Segunda prueba
Trabajo de investigación

Jose Emilio Labra Gayo

Universidad de Oviedo

Creación de subconjuntos de grafos de conocimiento mediante Shape Expressions

Jose Emilio Labra Gayo

WESO

Razones para elección del tema

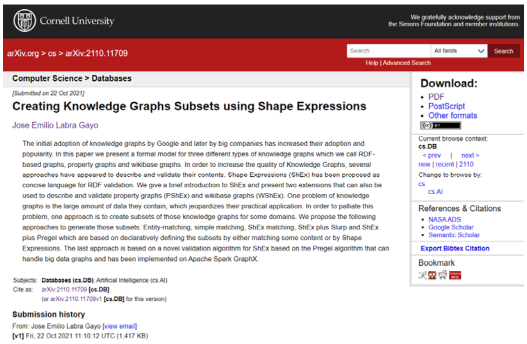
- Continúa investigación sobre *Shape Expressions* y sus aplicaciones
- Resolver problema práctico, propuesto por la comunidad
- Participación en biohackathones con esa temática
  - Virtual Biohackathon 2020
  - SWAT4(HC)LS Hackathon 2021
- Adopción de *Shape Expressions* por Wikidata
- Contrato "*Robustifying Scholia*", Univ. Virginia, *Sloan Foundation*
- Interés de Andra Waagmeester y proyecto GeneWiki
- Parte de entregable proyecto ANGLIRU
- Avance en la disciplina: Mejorar/facilitar consumo de datos semánticos

Principales resultados del artículo

- Definición formal de grafos RDF, *property graphs* y *wikbase graphs*
- Extensión de ShEx para *property graphs* (PShEx)
- Extensión de ShEx para *wikibase graphs* (WShEx)
- Definición de 5 técnicas de generación de subconjuntos
  - Entity-matching
  - Simple matching
  - ShEx-based matching
  - Shex+Slurp
  - ShEx+Pregel
- Definición e implementación algoritmo validación a gran escala sobre Spark GraphX

Formato del trabajo

- Artículo científico en inglés
  - La convocatoria no indica requisitos de idioma
  - El inglés permite su posterior presentación en revista/congreso
- Publicado en abierto como preprint en Arxiv
  - https://arxiv.org/abs/2110.11709
- Objetivos a corto plazo:
  - Optimizar implementaciones
  - Terminar experimentos
  - Enviar a publicación



Cornell University

We gratefully acknowledge support from the Simons Foundation and member institutions.

arXiv.org > cs > arXiv:2110.11709

Search... All fields Search

Help | Advanced Search

Computer Science > Databases

[Submitted on 22 Oct 2021]

Creating Knowledge Graphs Subsets using Shape Expressions

Jose Emilio Labra Gayo

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.

Subjects: **Databases (cs.DB)**; Artificial Intelligence (cs.AI)
Cite as: arXiv:2110.11709 **[cs.DB]**
(or arXiv:2110.11709v1 **[cs.DB]** for this version)

Submission history

From: Jose Emilio Labra Gayo [view email]
[v1] Fri, 22 Oct 2021 11:10:12 UTC (1,417 KB)

Download:
- PDF
- PostScript
- Other formats

Current browse context:
cs.DB
< prev | next >
new | recent | 2110
Change to browse by:
cs
cs.AI

References & Citations
- NASA ADS
- Google Scholar
- Semantic Scholar

Export Bibtex Citation

Bookmark

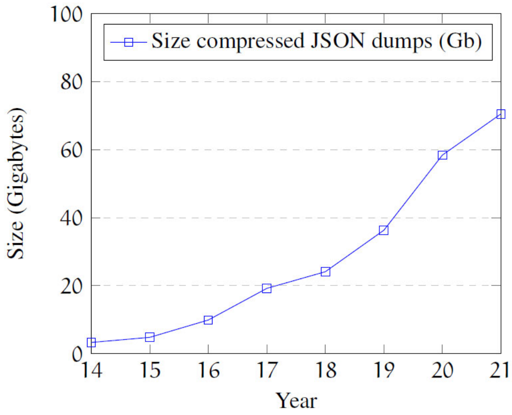
Creating Knowledge Graphs subsets using Shape Expressions

Jose Emilio Labra Gayo

WESO

Motivation and main idea (1)

- Since its proposal in 2012, KGs are very successful
- Different kinds of KGs
  - Free/open and collaborative like Wikidata
  - Enterprise/proprietary and owned by companies: Google, Amazon, eBay, Facebook,Microsoft,...
- Size of their contents is continually growing
- Problem: KGs victims of their own success?
- Example: size of Wikidata dumps almost doubling
  - 30GB (2014 - uncompressed)
  - 1.256GB (2021 - uncompressed)
- Difficult for produce/consume/handle KGs



Idea 1: create subset of KGs for some domain

Motivation and main idea (2)

- Shape Expressions: language to describe and validate RDF
  - Concise and human-friendly
  - Automatically procesable
- Several tools already exist around ShEx
- Target audience: domain data experts
- Already adopted by Wikidata to validate RDF serialization of Wikibase entities

```
<Researcher> {
  :name        xsd:string ;
  :birthDate   xsd:date ? ;
  :birthPlace  @<Place> ? ;
  :knows       @<Researcher> *
}
<Place> {
  :name        xsd:string ;
  :country     @<Country>
}
<Country> {
  :name        xsd:string ;
}
```

Example of a ShEx schema

Idea 2: use Shape expressions to define those subsets

Structure of the presentation

3 main types of Knowledge graphs:
- RDF graphs
- Property graphs
- Wikibase graphs

ShEx for those types of Knowledge Graphs:
- ShEx
- PShEx
- WShEx

Review and propose approaches to generate subsets:
- Entity matching
- Simple matching
- ShEx−based matching
- ShEx+Slurp
- ShEx+Pregel

RDF graphs

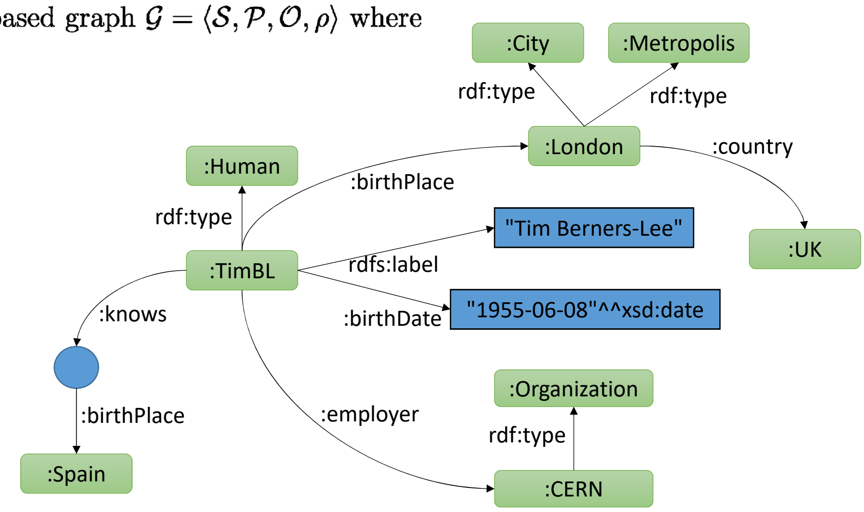
Given a set of IRIs $\mathcal{I}$, a set of blank nodes $\mathcal{B}$ and a set of literals *Lit* an *RDF graph* is a triple based graph $\mathcal{G} = \langle \mathcal{S}, \mathcal{P}, \mathcal{O}, \rho \rangle$ where

$\mathcal{S} = \mathcal{I} \cup \mathcal{B}$,

$\mathcal{P} = \mathcal{I}$,

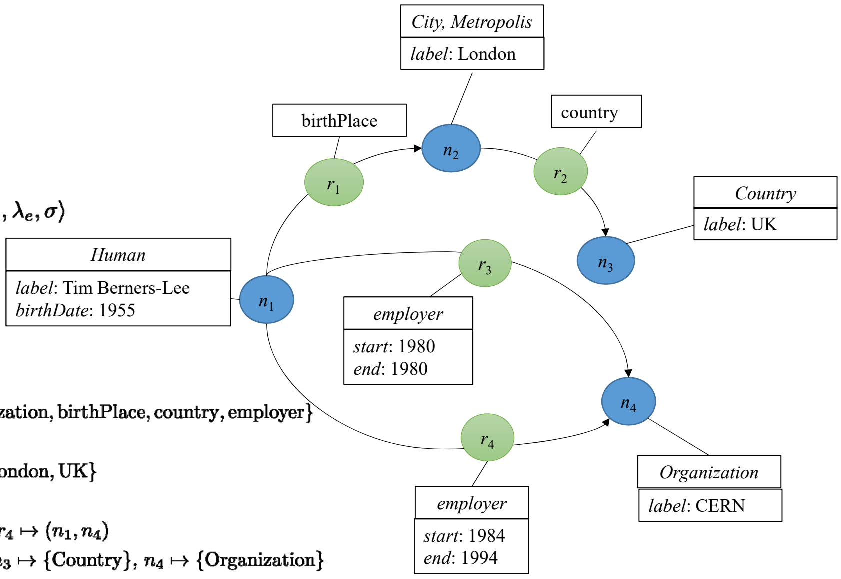
$\mathcal{O} = \mathcal{I} \cup \mathcal{B} \cup Lit$

$\rho \subseteq \mathcal{S} \times \mathcal{P} \times \mathcal{O}$



Property graphs

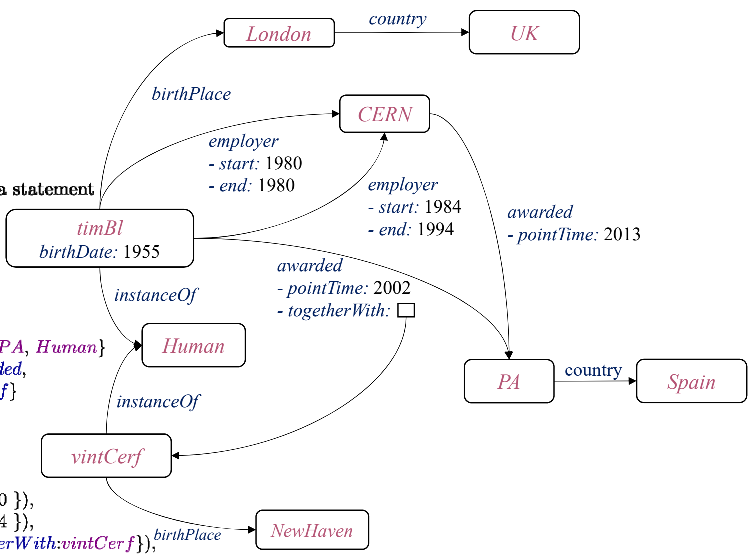
Given a set of types $\mathcal{T}$,
a set of properties $\mathcal{P}$,
and a set of values $\mathcal{V}$,
a *property graph* $\mathcal{G}$ is a tuple $\langle \mathcal{N}, \mathcal{E}, \rho, \lambda_n, \lambda_e, \sigma \rangle$
where $\mathcal{N} \cap \mathcal{E} = \emptyset$,
$\rho : \mathcal{E} \mapsto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



$\mathcal{T} = \{\text{Human, City, Metropolis, Country, Organization, birthPlace, country, employer}\}$
$\mathcal{P} = \{\text{label, birthDate, start, end}\}$
$\mathcal{V} = \{\text{Tim Berners-Lee, 1955, 1980, 1984, 1994, London, UK}\}$
$\mathcal{N} = \{n_1, n_2, n_3, n_4\} \qquad \mathcal{E} = \{r_1, r_2, r_3, r_4\}$
$\rho = r_1 \mapsto (n_1, n_2), r_2 \mapsto (n_2, n_3), r_3 \mapsto (n_1, n_4), r_4 \mapsto (n_1, n_4)$
$\lambda_n = n_1 \mapsto \{\text{Human}\},\ n_2 \mapsto \{\text{City,Metropolis}\},\ n_3 \mapsto \{\text{Country}\},\ n_4 \mapsto \{\text{Organization}\}$
$\lambda_e = r_1 \mapsto \text{birthPlace}, r_2 \mapsto \text{country}, r_3 \mapsto \text{employer}, r_4 \mapsto \text{employer}$
$\sigma = (n_1, \mathit{label}) \mapsto \text{Tim Berners-Lee},\ (n_1, \mathit{birthDate}) \mapsto 1955$
$(n_2, \text{label}) \mapsto \text{London}\},\ (n_3, \text{label}) \mapsto UK,\ (n_4, \text{label}) \mapsto \text{CERN}$
$(r_3, \text{start}) \mapsto 1980,\ (r_3, \text{end}) \mapsto 1980,\ (r_4, \text{start}) \mapsto 1984,\ (r_4, \text{end}) \mapsto 1994$

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
$\rho \subseteq \mathcal{E} \times \mathcal{P} \times \mathcal{V} \times \mathit{FinSet}(\mathcal{P} \times \mathcal{V})$ where
$\mathcal{E} = \mathcal{Q} \cup \mathcal{P}$ is the set of entities which can be subjects of a statement
$\mathcal{V} = \mathcal{E} \cup \mathcal{D}$ is the set of possible values of a property.

