# Industry Survey

Property Graph Schema Work Group

## Table of contents

# Introduction

## Participants

Participants in the industry survey are:

- Alberto Delazzari (Larus)
- Alastair Green (Neo4j)
- Victor Lee (TigerGraph)
- Petra Selmer (Neo4j)
- Dominik Tomaszuk (University of Bialystok)
- Oskar van Rest (Oracle)
- Mingxi Wu (TigerGraph)
- Hannes Voigt (Neo4j)
- Joshua Shinavier (Uber)

Simply add your name if you want to contribute.

## Languages

The focus is on the following languages:

Property Graph:

- Gremlin's Graph.Features (Josh S)
- Cypher for Apache Spark Graph DDL (Petra, Alastair)
- GSQL DDL from TigerGraph (Victor)
- Oracle PGX "heterogeneous graph" schema
- Neo4j/Larus ETL
- Gremlin Datastax schema: https://docs.datastax.com/en/dse/5.1/dse-dev/datastax_enterprise/graph/using/createSchemaGremlin.html
- Gremlin JanusGraph schema: https://docs.janusgraph.org/0.3.0/schema.html
  Also see http://tinkerpop.apache.org/docs/current/reference/
- Graph Processing in SQL Server 2017 https://docs.microsoft.com/en-us/sql/relational-databases/graphs/sql-graph-overview?view=sql-server-2017

RDF

- OWL (Oskar)
- SHACL (Hannes, Dominik)
- ShEx

Relational

- SQL Schema (Oskar)

XML

- XML Schema - W3C XSD (There are several other schema languages for XML) (Peter)

JSON

- JSON Schema (Alastair plus small editions by Dominik)

Other

- GraphQL: https://facebook.github.io/graphql/draft/#sec-Type-System (Dominik)

# Overview

TODO: create table

- General concepts
    - Oskar (Juan backup)
- Type hierarchy (subtypes, supertype)
    - Hannes
- Property / data value constraints
    -
- Structural constraints
    - Victor

## General concepts

See https://docs.google.com/presentation/d/1Aa1J0FcxcpVGvwMMoNUzLKWjCMobqW3cXE8PMnDLy-c/edit?usp=sharing

| | | | | | |
|---|---|---|---|---|---|
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |

| | | | | | |
|---|---|---|---|---|---|
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |

## Types Hierarchy

| concept<br><br>language | Vertex sub-/super-typing (form of hierarchy) | Edge sub-/super-typing (form of hierarchy) | Graph sub-/super-typing (form of hierarchy) |
|---|---|---|---|
| **Gremlin's Graph.Features** | no | no | no |
| **Cypher for Apache Spark Graph DDL** | Yes (Mixins, DAG) | Yes (Mixins, DAG) | ? |
| **TigerGraph GSQL DDL** | no | no | |
| **Oracle PGX "heterogeneous graph" schema** | no | no | no |
| **Neo4j/Larus ETL** | | | |
| **Gremlin Datastax schema** | no | no | no |

| | | | |
|---|---|---|---|
| **Gremlin JanusGraph schema** | no | no | no |
| **Graph Processing in SQL Server** | no | no | no |
| **OWL (incl. RDF Schema)** | RDFS: yes (arbitrary, graph)<br>OWL: uses RDFS | RDFS: yes (arbitrary, graph)<br>OWL: uses RDFS | RDFS: no<br>OWL: no |
| **SHACL** | As in RDFS | As in RDFS | As in RDFS |
| **ShEx** | As in RDFS | As in RDFS | As in RDFS |
| **SQL** | User-defined types in relational SQL support sub-/super-typing (single inheritance, tree) | | |
| **XML Schema** | Extension and restriction (single inheritance, tree) | no | no |
| **JSON Schema** | no | no | no |
| **GraphQL** | Yes, (interface implementation: Mixins, DAG; interface and type extension: single inheritance, tree) | No | Yes (Schema extension: single inheritance, tree) |

# Types

## ID types (Graph.Features)

- Any id
- Numeric id
- String id
- UUID id

## Property value types (Graph.Features)

Basic data types:

- Boolean
- Byte
- Double

- Float
- Integer
- Long
- String

Additional data types:

