definition 16 (Wikibase subset). *Given a wikibase graph $\mathcal{G} = \langle \mathcal{Q}, \mathcal{P}, \mathcal{D}, \rho \rangle$, a wikibase subgraph is defined as $\mathcal{G}' = \langle \mathcal{Q}', \mathcal{P}', \mathcal{D}', \rho' \rangle$ such that: $\mathcal{Q}' \subseteq \mathcal{Q}$, $\mathcal{P}' \subseteq \mathcal{P}$, $\mathcal{D}' \subseteq \mathcal{D}$ and $\rho' \subseteq \rho$*

Example 12 (Example of wikibase subgraph). Given the wikibase graph from example 4 $\mathcal{G}' = \langle \mathcal{Q}', \mathcal{P}', \mathcal{D}', \rho' \rangle$ where

$$\begin{aligned}
\mathcal{Q}' &= \{timBl, London, CERN\} \\
\mathcal{P}' &= \{birthPlace, employer, start\} \\
\mathcal{D} &= \{1980, 1984\} \\
\rho &= \{(timBl, birthPlace, London, \{\}), \\
&\quad (timBl, employer, CERN, \{start : 1980\}), \\
&\quad (timBl, employer, CERN, \{start : 1984\})\}
\end{aligned}$$

is a wikibase subgraph of $\mathcal{G}$.

5.2   Entity-generated subsets

Wikibase subgraphs can be generated from a set of entities (items or properties), where we collect the subgraph associated with those entities.

Definition 17 (Item-generated subgraph). *Given a wikibase graph* $\mathcal{G} = \langle \mathcal{Q}, \mathcal{P}, \mathcal{D}, \rho \rangle$ *and a subset of items* $\mathcal{Q}_s \subset \mathcal{Q}$ *generates an* item-generated sub-graph $\langle \mathcal{Q}', \mathcal{P}', \mathcal{D}', \rho' \rangle$ *such that:*

$$
\begin{aligned}
\mathcal{Q}' &= \{q \in \mathcal{Q} \mid (q, \_, \_, \_) \vee (\_, \_, q, \_) \in \rho'\} \\
&\cup \{q \in \mathcal{Q} \mid (\_, \_, \_, q_s) \in \rho' \wedge (\_, q) \in q_s\} \\
\mathcal{P}' &= \{p \in \mathcal{P} \mid (\_, p, \_, \_) \in \rho'\} \\
&\cup \{p \in \mathcal{P} \mid (\_, \_, \_, q_s) \in \rho' \wedge (p, \_) \in q_s\} \\
\mathcal{D}' &= \{d \in \mathcal{D} \mid (\_, \_, d, \_) \in \rho'\} \\
&\cup \{d \in \mathcal{D} \mid (\_, \_, \_, q_s) \in \rho' \wedge (\_, d) \in q_s\} \\
\rho' &= \{(q, \_, \_, \_) \in \rho \mid q \in \mathcal{Q}_s\} \\
&\cup \{(\_, \_, q, \_) \in \rho \mid q \in \mathcal{Q}_s\} \\
&\cup \{(\_, \_, \_, q_s) \in \rho \wedge \exists q \in \mathcal{Q}_s \mid (\_, q) \in q_s\}
\end{aligned}
$$

Notice that the item-generated subgraph usually contains more items than the items provided by $\mathcal{Q}_s$.

Example 13 (Example of item-generated subgraph).

Given the wikibase graph from example 4 and $\mathcal{Q}_s = \{timBl\}$ the item generated subgraph is:

$\mathcal{Q}' = \{timBl, CERN, vintCerf, PA\}$
$\mathcal{P}' = \{birthDate, birthPlace, employer, awarded,$
$\quad start, end, togetherWith\}$
$\mathcal{D}' = \{1984, 1994, 1980, 1955\}$
$\rho' = \{(timBl, birthDate, 1955, \{\}),$
$\quad (timBl, birthPlace, London, \{\}),$
$\quad (timBl, employer, CERN, \{start : 1980, end : 1980\}),$
$\quad (timBl, employer, CERN, \{start : 1984, end : 1994\}),$
$\quad (timBl, awarded, PA, \{togetherWith : vintCerf\})\}$

Definition 18 (Property-generated subgraph). *Given a wikibase graph* $\mathcal{G} = \langle \mathcal{Q}, \mathcal{P}, \mathcal{D}, \rho \rangle$, *with the set of entities* $\mathcal{E} = \mathcal{Q} \cup \mathcal{P}$ *a subset of properties* $\mathcal{P}_s \subset \mathcal{P}$

generates a property generated subgraph $\langle Q', P', D', \rho' \rangle$ *such that:*

$$Q' = \{q \in Q \mid \exists p \in P_s \mid (q, p, \_, \_) \in \rho\}$$
$$\cup \{q \in Q \mid \exists p \in P_s \mid (\_, p, q, \_) \in \rho\}$$
$$\cup \{q \in Q \mid (\_, \_, \_, q_s) \in \rho \wedge \exists p \in P_s \mid (p, \_) \in q_s\}$$
$$P' = \{p \in P_s \mid (\_, p, \_, \_) \in \rho\}$$
$$\cup \{p \in P_s \mid \exists q_s \mid (\_, \_, \_, qs) \in \rho \wedge (p, \_) \in q_s\}$$
$$D' = \{d \in D \mid \exists p \in P_s \mid (\_, p, d, \_) \in \rho\}$$
$$\cup \{d \in D \mid (\_, \_, \_, q_s) \in \rho \wedge \exists p \in P_s \mid (p, d) \in q_s\}$$
$$\rho' = \{(\_, p, \_, \_) \in \rho \mid p \in P_s\}$$
$$\cup \{(\_, \_, \_, q_s) \in \rho \mid \exists p \in P_s \mid (p, \_) \in q_s\}$$

The property generated subgraph usually contains more properties than the properties provided by P_s.

Example 14 (Example of property-generated subgraph). Given the wikibase graph from example 4 and $P_s = \{birthDate, togetherWith\}$ the property generated subgraph is:

$$Q' = \{timBl, vintCerf, PA\}$$
$$P' = \{birthDate, awarded, togetherWith\}$$
$$D' = \{1955\}$$
$$\rho' = \{(timBl, birthDate, 1955, \{\}),$$
$$(timBl, awarded, PA, \{togetherWith : vintCerf\})\}$$

Notice that it is possible to define a *Datatype-generated subgraph* in a similar way than the previous definitions.

Definition 19 (Entity-generated subgraph). *Given a subset of entities $E_s \subset Q \cup P$, the entity-generated subgraph is defined as the union of the item-generated subgraph with all the items in E_s and the property-generated subgraph with all the properties in E_s.*

5.3   Simple Matching-generated subsets

Definition 20 (Matching expression). *Given a wikibase graph $G = \langle Q, P, D, \rho \rangle$ where $E = Q \cup P$ and $V = E \cup D$, a matching expression M_s is a set of matchers where each matcher m follows the grammar:*

$$
\begin{aligned}
m ::= \ & subject(e) && Subject\ e \in \mathcal{E}\\
| \ & property(p) && Property\ p \in \mathcal{P}\\
| \ & value(v) && Value\ v \in \mathcal{V}\\
| \ & qualifier(p,v) && Qualifier\ with\ property\ p \in \mathcal{P}\ and\ value\ v \in \mathcal{V}\\
| \ & qualifiedProp(p) && Qualifier\ with\ property\ p \in \mathcal{P}\\
| \ & qualifiedValue(v) && Qualifier\ with\ value\ v \in \mathcal{V}
\end{aligned}
$$

Example 15 (Example of a matching expression). An example of a matching expression is $M_s = \{\text{property(country)}, \text{qualifiedProp(togetherWith)}\}$

Definition 21 (Matching-generated subgraph). *Given a matching expression M_s over a wikibase graph $\mathcal{G} = \langle \mathcal{Q}, \mathcal{P}, \mathcal{D}, \rho \rangle$ we can define the matching-generated subgraph as a wikibase graph $\mathcal{G}' = \langle \mathcal{Q}'\mathcal{P}'\mathcal{D}'\rho' \rangle$ such that:*

$$
\begin{aligned}
\mathcal{Q}' = \ & \{q \in \mathcal{Q} \mid (q, \_, \_, \_) \in \rho' \cup \{q \in \mathcal{Q} \mid (\_, \_, q, \_) \in \rho'\}\\
& \cup \{q \in \mathcal{Q} \mid (\_, \_, \_, q_s) \in \rho' \wedge (\_, q) \in q_s\}\\
\mathcal{P}' = \ & \{p \in \mathcal{P} \mid (\_, p, \_, \_) \in \rho' \cup \{p \in \mathcal{P} \mid (\_, \_, \_, q_s) \in \rho' \wedge (p, \_) \in q_s\}\\
\mathcal{D}' = \ & \{d \in \mathcal{D} \mid (\_, \_, d, \_) \in \rho'\} \cup \{d \in \mathcal{D} \mid (\_, \_, \_, q_s) \in \rho' \wedge (\_, d) \in q_s\}\\
\rho' = \ & \{(q, \_, \_, \_) \in \rho \mid subject(q) \in M_s\}\\
& \cup \{(\_, p, \_, \_) \in \rho \mid property(p) \in M_s\}\\
& \cup \{(\_, \_, v, \_) \in \rho \mid value(v) \in M_s\}\\
& \cup \{(\_, \_, \_, q_s) \in \rho \mid qualifier(p,v) \in M_s \wedge \exists (p,v) \in q_s\} \qquad \cup \{(\_, \_, \_, q_s) \in \rho \mid qualifiedProp(p) \in M_s
\end{aligned}
$$

Example 16 (Example of matching-generated subgraph). Given the wikibase graph $\mathcal{G}$ of example 4 and the matching-expression M_s in example **??**, the matching-generated subgraph of $\mathcal{G}$ from M_s is the wikibase graph $\mathcal{G}' = \langle \mathcal{Q}', \mathcal{P}', \mathcal{D}', \rho' \rangle$ such that:

$$
\begin{aligned}
\mathcal{Q}' = \ & \{PA, Spain, London, UK, timBl, vintCerf\}\\
\mathcal{P}' = \ & \{country, awarded, togetherWith\}\\
\mathcal{D}' = \ & \{\}\\
\rho' = \ & \{(timBl, awarded, PA, \{togetherWith : vintCerf\}),\\
& (vintCerf, awarded, PA, \{togetherWith : timBl\})\\
& (PA, country, Spain, \{\})\\
& (London, country, UK, \{\})\}
\end{aligned}
$$

The matching approach is followed by WDumper [25] and WDSub[26].

WDumper defines the expected patterns using a JSON configuration file that describes them or filling a web form which internally generates the JSON file.