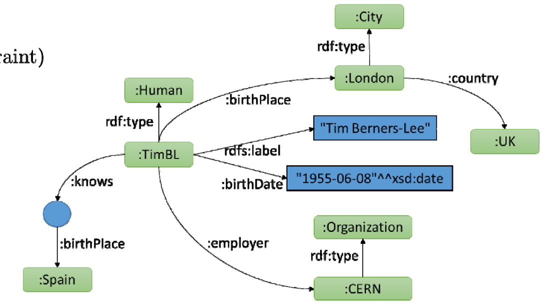


$\mathcal{Q} = \{\ \mathit{timBl}, \mathit{vintCerf}, \mathit{London}, \mathit{CERN}, \mathit{UK}, \mathit{Spain}, \mathit{PA}, \mathit{Human}\}$
$\mathcal{P} = \{\ \mathit{birthDate}, \mathit{birthPlace}, \mathit{country}, \mathit{employer}, \mathit{awarded}, \mathit{start}, \mathit{end}, \mathit{pointTime},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

ShEx for RDF

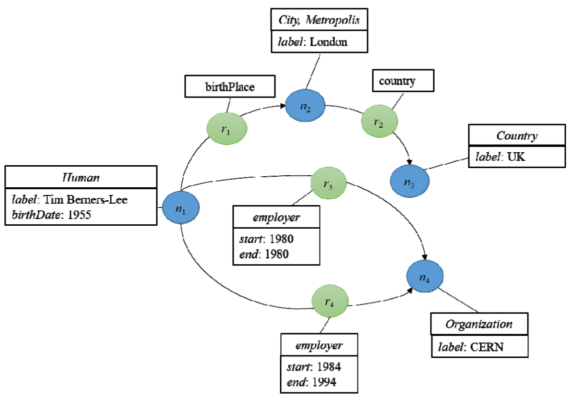
A ShEx Schema is a tuple $\langle \mathcal{L}, \delta \rangle$ where
$\mathcal{L}$ set of shape labels
$\delta : \mathcal{L} \to se$

se	$::=$	cond	Basic boolean condition on nodes (node constraint)
	$\mid$	s	Shape
	$\mid$	se_1 AND se_2	Conjunction
	$\mid$	se_1 OR se_2	Disjunction
	$\mid$	NOT se	Negation
	$\mid$	$@l$	Shape label reference for $l \in \mathcal{L}$
s	$::=$	CLOSED $\{te\}$	Closed shape
	$\mid$	$\{te\}$	Open shape
te	$::=$	$te_1 ; te_2$	Each of te_1 and te_2
	$\mid$	$te_1 \mid te_2$	Some of te_1 or te_2
	$\mid$	$te*$	Zero or more te
	$\mid$	ϵ	Empty triple expression
	$\mid$	$\_ \xrightarrow{p} @l$	Triple constraint with predicate p



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

PShEx for property graphs



A PShEx Schema is a tuple $\langle \mathcal{L}, \delta \rangle$ where
$\mathcal{L}$ set of shape labels
$\delta : \mathcal{L} \to se$

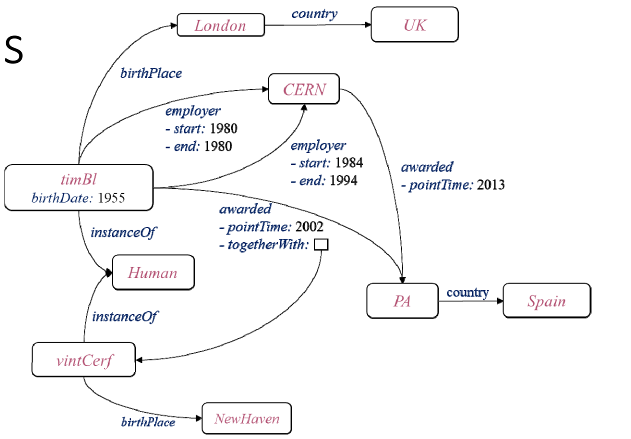
se	$::=$	$cond_{t_s}$	Basic boolean condition on set of types $t_s \subseteq \mathcal{T}$
	$\mid$	s	Shape
	$\mid$	se_1 AND se_2	Conjunction
	$\mid$	se_1 OR se_2	Disjunction
	$\mid$	NOT se	Negation
	$\mid$	$@l$	Shape label reference for $l \in \mathcal{L}$
	$\mid$	qs	Qualifiers of that node
s	$::=$	CLOSED $\{te\}$	Closed shape
	$\mid$	$\{te\}$	Open shape
te	$::=$	$te_1 ; te_2$	Each of te_1 and te_2
	$\mid$	$te_1 \mid te_2$	Some of te_1 or te_2
	$\mid$	$te*$	Zero or more te
	$\mid$	$\_ \xrightarrow{p} @l \; qs$	Triple constraint with property type p whose nodes satisfy the shape l and qualifiers qs
qs	$::=$	$\lfloor ps \rfloor$	Open qualifier specifiers ps
	$\mid$	$\lceil ps \rceil$	Closed qualifier specifiers ps
ps	$::=$	ps_1 , ps_2	Each of ps_1 and ps_2
	$\mid$	$ps_1 \mid ps_2$	OneOf of ps_1 or ps_2
	$\mid$	$ps*$	zero of more ps
	$\mid$	$p : cond_v$	Property p with value conforming to $cond_v$
			$cond_{v_s}$ is a boolean condition on sets of values $v_s \subseteq \mathcal{V}$

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

WShEx for Wikibase graphs

A WShEx Schema is a tuple $\langle \mathcal{L}, \delta \rangle$ where
$\mathcal{L}$ set of shape labels
$\delta : \mathcal{L} \rightarrow se$

se	::=	cond	Basic boolean condition on nodes (node constraint)
	|	s	Shape
	|	se_1 AND se_2	Conjunction
	|	se_1 OR se_2	Disjunction
	|	NOT se	Negation
	|	$@l$	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
	|	$te_1 \mid te_2$	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
	|	$ps \mid ps$	*OneOf* property specifiers
	|	ps*	zero of more property specifiers
	|	ϵ	Empty property specifier
	|	p:@l	Property p with value conforming to shape l



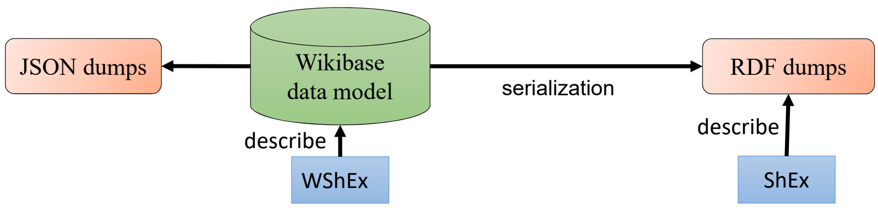
$$
\begin{array}{lcl}
\mathcal{L} & = & \{ Person, Place, Country, Organization, Date, Award \} \\
\delta(Person) & = & \{ \ \_ \xrightarrow{birthDate} @Date;\ \_ \xrightarrow{birthPlace} @Place; \\
 & & \ \_ \xrightarrow{employer} @Organization\ \lfloor start : @Date,\ end : @Date \rfloor * \\
 & &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

What's the role of WShEx

WShEx schemas describe Wikibase data model

- Closer to JSON dumps

ShEx (entity schemas) describe RDF serializacion of Wikibase data model

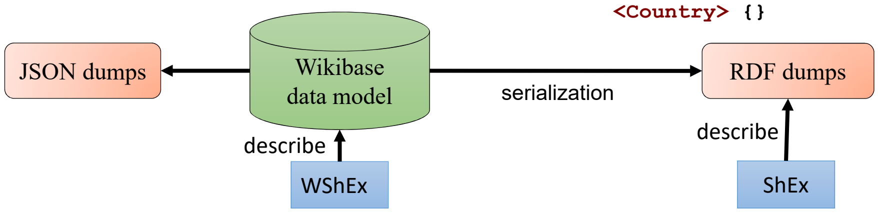


WShEx

```
<Researcher> {
 <birthPlace>        @<Place> ;
 <awarded>           @<Award> {{
    <togetherWith> @<Researcher> *
  }} *
}
<Place> {
 <country>           @<Country>
}
<Country> {}
```

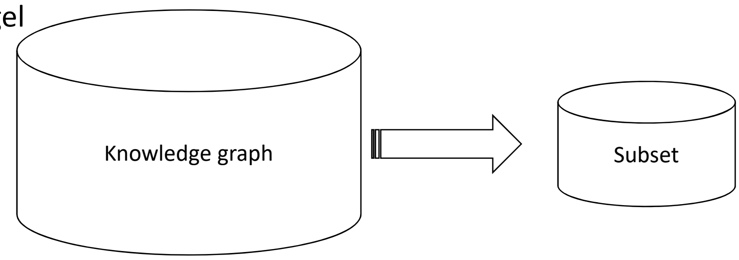
ShEx

```
<Researcher> {
 wdt:birthPlace @<Place> ;
 p:birthPlace    {
    ps:birthPlace    @<Place>
 } ;
 wdt:awarded          @<Awarded> ;
 p:awarded {
   ps:awarded         @<Awarded> ;
   pq:togetherWith @<Researcher> *;
 } *
}
<Place> {
 wdt:country          @<Country> ;
 p:country {
  ps:country          @<Country> }
}
<Country> {}
```



Knowledge graphs subsets

Different approaches
- Entity-matching
- Simple-matching
- ShEx-based matching
- ShEx+Slurp
- ShEx+Pregel

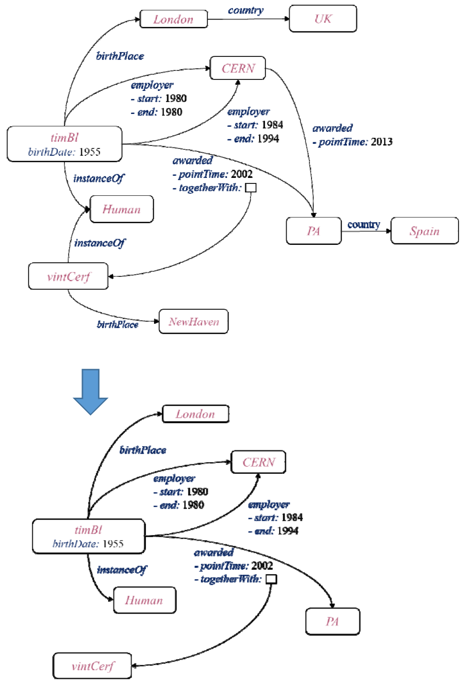


Entity-matching

Directly matches an entity like an ítem, property, etc.