- Array <basic data type>
- Map -- arbitrary serializable values
- Mixed list -- arbitrary serializable values, not necessarily of the same type
- Uniform list -- arbitrary serializable values, all of the same type
- Serializable -- any serializable value

## Data value constraints

| concept<br><br>language | Range constraints | Keys | Property pair constraints | Multiple values |
|---|---|---|---|---|
| **Gremlin's Graph.Features** | | | | |
| **Cypher for Apache Spark Graph DDL** | unsupported | | | |
| **TigerGraph GSQL DDL** | For numerics | | | |
| **Oracle PGX "heterogeneous graph" schema** | unsupported | Keys are unique within a vertex/edge provider (e.g. database table or CSV file) | unsupported | unsupported |
| **Neo4j/Larus ETL** | | | | |
| **Gremlin Datastax schema** | | | | |

| | | | | |
|---|---|---|---|---|
| **Gremlin JanusGraph schema** | | | | |
| **Graph Processing in SQL Server** | For any data type (CHECK) | Same as SQL. | | |
| **OWL (incl. RDF Schema)** | | HasKey axiom states that each named instance of a class is uniquely identified by a property | Supported | Supported (RDF lists) |
| **SHACL** | For numerics and strings | unsupported | Supported | Supported (RDF lists) |
| **ShEx** | For numerics and strings | unsupported | unsupported | Supported |
| **SQL** | For any data type (CHECK) | Keys are unique within a table. Keys may be composites. Can auto-generate keys using AUTO INCREMENT | Supported using CHECK | Supported (in SQL:1999 Structured type) |
| **XML Schema** | For numerics and strings | Supported | | supported |
| **JSON Schema** | For numerics and strings | Only uniqueItems for arrays | | supported |
| **GraphQL** | | Unique IDs | unsupported | supported |

### Graph.Features

- Supported subset of id types (see above) for vertices
- Supported subset of id types (see above) for edges
- Supported subset of property value types (see above)

## Structural constraints

Graph.Features

- Whether vertex properties are supported
- Whether edge properties are supported
- Whether vertex meta-properties are supported
- Whether vertex multi-properties are supported (i.e. multiple properties for a given vertex with the same key but different values)

# General concepts

## Schema management

In OWL, ontologies (i.e. schemas) have a name, which is generally the place where the ontology document is located in the web.
OWL ontologies are placed into OWL documents, which are then placed into logical filesystems, on the web, or in the triplestore.
OWL makes no distinction between classes and instances, so, any graph instance could be considered as a schema.

In GSQL, the DDL adopts traditional relational database data independence design paradigm. That is, the meta data is separate from the binary data. The user pre-defines the metadata model before loading data. Schema can evolve over time via schema change DDL. The user can create multiple named graph schemas. Each graph schema has a name, which is a grouping of a set of vertex types and edge types. Each vertex/edge type has a predefined schema, which is similar to the table schema in SQL, with the constraint that a vertex type must have a primary key, and an edge type has a default composite primary key, composed of the source vertex type's primary key and the destination vertex type's primary key (additional key column can be added based on needs). A vertex type or edge type may belong to more than one graph schema-- **a graph is a namespace holder, where it can serve as view as well as a physical container.**

SHACL uses RDF vocabulary so in can be stored OWL (that can be also serialized in RDF) into logical filesystems, on the web, or in the triplestore.

In Cypher for Apache Spark, the DDL for the metadata is separate from the instance data. Through the use of a "graph type", multiple sets of instance data conforming to a schema (or graph type) can exist.