---

[25] https://github.com/bennofs/wdumper
[26] https://github.com/weso/wdsub

In the case of WDSub, the input format is a WShEx file with a set of shapes and the system processes a Wikidata dump trying to match each entity with any of the Shapes defined in the WShEx file. The algorithm employed in WDSub to generate a matching expression from a Shape Expression is the following:

5.4   ShEx-based Matching generated subsets

ShEx-based matching consists on taking as input a WShEx schema $\mathcal{S}$ and include in the generated subset the nodes whose neighborhood matches any of the shapes from $\mathcal{S}$ after replacing any shape references by a condition that always returns true. The goal of this approach is to use ShEx as a basic description language of the topology of nodes ignoring shape references so the algorithm can be used to check dumps that contain include the information about a node and its neighborhood in a single line. In this way, the subset generator only needs to traverse the dump sequentially one time.

Example 17 (ShEx-based matching). Giveb the following WShEx Schema:

$$\mathcal{L} = \{Researcher, Place, Country, Date, Human\}$$

$$\delta(Researcher) = \{ \_ \xrightarrow{instanceOf} @\text{Human};$$
$$\_ \xrightarrow{birthDate} @\text{Date}?;$$
$$\_ \xrightarrow{birthPlace} @\text{Place}$$
$$\}$$

$$\delta(Place) = \{ \_ \xrightarrow{country} @\text{Country}\}$$
$$\delta(Date) = \in xsd:date$$
$$\delta(Human) = \in \{Human\}$$

The result of ShEx-based matching on example 4 is:

$$\rho = \{ \ (timBl, instanceOf, Human, \{\}),$$
$$(timBl, birthDate, 1955, \{\}),$$
$$(timBl, birthPlace, London, \{\}),$$
$$(London, country, UK, \{\}),$$
$$(vintCerf, instanceOf, Human, \{\})$$
$$(vintCerf, birthPlace, NewHaven, \{\})$$
$$\}$$

Notice that *vintCerf* is included although the node doesn't conform to the shape person because it has a *birthPlace* declaration whose value is *NewHaven* but there is no *country* property for *NewHaven*.

In the previous example, the ShEx-based matching consisted on validating each node with any of the following shapes:

$$\delta(Researcher) = \{ \_ \xrightarrow{instanceOf} \texttt{true};$$
$$\_ \xrightarrow{birthDate} \texttt{true}?;$$
$$\_ \xrightarrow{birthPlace} \texttt{true}$$
$$\}$$

$$\delta(Place) = \{ \_ \xrightarrow{country} \texttt{true}\}$$

Notice that if the original ShEx schema had included the following shape:

$$\delta(Country) = \{\}$$

Then, every node would be included in the generated subset because every node would match the *Country* shape.

ShEx-based matching generation has been implemented in WDSub[27].

5.5   ShEx + Slurp generated subsets

The concept of *slurp* was introduced in the shex.js[28] implementation as a mechanism to collect the nodes and triples visited during validation.

In this way, if we collect that data, the result will be a subset of the graph which contains the portion of the graph that relates to a given ShEx schema. Although the slurp option was not formally defined, we can define it modifying the semantics of ShEx adding a new parameter to the conformance relationship.

We define a conformance relation parameterized by a shape assignment $\mathcal{G}, n, \tau \vDash se \rightsquigarrow \mathcal{G}'$ with the meaning that node n in graph $\mathcal{G}$ conforms to shape expression se with shape assignment τ and generates a slurp graph $\mathcal{G}'$. The conformance relation follows the rules 10.

$$Cond \frac{cond(n) = true}{\mathcal{G}, n, \tau \vDash cond \rightsquigarrow \langle \{n\}, \{\} \rangle} \qquad AND \frac{\mathcal{G}, n, \tau \vDash se_1 \rightsquigarrow \mathcal{G}_1 \qquad \mathcal{G}, n, \tau \vDash se_2 \rightsquigarrow \mathcal{G}_2}{\mathcal{G}, n, \tau \vDash se_1 \ \mathtt{AND} \ se_2 \rightsquigarrow \mathcal{G}_1 \cup \mathcal{G}_2}$$

$$ClosedShape \frac{neighs(n, \mathcal{G}) = ts \qquad \mathcal{G}, ts, \tau \Vdash s' \rightsquigarrow \mathcal{G}'}{\mathcal{G}, n, \tau \vDash \mathtt{CLOSED} \ s' \rightsquigarrow \mathcal{G}'}$$

$$OpenShape \frac{ts = \{ \langle x, p, y \rangle \in neighs(n, \mathcal{G}) \mid p \in preds(te) \} \qquad \mathcal{G}, ts, \tau \Vdash s' \rightsquigarrow \mathcal{G}'}{\mathcal{G}, n, \tau \vDash s' \rightsquigarrow \mathcal{G}'}$$

Table 10. Inference rules for WShEx+slurp shape expressions

We also define a conformance relation $\mathcal{G}, ts, \tau \Vdash te \rightsquigarrow \mathcal{G}'$ which declares that the triples ts in graph $\mathcal{G}$ conform to the triple expression te with the shape assignment τ generating a slurp $\mathcal{G}'$. The relation is defined using the rules 11.

The conformance relationship $\mathcal{G}, s, \tau \vdash qs \rightsquigarrow (qs', \mathcal{G}')$ between a graph $\mathcal{G}$ a set $s \in P \times V$ of property-value elements, a shape assignment τ and a qualifier specifier qs generates a slurp that consists of a pair $(qs', \mathcal{G}')$ where qs' is a set of qualifiers slurped and $\mathcal{G}'$ is the graph slurped. It is defined according to the rules 12.

Example 18 (ShEx+Slurp example).
   Given the following WShEx Schema:

---

[27] https://github.com/weso/wdsub
[28] https://github.com/shexjs/shex.js

$$EachOf \frac{(ts_1, ts_2) \in part(ts) \quad \mathcal{G}, ts_1, \tau \Vdash te_1 \rightsquigarrow \mathcal{G}_1 \quad \mathcal{G}, ts_2, \tau \Vdash te_2 \rightsquigarrow \mathcal{G}_2}{\mathcal{G}, ts, \tau \Vdash te_1; te_2 \rightsquigarrow \mathcal{G}_1 \cup \mathcal{G}_2}$$

$$OneOf_1 \frac{\mathcal{G}, ts, \tau \Vdash te_1 \rightsquigarrow \mathcal{G}_1}{\mathcal{G}, ts, \tau \Vdash te_1 \mid te_2 \rightsquigarrow \mathcal{G}_1} \qquad OneOf_2 \frac{\mathcal{G}, ts, \tau \Vdash te_2 \rightsquigarrow \mathcal{G}_2}{\mathcal{G}, ts, \tau \Vdash te_1 \mid te_2 \rightsquigarrow \mathcal{G}_2}$$

$$Star_1 \frac{}{\mathcal{G}, \emptyset, \tau \Vdash te* \rightsquigarrow \emptyset}$$

$$Star_2 \frac{(ts_1, ts_2) \in part(ts) \quad \mathcal{G}, ts_1, \tau \Vdash te \rightsquigarrow \mathcal{G}_1 \quad \mathcal{G}, ts_2, \tau \Vdash te* \rightsquigarrow \mathcal{G}_2}{\mathcal{G}, ts, \tau \Vdash te* \rightsquigarrow \mathcal{G}_1 \cup \mathcal{G}_2}$$

$$TripleConstraint \frac{ts = \{\langle x, p, y, s \rangle\} \quad \mathcal{G}, y, \tau \vDash @l \rightsquigarrow \langle \mathcal{V}, \mathcal{E} \rangle \quad \mathcal{G}, s, \tau \vdash qs \rightsquigarrow (qs', \mathcal{G}_{qs})}{\mathcal{G}, ts, \tau \Vdash \_ \xrightarrow{p} @l \ qs \rightsquigarrow \langle \mathcal{V} \cup \{x\} \cup \{y\}, \mathcal{E} \cup (x, p, y, qs') \rangle \cup \mathcal{G}_{qs}}$$

Table 11. Inference rules for WShEx+slurp triple expressions

$$\mathcal{L} \quad = \{Researcher, Place, Country, Date, Human\}$$
$$\delta(Researcher) = \{ \_ \xrightarrow{instanceOf} @Human;$$
$$\_ \xrightarrow{birthDate} @Date;$$
$$\_ \xrightarrow{birthPlace} @Place$$
$$\}$$
$$\delta(Place) \quad = \{ \_ \xrightarrow{country} @Country\}$$
$$\delta(Country) = \{\}$$
$$\delta(Date) \quad = \in xsd : date$$
$$\delta(Human) \quad = \in \{Human\}$$

The result of running the ShEx+Slurp on example 4 is:
$$\rho = \{ \ (timBl, instanceOf, Human, \{\}),$$
$$(timBl, birthDate, 1955, \{\}),$$
$$(timBl, birthPlace, London, \{\}),$$
$$(London, country, UK, \{\}),$$

The main difference between this approach and the previous one is that it retrieves the valid subset according to the ShEx schema. In this case, the node *vintCerf* is not generated because the value of the property *birthPlace* is *NewHaven* and it has no *country* declaration, so *NewHaven* doesn't conform to the *Place* shape and subsequently, *vintCerf* doesn't conform to the *Person* shape as they are declared in that Schema.

Although the ShEx+Slurp approach has not yet been implemented for WShEx, is has already been implemented for ShEx in shex.js and in PyShEx [29].

One problem of this approach is that it is difficult to scale as it needs to traverse the graph while validating and collecting the slurped graph. The complexity also increases if the implementation wants to adjust the collected triples when checking the different partitions of a node neighborhood. If one of the partitions fails, following the definition it would need to discard the corresponding portion of the graph, which would make the whole process more complex.