Example: $\mathcal{Q}_s = \{timBl\}$

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



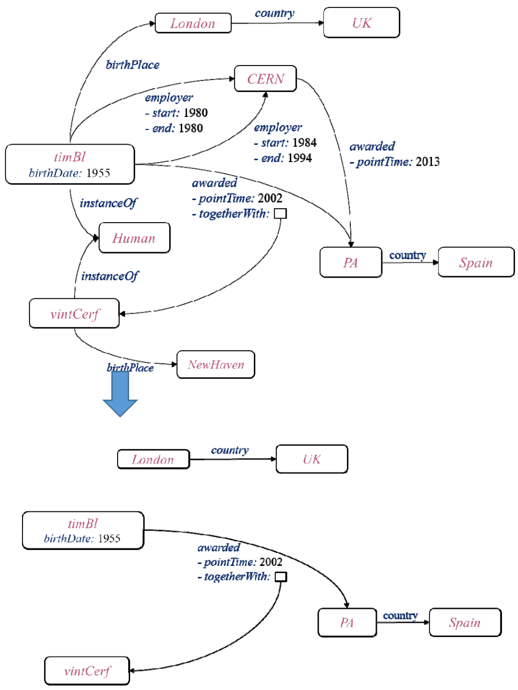
Simple matching

Defines a matching language:

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

Example: $M_s = \{\text{property(country)}, \text{qualifiedProp(togetherWith)}\}$

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



ShEx-based matching

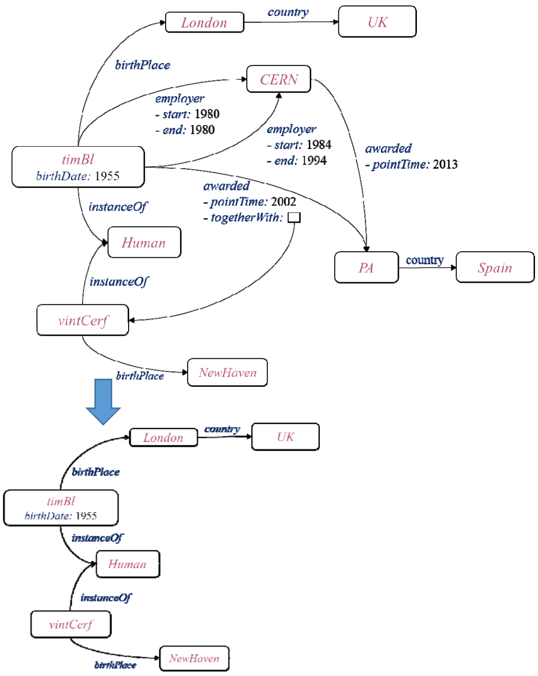
Uses ShEx definitions to match entities

Replaces shape references by *true*

Doesn't require graph traversal

$$
\begin{aligned}
\mathcal{L} &= \{\ Researcher, Place, Country, Date, Human\} \\
\delta(Researcher) &= \{\ \llcorner \xrightarrow{instanceOf} \text{@Human}; \\
&\quad\ \llcorner \xrightarrow{birthDate} \text{@Date?}; \\
&\quad\ \llcorner \xrightarrow{birthPlace} \text{@Place} \\
&\ \ \} \\
\delta(Place) &= \{\ \llcorner \xrightarrow{country} \text{@Country}\} \\
\delta(Date) &= \ \in xsd:date \\
\delta(Human) &= \ \in \{Human\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{S} = \{\ & (timBl, instanceOf, Human], \{\}), \\
& (timBl, birthDate, 1955, \{\}), \\
& (timBl, birthPlace, London, \{\}), \\
& (London, country, UK, \{\}), \\
& (vintCerf, instanceOf, Human], \{\}) \\
& (vintCerf, birthPlace, newHaven, \{\}) \\
\}
\end{aligned}
$$



ShEx-based matching

Can be implemented replacing shape references by *true*

$$
\begin{aligned}
\delta(Researcher) &= \{\ \llcorner \xrightarrow{instanceOf} \text{@Human}; \\
&\quad\ \llcorner \xrightarrow{birthDate} \text{@Date?}; \\
&\quad\ \llcorner \xrightarrow{birthPlace} \text{@Place} \\
&\ \ \} \\
\delta(Place) &= \{\ \llcorner \xrightarrow{country} \text{@Country}\} \\
\delta(Date) &= \ \in xsd:date \\
\delta(Human) &= \ \in \{Human\}
\end{aligned}
$$

$$
\begin{aligned}
\delta(Researcher) &= \{\ \llcorner \xrightarrow{instanceOf} \texttt{true}; \\
&\quad\ \llcorner \xrightarrow{birthDate} \texttt{true}?; \\
&\quad\ \llcorner \xrightarrow{birthPlace} \texttt{true} \\
&\ \ \} \\
\delta(Place) &= \{\ \llcorner \xrightarrow{country} \texttt{true}\}
\end{aligned}
$$

ShEx-based matching has been implemented in WDSub

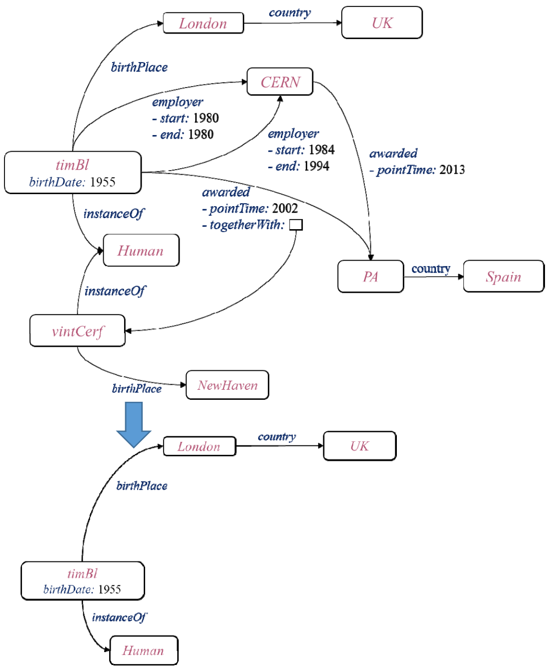
It allows sequential processing of the dump

It can process the latest Wikidata dump in 5h 15'

ShEx+slurp

Slurp = Shex option that collects nodes/triples while validating

Example:



$$
\begin{array}{lcl}
\mathcal{L} & = & \{ \; Researcher, Place, Country, Date, Human\} \\
\delta(Researcher) & = & \{ \; \llcorner \xrightarrow{instanceOf} @\text{Human}; \\
 & & \quad \llcorner \xrightarrow{birthDate} @\text{Date}; \\
 & & \quad \llcorner \xrightarrow{birthPlace} @\text{Place} \\
 & & \} \\
\delta(Place) & = & \{ \; \llcorner \xrightarrow{country} @\text{Country}\} \\
\delta(Country) & = & \{ \; \} \\
\delta(Date) & = & \in xsd : date \\
\delta(Human) & = & \in \{Human\}
\end{array}
$$

ShEx+Slurp

Formal definition generalizes ShEx semantics

New conformance relation:

$$\mathcal{G}, n, \tau \vDash se \rightsquigarrow \mathcal{G}'$$

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

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

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

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

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

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

$$EachOfQs\frac{\mathcal{G}, s, \tau \vdash ps_1 \rightsquigarrow (qs_1, \mathcal{G}_1) \quad \mathcal{G}, s, \tau \vdash ps_2 \rightsquigarrow (qs_2, \mathcal{G}_2)}{\mathcal{G}, s, \tau \vdash ps_1 \,,\, ps_2 \rightsquigarrow (qs_{1 \cup 2}, \mathcal{G}_1 \cup \mathcal{G}_2)}$$

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

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

$$Star_1\frac{}{\mathcal{G}, \emptyset, \tau \Vdash te* \rightsquigarrow \emptyset} \qquad StarQs_1\frac{}{\mathcal{G}, \emptyset, \tau \vdash ps* \rightsquigarrow (\{\}, \emptyset)}$$

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

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

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

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

ShEx+Slurp

- Implemented by shex.js and pyshex
  - Only for ShEx, not for PShEx or WShEx
- It does graph traversal while validating
  - Difficult to use for processing Wikidata dumps
- Subsets include only valid data according to the ShEx
- Problems:
  - Difficult to run behind SPARQL endpoint
  - Naïve implementation generates too many requests
  - Optimizations to improve number of SPARQL queries can generate timeouts

ShEx+Pregel

- Large scale validation
- Based on Pregel algorithm
  - Developed by Google 2010
- Based on Bulk Synchronous Parallel model
  - Supersteps = units of parallel computation
  - Communication through messages
- Implemented using Apache Spark GraphX library

Think like a vertex

Vertex-centric approach

Each vertice has an id and a state

All vertexes receive an initial message

While messages received do

- Run vProg to update vertex state according to messages received
- Run sendMsg for each triplet to generate messages
- Merge messages by Vertex

GraphX

Graph API from Apache Spark

- RDD (Resilient distributed datasets)
- MapReduce in memory

$Graph[\mathcal{V},\mathcal{E}]$ = graph with vertices $\mathcal{V}$ and edges $\mathcal{E}$

- Vertices are represented as $RDD[(Id,\mathcal{V})]$ where $Id = Long$
- Edges are represented as $RDD[(Id,Id,\mathcal{E})]$
- *triplets* view represents edges as collections $RDD[(\mathcal{V},\mathcal{E},\mathcal{V})]$.

Several built-in operators:

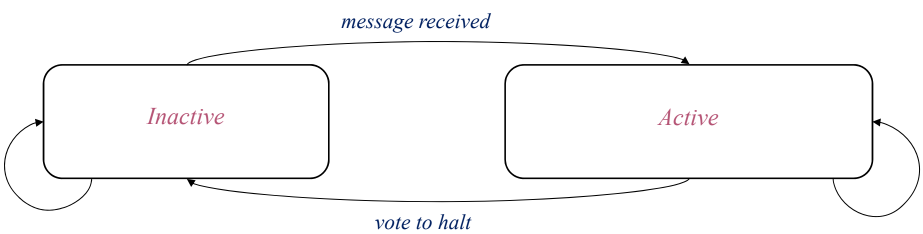
- `mapVertices`(`g: Graph`$[\mathcal{V},\mathcal{E}]$, `f: `$(\texttt{Id},\mathcal{V}) \rightarrow \mathcal{V}$): `Graph`$[\mathcal{V},\mathcal{E}]$
- `mapReduceTriples`(`g:Graph`$[\mathcal{V},\mathcal{E}]$, `m: `$(\mathcal{V},\mathcal{E},\mathcal{V}) \rightarrow [(\texttt{Id},\mathcal{M})]$, `r:`$(\mathcal{M},\mathcal{M}) \rightarrow \mathcal{M}$):`RDD`$[(\mathrm{Id},\mathcal{M})]$
- `joinVertices`(`g:Graph`$[\mathcal{V},\mathcal{E}]$, `msgs:RDD`$[(\texttt{Id}, \mathcal{M})]$, `f:`$(\texttt{Id}, \mathcal{V},\mathcal{M}) \rightarrow \mathcal{V}$): `Graph`$[\mathcal{V},\mathcal{E}]$
- pregel (see next slide)
- ...

GraphX Pregel pseudo-code

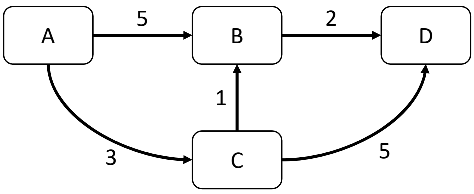
Algorithm 1: Pregel algorithm pseudocode as implemented in GraphX

```
Input parameters:
    g: Graph[V,E]
    initialMsg: M
    vProg: (Id,V,M)→V
    sendMsg: Triplet→[(Id,M)]
    mergeMsg: (M,M)→M
Output: g:Graph[V,E]
g = mapVertices(g,λ(id,v)→vProg(id,v,initialMsg))
msgs = mapReduceTriples(g,sendMsg,mergeMsg)
while size(msgs)> 0 do
    g = joinVertices(g,msgs,vProg)
    msgs = mapReduceTriples(g,sendMsg,mergeMsg)
return g
```

Pregel's vertex state diagram



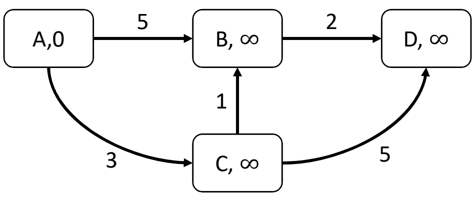
Shortest path to a node




```
val initialGraph = graph.mapVertices(
 (id, _) => if (id == source) 0 else ∞
)

initialMsg = ∞

def vProg(id: VertexId, dist: Int, msg: Int) =
  min(dist, msg)

def sendMsg(t: Triplet) = {
  if (t.srcAttr + t.attr < t.dstAttr) {
    (t.dstId, t.srcAttr + t.attr)
  }
  else { }

def mergMsg(a: Int, b: Int) = min(a, b)

initialGraph.pregel(initialMsg)
                   (vProg,sendMsg,mergeMsg)
```