## Open world vs. closed world assumption

[The closed] world assumption implies that everything we don't know is *false*, while the open world assumption states that everything we don't know is *undefined*. [1]

Under the open world assumption, if a statement cannot be proven to be true with current knowledge, we cannot draw the conclusion that the statement is false.
Languages based on the **open world** assumption are:

- OWL
- RDF Schema

Languages based on the **closed world** assumption are:

- Gremlin
    - Supports closed-world operations such as count() and groupCount().
- SQL
- Cypher for Apache Spark Graph DDL
    - as this is bound to an underlying relational model (in Spark), this is closed world
- Relax NG
    - when the document specifies a schema
- XML XSD
    - when the document specifies a schema.
- SHACL
    - allows partial schemas, though (cf. Closed Constraint)
- ShEx
    - allows partial schemes
- GraphQL
    - do not allow partial schemas

[1] https://web.archive.org/web/20090624113015/http://www.betaversion.org/~stefano/linotype/news/91/

## Unique name assumption

OWL assumes NO unique names. So, different URLs may correspond to the same entity in the real world. OWL provides capabilities for relating entities, such as EquivalentClass, equivalentProperty, sameAs, differentFrom, AllDifferent, distinctMembers.

GSQL namespaces are modeled after those in SQL. A GSQL instance is one world which can contain several graphs, vertex types, and edge types (analogous to SQL tables). Names of graphs, vertex types, and edge types are unique within that world. Each object instance has a unique ID (primary key) within its type, but there is no constraint across types. For example, each vertex type could have a unique instance with ID = 001.

Cypher for Apache Spark mandates that within a graph type (analogous to a schema), names of node and edge types are unique.

While Gremlin does not specify whether two vertex IDs may correspond to the same logical entity, in practice it tends to be assumed that distinct vertices are distinct entities. Properties

belong exclusively to an individual edge or vertex, and vertices cannot easily be merged based on logical identity.

## Multiple Graphs

GSQL:

- A graph is a collection of vertex types and edge types.
- A GSQL instance can contain multiple graphs. The graphs can be disjoint or they can overlap. For example:
  CREATE GRAPH workplace (Person, Business, Job)
  CREATE GRAPH social (Person, Relationship)
- MultiGraph model:
  - Vertex types and edge types can be global or local.
  - A global type can potentially be included in any graph.
  - A local type is created within one graph and is only accessible within that graph.
  - Each graph is also an access control domain.

## Schema reuse

Through subtyping, through recursion (XML, JSON), etc.

XMLXSD: can import or include other schems or even parts of schemas. “Import” references declarations or definitions that are in a different target namespace. “Include” references declarations or definitions that are (or will be) brought into the target namespace of the schema.
JSON: can import parts of other schemas with “$ref”
OWL: can import entire ontologies in the OWL Header
SHACL: can import entire ontologies (using OWL) and support linking to shapes graphs
RELAX NG: can import entire or partial schema

GSQL does not provide schema declaration by reference. This fits with the SQL-like assumption that each database is its own world. One can copy the schema definitions statements and reuse them for another database, of course.

SQL has the LIKE construct for copying tables. It doesn’t reference, it copies.

Languages that **allow** recursive schema definition:

- XML XSD, Document Type Definition (DTD), RELAX NG
- JSON

Languages that **don’t allow** recursive schema definition:

- OWL

Cypher for Apache Spark provides for schema reuse. Graphs may refer to an existing graph type, or define new element, node and edge types, and extend these with mapping information.

## Schema Change

GSQL:
Modeled after SQL DDL syntax and semantics:

- Keywords ADD and DROP: You may ADD a new vertex or edge type or DROP an existing one.
- You may ALTER an existing vertex or edge type by performing ADD or DROP to its properties.
- You can ADD or DROP vertex and edge types to/from a schema.
- Schema changes must be wrapped in a SCHEMA_CHANGE JOB.
- There are additional subtleties for managing local vs. global types.
- Syntax:

```
ADD VERTEX vname (PRIMARY_ID id type …) // same syntax as CREATE VERTEX
ADD UNDIRECTED EDGE ename (FROM vname1…) // same syntax as CREATE UNDIRECTED EDGE
ADD DIRECTED EDGE ename (FROM vname1… ) // same syntax as CREATE DIRECTED EDGE
ALTER VERTEX|EDGE name ADD (attribute_name type DEFAULT default_value]
                          [, attribute_name type [DEFAULT default_value]]* );
ALTER VERTEX|EDGE name DROP (attribute_name [, attribute_name]* );
DROP VERTEX vname [, vname]*; DROP EDGE ename [, ename]*;
```

Graphs in SQL Server:

- Node and edge tables can be altered the same way a relational table is, using the `ALTER TABLE`. Users can add or modify user-defined columns, indexes or constraints. However, altering internal graph columns, like `$node_id` or `$edge_id`, will result in an error.