---

[29] https://github.com/hsolbrig/PyShEx

$$OpenQs \dfrac{s' = \{(p,v) \in s | p \in preds(ps)\} \quad \mathcal{G}, s', \tau \vdash ps \rightsquigarrow (qs, \mathcal{G}')}{\mathcal{G}, s, \tau \vdash \lfloor ps \rfloor \rightsquigarrow (qs, \mathcal{G}')}$$

$$CloseQs \dfrac{\mathcal{G}, s, \tau \vdash ps \rightsquigarrow (qs, \mathcal{G}')}{\mathcal{G}, s, \tau \vdash \lceil ps \rceil \rightsquigarrow (qs, \mathcal{G}')}$$

$$EachOfQs \dfrac{\mathcal{G}, s, \tau \vdash ps_1 \rightsquigarrow (qs_1, \mathcal{G}_1) \quad \mathcal{G}, s, \tau \vdash ps_2 \rightsquigarrow (qs_2, \mathcal{G}_2)}{\mathcal{G}, s, \tau \vdash ps_1 , ps_2 \rightsquigarrow (qs_1 \cup_2, \mathcal{G}_1 \cup \mathcal{G}_2)}$$

$$OneOfQs_1 \dfrac{\mathcal{G}, s, \tau \vdash ps_1 \rightsquigarrow (qs_1, \mathcal{G}_1)}{\mathcal{G}, s, \tau \vdash ps_1 \mid ps_2 \rightsquigarrow (qs_1, \mathcal{G}_1)} \qquad OneOfQs_2 \dfrac{\mathcal{G}, s, \tau \vdash ps_2 \rightsquigarrow (qs_2, \mathcal{G}_2)}{\mathcal{G}, s, \tau \vdash ps_1 \mid ps_2 \rightsquigarrow (qs_2, \mathcal{G}_2)}$$

$$StarQs_1 \dfrac{}{\mathcal{G}, \emptyset, \tau \vdash ps* \rightsquigarrow (\{\}, \emptyset)}$$

$$StarQs_2 \dfrac{(s_1, s_2) \in part(s) \quad \mathcal{G}, s_1, \tau \vdash ps \rightsquigarrow (qs_1, \mathcal{G}_1) \quad \mathcal{G}, s_2, \tau \vdash ps* \rightsquigarrow qs_2, \mathcal{G}_2}{\mathcal{G}, s, \tau \vdash ps* \rightsquigarrow (qs_1 \cup qs_2, \mathcal{G}_1 \cup \mathcal{G}_2)}$$

$$EmptyQs \dfrac{}{\mathcal{G}, \emptyset, \tau \vdash \epsilon \rightsquigarrow (\{\}, \emptyset)} \qquad PropertyQs \dfrac{s = \{(p,v)\} \quad \mathcal{G}, v, \tau \vDash @l \rightsquigarrow \mathcal{G}'}{\mathcal{G}, s, \tau \vdash p : @l \rightsquigarrow (\{p : v\}, \mathcal{G}')}$$

Table 12. Inference rules for WShEx+slurp qualifiers

In practice, implementations just collect the visited nodes and triples without discarding the ones that shouldn't be part of the result.

5.6   ShEx + Pregel generated subsets

Pregel [42] has been proposed as an scalable computational model created by Google to handle large graphs. It is based on Bulk Synchronous Parallel (BSP) model which simplifies parallel programming having different computation and communication phases. Pregel is an iterative algorithm where each phase is called a superstep. Following the lemma *think like a vertex*, it is a vertex-centric abstraction where at each superstep, a vertex executes a user defined function (called vertex program) which can update its status and later sends messages to neighbors along graph edges. Supersteps end with a synchronization barrier that guarantees that messages sent at one superstep are received at the beginning of the next superstep. Vertices may change status between active and inactive and the algorithm terminates when all vertices are inactive and no more messages are sent.

GraphX was proposed in 2014 as a graph processing framework embedded in Apache Spark. Its API includes a variant of Pregel which is used to implement several graph algorithms like PageRank, connected components, triangle counting, etc.

GraphX defines an API for graphs based on RDDs (resilient distributed datasets). An $RDD[\mathcal{V}]$ is an abstraction of a collection of values of type $\mathcal{V}$ which are immutable and can be partitioned to run data-parallel operations like *map* and *reduce*.

A graph $Graph[\mathcal{V},\mathcal{E}]$ represents and abstraction of vertices with values of type $\mathcal{V}$ and edges of type $\mathcal{E}$ where internally the vertices are represented as $RDD[(Id,\mathcal{V})]$, i.e. a collection of a tuple with an Id (a *Long* value) and a $\mathcal{V}$, and edges are represented as $RDD[(Id,Id,\mathcal{E})]$, i.e. a triple where the first and second components are the Id of the source and destiny respectively, and the third component is the edge property $p \in \mathcal{E}$. A graph $Graph[\mathcal{V},\mathcal{E}]$ also provides what is called a *triplets* view which represents edges as collections of triplets of the form $RDD[(\mathcal{V},\mathcal{E},\mathcal{V})]$. A triplet t will be denoted by the type `Triplet` and provides access to the source vertex (using `t.srcAttr`), the destiny (`t.dstAttr`) and the edge property (`t.attr`).

GraphX provides several built-in operators for graphs[30]. We will use the following in the rest of the paper:

- `mapVertices(g: Graph`$[\mathcal{V},\mathcal{E}]$`, f: (Id,`$\mathcal{V}$`)`$\rightarrow \mathcal{V}$`): Graph`$[\mathcal{V},\mathcal{E}]$ maps every pair `(id,v)` in the vertices of `g` to `(id, f(v))`.
- `mapReduceTriples(g:Graph`$[\mathcal{V},\mathcal{E}]$`, m: (`$\mathcal{V},\mathcal{E},\mathcal{V}$`)`$\rightarrow$`[(Id,`$\mathcal{M}$`)], r:(`$\mathcal{M},\mathcal{M}$`)`$\rightarrow\mathcal{M}$`):RDD[(Id,`$\mathcal{M}$`)]`, encodes the two-stage parallel computation process commonly known as mapReduce using the triplets view. It takes as parameters, a grapg `g`, a map function `m` and a reduce function `r`.
  In the first stage it applies the `m` to each triplet in the graph to generate a list of messages that will be sent to the vertices identified a given `id`.
  In the second stage, it groups all the messages sent to a given vertex applying the reduce function `r` to each pair of messages.
- `joinVertices(g:Graph`$[\mathcal{V},\mathcal{E}]$`, msgs:RDD[(Id, `$\mathcal{M}$`)], f:(Id, `$\mathcal{V},\mathcal{M}$`)`$\rightarrow\mathcal{V}$`): Graph`$[\mathcal{V},\mathcal{E}]$`,` joins the collection of messages sent to a the vertices which have a value `(id,m)` with the vertex `v` identified by `id` and replaces that vertex by `f(id,v,m)`.

The GraphX Pregel algorithm is defined iteratively where each iteration is usually called a superstep as follows:

It takes as input a `Graph`$[\mathcal{V},\mathcal{E}]$ and the following parameters:

- `initialMsg`: initial message sent to all the vertices
- `vprog` is the vertex program. It is run by each vertex at the beginning of the algorithm using the `initialMsg` and in each superstep using the collected messages sent by the neighbors in the previous superstep.
- `sendMsg` takes as parameter an triplet and returns an iterator with a pair `(id, msg)` where `id` represents the id of the vertex which will receive the message and `msg` represents the message that will be sent.
- `mergeMsg` is a function that defines how to merge 2 messages into one. This function must be associative and commutative, and will be invoked to collect all the messages that are sent to a vertex in each superstep.

We have implemented a ShEx validation algorithm based on the Pregel algorithm. The algorithm assumes that there is a ShEx schema $\langle \mathcal{L}, \delta \rangle$ where each label $l \in \mathcal{L}$ identifies a shape expression.

---

[30] https://spark.apache.org/docs/latest/graphx-programming-guide.html

---

Algorithm 1: Pregel algorithm pseudocode as implemented in GraphX

Input parameters:
> g: Graph$[\mathcal{V},\mathcal{E}]$
> initialMsg: $\mathcal{M}$
> vProg: $(\text{Id},\mathcal{V},\mathcal{M}){\rightarrow}\mathcal{V}$
> sendMsg: $\text{Triplet}{\rightarrow}[(\text{Id},\mathcal{M})]$
> mergeMsg: $(\mathcal{M},\mathcal{M}){\rightarrow}\mathcal{M}$

Output: g:Graph$[\mathcal{V},\mathcal{E}]$

1 g $=$ mapVertices(g,λ(id,v)$\rightarrow$vProg(id,v,initialMsg))
2 msgs $=$ mapReduceTriples(g,sendMsg,mergeMsg)
3 while size(msgs)> 0 **do**
4 $\quad$ g $=$ joinVertices(g,msgs,vProg)
5 $\quad$ msgs $=$ mapReduceTriples(g,sendMsg,mergeMsg)

6 return g

The algorithm annotates each node $n \in \mathcal{V}$ with a status map that represents the validation status with regards to some labels. The new nodes in the graph will be tuples (n, m) where $n \in \mathcal{V}$, and $m : \mathcal{L} \mapsto Status$ associates a status for each shape label.

A *Status* is defined as:

$Status ::=$ `Undefined`   Default status
$\quad\quad\quad$ | `Ok`   Node conforms
$\quad\quad\quad$ | `Failed`   Node doesn't conform
$\quad\quad\quad$ | `Pending`   Requested to conform
$\quad\quad\quad$ | `WaitingFor`(ds, oks, fs) Waiting for some neighbours
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad$ $ds =$ list dependants neighbours
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad$ $oks =$ list of conformant neighbours
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad$ $fs =$ list of non conformant neighbours
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad$ where $ds, oks, failed \in \mathcal{V} \times \mathcal{P} \times \mathcal{L}$

The status can be `Undefined` if there is no information yet (this is the default value) `Ok` if the node conforms to the shape identified by l, `Failed` if it doesn't conform to the shape, `Pending` if the node has been requested to be validated with that label or `WaitingFor`$(ds, oks, failed)$ if the validation of node n depends on the validation of a set of neighbour nodes ds. Each neighbour node is represented by a triple (v, p, l) where v is the neighbour node, p is the property which links n with v, and l is the shape label that the node must conform. During the validation, we may receive information that some of those neighbour nodes have been validated or failed. That information is collected in the set oks which is the set of conforming neighbour nodes and $failed$ is the set of failed neighbour nodes.

A message can be represented as a map which assigns to each label the following requests:

$$\begin{aligned}
\text{Msg} ::= \; & \texttt{Validate} && \text{Request to validate} \\
| \; & \texttt{Checked}(oks, fs) && \text{Some neighbours have been checked} \\
& && oks = \text{neighbours that have been checked as conformant} \\
& && fs = \text{neighbours that have been checked as non-conformant} \\
& && \text{where } oks, fs \in \mathcal{V} \times \mathcal{P} \times \mathcal{L} \\
| \; & \texttt{WaitFor}(ds) && \text{Request to wait for some neighbours} \\
& && \text{where } ds \in \mathcal{V} \times \mathcal{P} \times \mathcal{L}
\end{aligned}$$

The ShEx+Pregel validation traversal is defined with the following pseudo-code.