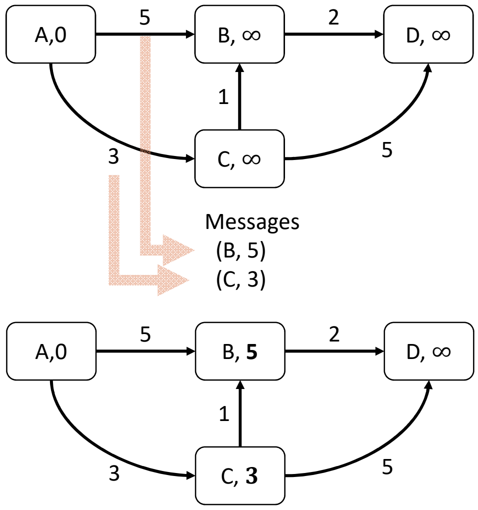
Shortest path to a node

initialGraph



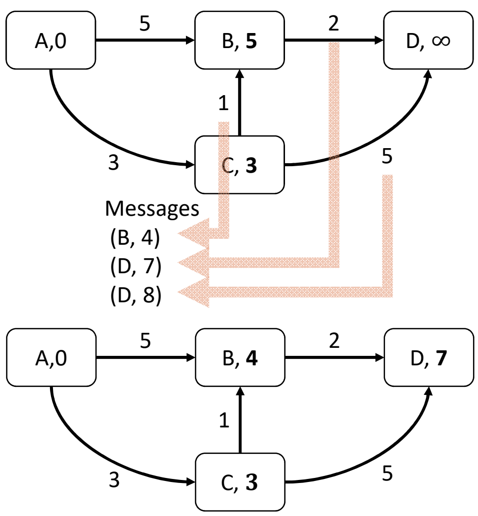

```
val initialGraph = graph.mapVertices(
 (id, _) => if (id == source) 0 else ∞
)

initialMsg = ∞

def vProg(id: VertexId, dist: Int, msg: Int) =
  min(dist, msg)

def sendMsg(t: Triplet) = {
  if (t.srcAttr + t.attr < t.dstAttr) {
    (t.dstId, t.srcAttr + t.attr)
  }
  else { }

def mergMsg(a: Int, b: Int) = min(a, b)

initialGraph.pregel(initialMsg)
                   (vProg,sendMsg,mergeMsg)
```

Shortest path to a node



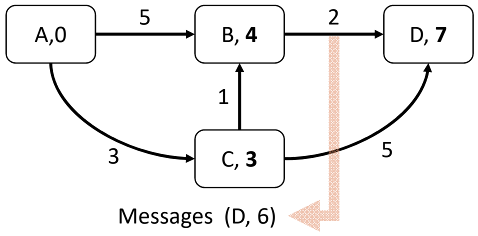

```
val initialGraph = graph.mapVertices(
 (id, _) => if (id == source) 0 else ∞
)

initialMsg = ∞

def vProg(id: VertexId, dist: Int, msg: Int) =
  min(dist, msg)

def sendMsg(t: Triplet) = {
  if (t.srcAttr + t.attr < t.dstAttr) {
    (t.dstId, t.srcAttr + t.attr)
  }
  else { }

def mergMsg(a: Int, b: Int) = min(a, b)

initialGraph.pregel(initialMsg)
                   (vProg,sendMsg,mergeMsg)
```

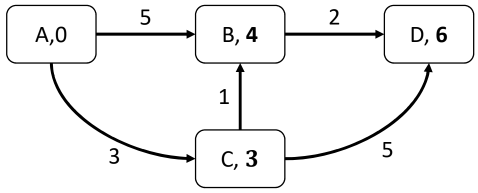
Shortest path to a node




```
val initialGraph = graph.mapVertices(
 (id, _) => if (id == source) 0 else ∞
)

initialMsg = ∞

def vProg(id: VertexId, dist: Int, msg: Int) =
  min(dist, msg)

def sendMsg(t: Triplet) = {
  if (t.srcAttr + t.attr < t.dstAttr) {
    (t.dstId, t.srcAttr + t.attr)
  }
  else { }

def mergMsg(a: Int, b: Int) = min(a, b)

initialGraph.pregel(initialMsg)
                   (vProg,sendMsg,mergeMsg)
```

Shortest path to a node



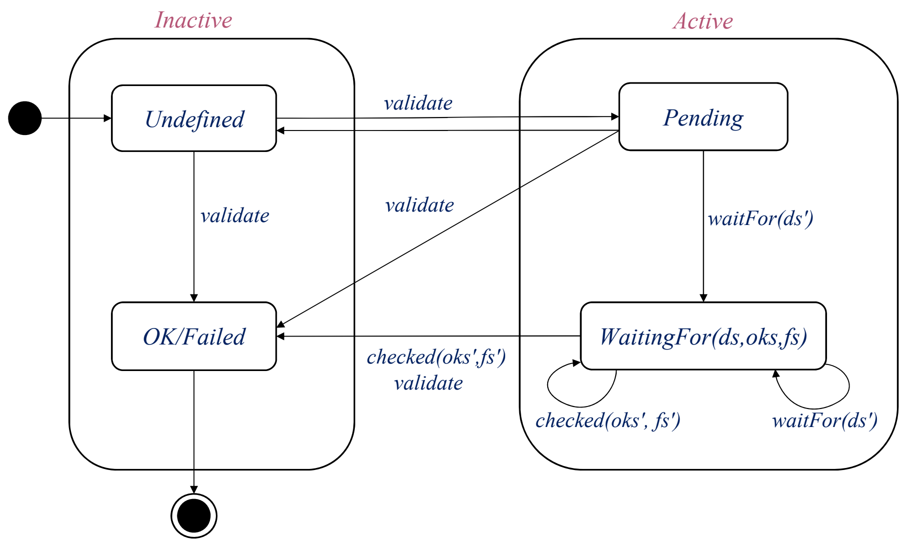



```
val initialGraph = graph.mapVertices(
 (id, _) => if (id == source) 0 else ∞
)

initialMsg = ∞

def vProg(id: VertexId, dist: Int, msg: Int) =
  min(dist, msg)

def sendMsg(t: Triplet) = {
  if (t.srcAttr + t.attr < t.dstAttr) {
    (t.dstId, t.srcAttr + t.attr)
  }
  else { }

def mergMsg(a: Int, b: Int) = min(a, b)

initialGraph.pregel(initialMsg)
                   (vProg,sendMsg,mergeMsg)
```

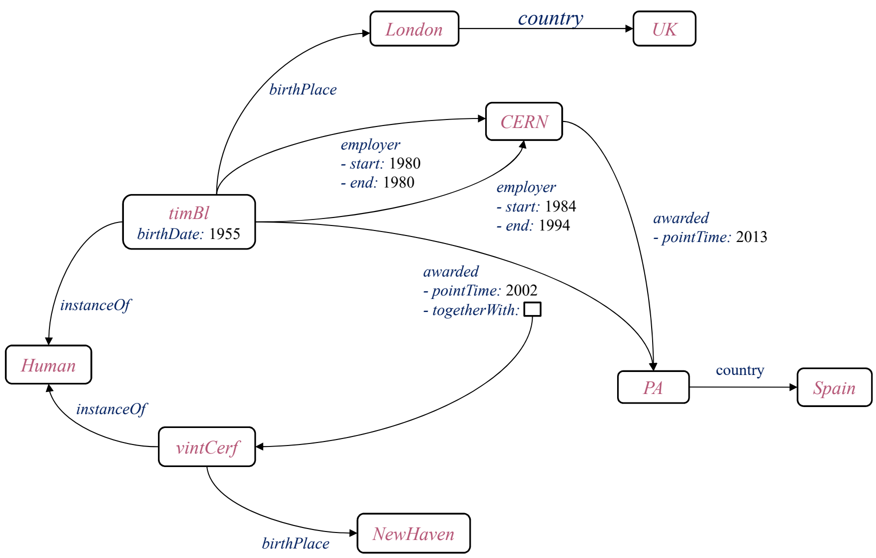
Idea: Validate graphs using Pregel?

- Associate each node with a status map
  - Status map contains information about node conformance to shapes
  - Messages =
    - Requests to validate
    - Request to wait for neighbours
    - Information about validated/failed neighbours
  - Initial message = request to validate `start` shape
  - In each iteration we collect information from neighbours conformance
  - We provide 3 parameters
    - checkLocal: tries to check conformance of a node locally or return pending neighbours
    - checkNeighs: checks neighbours with regular expression defined by a shape
    - tripleConstraints: return the triple constraints associated with a shape label

State diagram



Trace of PSchema



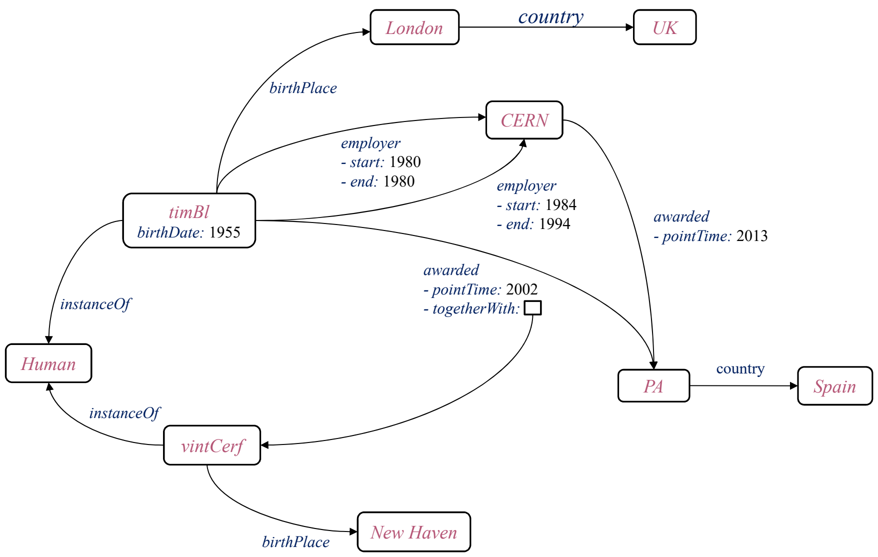
WShEx Schema

```
start = <Researcher> {
  instanceOf    @<Human>  ;
  birthDate     Date? ;
  birthPlace    @<Place>  ;
}
<Human> [ Human ]
<Place> {
  country @<Country>
}
<Country> { }
```

Trace of PSchema

WShEx Schema

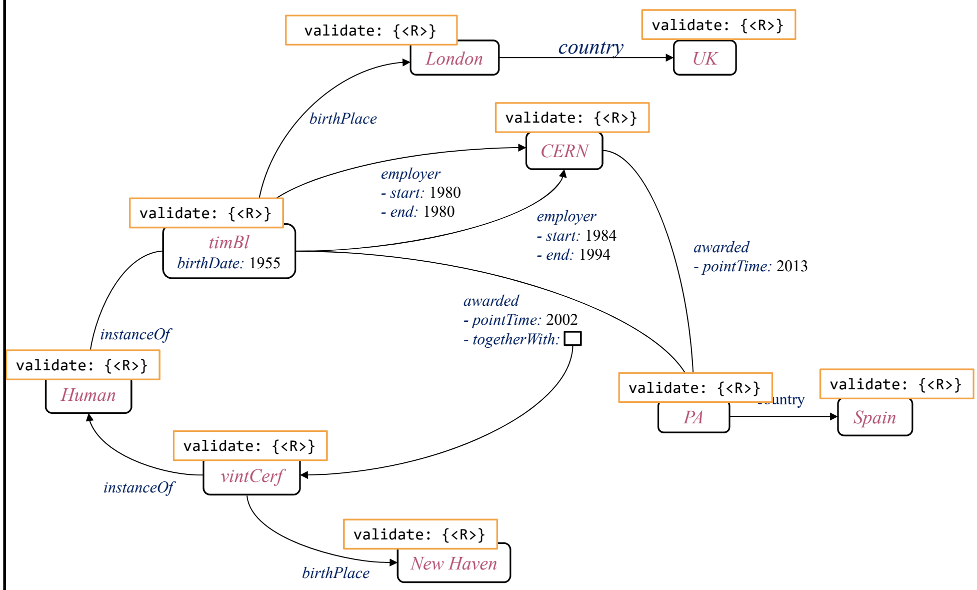
```
start = <R> {
  instanceOf   @<H>  ;
  birthDate    Date? ;
  birthPlace   @<P>  ;
}
<H> [ Human ]
<P> {
  country @<C>
}
<C> { }
```