OWL:

- TODO Oskar

SQL:

- ALTER, ADD, DROP, CREATE

XML:

- The schema is itself an XML document, so editing the schema document changes the schema.

# Types and Instances

## Types

In OWL:

```
ClassAssertion( :Person :Mary )
ClassAssertion( :Woman :Mary )
```

This says that the instance Mary belongs to the class Person as well as the class Woman

In SHACL:
Types are SHACL classes similar to classes know from plain RDF and RDFS (cf. SHACL Terminology). Instances are assigned to classes, as in plain RDF, by an RDF triples, where the subject is an instance, the predicate is rdf:type, and the object is a class.
SHACL itself allow to express, so called, shaps, which are a conjunction of constraints that shall apply to either a type (Class-based Targets) or one or more instances (Node targets).

```
# schema
ex:ExampleShape
   a sh:NodeShape ;
   sh:targetClass ex:Person ;
   sh:property [
         sh:path ex:hasEmail ;
         sh:class ex:Email ;
   ] .
```

In GSQL:

```
CREATE VERTEX Person(SSN string PRIMARY_ID, name string)
CREATE UNDIRECTED EDGE Friendship(FROM Person, TO Person, start  datetime)
CREATE DIRECTED EDGE AnyRelation(FROM *, TO *, weight double)
CREATE GRAPH social (Person, Friendship, AnyRelation)
```

In XML XSD:
Complex types contain nested elements (of complex or simple type) as *sequence* (ordered), *all* (unordered) or *choice* (one of N alternatives). Cardinality of the nested elements in *sequence* and *all* is defined by the minOccurs, maxOccurs attributes of the element.

In JSON Schema:
Objects can have nested objects.

In GraphQL:
There are object types.

In Cypher for Apache Spark:

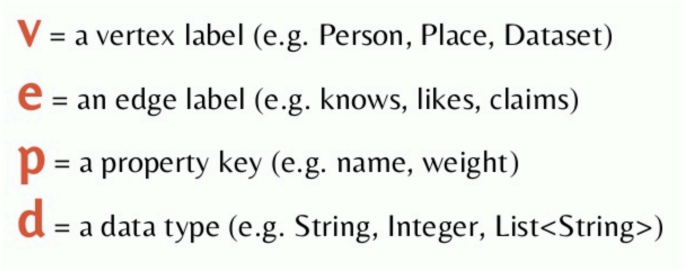
- The basic underlying type is called an **element type**
- An element type has a name, and can have 0 or more properties (name-value pairs)
    - A property may be either mandatory or optional
    - A property must be defined as having a fixed type (string, integer, etc)

- An element type can inherit from another element type (cycles are not permitted)
- Element types form the basis for **node** and **edge types**
- A node type is formed from an element type
- A node type is defined thus: `(node_type)`
- An edge type is comprised of two node types - one for the tail, and one for the head node - and one element type
- An edge type is defined thus:
  `(tail_node_type)-[ELEMENT_TYPE]->(head_node_type)`
- Node and edge types inherit all the properties defined by the constituent element types
- All the preceding definitions - comprising element, node and edge types - are contained within a **graph type**
- A graph type is always named; e.g. “finance_department”, and is defined thus:
  `CREATE GRAPH TYPE` **`finance_department`**

## Types in Property Graphs

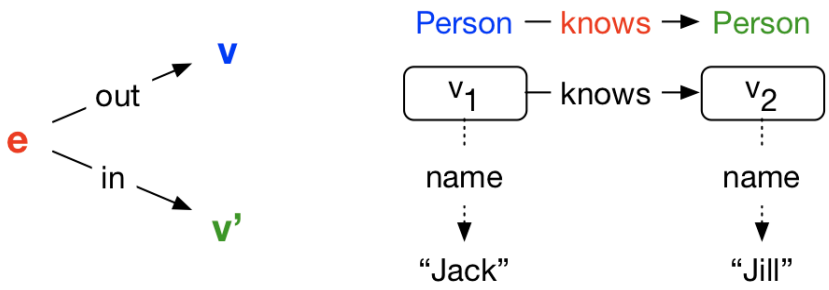
In the basic Property Graph data model, every edge and every property has a type (the *edge label* or *property key*), while a given vertex may or may not have a type (the *vertex label*). Alternatively, one can think of a graph as supporting a default (empty) vertex label in addition to explicit vertex labels. Below are a series of diagrams which illustrate various types of property graph elements in terms of category theory (source: slide show from the Global Graph Summit).