Algorithm 2: Pregel-based ShEx validation pseudocode

Input parameters:
  g: $\texttt{Graph}[\mathcal{V}, \mathcal{E}]$
  initialLabel: $\mathcal{L}$
  checkLocal: $(\mathcal{L}, \mathcal{V}) \rightarrow \texttt{Ok}| \texttt{Failed}| \texttt{Pending}(\texttt{Set}[\mathcal{L}])$
  checkNeighs: $(\mathcal{L}, \texttt{Bag}[(\mathcal{E}, \mathcal{L})], \texttt{Set}[(\mathcal{E}, \mathcal{L})]) \rightarrow \texttt{Ok}|\texttt{Failed}$
  tripleConstraints: $\mathcal{L} \rightarrow \texttt{Set}[(\mathcal{E}, \mathcal{L})]$

Output: g:$\texttt{Graph}[(\mathcal{V}, \mathcal{L} \mapsto \text{Status}), \mathcal{E}]$

gs $=$ mapVertices(g, λ(id, v)$\rightarrow$(id, (v, λv$\rightarrow$Undefined)))
gs $=$ pregel(Validate, gs, vProg, sendMsg, mergeMsg)
gs $=$ mapVertices(gs, checkUnsolved)
return gs

def checkUnsolved(v,m) $=$ (v,m') where

$$\text{m'}(l) = \begin{cases} \texttt{checkNeighs}(l, \emptyset, \emptyset) & \text{if } m(l) = \texttt{Pending} \\ \texttt{checkNeighs}(l, \text{oks}, \text{fs} \cup \text{ds}) & \text{if } m(l) = \texttt{WaitingFor}(ds, oks, fs)\} \\ m(l) & \text{otherwise} \end{cases}$$

def vProg:$(\texttt{Id}, \mathcal{V}, \mathcal{M}) \rightarrow \mathcal{V} = $ ...see 13

The algorithm takes as input the parameters:

- `initialLabel` is the initial shape label that is requested to validate every node in the graph. In Shape Expressions, this label is usually annotated with the `start` keyword.
- `checkLocal` checks if the shape expression associated with a label can validate a node locally. It returns `Ok` if the node validates without further dependencies, `Failed`, if it doesn't validate, and `Pending`(ls) if the validation of the node depends on a list of shape labels ls.
- `checkNeighs` checks if the bag of neighbors of a node matches the regular bag expression associated with the label in the schema.
- `tripleConstraints` returns the list of triple constraints associated with the shape expression indicated by the label.

The algorithm starts by mapping every node to the status which associates any label $l \in \mathcal{L}$ to undefined (`Undefined`). After that, it runs the iterative Pregel

algorithm using the `vProg`, `sendMsg` and `mergeMsg` functions defined as above. Once the Pregel algorithm finishes, it replaces the status of any node that is pending or waiting for some neighbours by a last check based on the current information of the neighbours, assuming that if the node didn't receive information that a pending neighbour has validated, it means that there was no evidence of it's validation, and it failed.

`vProg` changes the status map of a node with regards to a label when it receives a message for that label. It can be defined as:

$vProg(id,(n,m), msg) = (n,m')$ where $m'(l) = m(l)$ except for the cases indicated by the following rules:

$$\frac{(n,m), l \rightsquigarrow \texttt{Validate} \qquad \texttt{checkLocal}(l,n) = r \in \{\texttt{Ok}, \texttt{Failed}\}}{m(l) = s \in \{\texttt{Undefined}, \texttt{Pending}\}}{m'(l) = r}$$

$$\frac{(n,m), l \rightsquigarrow \texttt{Validate} \qquad \texttt{checkLocal}(l,n) = \texttt{Pending}(ls)}{m(l) = r \in \{\texttt{Undefined}, \texttt{Pending}\}}{m'(l) = \texttt{Undefined}}$$
$$m'(l') = \texttt{Pending} \ \forall l' \in ls$$

$$\frac{(n,m), l \rightsquigarrow \texttt{Validate}}{m(l) = r \in \{\texttt{Ok}, \texttt{Failed}\}}{m'(l) = r}$$

$$\frac{(n,m), l \rightsquigarrow \texttt{Validate}}{m(l) = \texttt{WaitingFor}(ds, oks, fs)}{m'(l) = \texttt{Ok}}$$

$$\frac{(n,m), l \rightsquigarrow \texttt{Checked}(oks, fs) \qquad ds \setminus (oks \cup fs) \neq \emptyset}{m(l) = \texttt{WaitingFor}(ds, oks', fs')}{m'(l) = \texttt{WaitingFor}(ds, oks \cup oks', fs \cup fs')}$$

$$\frac{(n,m), l \rightsquigarrow \texttt{Checked}(oks, fs) \qquad ds \setminus (oks \cup fs) = \emptyset}{m(l) = \texttt{WaitingFor}(ds, oks', fs')}{m'(l) = \texttt{checkNeighs}(l, oks \cup oks', fs \cup fs')}$$

Table 13. Definition of `vProg` for Pregel-based ShEx validation

Figure 9 represents a state diagram which shows the different status that a node can have with regards to a shape label. Initially, all nodes have status `Undefined` until they get a message request to validate against some label. If it is possible to validate locally those nodes, then they will go directly to the end state which can be `Ok` or `Failed`. Otherwise, if their validation depends on the neighbours, they will enter the status `Pending` whose nodes are active in the Pregel algorithm and will be activated in the messages generation phase. If they

receive a request to wait for some other nodes to be validated, they will go to the state $\mathtt{WaitingFor}(ds, oks, fs)$ which means that they are waiting for the status of the neighbour nodes ds.

In subsequent phases, they can receive notifications that some of those neighbour nodes have either been validated or not updating the corresponding values of oks and fs. Once all the pending neighbours have either been validated or failed, it will invoke $\mathtt{checkNeighs}(l, oks, fs)$ to check if the regular expression matches taking into account which neighbours conform or don't conform and passing to the state $\mathtt{Ok}$ or $\mathtt{Failed}$ which is inactive.

Once executed the Pregel algorithm, it is possible that some nodes are in state $\mathtt{Pending}$and don't receive any message, which means that their validation depends on the existence of some arcs pointing to some neighbours and they didn't receive messages from those arcs, i.e. there are no arcs in the graph. In that case, a last step in the algorithm checks if those nodes can validate with an empty neighbourhood.
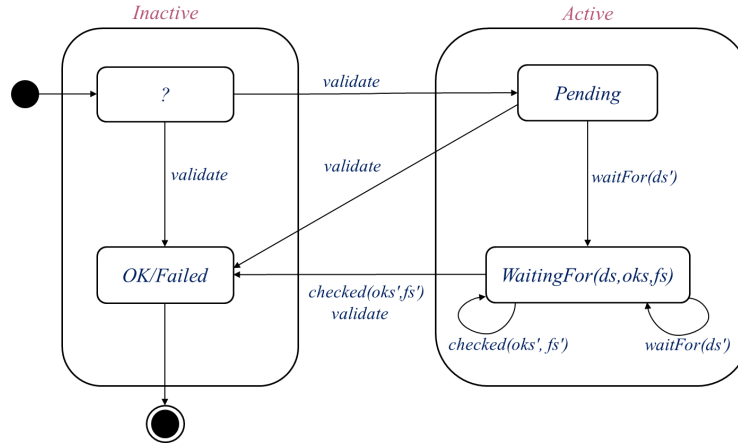

Fig. 9. State diagram representing the different states in vProg

In order to define $\mathtt{sendMsg}(Triplet){:}[(Id,Msg)]$ we will use the notation $(x, m_x), l \rightsquigarrow Msg$ to represent that message Msg is sent to the node x with status map m_x for label l.

Table 13 represents the rules that declare which messages are sent for each triplet view which is represented as $\langle(s, m_s), p, (o, m_o)\rangle$ where (s, m_s) is the subject, p the predicate and (o, m_o) the object:

Finally $\mathtt{mergeMsg}$ merges the messages that arrive to the same node and can be defined as:

$\mathtt{mergeMsg}((n, m), l \rightsquigarrow msg_1, (n, m), l \rightsquigarrow msg_2) = (n, m), l \rightsquigarrow msg_1 \oplus msg_2$
where

$$\frac{\langle(s, m_s), p, (o, m_o)\rangle \in \mathcal{G} \quad m_s(l) = \mathtt{Pending} \quad tcs(l, \mathcal{S}) = \_ \xrightarrow{p} @l'}{(s, m_s), l \rightsquigarrow \mathtt{WaitFor}((o, p, l'))}$$
$$(o, m_o), l \rightsquigarrow \mathtt{Validate}$$

$$\frac{\langle(s, m_s), p, (o, m_o)\rangle \in \mathcal{G} \quad m_s(l) = \mathtt{WaitingFor}(ds, oks, fs) \quad (o, p, l') \in ds \quad m_o(l') = \mathtt{Ok}}{(s, m_s), l \rightsquigarrow \mathtt{Checked}((o, p, l'), \emptyset)}$$

$$\frac{\langle(s, m_s), p, (o, m_o)\rangle \in \mathcal{G} \quad m_s(l) = \mathtt{WaitingFor}(ds, oks, fs) \quad (o, p, l') \in ds \quad m_o(l') = \mathtt{Failed}}{(s, m_s), l \rightsquigarrow \mathtt{Checked}(\emptyset, (o, p, l'))}$$

Table 14. Definition of `sendMsg` for Pregel-based ShEx validation

$$\mathtt{Validate} \oplus y = y$$
$$\mathtt{Validate} \oplus \mathtt{Checked}(oks, fs) = \mathtt{Checked}(oks, fs)$$
$$\mathtt{Validate} \oplus \mathtt{WaitFor}(ds) = \mathtt{WaitFor}(ds)$$
$$\mathtt{Checked}(oks, fs) \oplus \mathtt{Validate} = \mathtt{Checked}(oks, fs)$$
$$\mathtt{Checked}(oks, fs) \oplus \mathtt{Checked}(oks', fs') = \mathtt{Checked}(oks \cup oks', fs \cup fs')$$
$$\mathtt{Checked}(oks, fs) \oplus \mathtt{WaitFor}(ds) = \mathtt{Checked}(oks \cup ds, fs \cup fs)$$
$$\mathtt{WaitFor}(ds) \oplus \mathtt{Validate} = \mathtt{WaitFor}(ds)$$
$$\mathtt{WaitFor}(ds) \oplus \mathtt{Checked}(oks, fs) = \mathtt{Checked}(oks \cup ds, fs)$$
$$\mathtt{WaitFor}(ds) \oplus \mathtt{WaitFor}(ds') = \mathtt{WaitFor}(ds \cup ds')$$

The algorithm presented in figure 2 required as parameters a function `checkLocal`: $(\mathcal{L}, \mathcal{V}) \to \mathtt{Ok}|\, \mathtt{Failed}|\, \mathtt{Pending}(\mathtt{Set}[\mathcal{L}])$ that returns `Ok` if it is possible to check that the node conforms to a shape label locally, `Failed` if it is possible to check that a node doesn't conform to a shape label locally, and `Pending`(ls) if the conformance of a node depends on the arcs in ls.