Superstep 0. All nodes asked to validate start = <R>

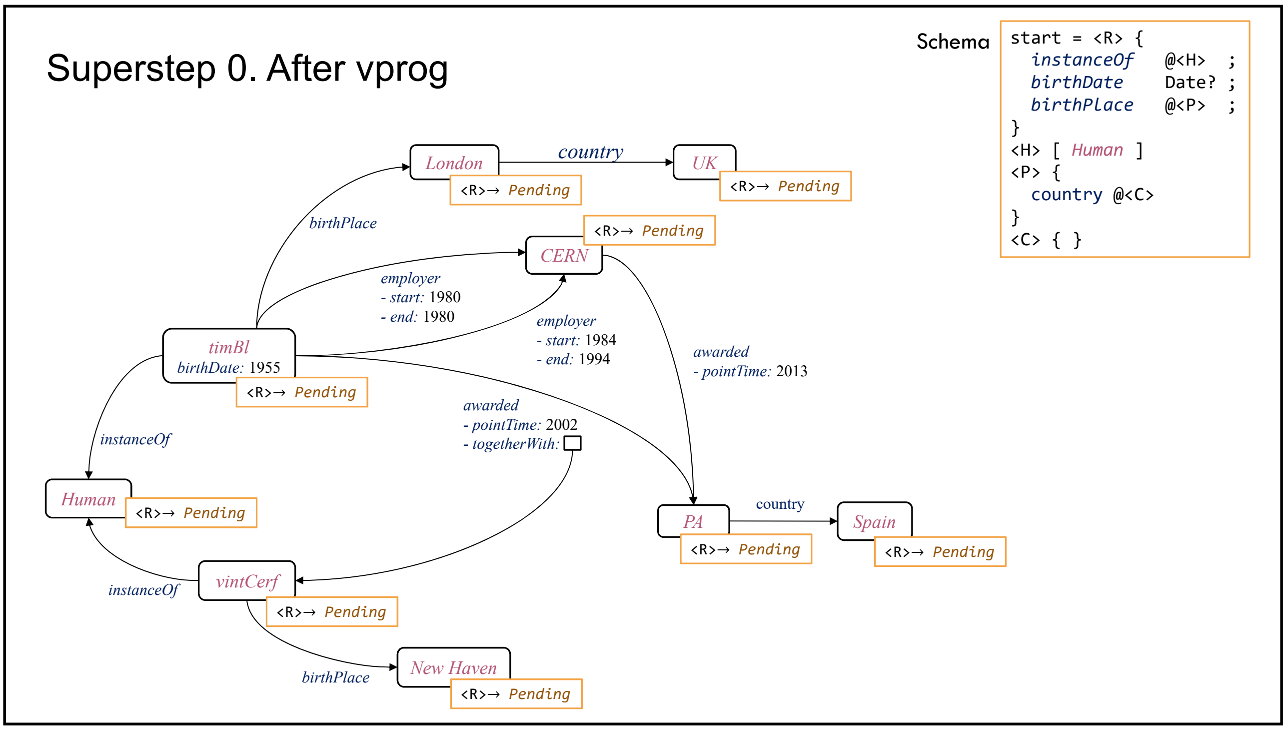
Schema

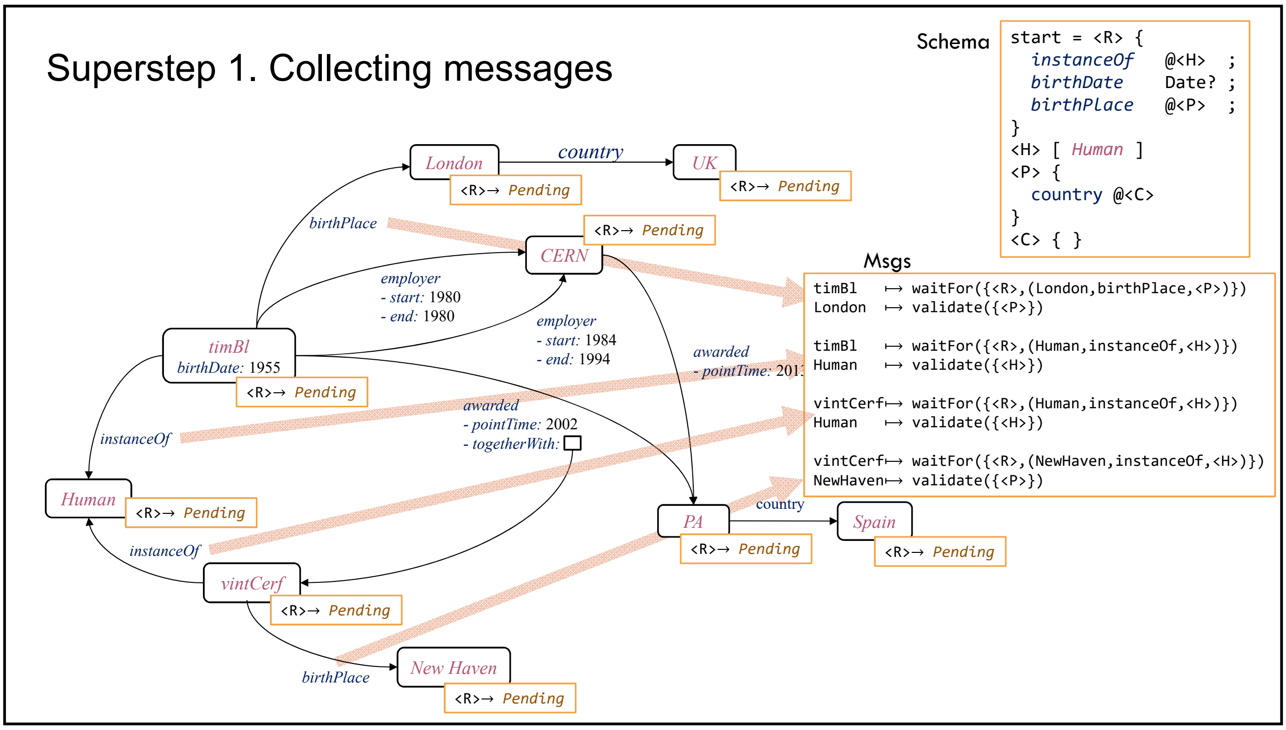
```
start = <R> {
  instanceOf   @<H>  ;
  birthDate    Date? ;
  birthPlace   @<P>  ;
}
<H> [ Human ]
<P> {
  country @<C>
}
<C> { }
```



Messages at superstep 0

```
(timBl, validate: <R>),
(London, validate: <R>),
(UK, validate: <R>),
(CERN, validate: <R>),
(PA, validate: <R>),
(Spain, validate: <R>),
(Human, validate: <R>),
(vintCerf, validate: <R>)
(newHaven, validate: <R>)
```


Superstep 0. After vprog
Schema
start = <R> {
  instanceOf   @<H> ;
  birthDate   Date? ;
  birthPlace   @<P> ;
}
<H> [Human]
<P> {
  country @<C>
}
<C> { }
London
country
UK
<R>→ Pending
birthPlace
CERN
employer
- start: 1980
- end: 1980
employer
- start: 1984
- end: 1994
awarded
- pointTime: 2013
timBl
birthDate: 1955
awarded
- pointTime: 2002
- togetherWith:
instanceOf
Human
PA
country
Spain
vintCerf
instanceOf
New Haven
birthPlace


Superstep 1. Collecting messages
Schema
start = <R> {
  instanceOf   @<H> ;
  birthDate   Date? ;
  birthPlace   @<P> ;
}
<H> [Human]
<P> {
  country @<C>
}
<C> { }
Msgs
timBl   ↦ waitFor({<R>,(London,birthPlace,<P>)})
London   ↦ validate({<P>})
timBl   ↦ waitFor({<R>,(Human,instanceOf,<H>)})
Human   ↦ validate({<H>})
vintCerf↦ waitFor({<R>,(Human,instanceOf,<H>)})
Human   ↦ validate({<H>})
vintCerf↦ waitFor({<R>,(NewHaven,instanceOf,<H>)})
NewHaven↦ validate({<P>})
London
country
UK
<R>→ Pending
birthPlace
CERN
employer
- start: 1980
- end: 1980
employer
- start: 1984
- end: 1994
awarded
timBl
birthDate: 1955
awarded
- pointTime: 2002
- togetherWith:
instanceOf
Human
PA
country
Spain
vintCerf
instanceOf
New Haven
birthPlace

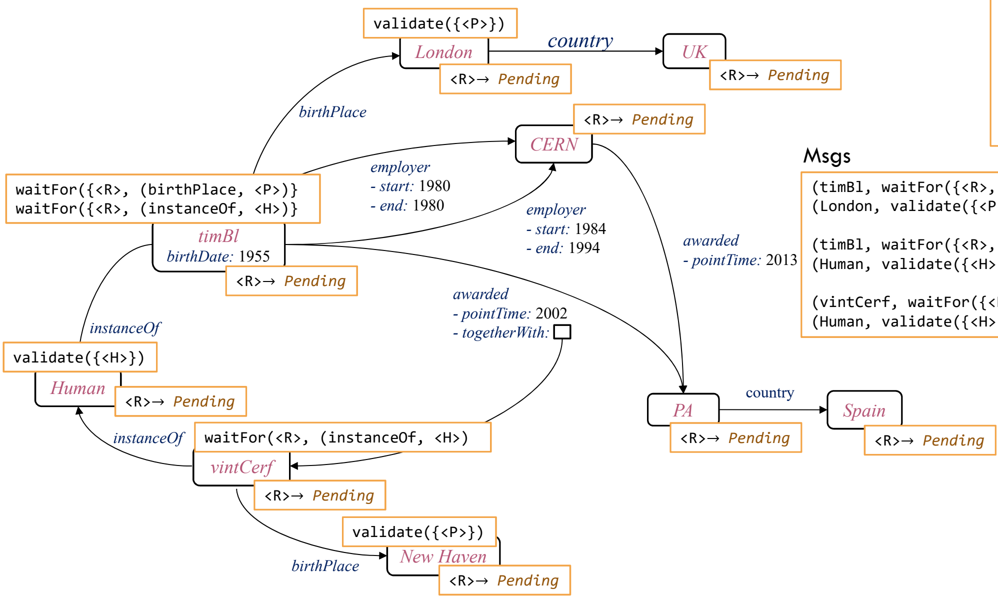
Superstep 1. Collecting messages

Schema

```
start = <R> {
  instanceOf   @<H>  ;
  birthDate    Date? ;
  birthPlace   @<P>  ;
}
<H> [ Human ]
<P> {
  country @<C>
}
<C> { }
```

Msgs

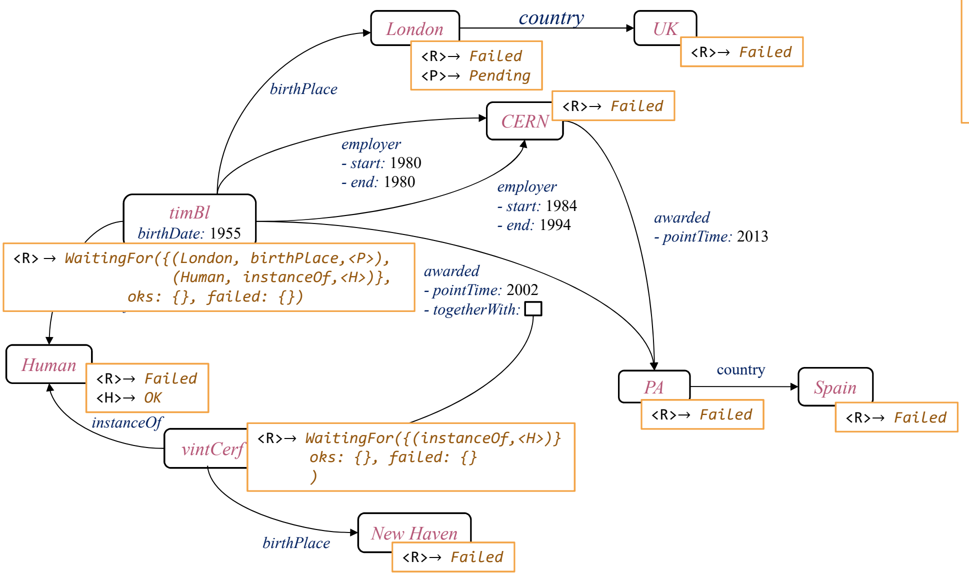
```
(timBl, waitFor({<R>, (London,birthPlace,<P>)})
(London, validate({<P>})

(timBl, waitFor({<R>, (Human,instanceOf,<H>)})
(Human, validate({<H>})

(vintCerf, waitFor({<R>, (Human,instanceOf,<H>)})
(Human, validate({<H>})
```



Superstep 1. After vprog.