### High-level types



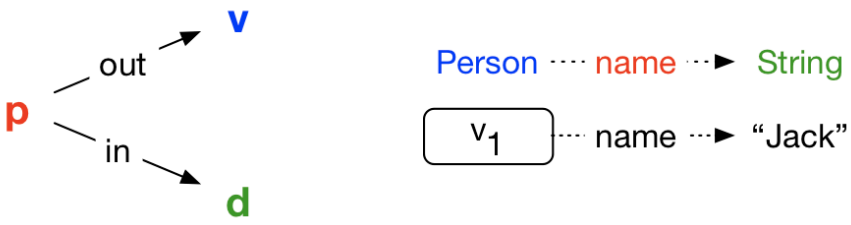

### Edge labels

Supported by every property graph; an edge connects a vertex with another vertex. In terms of schema, an edge label connects a vertex label with another vertex label.



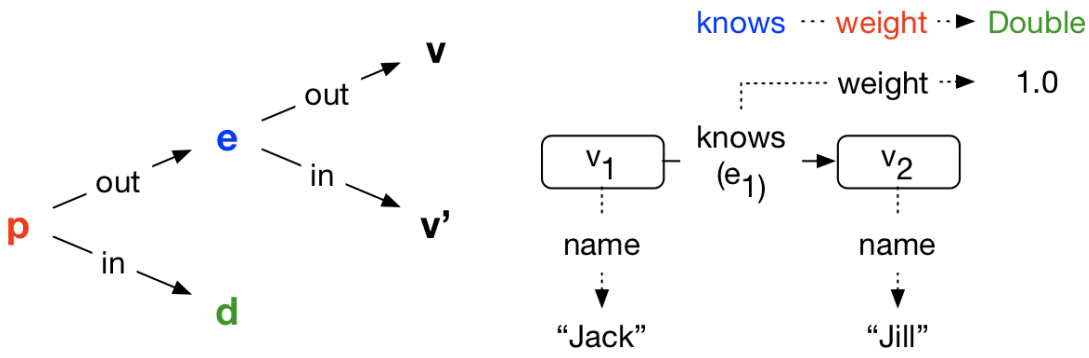

### Vertex properties

Optionally supported in property graphs. A property is similar to an edge, but it connects a vertex to a primitive value as opposed to another vertex.



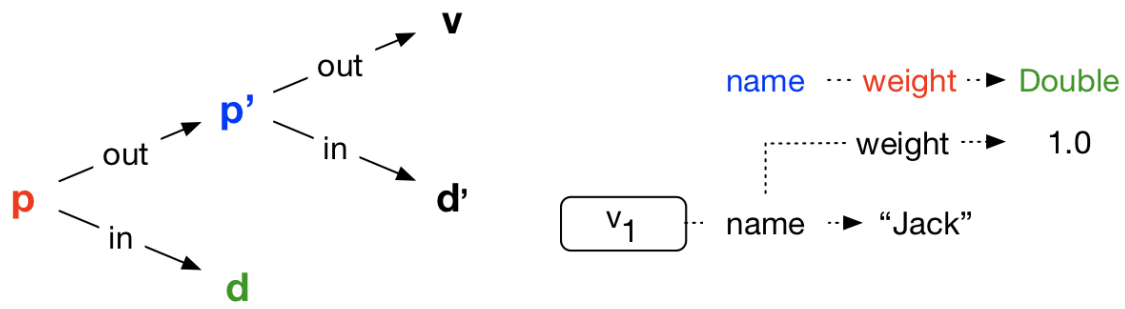

### Edge properties

Optionally supported in property graphs. Similar to a vertex property, but the outgoing element is an edge instead of a vertex.