Figure 3 presents a possible implementation of `checkLocal` for WShEx.

The definition of `checkNeighs`: $(\mathcal{L}, \mathtt{Bag}[(\mathcal{E}, \mathcal{L})], \mathtt{Set}[(\mathcal{E}, \mathcal{L})]) \to \mathtt{Ok}|\mathtt{Failed}$ is shown in figure 4.

Finally, `tripleConstraints`: $\mathcal{L} \to \mathtt{Set}[(\mathcal{E}, \mathcal{L})]$— returns the triple constraints associated with a shape label. It is defined in figure 5

Example 19 (Pregel+ShEx example).

As an example, we will use the Wikibase graph from example 4 to validate the ShEx schema from example 18. We replace the shape labels by their initial so we will use:

$$\mathcal{L} = \{ \textit{Researcher, Place, Country, Date, Human}\}$$
$$\delta(R) = \{ \_ \xrightarrow{instanceOf} @\mathrm{H}; \_ \xrightarrow{birthDate} @\mathrm{D}; \_ \xrightarrow{birthPlace} @\mathrm{P} \}$$
$$\delta(P) = \{ \_ \xrightarrow{country} @\mathrm{C}\}$$
$$\delta(C) = \{ \}$$
$$\delta(D) = \in xsd : date$$
$$\delta(H) = \in \{Human\}$$

The first step of the algorithm will send a message to every node requesting it to validate with the shape *Researcher*, and after running *vProg* the status

---

Algorithm 3: Definition of checkLocal:$(\mathcal{L}, \mathcal{V}) \rightarrow$ Ok | Failed | Pending(Set$[\mathcal{L}]$)

---

def checkLocal(l, n) = checkLocal$(\delta(l), n)$
def *checkLocal*(se, n)=**match** se
   | **case** se_1 AND $se_2 \Rightarrow$ combine(checkLocal(se_1, n), checkLocal(se_2, n))
   | **case** @$l \Rightarrow$ checkLocal$(\delta(l), n)$
   | **case** { te } $\Rightarrow$ checkLocal(te, n)
   | **case** CLOSED { te } $\Rightarrow$ checkLocal(te, n)

def *checkLocal*(te, n) = **match** te
   | **case** $te_1; te_2 \Rightarrow$ combine(checkLocal(te_1, n), checkLocal(te_2, n))
   | **case** $te_1 \mid te_2 \Rightarrow$ combine(checkLocal(te_1, n), checkLocal(te_2, n))
   | **case** $te* \Rightarrow$ checkLocal$(te,)$
   | **case** $\_ \xrightarrow{p}$ @l $qs \Rightarrow$ Pending$\{l\}$

def *combine*(r_1, r_2) = **match** (r_1, r_2)
   | **case** (Ok,Ok) $\Rightarrow$ Ok
   | **case** (Ok,Pendingls) $\Rightarrow$ Pendingls
   | **case** (Ok,Failed) $\Rightarrow$ Failed
   | **case** (Pending(ls),Ok) $\Rightarrow$ Pendingls
   | **case** (Pending(ls_1),Pending(ls_2)) $\Rightarrow$ Pending$(ls_1 \cup ls_2)$
   | **case** (Pending(ls),Failed) $\Rightarrow$ Failed
   | **case** (Failed,$\_$) $\Rightarrow$ Failed

Algorithm 4: Definition of checkNeighs: $(\mathcal{L}, $ Bag$[(\mathcal{E}, \mathcal{L})]$, Set$[(\mathcal{E}, \mathcal{L})])$ $\rightarrow$ Ok | Failed

---

def checkNeighs(l, w, fs) = checkNeighs$(\delta(l), w, fs)$
def *checkNeighs*(se, w, fs)=**match** se
   | **case** $cond \Rightarrow$ Ok
   | **case** se_1 AND $se_2 \Rightarrow$ checkNeighs$(se_1, w, fs) \wedge$ checkNeighs(se_2, w, fs)
   | **case** @$l \Rightarrow$ checkNeighs$(\delta(l), w, fs)$
   | **case** { te } $\Rightarrow$ $w \cong$ rbe(te)
   | **case** CLOSED { te } $\Rightarrow$ $w \cong$ rbe$(te) \wedge fs = \emptyset$

def *rbe*(te)=**match** te
   | **case** $te_1; te_2 \Rightarrow$ rbe(te_1); rbe(te_2)
   | **case** $te_1 \mid te_2 \Rightarrow$ rbe$(te_1) \mid$ rbe(te_2)
   | **case** $te* \Rightarrow$ rbe$(te)*$
   | **case** $\_ \xrightarrow{p}$ @l $qs \Rightarrow (p, l)$

---

of every node will be Pending on shape *Researcher*. In the first superstep, the messages that will be generated by each triple are[31]:

---

[31] For simplicity, for each node (x, m_x) we show only x and we omit qualifiers in the triples as they are always empty

---

Algorithm 5: Definition of `tripleConstraints`: $\mathcal{L} \rightarrow$ `Set`$[(\mathcal{E}, \mathcal{L})]$

---

def *tripleConstraints(se)* =**match** *se*

    case *cond* $\Rightarrow$ {}

    case *se₁* `AND` *se₂* $\Rightarrow$
    `tripleConstraints(se_1)` $\cup$ `tripleConstraints(se_2)`

    case @*l* $\Rightarrow$ `tripleConstraints`$(\delta(l))$

    case { *te* } $\Rightarrow$ `tripleConstraints(te)`

    case `CLOSED` { *te* } $\Rightarrow$ `tripleConstraints(te)`

def *tripleConstraints(te)* =**match** *te*

    case *te₁*; *te₂* $\Rightarrow$
    `tripleConstraints(te_1)` $\cup$ `tripleConstraints(te_2)`

    case *te₁* | *te₂* $\Rightarrow$
    `tripleConstraints(te_1)` $\cup$ `tripleConstraints(te_2)`

    case *te*∗ $\Rightarrow$ `tripleConstraints(te)`

    case $\_ \xrightarrow{p}$ @*l qs* $\Rightarrow$ $\{(p, l)\}$

Triple	Messages
$(timBl, birthPlace, London)$	$timBl, R \rightsquigarrow$ `WaitFor`$((London, birthPlace, P))$ $London, P \rightsquigarrow$ `Validate`
$(timBl, instanceOf, Human)$	$timBl, H \rightsquigarrow$ `WaitFor`$((Human, instanceOf, H))$ $Human, H \rightsquigarrow$ `Validate`
$(vintCerf, birthPlace, NewHaven)$	$vintCerf, P \rightsquigarrow$ `WaitFor`$((NewHaven, birthPlace, P))$ $Human, H \rightsquigarrow$ `Validate`
$(vintCerf, instanceOf, Human)$	$vintCerf, H \rightsquigarrow$ `WaitFor`$((Human, instanceOf, H))$ $Human, H \rightsquigarrow$ `Validate`

After running *vProg* the status of all nodes except *timBl* and *vintCerf* with regards to the label R will be `Failed` because they will fail to *checkLocal*. The status of both *timBl* and *vintCerf* will be waiting for the validation of their neighborhood nodes.

After superstep 2, the messages generated will be:

Triple	Messages
$(London, country, UK)$	$London, P \rightsquigarrow$ `WaitFor`$((UK, country, C))$ $UK, C \rightsquigarrow$ `Validate`
$(timBl, instanceOf, Human)$	$timBl, R \rightsquigarrow$ `Checked`$((Human, instanceOf, H), \{\})$
$(vintCerf, instanceOf, Human)$	$vintCerf, P \rightsquigarrow$ `Checked`$((Human, instanceOf, H), \{\})$
$(vintCerf, birthPlace, NewHaven)$	$vintCerf, P \rightsquigarrow$ `Checked`$((\{\}), (NewHaven, birthPlace, P))$

After running *vProg*, the status of UK will be `Ok` for shape label C, the status of *London* will be waiting for UK to validate as C, the status of *vintCerf* will be `Failed` for shape label R (it fails because the value *birthPlace* failed). In the third superstep, the messages generated will be:

Triple	Messages
$(London, country, UK)$	$London, P \rightsquigarrow$ `Checked`$((UK, country, C), \{\})$

Which will change the status of *London* to `Ok` for shape label P. In the fourth superstep, the messages that will be sent are:

Triple	Messages
$(timBl, birthPlace, London)$	$timBl, R \rightsquigarrow$ `Checked`$((London, birthPlace, P), \{\})$

And after running $vProg$ the status of $timBl$ for shape label R will be `Ok`. In the next superstep, no more messages will be sent and all the nodes will be inactive, finalizing the Pregel algorithm. The nodes that have a `Ok` status for some shape are:

Node	Shape Label
$timBl$	R
$London$	P
UK	C

And the generated subset will consist of collecting information about those nodes:

$\rho = \{$ $(timBl, instanceOf, Human, \{\})$,
$(timBl, birthDate, 1955, \{\})$,
$(timBl, birthPlace, London, \{\})$,
$(London, country, UK, \{\})$,

Representing Wikidata in Spark GraphX Spark GraphX supports a kind of property graphs where vertices and edges can have an associated value. The type `Graph[VD,ED]` is used to represent a graph whose vertices are pairs of values of type `(VertexId,VD)` where `VertexId=Long` represents a unique vertex identifier and `VD` represents the value associated with the vertex.

When representing Wikidata graphs in Spark GraphX it is necessary to take into account which entities will be represented as vertices. We opted to include as nodes in the graph only those nodes that can be subjects of statements, i.e. wikidata entities (items and properties), leaving out primitive values like literals, dates, etc. We separated the statements associated with an entity in two kind of statements: local statements, which are embedded inside the value of a node and whose values can be accessed without traversing the graph, and entity statements, whose values are other entities which have their corresponding vertex in the graph.

In this way, the WShEx also needs to distinguish triple expressions between local ones and entity ones.

6   Implementations and preliminary results

Although we have implemented some of the approaches presented in the paper, further work needs to be done in order to have all the possible implementations. In table 15, we present an overview of the approaches presented in the paper and their implementation.

The two implementations created by the authors of this paper are WDSub and SparkWDSub. They have been implemented in Scala and are mainly proof-of-concepts which have not yet been thoroughly tested and optimized so the results presented here are preliminary.