Schema

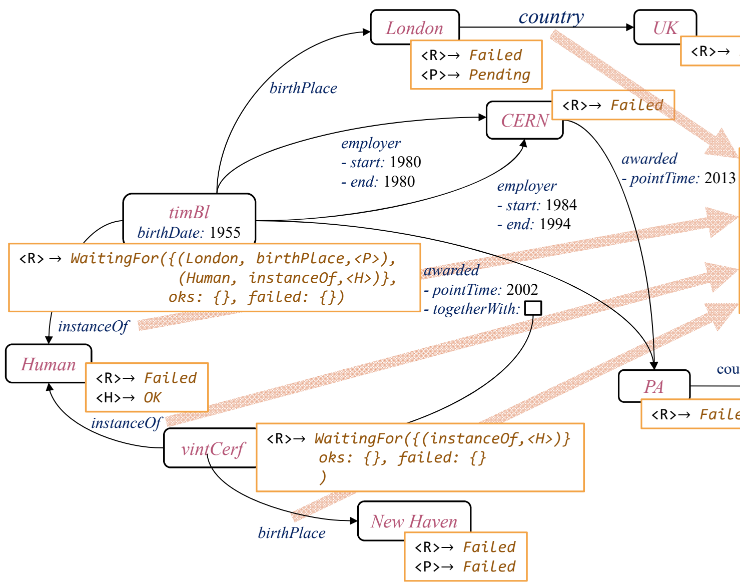
```
start = <R> {
  instanceOf   @<H>  ;
  birthDate    Date? ;
  birthPlace   @<P>  ;
}
<H> [ Human ]
<P> {
  country @<C>
}
<C> { }
```



Superstep 2. Collecting new messages.

Schema

```
start = <R> {
  instanceOf   @<H>  ;
  birthDate    Date? ;
  birthPlace   @<P>  ;
}
<H> [ Human ]
<P> {
  country @<C>
}
<C> { }
```



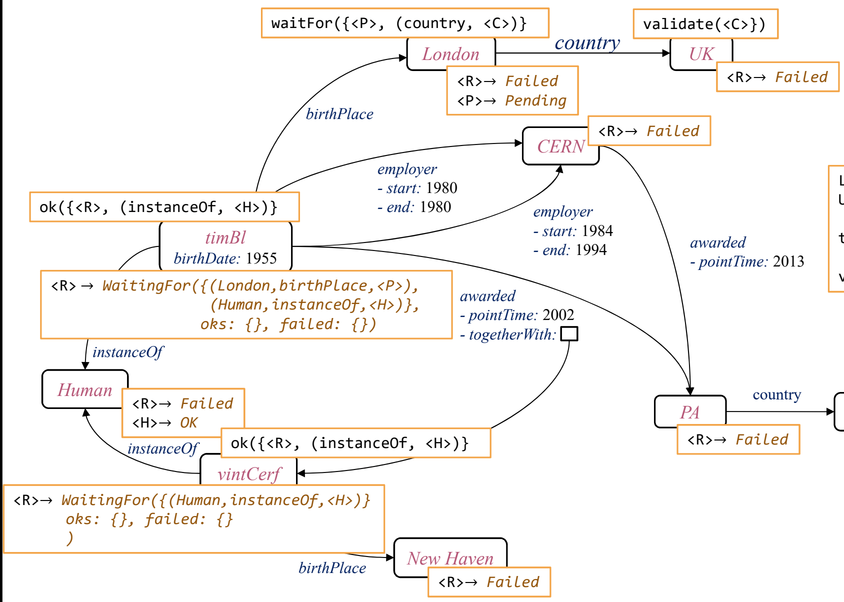
Msgs

```
London    ↦ waitFor({<P>,(London,country,<C>)})
UK        ↦ validate({<C>})

timBl ↦Checked({<R>,(Human,instanceOf,<H>)},{})

vintCerf↦Checked({<R>,(Human,instanceOf,<H>)},{})

vintCerf↦Checked({<R>,{},{(NewHaven,birthPlace,<P>)})
```

Superstep 2. After vprog

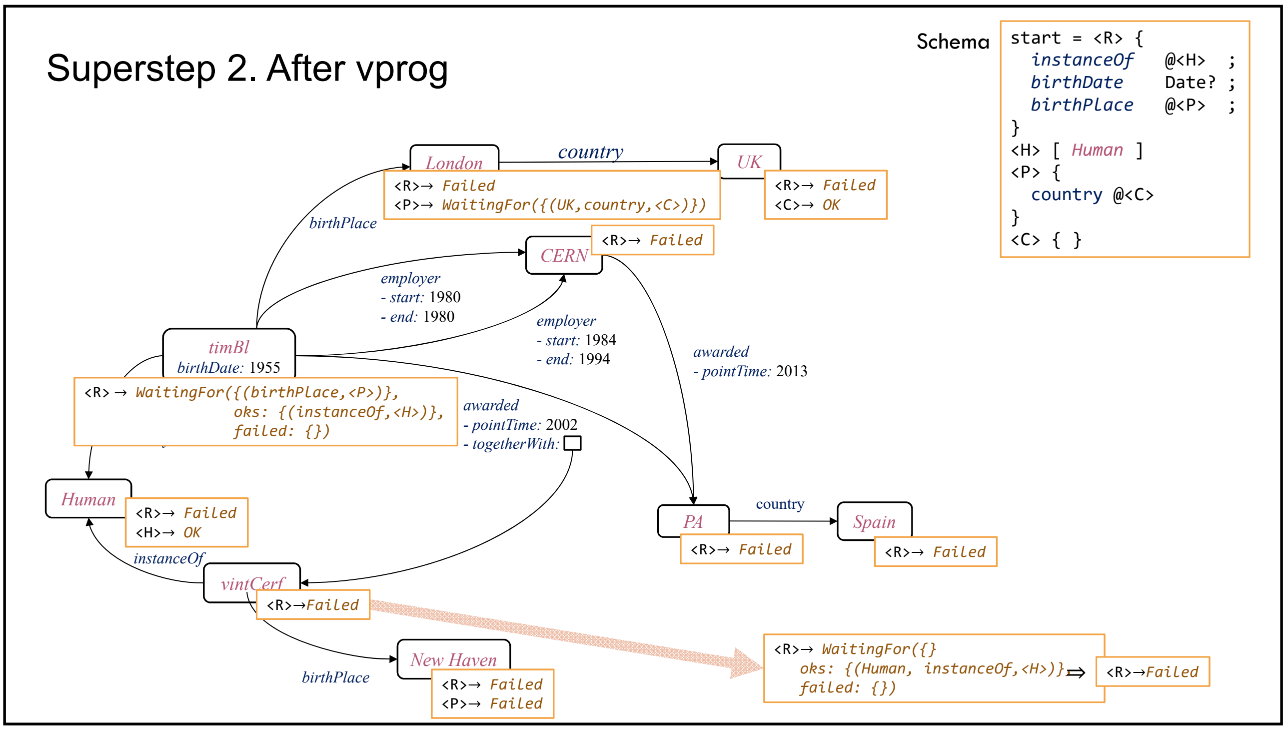
Schema

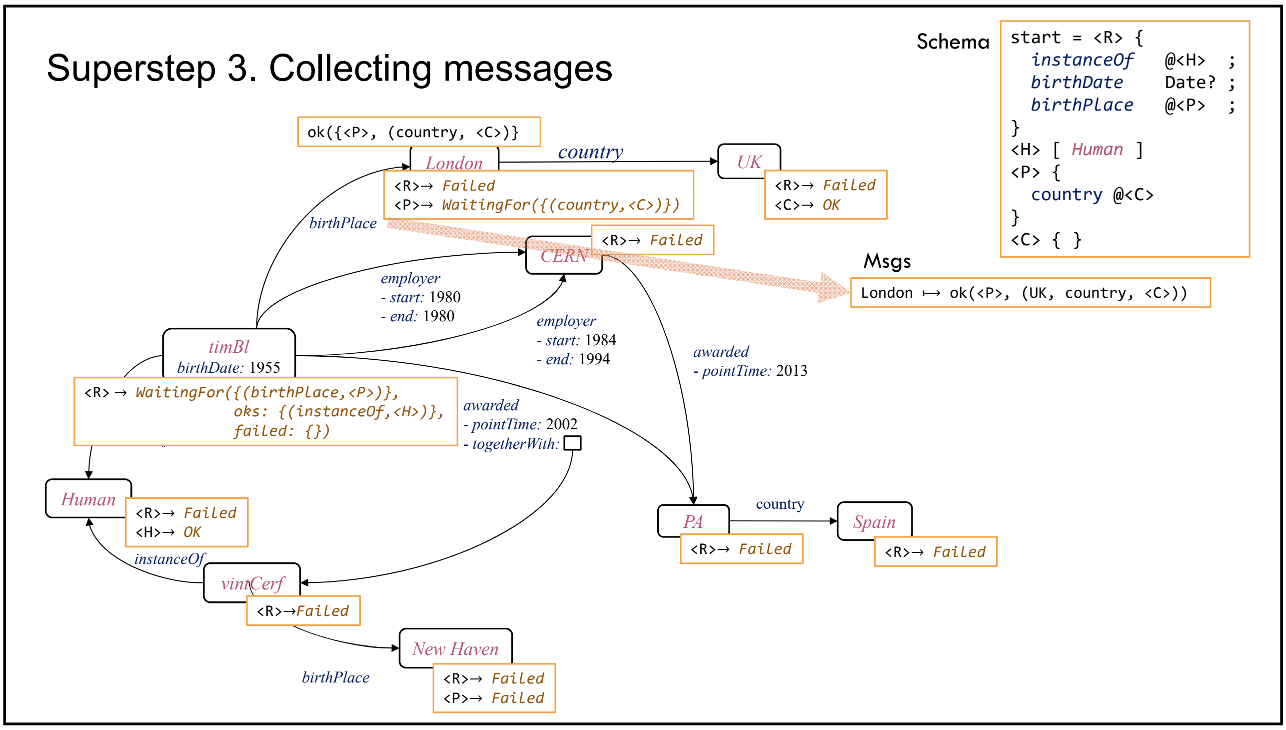
```
start = <R> {
  instanceOf   @<H>  ;
  birthDate    Date? ;
  birthPlace   @<P>  ;
}
<H> [ Human ]
<P> {
  country @<C>
}
<C> { }
```