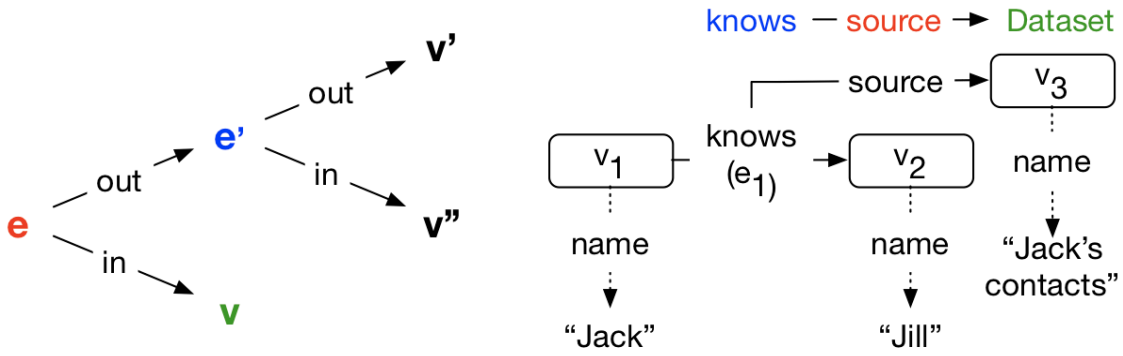

### Vertex meta-properties

Optionally, but not commonly supported in property graphs. A meta-property is a property outgoing from another property. In terms of Graph.Features, meta-properties can only be explicitly supported for vertices, not edges.



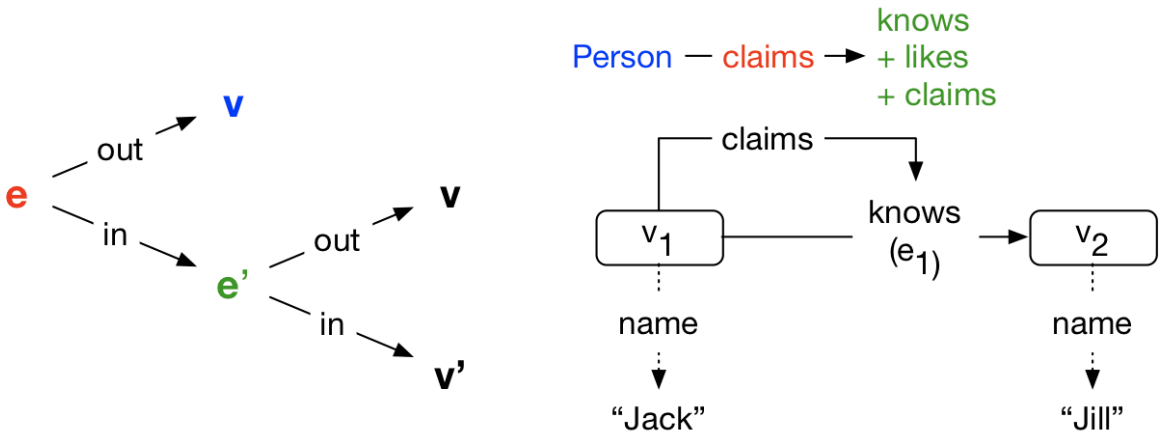

### Meta-edges: edge to vertex

Unidirected edges outgoing from other edges were supported in early Titan; see Advanced Schema.