WDSub takes as input a simple ShEx file containing the schema and another file with a Wikibase dump. It extracts the subset that matches the ShEx without taking into account shape references. It uses two approaches, one based

	RDF graphs	*Property graphs*	*Wikibase graphs*
Formal definition	Done, ex.[25]	Done, [62, 37]	[This paper]
Description	ShEx	PShEx	WShEx
with Shape Expressions	[62, 72, 6]	[This paper]	[This paper]
Entity-generated subsets	–	–	WDumper
Matching generated subsets	–	–	WDumper
ShEx-based matching	–	–	WDSub
ShEx+Slurping	shex.js PyShEx	–	–
ShEx + Pregel	–	–	SparkWDSub

Table 15. Overview of implementation/formal definition status

on Wikidata toolkit[32], and another one based on the fs2 library[33]. We tested it using a virtual server running on XCP-ng 8.2 (CentOS 8 stream) with 32 cores and 64Gb of RAM and it was able to process the JSON Wikidata dump from 2021, which has 1.256,55Gb uncompressed, in 5h 15min.

SparkWDSub implements the Pregel+ShEx approach. It has been built on top of the Spark GraphX library [34]. In order to obtain greater flexibility, it defines a generic `PSchema` class[35] which defines a generic graph validation algorithm.

Given a target graph `Graph[VD,ED]` of vertex `VD` and edges `ED`, after running Pregel, it will return `ShapedGraph[VD,L,E,P]` which will contain information about the shapes labels `L` associated with a vertex `VD` in a graph with properties `P` and possible errors `E` and the `MsgMap[L,E,P]` class that represents the messages sent to a vertex.

The `PSChema` class is defined as follows:

```scala
class PSchema[VD, ED, L, E, P]
    (checkLocal: (L, VD) => Either[E, Set[L]],
     checkNeighs: (L, Bag[(P,L)], Set[(P,L)]) => Either[E, Unit],
     getTripleConstraints: L => List[(P,L)],
     cnvEdge: ED => P,
    ) {
  def vprog(id: VertexId,
            g: Shaped[VD, L, E, P],
            msg: MsgMap[L, E, P]): Shaped[VD, L, E, P] = {
            . . . }

  def sendMsg(t: EdgeTriplet[Shaped[VD, L, E, P],ED]
            ): Iterator[(VertexId, MsgMap[L, E, P])] = {
            . . . }

  def mergeMsg(p1: MsgMap[L,E,P],
            p2: MsgMap[L,E,P]): MsgMap[L,E,P] = {
```

---

[32] https://www.mediawiki.org/wiki/Wikidata_Toolkit
[33] https://fs2.io/
[34] http://spark.apache.org/graphx/
[35] https://github.com/weso/sparkwdsub/blob/master/src/main/scala/es/weso/pschema/PSchema.scala

```
                . . . }

  def runPregel(graph: Graph[VD,ED],
                initialLabel: L,
                maxIterations: Int = Int.MaxValue,
              ): Graph[Shaped[VD,L,E,P],ED] = {
    Pregel(shapedGraph(graph),
           initialMsg(initialLabel),
           maxIterations)(vprog, sendMsg, mergeMsg)
    .mapVertices(checkRemaining)
  }
}
```

where `checkRemaining` checks which of the nodes with `Pending` status are valid
or not.

The previous interface enables the possible implementation with different
validation languages like ShEx, PShEx or WShEx.

SparkWDSub currently offers an implementation for a simplified version of
WShEx which implements the `checkLocal`, `checkNeighs`, `getTripleConstraints`
and `cnvEdge` to convert an edge `ED` into a property `P`.

We have run an experiment on AWS using 512 cores, 3.904Gb RAM and
121.600Gb hard disk, and it was possible to generate subsets for the 2014 Wiki-
data dump (31.3 GB) in 3 minutes, while with the 2021 Wikidata dump (1.256,55
Gb uncompressed) it took 36 minutes.

In the experiments we used the following ShEx schema to obtain cities:

```
prefix wde: <http://www.wikidata.org/entity/>

Start = @<City>

<City> {
 wde:P31 @<CityCode>
}
<CityCode> [ wde:Q515 ]
```

7   Related work

Knowledge graphs
An introduction to Knowledge graphs is provided by [30], which cites other
books [58, 33, 20] and surveys like [59, 78, 80, 22, 79, 2]. In this paper, we follow
two of the graph models provided in that survey: directed labeled graphs, which
we call RDF-graphs, and property graphs; and add a new one: wikibase graphs.
Our definition of wikibase graphs has been inspired by MARS (Multi-Attributed
Relational Structures) [51], which are a a generalized notion of property graphs.
In that paper, they also define MAPL (Multi-Attributed Predicate Logic) as a
logical formalism that can be used for ontological reasoning.

Knowledge graph descriptions

Since the appearance of ShEx in 2014, there has been a lot of interest about RDF validation and description. In 2017, the data shapes working group proposed SHACL (Shapes Constraint Language) as a W3C recommendation [34]. Although SHACL can be used to describe RDF, its main purpose is to validate and check constraints about RDF data makes it less usable to describe RDF subsets.

ShEx was adopted by Wikidata in 2019 to define entity schemas [75]. We consider that ShEx adapts better to describe data models than SHACL, which is more focused on constraint violations. A comparison between both is provided in [39] while in [37], a simple language is defined that can be used as a common subset of both.

Improving quality of Knowledge graphs in general, and Wikidata in particular, has been the focus of some recent research like [60, 76, 69].

Following the work on MARS, there has been some recent work about adding an inference layer on top of Wikidata. The project SQID [50] combines inference and visualization to create a Wikidata browser. Another possibility that has been explored is to use MARS reasoning to define constraints [46].

Big data processing and graphs

There has been a lot of interest in the last decade to develop scalable algorithms that can process big data graphs. In 2010, Pregel was proposed by Google [42] as a suitable model for large-scale graph computing. Following that publication, several systems were developed like GraphLab [41], PowerGraph [24] and GraphX [23] which followed the lemma *thinking like a vertex*[52] where scalable graph are based on local iterations over the nodes of a graph and their neighbors.

GraphX was a framework that internally represented graphs using Apache Spark's Resilient Distributed Datasets (RDDs) [81] enabling to implement graph-parallel abstractions and algorithms like Pregel. A functional definition of Spark using Haskell has been proposed in [12]. It would be interesting to use that functional specification to prove the correctness of the algorithm proposed in this paper. Another related line of work is the creation of domain specific languages to facilitate encoding of Pregel-based algorithms like Palgol [82] or Fregel [18].

Knowledge graphs subsets Although it is possible to create subsets of RDF graphs using SPARQL construct queries, the approach usually requires some scripts to launch the queries as SPARQL doesn't support recursion so it is not possible to represent cyclic data models. There has been a proposal to extend SPARQL with recursion [64], but it is not part of most existing SPARQL processors.

RDFSlice [49, 47] was proposed as a system that generated RDF data fragments from large endpoints like DBpedia. It defines a subset of SPARQL called sliceSPARQL. A new version, called Torpedo, was proposed in [48], which improves the perfomance and adds further expressivity. The use of SPARQL to generate subsets is one important difference with the work presented in this paper.

We consider that ShEx improves the expressiveness and usability of SPARQL to describe data models and subsets as it allows cyclic or recursive data models and has a declarative syntax that is specifically defined fur such endeavor.

The creation of Wikidata subsets has been a topic of interest since the 12th International SWAT4HCLS Conference[36]. It was later selected as a topic in the Elixir Europe Biohackathon 2020[37] and the SWAT4HCLS 2021 hackathon from which a prepring was generated that collects the different approaches proposed [38].

One of the approaches was WDumper [38], which was created as a tool that processed Wikidata dumps and generates subsets from those dumps. It takes two inputs: a JSON compressed dump and a JSON configuration file that describes the different filters, and generates as output an RDF compressed dump. The tool can be run as a web service locally and is also deployed at https://tools.wmflabs.org/wdumps. It also contains a web service allows the user to introduce the different filters filling a form which generates a dump generation request that is added to a queue. It is possible to see the list of previously requested dumps. Once the dump has been generated, it can be uploaded to Zenodo. WDumper is divided in two main modules, the backend, which has been implemented in Java using the Wikidata Toolkit library [39] and the frontend that has been implemented in Typescript. The use of WDumper to generate Wikidata subsets is described in [31] where 4 subsets are created about the topics: politicians, military politicians, UK universities and GeneWiki data. That paper also presents several use case scenarios and discusses some strengths and weaknesses. In this paper, we present a formal definition of WDumper in the context of other subset generation approaches like the ShEx-based ones.

The Python library WikidataSets [8] generates Wikidata subsets from specific topics. In the paper the authors generated subsets for the following topics: humans, countries, companies, animal species and films. The tool obtains items following the instances of a topic or subclasses of the topic.

KGTK (Knowledge graph toolkit) [32] is a tool that works with knowledge graphs by defining a common format called KGTK format based on hypergraphs. It is possible to import and export data from different formats like Wikidata or ConceptNet, do several operations over those graphs like: validation, cleaning, graph manipulation (sort, column removal, edge filtering) and graph merging (join, cat) operations. The tool also supports graph querying and analytics operations. Given that KGTK can take as input Wikidata dumps and generate Wikidata dumps as output, it is possible to use KGTK to generate subsets of Wikidata. More recently, the authors have published a paper where they apply KGTK to create personalized versions of wikidata using a query language that

---

[36] This page was created to collect information: https://www.wikidata.org/wiki/Wikidata:WikiProject_Schemas/Subsetting

[37] https://github.com/elixir-europe/BioHackathon-projects-2020/tree/master/projects/35

[38] https://github.com/bennofs/wdumper

[39] https://github.com/Wikidata/Wikidata-Toolkit

they call Kypher[10], which is an adapted version of Cypher for KGTK. Internally, it uses a tabular data model which allows to translate Kypher queries to SQLite[40]. According to the paper, it can create very efficient queries processing the whole Wikidata in a laptop system. The main difference with our approach is that we use Shape Expressions to describe the data model of the subset. Further work could be done to see if it was possible to translate Shape Expression definitions to Kypher.

8   Conclusions

In this paper, we have presented three formal models for knowledge graphs: RDF-based graphs, property graphs and wikibase graphs. We also defined a shape expressions language that can be used to describe and validate data in those models: ShEx for RDF-based graphs, PShEx for property graphs and WShEx for wikibase graphs.

Given the success of knowledge graphs, their size has been increasing in a way that it is not possible to process their contents using conventional tools making it necessary to have some mechanism to extract subsets from them. Finally, we review some approaches to generate subsets from Wikibase graphs. The first two approaches, entity-matching and simple matching can be implemented by processing Wikibase dumps sequentially. The third approach takes as input a WShEx schema, and matches the different entities and their local neighbors with the shapes ignoring shape references. This approach can be used to efficiently process dumps sequentially but doesn't take into account the relations in the graph. The fourth approach, ShEx+Slurp, adds an option to the ShEx processor to collect the visited triples while it is validating the data. This approach can do graph traversal but also require a large number of requests for nodes neighbors which may not be possible to apply it behind endpoints that limit the number of requests. The final approach that we proposed applies the Pregel algorithm to validate all nodes. This approach does graph traversal and can also handle large graphs.