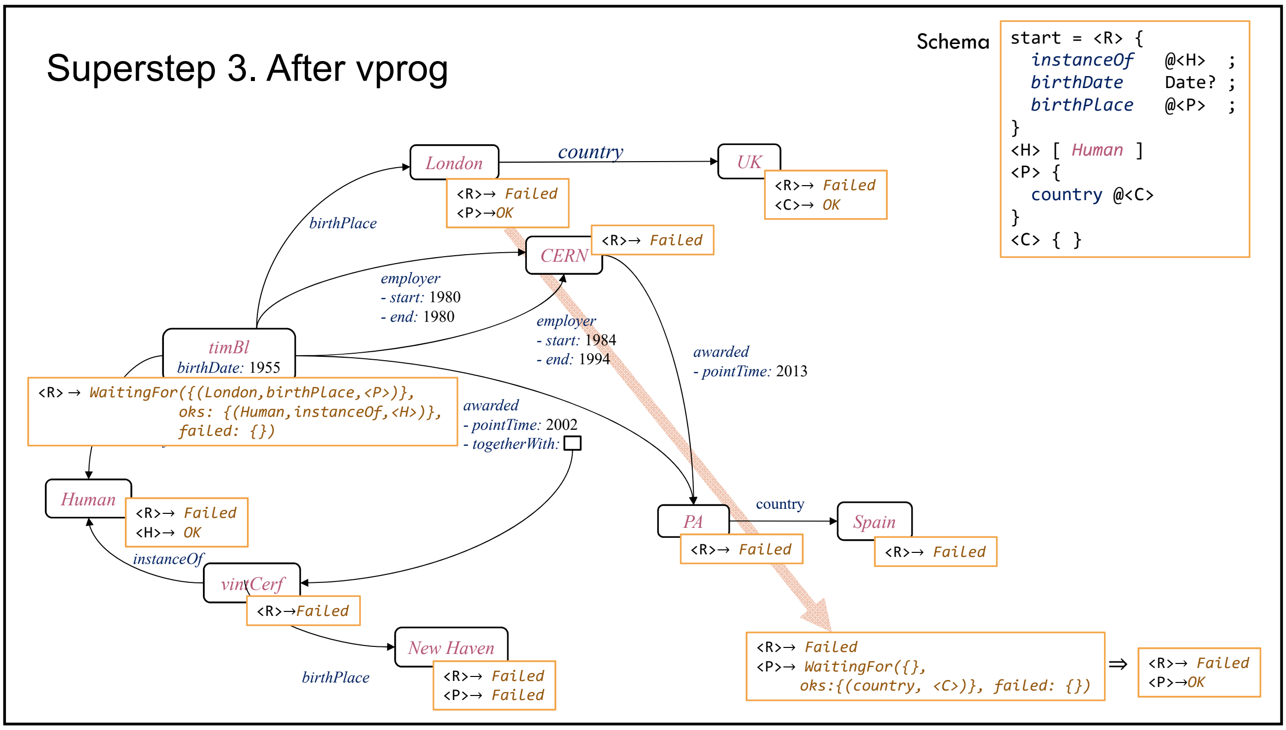


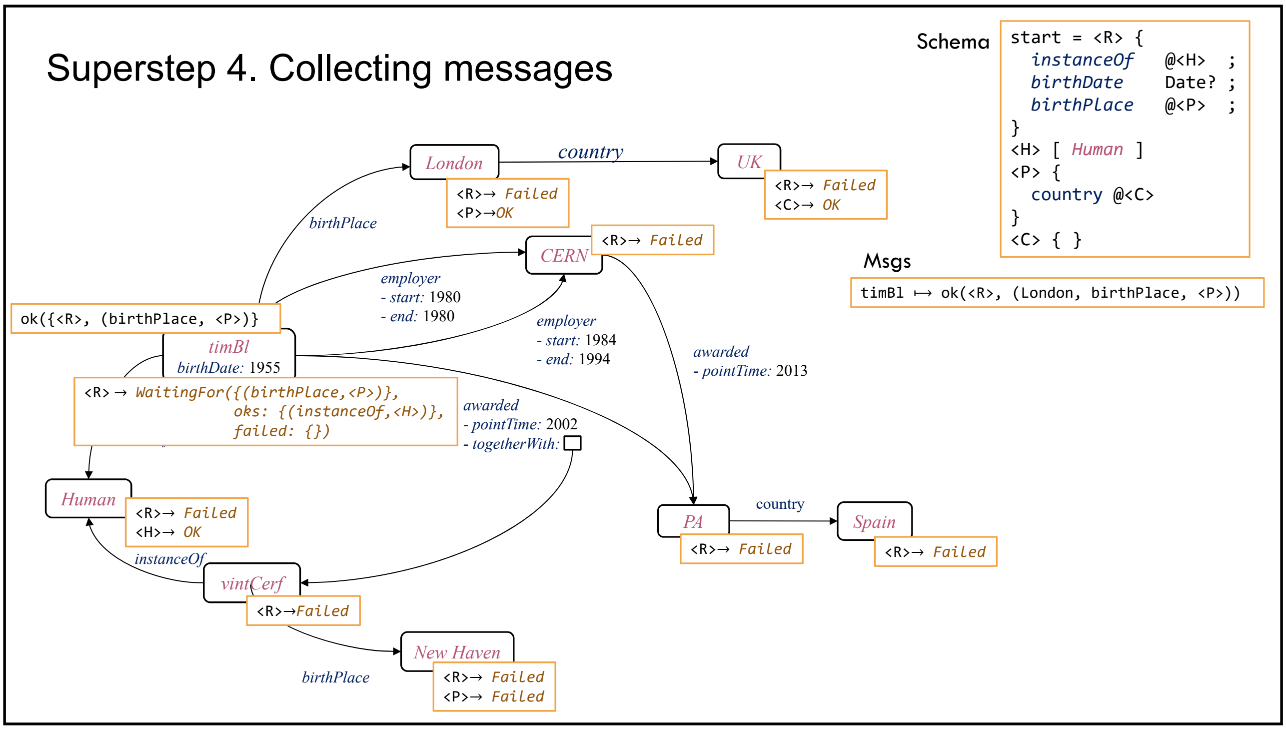
Msgs

```
London    ↦ waitFor({<P>, (UK, country, <C>)})
UK        ↦ validate({<C>})

timBl     ↦ ok({<R>, (Human, instanceOf, <H>)})

vintCerf  ↦ ok({<R>, (Human, instanceOf, <H>)})
```

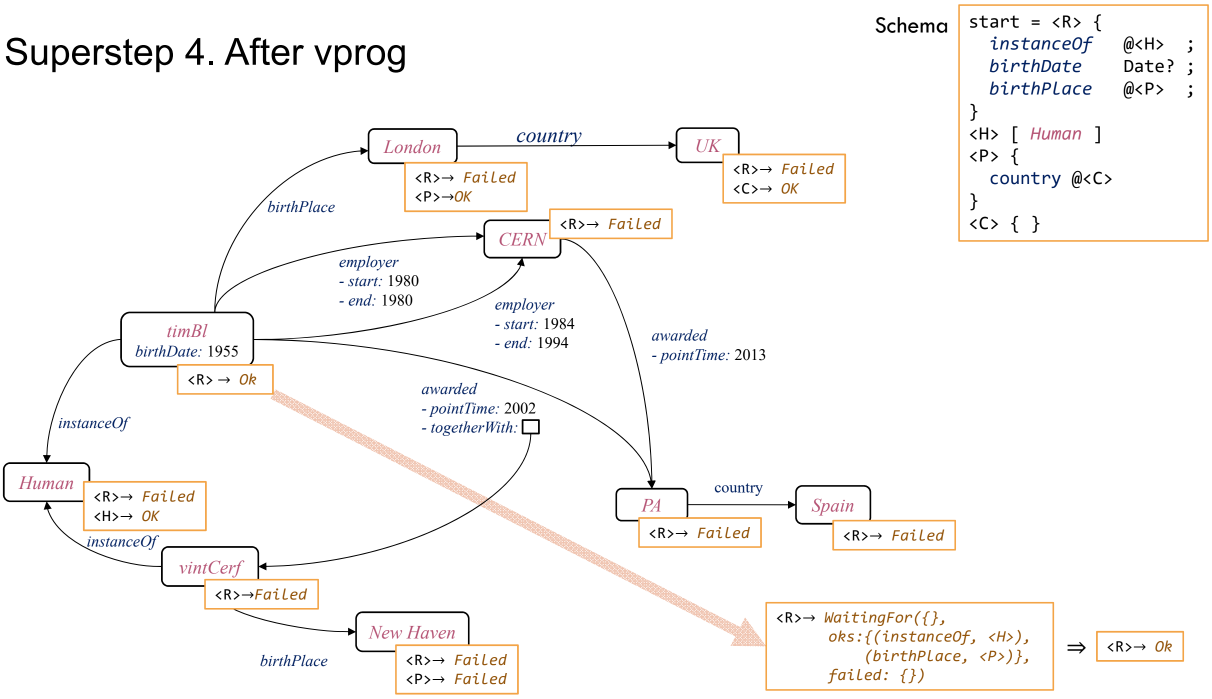

Superstep 2. After vprog
Schema
start = <R> {
instanceOf @<H> ;
birthDate Date? ;
birthPlace @<P> ;
}
<H> [Human]
<P> {
country @<C>
}
<C> { }
London
country
UK
<R>→ Failed
<P>→ WaitingFor({(UK,country,<C>)})
<R>→ Failed
<C>→ OK
birthPlace
CERN
<R>→ Failed
employer
- start: 1980
- end: 1980
employer
- start: 1984
- end: 1994
awarded
- pointTime: 2013
timBl
birthDate: 1955
<R> → WaitingFor({(birthPlace,<P>)},
oks: {(instanceOf,<H>)},
failed: {})
awarded
- pointTime: 2002
- togetherWith:
Human
<R>→ Failed
<H>→ OK
instanceOf
PA
country
Spain
<R>→ Failed
<R>→ Failed
vintCerf
<R>→Failed
New Haven
<R>→ Failed
<P>→ Failed
birthPlace
<R>→ WaitingFor({}
oks: {(Human, instanceOf,<H>)}
failed: {})
⇒
<R>→Failed


Superstep 3. Collecting messages
Schema
start = <R> {
instanceOf @<H> ;
birthDate Date? ;
birthPlace @<P> ;
}
<H> [Human]
<P> {
country @<C>
}
<C> { }
ok({<P>, (country, <C>)}
London
country
UK
<R>→ Failed
<P>→ WaitingFor({(country,<C>)})
<R>→ Failed
<C>→ OK
birthPlace
CERN
<R>→ Failed
Msgs
London ↦ ok(<P>, (UK, country, <C>))
employer
- start: 1980
- end: 1980
employer
- start: 1984
- end: 1994
awarded
- pointTime: 2013
timBl
birthDate: 1955
<R> → WaitingFor({(birthPlace,<P>)},
oks: {(instanceOf,<H>)},
failed: {})
awarded
- pointTime: 2002
- togetherWith:
Human
<R>→ Failed
<H>→ OK
instanceOf
PA
country
Spain
<R>→ Failed
<R>→ Failed
vintCerf
<R>→Failed
New Haven
<R>→ Failed
<P>→ Failed
birthPlace

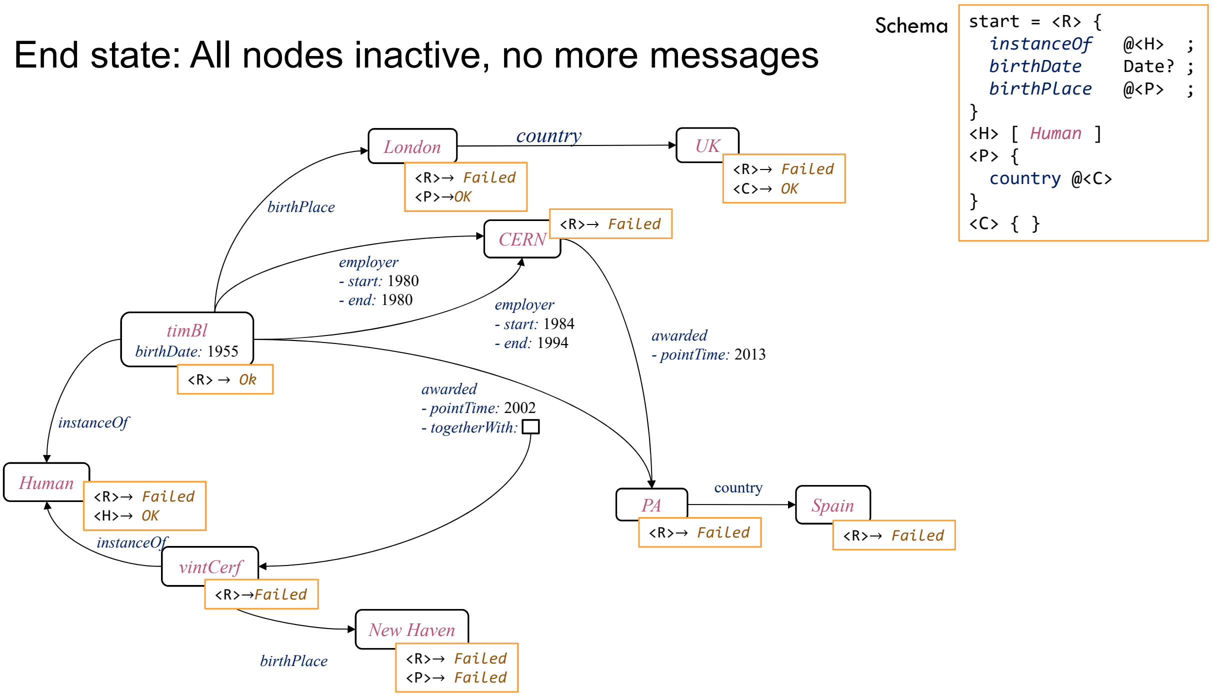

Superstep 3. After vprog
Schema
start = <R> {
instanceOf @<H> ;
birthDate Date? ;
birthPlace @<P> ;
}
<H> [Human]
<P> {
country @<C>
}
<C> { }
London
<R>→ Failed
<P>→OK
country
UK
<R>→ Failed
<C>→ OK
CERN
<R>→ Failed
birthPlace
employer
- start: 1980
- end: 1980
employer
- start: 1984
- end: 1994
awarded
- pointTime: 2013
timBl
birthDate: 1955
<R> → WaitingFor({(London,birthPlace,<P>)},
oks: {(Human,instanceOf,<H>)},
failed: {})
awarded
- pointTime: 2002
- togetherWith:
Human
<R>→ Failed
<H>→ OK
instanceOf
vintCerf
<R>→Failed
PA
<R>→ Failed
country
Spain
<R>→ Failed
New Haven
<R>→ Failed
<P>→ Failed
birthPlace
<R>→ Failed
<P>→ WaitingFor({},
oks:{(country, <C>)}, failed: {})
⇒
<R>→ Failed
<P>→OK