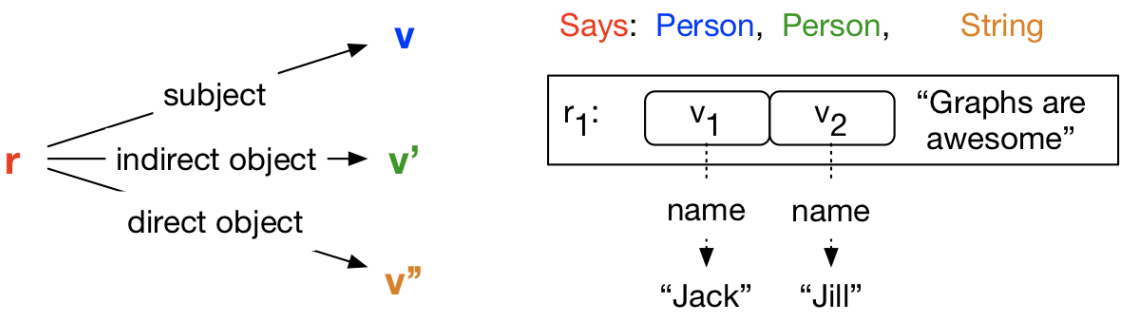

### Meta-edges: vertex to edge

Currently unsupported by any strictly PG database.

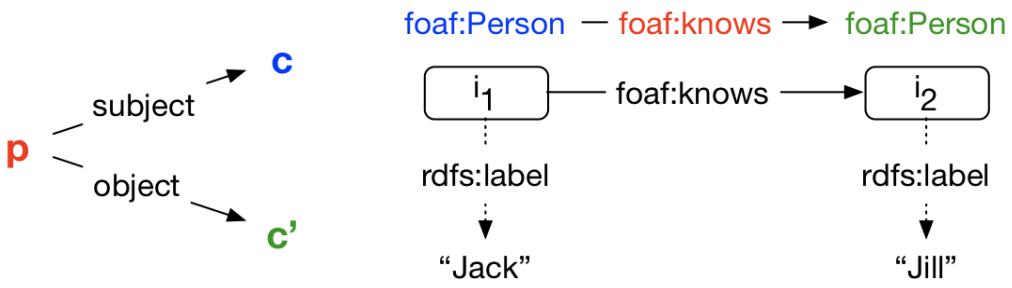



### Compare to: hyper-edge

See for example GRAKN.AI schemas.




Compare to: RDF property




## Subtyping

In OWL:

SubClassOf( :Woman :Person )

This says that Woman is a subclass of Person.

In SHACL:

Sub/superclassing is not expressed in SHACL, but rather by RDFS means (rdf:subClassOf and rdfs:subPropertyOf). However, SHACL validates a shape with class-based target also on all subclasses of the shape's target classes.

Additionally, shapes can be logically combined, which also also to include a more general shap into more specialized ones.

In XML Schema, type extensions or restrictions can be used to derive types from other types. This can happen both for "simple types" (integers, strings, …) and "complex types" (which are structural, i.e. describe how nodes in XML trees are allowed to be nested).

GSQL does not have subtyping or inheritance yet. It is unclear whether the property graph label concept can satisfy all subclass typing polymorphism query requirement at this point.

In GraphQL:

GraphQL does not include this semantic. But it has *implements* keyword which means that it needs to have the exact fields.

In Cypher for Apache Spark:
    Subtyping is permitted. See here for more details

## Type equivalence

In OWL:
    EquivalentClasses( :Person :Human )
This says that Person and Human can be used interchangeably at the instance level.

In SHACL:
SHACL does not allow to express type equivalence.

Property Graphs - I (Victor) do not know of any property graphs that include this semantic.

In GraphQL:
GraphQL does not include this semantic.

In Cypher for Apache Spark:
    This semantic is not included

## Type Disjointness

In OWL:
    DisjointClasses( :Woman :Man )
This says that instances cannot be both Man and Woman at the same time.

In SHACL:
SHACL does not allow to express type disjointness.

Property Graphs - I (Victor) do not know of any property graphs that include this semantic.

In GraphQL:
GraphQL does not include this semantic.

In Cypher for Apache Spark:
    This semantic is not included

## Data types

**XML XSD:**

- String : *string.* Various derivations: *normalisedString* (no CR/LF/tab), *token* (identifier-like, allows single spaces) and several for the names of XML and web constructs
- Numeric: *decimal, integer, float, double;* various signed and unsigned integers of defined length; integers contrained to positive (>0), negative (<0), nonNegative (>= 0), nonPositive (<=0)
- Date and time : *date*, *time*, *dateTime*, *duration* and a number of partial date (day, month, month+day etc.) types
- Other: *boolean*, binary (base64 or hex encoded)

# Data value constraints

**OWL:**
"Datatype properties" in OWL are like properties in the property graph model. However, OWL defines these properties independent of any class type. After having defined them, they can be assigned to an instance of any class. Another instance of the same class may not have the property. So in OWL, note that the properties that instances have are not described in their class types, but their instances.