All the approaches have been implemented, although not all of them within the same system, so a proper comparison is not yet possible. Further work needs to be done on improving the implementations, applying them to some use cases and assessing their advantages and challenges.

9   Acknowledgments

---

[40] https://www.sqlite.org/

References

[1]   Bilal Abu-Salih. "Domain-specific knowledge graphs: A survey". In: *Journal of Network and Computer Applications* 185 (July 2021), p. 103076. DOI: 10.1016/j.jnca.2021.103076.

[2]   Tareq Al-Moslmi et al. "Named Entity Extraction for Knowledge Graphs: A Literature Overview". In: *IEEE Access* 8 (2020), pp. 32862–32881. DOI: 10.1109/ACCESS.2020.2973928. URL: https://doi.org/10.1109/ACCESS.2020.2973928.

[3]   Renzo Angles et al. "Foundations of Modern Query Languages for Graph Databases". In: *ACM Computing Surveys* 50.5 (2017), 68:1–68:40. DOI: 10.1145/3104031.

[4]   Sören Auer et al. "DBpedia: A Nucleus for a Web of Open Data". In: *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007.* Ed. by Karl Aberer et al. Vol. 4825. Lecture Notes in Computer Science. Springer, 2007, pp. 722–735. ISBN: 978-3-540-76297-3.

[5]   Adrian Bielefeldt, Julius Gonsior, and Markus Krötzsch. "Practical Linked Data Access via SPARQL: The Case of Wikidata". In: *Proceedings of the WWW2018 Workshop on Linked Data on the Web (LDOW-18).* Ed. by Tim Berners-Lee et al. Vol. 2073. CEUR Workshop Proceedings. CEUR-WS.org, 2018.

[6]   Iovka Boneva, Jose Emilio Labra Gayo, and Eric Prud'hommeaux. "Semantics and Validation of Shapes Schemas for RDF". In: *International Semantic Web Conference.* 2017.

[7]   Iovka Boneva, Jose Emilio Labra Gayo, and Eric G. Prud'hommeaux. "Semantics and Validation of Shapes Schemas for RDF". In: *The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part I.* Ed. by Claudia d'Amato et al. Vol. 10587. Lecture Notes in Computer Science. Springer, Oct. 2017, pp. 104–120. ISBN: 978-3-319-68287-7.

[8]   Armand Boschin and Thomas Bonald. "WikiDataSets: Standardized subgraphs from Wikidata". In: *arXiv:1906.04536 [cs, stat]* (Oct. 2019). arXiv: 1906.04536. URL: http://arxiv.org/abs/1906.04536.

[9]   Sebastian Burgstaller-Muehlbacher et al. "Wikidata as a semantic framework for the Gene Wiki initiative". In: *Database* 2016 (2016), baw015. DOI: 10.1093/database/baw015.

[10] H. Chalupsky et al. "Creating and Querying Personalized Versions of Wikidata on a Laptop". In: *The 2nd Wikidata Workshop* (2021). URL: https://arxiv.org/abs/2108.07119.

[11] Spencer Chang. *Scaling Knowledge Access and Retrieval at Airbnb*. AirBnB Medium Blog. https://medium.com/airbnb-engineering/scaling-knowledge-access-and-retrieval-at-airbnb-665b6ba21e95. Sept. 2018.

[12] Yu-Fang Chen et al. "An Executable Sequential Specification for Spark Aggregation". In: *Networked Systems*. Springer International Publishing, 2017, pp. 421–438. DOI: 10.1007/978-3-319-59647-1_31.

[13] Constance Crompton et al. "Familiar Wikidata: The Case for Building a Data Source We Can Trust". en. In: (2020). DOI: 10.48404/POP.2020.02.

[14] Richard Cyganiak, David Wood, and Markus Lanthaler. *RDF 1.1 Concepts and Abstract Syntax, W3C Recommendation 25 February 2014*. W3C Recommendation. World Wide Web Consortium, Feb. 2014. URL: https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.

[15] Richard Cyganiak, David Wood, and Markus Lanthaler. *RDF-star and SPARQL-star*. Draft Community Group Report. World Wide Web Consortium, July 2021. URL: https://w3c.github.io/rdf-star/cg-spec.

[16] Deepika Devarajan. *Happy Birthday Watson Discovery*. IBM Cloud Blog. https://www.ibm.com/blogs/bluemix/2017/12/happy-birthday-watson-discovery/. Dec. 2017.

[17] Martin Dürst and Michel Suignard. *Internationalized Resource Identifiers (IRIs)*. RFC 3987. Internet Engineering Task Force, Jan. 2005. URL: http://www.ietf.org/rfc/rfc3987.txt.

[18] Kento Emoto et al. "Think like a vertex, behave like a function! a functional DSL for vertex-centric big graph processing". In: 51.9 (Dec. 2016), pp. 200–213. DOI: 10.1145/3022670.2951938.

[19] Fredo Erxleben et al. "Introducing Wikidata to the Linked Data Web". In: *Proceedings of the 13th International Semantic Web Conference (ISWC 2014)*. Ed. by Peter Mika et al. Vol. 8796. LNCS. Springer, Oct. 2014, pp. 50–65. DOI: 10.1007/978-3-319-11964-9_4.

[20] Dieter Fensel et al. *Knowledge Graphs - Methodology, Tools and Selected Use Cases*. Springer, 2020. ISBN: 978-3-030-37438-9. DOI: 10.1007/978-3-030-37439-6.

[21] Nadime Francis et al. "Cypher". In: *Proceedings of the 2018 International Conference on Management of Data*. ACM, May 2018. DOI: 10.1145/3183713.3190657.

[22] Genet Asefa Gesese, Russa Biswas, and Harald Sack. "A Comprehensive Survey of Knowledge Graph Embeddings with Literals: Techniques and Applications". In: *Proceedings of the Workshop on Deep Learning for Knowledge Graphs (DL4KG2019)*. 2019, pp. 31–40.

[23] Joseph E. Gonzalez et al. "GraphX: Graph Processing in a Distributed Dataflow Framework". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 599–613. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez.

[24] Joseph E. Gonzalez et al. "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs". In: *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX Association, Oct. 2012, pp. 17–30. ISBN: 978-1-931971-96-6. URL: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez.

[25] Claudio Gutierrez, Carlos Hurtado, and Alberto Mendelzon. "Formal aspects of querying RDF databases". In: *SWDB'03: Proceedings of the First International Conference on Semantic Web and Databases*. Berlin, Germany: CEUR-WS.org, 2003.

[26] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. *SPARQL 1.1 Query Language, W3C Recommendation 21 March 2013*. W3C Recommendation. World Wide Web Consortium, Mar. 2013. URL: https://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[27] Qi He, Bee-Chung Chen, and Deepak Agarwal. *Building The LinkedIn Knowledge Graph*. LinkedIn Blog. https://engineering.linkedin.com/blog/2016/10/building-the-linkedin-knowledge-graph. Oct. 2016.

[28] Daniel Hernández, Aidan Hogan, and Markus Krötzsch. "Reifying RDF: What Works Well With Wikidata?" In: *Proceedings of the 11th International Workshop on Scalable Semantic Web Knowledge Base Systems*. Ed. by Thorsten Liebig and Achille Fokoue. Vol. 1457. CEUR Workshop Proceedings. CEUR-WS.org, 2015, pp. 32–47.

[29] Aidan Hogan et al. "Everything you always wanted to know about blank nodes". In: *Journal of Web Semantics* 27–28 (2014), pp. 42–69. DOI: 10.1016/j.websem.2014.06.004.

[30] Aidan Hogan et al. "Knowledge Graphs". In: *ACM Computing Surveys* 54.4 (July 2021), pp. 1–37. DOI: 10.1145/3447772.

[31] Seyed Amir Hosseini Beghaeiraveri, Alasdair J. G. Gray, and Fiona J. McNeill. "Experiences of Using WDumper to Create Topical Subsets from Wikidata". English. In: *CEUR Workshop Proceedings* 2873 (June 2021). ISSN: 1613-0073.

[32] Filip Ilievski et al. "KGTK: A Toolkit for Large Knowledge Graph Manipulation and Analysis". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2020, pp. 278–293. DOI: 10.1007/978-3-030-62466-8_18.

[33] Mayank Kejriwal. *Domain-Specific Knowledge Graph Construction*. Springer Briefs in Computer Science. Springer, 2019. ISBN: 978-3-030-12374-1. DOI: 10.1007/978-3-030-12375-8.

[34] Holger Knublauch and Dimitris Kontokostas. *Shapes Constraint Language (SHACL), W3C Recommendation 20 July 2017*. W3C Recommendation. World Wide Web Consortium, June 2017. URL: https://www.w3.org/TR/2017/REC-shacl-20170720/.

[35] Arun Krishnan. *Making search easier: How Amazon's Product Graph is helping customers find products more easily*. Amazon Blog. https://blog.aboutamazon.com/innovation/making-search-easier. Aug. 2018.

[36] Jose Emilio Labra Gayo, Daniel Fernández Álvarez, and Herminio García-González. "RDFShape: An RDF Playground Based on Shapes". In: *Proceedings of the ISWC 2018 Posters and Demonstrations, Industry and Blue Sky Ideas Tracksco-located with 17th International Semantic Web Conference*. Vol. 2180. CEUR Workshop Proceedings. 2018.

[37] Jose Emilio Labra Gayo et al. "Challenges in RDF Validation". In: *Current Trends in Semantic Web Technologies: Theory and Practice*. Ed. by Giner Alor-Hernández et al. Studies in Computational Intelligence. Springer, 2019, pp. 121–151. DOI: 10.1007/978-3-030-06149-4_6.

[38] Jose Emilio Labra Gayo et al. "Knowledge graphs and wikidata subsetting". In: *BioHackrXiv Preprints* (2021). DOI: https://doi.org/10.37044/osf.io/wu9et.

[39] Jose Emilio Labra Gayo et al. *Validating RDF Data*. Vol. 7. Synthesis Lectures on the Semantic Web: Theory and Technology. Morgan & Claypool, Sept. 2017, pp. 1–328. DOI: 10.2200/s00786ed1v01y201707wbe016. URL: https://doi.org/10.2200/s00786ed1v01y201707wbe016.