Superstep 4. Collecting messages
Schema
start = <R> {
instanceOf @<H> ;
birthDate Date? ;
birthPlace @<P> ;
}
<H> [Human]
<P> {
country @<C>
}
<C> { }
Msgs
timBl ↦ ok(<R>, (London, birthPlace, <P>))
London
<R>→ Failed
<P>→OK
country
UK
<R>→ Failed
<C>→ OK
CERN
<R>→ Failed
birthPlace
ok({<R>, (birthPlace, <P>)}
employer
- start: 1980
- end: 1980
employer
- start: 1984
- end: 1994
awarded
- pointTime: 2013
timBl
birthDate: 1955
<R> → WaitingFor({(birthPlace,<P>)},
oks: {(instanceOf,<H>)},
failed: {})
awarded
- pointTime: 2002
- togetherWith:
Human
<R>→ Failed
<H>→ OK
instanceOf
vintCerf
<R>→Failed
PA
<R>→ Failed
country
Spain
<R>→ Failed
New Haven
<R>→ Failed
<P>→ Failed
birthPlace

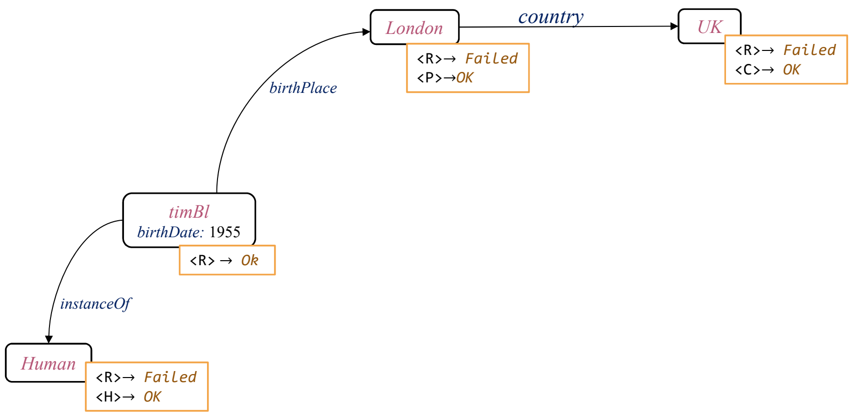
Superstep 4. After vprog



End state: All nodes inactive, no more messages



Wikibase Subgraph: entities with some OK shape



Schema

```
start = <R> {
  instanceOf   @<H>  ;
  birthDate    Date? ;
  birthPlace   @<P>  ;
}
<H> [ Human ]
<P> {
  country @<C>
}
<C> { }
```

Status and Messages

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

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

Adapting Pregel to validate graphs using ShEx

Algorithm 1: Pregel-based ShEx validation pseudocode

Input parameters:

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

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

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

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

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

Parameters

- InitialLabel, start label
- checkLocal, attempts to check if a node conforms locally. Possible results:
  - Ok
  - Failed
  - Pending(ls)
- checkNeighs, attempts to check the neighbourhood of a node. Result:
  - Ok
  - Failed
- tripleConstraints: set of constraints associated with a label

vProg definition

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

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

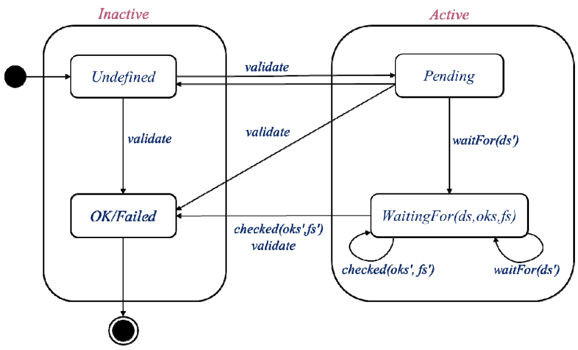
$$\frac{(n,m), l \rightsquigarrow Validate \quad m(l) = r \in \{Undefined, Pending\} \quad checkLocal(l,n) = Pending(ls)}{m'(l) = Undefined \quad m'(l') = Pending\ \forall l' \in ls}$$

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

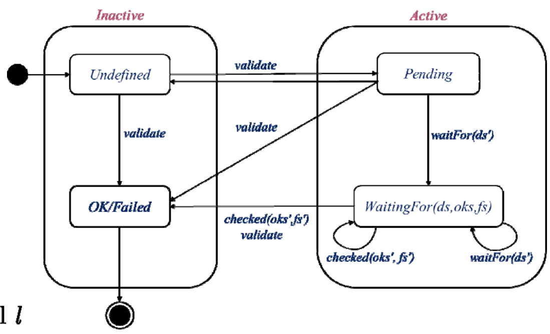
$$\frac{(n,m), l \rightsquigarrow Validate \quad m(l) = WaitingFor(ds, oks, fs)}{m'(l) = Ok}$$

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

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



sendMsg



$\langle (s, m_s), p, (o, m_o) \rangle =$
triplet view with
- subject (s, m_s)
- predicate p
- object (o, m_o)

$(x, m_x), l \rightsquigarrow Msg =$
message Msg sent to node x with status map m_x for label l

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

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

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

mergeMsg

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

$$\begin{aligned}
Validate \oplus y &= y \\
Validate \oplus Checked(oks, fs) &= Checked(oks, fs) \\
Validate \oplus WaitFor(ds) &= WaitFor(ds) \\
Checked(oks, fs) \oplus Validate &= Checked(oks, fs) \\
Checked(oks, fs) \oplus Checked(oks', fs') &= Checked(oks \cup oks', fs \cup fs') \\
Checked(oks, fs) \oplus WaitFor(ds) &= Checked(oks \cup ds, fs \cup fs) \\
WaitFor(ds) \oplus Validate &= WaitFor(ds) \\
WaitFor(ds) \oplus Checked(oks, fs) &= Checked(oks \cup ds, fs) \\
WaitFor(ds) \oplus WaitFor(ds') &= WaitFor(ds \cup ds')
\end{aligned}$$

checkLocal

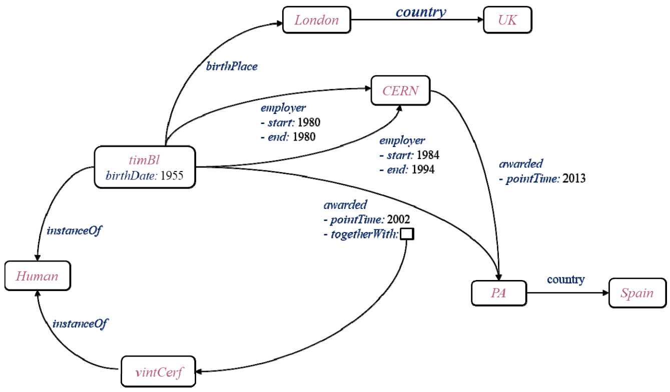
$\texttt{checkLocal}\colon\ (\mathcal{L},\,\mathcal{V}) \to Ok|\ Failed|\ Pending(\texttt{Set}[\mathcal{L}])$

Checks if it is possible to validate a node locally

Returns

- *Ok*: Node conforms
- *Failed*: Node doesn't conform
- *Pending(ls)* conformance depends on neighbours

```
checkLocal(<Human>, Human) = Ok
checkLocal(<Researcher>, Human) = Failed
checkLocal(<Researcher>, timBL) = Pending { <Human>, <Place> }
```




```
start = <Researcher>
<Researcher> {
  instanceOf    @<Human>  ;
  birthDate     Date? ;
  birthPlace    @<Place>  ;
}
<Human> [ Human ]
<Place> {
  country @<Country>
}
<Country> { }
```

checkNeighs

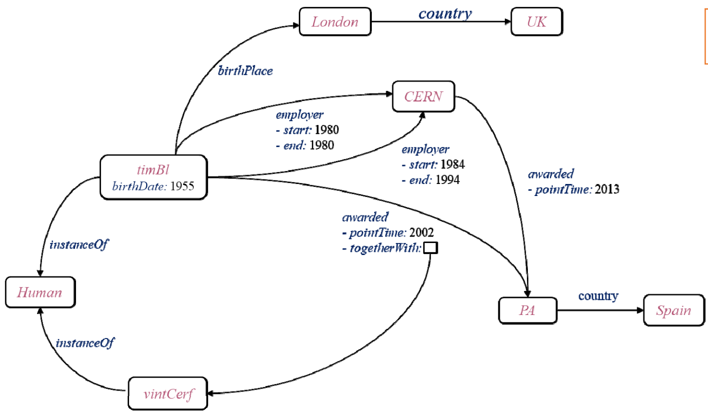
$\texttt{checkNeighs}\colon\ (\mathcal{L},\,\texttt{Bag}[(\mathcal{E},\,\mathcal{L})],\,\texttt{Set}[(\mathcal{E},\,\mathcal{L})]) \to Ok|Failed$

Checks if the bag of neighbours of a node conform to the regular bag expression defined by a shape

Returns: ok (conforms), failed (doesn't)

```
checkNeighs(<Researcher>, {(instanceOf, <Human>), (birthPlace, <London>)}, {}) = Ok
checkNeighs(<Researcher>, {(instanceOf, <Human>) }, {}) = Failed
```

```
rbe(<Researcher>) = (instanceOf, <Human>);
                    (birthPlace, <Place>)
```




```
start = <Researcher>
<Researcher> {
  instanceOf    @<Human>  ;
  birthDate     Date? ;
  birthPlace    @<Place>  ;
}
<Human> [ Human ]
<Place> {
  country @<Country>
}
<Country> { }
```

tripleConstraints

$\texttt{tripleConstraints}: \mathcal{L} \rightarrow \texttt{Set}[(\mathcal{E}, \mathcal{L})]$

Returns the triple constraints of a shape label

```
tripleConstraints(<Researcher>) = {(instanceOf, <Human>), (birthPlace, <Place>)}
tripleConstraints(<Human>) = {}
tripleConstraints(<Place>) = {(country, <Country>)}
tripleConstraints(<Country>) = {}
```

```
start = <Researcher>
<Researcher> {
  instanceOf   @<Human>  ;
  birthDate    Date? ;
  birthPlace   @<Place>  ;
}
<Human> [ Human ]
<Place> {
  country @<Country>
}
<Country> { }
```

Preliminary results

Implemented in SparkWDSub using Apache Spark GraphX

- No optimizations applied

Latest Wikidata dumps on AWS can be processed on AWS

- 512 cores, 3.904 Gb RAM, 121.600 Gb disk
- For 2014 Wikidata dump (31.3 GB uncompressed) in 3 minutes
- For 2021 Wikidata dump (1.256,55 Gb uncompressed) it took 36 minutes
- Only for simple schemas yet

Conclusions

Formal definition of Knowledge graphs
- RDF graphs, property graphs, wikibase graphs

ShEx extension for property graphs/wikibase graphs

Formal definition of wikibase graphs subsets

Scalable approach to validate big knowledge graphs
- Inspired by Pregel algorithm (Spark GraphX)

Repaso de contribuciones

	Grafos RDF	*Property graphs*	Grafos Wikibase
Definición formal	Realizado previamente	Realizado previamente	**Presente trabajo**
Descripción grafos conocimiento	ShEx Realizado anteriormente	**PShEx** **Presente trabajo**	**WShEx** **Presente trabajo**
Entity-generated (definición)	-	-	**Presente trabajo**
Entity-generated (implementación)	-	-	WDumper
Simple matching (definición)	-	-	**Presente trabajo**
Simple matching (implementación)	-	-	Wdumper
ShEx-based matching (definición)	-	-	**Presente trabajo**
ShEx-based matching (implementación)	-	-	**WDSub**
ShEx+Slurp (definición)	-	-	**Presente trabajo**
ShEx+Slurp (implementación)	ShEx.js, pyShEx	-	-
ShEx+Pregel (definición)	-	-	**Presente trabajo**
ShEx+Pregel (implementación)	-	-	**SparkWDSub**

Fin presentación ejercicio 1

Jose Emilio Labra Gayo

Universidad de Oviedo