[40] Jens Lehmann et al. "DBpedia - A large-scale, multilingual knowledge base extracted from Wikipedia". In: *Semantic Web Journal* 6.2 (2015), pp. 167–195.

[41] Yucheng Low et al. "Distributed GraphLab". In: *Proceedings of the VLDB Endowment* 5.8 (Apr. 2012), pp. 716–727. DOI: 10.14778/2212351.2212354.

[42] Grzegorz Malewicz et al. "Pregel: a system for large-scale graph processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, June 2010. DOI: 10.1145/1807167.1807184.

[43] Stanislav Malyshev et al. "Getting the Most out of Wikidata: Semantic Technology Usage in Wikipedia's Knowledge Graph". In: *Proceedings of the 17th International Semantic Web Conference (ISWC'18)*. Ed. by Denny Vrandečić et al. Vol. 11137. LNCS. Springer, 2018, pp. 376–394.

[44] Stanislav Malyshev et al. "Getting the most out of Wikidata: Semantic technology usage in Wikipedia's knowledge graph". In: *The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12*. Springer, 2018, pp. 376–394.

[45] Frank Manola and Eric Miller. *Resource Description Framework: RDF Primer*. W3C Recommendation. World Wide Web Consortium, Feb. 2004. URL: https://www.w3.org/TR/2004/REC-rdf-primer-20040210.

[46]   David Martin and Peter P. Schneider. "Wikidata Constraints on MARS". In: *Proceedings of the 1st Wikidata Workshop (Wikidata 2020)*. Vol. 2773. 2020. URL: http://ceur-ws.org/Vol-2773/paper-12.pdf.

[47]   Edgard Marx et al. "Large-scale RDF Dataset Slicing". In: *7th IEEE International Conference on Semantic Computing, September 16-18, 2013, Irvine, California, USA*. 2013. URL: http://svn.aksw.org/papers/2013/ICSC_SLICE/public.pdf.

[48]   Edgard Marx et al. "Torpedo: Improving the State-of-the-Art RDF Dataset Slicing". In: *11th IEEE International Conference on Semantic Computing, Jan 30-Feb 1, 2017, San Diego, California, USA*. 2017. URL: https://svn.aksw.org/papers/2017/Torpedo_ICSC/public.pdf.

[49]   Edgard Marx et al. "Towards an Efficient RDF Dataset Slicing". In: *International Journal of Semantic Computing* 07.04 (2013), pp. 455–477. DOI: 10.1142/S1793351X13400151.

[50]   Maximilian Marx and Markus Krötzsch. "SQID: Towards Ontological Reasoning for Wikidata". In: *Proceedings of the ISWC 2017 Posters & Demonstrations Track*. Ed. by Nadeschda Nikitina and Dezhao Song. CEUR Workshop Proceedings. CEUR-WS.org, Oct. 2017.

[51]   Maximilian Marx, Markus Krötzsch, and Veronika Thost. "Logic on MARS: Ontologies for generalised property graphs". In: *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17)*. Ed. by Carles Sierra. International Joint Conferences on Artificial Intelligence, Aug. 2017, pp. 1188–1194. DOI: 10.24963/ijcai.2017/165.

[52]   Robert Ryan McCune, Tim Weninger, and Greg Madey. "Thinking Like a Vertex". In: *ACM Computing Surveys* 48.2 (Nov. 2015), pp. 1–39. DOI: 10.1145/2818185.

[53]   Justin J. Miller. "Graph Database Applications and Concepts with Neo4j". In: *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA March 23rd-24th, 2013*. AIS eLibrary, 2013, pp. 141–147. URL: https://aisel.aisnet.org/sais2013/24.

[54]   Vinh Nguyen, Olivier Bodenreider, and Amit Sheth. "Don't like RDF reification?" In: *Proceedings of the 23rd international conference on World wide web - WWW '14*. ACM Press, 2014. DOI: 10.1145/2566486.2567973.

[55]   Finn Årup Nielsen, Daniel Mietchen, and Egon Willighagen. "Scholia, Scientometrics and Wikidata". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2017, pp. 237–259. DOI: 10.1007/978-3-319-70407-4_36.

[56]   Natasha F. Noy et al. "Industry-scale Knowledge Graphs: Lessons and Challenges". In: *ACM Queue* 17.2 (2019), p. 20.

[57]   Ora Lassila and Ralph R. Swick. *Resource Description Framework (RDF) Model and Syntax Specification*. https://www.w3.org/TR/1999/REC-rdf-syntax-19990222/. 1999. URL: https://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.

[58] Jeff Z. Pan et al., eds. *Exploiting Linked Data and Knowledge Graphs in Large Organisations*. Springer, 2017. ISBN: 978-3-319-45652-2. DOI: 10.1007/978-3-319-45654-6.

[59] Heiko Paulheim. "Knowledge graph refinement: A survey of approaches and evaluation methods". In: *Semantic Web Journal* 8.3 (2017), pp. 489–508. DOI: 10.3233/SW-160218.

[60] Alessandro Piscopo and Elena Simperl. "What we talk about when we talk about wikidata quality". In: ACM, Aug. 2019. DOI: 10.1145/3306446.3340822.

[61] R. J. Pittman et al. *Cracking the Code on Conversational Commerce*. eBay Blog. https://www.ebayinc.com/stories/news/cracking-the-code-on-conversational-commerce/. Apr. 2017.

[62] Eric Prud'hommeaux, Jose Emilio Labra Gayo, and Harold Solbrig. "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*. Ed. by Harald Sack et al. ACM Press, Sept. 2014, pp. 32–40. ISBN: 978-1-4503-2927-9. DOI: 10.1145/2660517.2660523.

[63] Eric Prud'hommeaux et al. *Shape Expressions Language 2.0*. Apr. 2017. URL: https://shexspec.github.io/spec/.

[64] Juan L. Reutter, Adrián Soto, and Domagoj Vrgoč. "Recursion in SPARQL". In: *International Semantic Web Conference (ISWC)*. Springer International Publishing, 2015, pp. 19–35. DOI: 10.1007/978-3-319-25007-6_2.

[65] Marko A. Rodriguez and Peter Neubauer. "Constructions from dots and lines". In: *Bulletin of the American Society for Information Science and Technology* 36.6 (Aug. 2010), pp. 35–41. DOI: 10.1002/bult.2010.1720360610.

[66] Edward W. Schneider. "Course Modularization Applied: The Interface System and Its Implications For Sequence Control and Data Analysis". In: *Association for the Development of Instructional Systems (ADIS), Chicago, Illinois, April 1972*. 1973.

[67] Dan Scott and Stacy Allison-Cassin. "Wikidata: a platform for your library's linked open data". In: *Code4Lib Journal* 40 (May 2018). URL: https://journal.code4lib.org/articles/13424.

[68] Philipp Seifer, Ralf Lämmel, and Steffen Staab. "ProGS: Property Graph Shapes Language (Extended Version)". In: *International Semantic Web Conference*. Vol. 12922. Springer, Oct. 2021, pp. 392–401. DOI: https://doi.org/10.1007/978-3-030-88361-4_23. arXiv: 2107.05566 [cs.DB].

[69] Kartik Shenoy et al. "A Study of the Quality of Wikidata". URL: https://arxiv.org/pdf/2107.00156.pdf.

[70] Saurabh Shrivastava. *Bring rich knowledge of people, places, things and local businesses to your apps*. Bing Blogs. https://blogs.bing.com/search-quality-insights/2017-07/bring-rich-knowledge-of-

`people-places-things-and-local-businesses-to-your-apps`. July 2017.

[71]   Amit Singhal. *Introducing the Knowledge Graph: things, not strings.* Google Blog. `https://www.blog.google/products/search/introducing-knowledge-graph-things-not/`. May 2012.

[72]   Slawek Staworko et al. "Complexity and Expressiveness of ShEx for RDF". In: *18th International Conference on Database Theory, ICDT 2015.* Vol. 31. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 195–211.

[73]   Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. "YAGO: A core of semantic knowledge unifying WordNet and Wikipedia". In: *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007.* Ed. by Carey L. Williamson et al. ACM Press, May 2007, pp. 697–706. ISBN: 978-1-59593-654-7.

[74]   Thomas Pellissier Tanon et al. "From Freebase to Wikidata". In: *Proceedings of the 25th International Conference on World Wide Web.* International World Wide Web Conferences Steering Committee, Apr. 2016. DOI: `10.1145/2872427.2874809`.

[75]   Katherine Thornton et al. "Using Shape Expressions (ShEx) to Share RDF Data Models and to Guide Curation with Rigorous Validation". In: *The Semantic Web - 16th International Conference, ESWC 2019, Portorož, Slovenia, June 2-6, 2019, Proceedings.* Ed. by Pascal Hitzler et al. Vol. 11503. Lecture Notes in Computer Science. Springer, June 2019, pp. 606–620. ISBN: 978-3-030-21347-3. DOI: `10.1007/978-3-030-21348-0\_39`.

[76]   Houcemeddine Turki et al. "Using logical constraints to validate information in collaborative knowledge graphs: a study of COVID-19 on Wikidata". en. In: (2020). DOI: `10.5281/ZENODO.4445363`.

[77]   Denny Vrandečić and Markus Krötzsch. "Wikidata: A Free Collaborative Knowledge base". In: *Communications of the ACM* 57.10 (2014), pp. 78–85.

[78]   Xiu-Qing Wang and Shun-Kun Yang. "A Tutorial and Survey on Fault Knowledge Graph". In: *International 2019 Cyberspace Congress, CyberDI and CyberLife, Beijing, China, December 16–18, 2019, Proceedings, Part II.* Ed. by Huansheng Ning. Springer, Dec. 2019, pp. 256–271.

[79]   Guohui Xiao et al. "Virtual Knowledge Graphs: An Overview of Systems and Use Cases". In: *Data Intelligence* 1.3 (2019), pp. 201–223. DOI: `10.1162/dint\_a\_00011`. URL: `https://doi.org/10.1162/dint%5C_a%5C_00011`.

[80]   Jihong Yan et al. "A retrospective of knowledge graphs". In: *Frontiers of Computer Science* 12.1 (2018), pp. 55–74. DOI: `10.1007/s11704-016-5228-9`. URL: `https://doi.org/10.1007/s11704-016-5228-9`.

[81]   Matei Zaharia et al. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". In: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12).* San

Jose, CA: USENIX Association, Apr. 2012, pp. 15–28. ISBN: 978-931971-92-8. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

[82]   Yongzhe Zhang, Hsiang-Shang Ko, and Zhenjiang Hu. "Palgol: A High-Level DSL for Vertex-Centric Graph Processing with Remote Data Access". In: Springer International Publishing, 2017, pp. 301–320. DOI: 10.1007/978-3-319-71237-6_15.