**SHACL:**
SHACL allow to constraints which the properties for a node shape (including datatype constraints, value range constraints, string value constraints, pair check constraints, and logical combination of these) with so called property shapes.
**SHACL:**
SHACL allow to express datatype constraints, value range constraints, string value constraints, pair check constraints, and logical combination of these.
**GSQL:**
Properties of vertex/edge types are similar to columns in relational table definition.

**Cypher for Apache Spark:**
Properties of node and edge types are similar to columns in relational table definition, as this system is based atop Spark SQL.

## Key constraints

**SQL**: PRIMARY KEY functions as a constraint that requires each key to be UNIQUE.

# Datatype-generic constraints

## UNIQUE

**SQL**: UNIQUE ensures that all values in a column are different.

**JSON Schema**: uniqueItems key - if it has boolean value true, the instance validates successfully if all of its elements are unique.

**GraphQL**: The ID scalar type represents a unique identifier.

## NOT NULL constraint

**SQL**: NOT NULL ensures that the column cannot contain NULL values.

**XML XSD:** The *nillable* attribute on an element declaration specifies whether an instance can be *null*

**JSON Schema**: type key can have values without *null*.

**GraphQL**: ! means that the field is non-nullable.

**Cypher for Apache Spark**: ? means the field is nullable (i.e. optional)

## DEFAULT

**SQL**: The DEFAULT constraint is used to provide a default value for a column.

**XML XSD:** A default value can specified for an element (if of simple type or text only)

**JSON Schema**: The *default* keyword specifies a default value for an item.

**GraphQL**: Only when an argument is optional, we can define a default value.

## General value range constraint

**SQL**:

- The CHECK constraint is used to limit the value range that can be placed in a column.
  Example:

```
ALTER TABLE Persons
ADD CONSTRAINT CHK_PersonAge CHECK (Age>=18 AND City='Sandnes');
```

# Numeric-specific Constraints

## Numeric range constraints

**SHACL:** value range constraints

- sh:minExclusive - it specifies the the minimum exclusive value
- sh:minInclusive - it specifies the the minimum inclusive value
- sh:maxExclusive - it specifies the the maximum exclusive value
- sh:maxInclusive - it specifies the the maximum inclusive value

**XML Schema:** Constraining Facets

- xs:maxExclusive
- xs:minExclusive
- xs:maxInclusive
- xs:minInclusive

**JSON Schema**: Validation Keywords for Numeric Instances (number and integer)

- multipleOf
- maximum
- exclusiveMaximum
- minimum
- exclusiveMinimum

**GSQL:**
no value-range constraints. Supports INT, UINT, FLOAT, and DOUBLE numeric types.

**SQL**:

- AUTO INCREMENT allows a unique number to be generated automatically when a new record is inserted into a table.

**Cypher for Apache Spark:**
There are no value-range constraints. SQL numeric types are supported.

## Other numeric constraints

**XML Schema:** Constraining Facets

- xs:fractionDigits
- xs:totalDigits
- xs:assertion
- xs:explicitTimezone

## String-specific Constraints

**SHACL:** string value constraints

- sh:minLength - it specifies the minimum literal length of each value node that satisfies the condition.
- sh:maxLength - it specifies the maximum literal length of each value node that satisfies the condition.
- sh:pattern - it specifies a regular expression that each value node matches to satisfy the condition.
- sh:languageIn - it specifies the allowed language tags for each value node
- sh:uniqueLang - it can be set to true to specify that no pair of value nodes may use the same language tag

**JSON Schema**: Validation Keywords for Strings

- minLength
- maxLength
- pattern

**XML Schema:** Constraining Facets

- xs:length
- xs:minLength
- xs:maxLength
- xs:pattern

**GSQL:**
No string-based constraints. All strings are variable length (VARCHAR, not CHAR).

**Cypher for Apache Spark:**
There are no string-based constraints. SQL string types are supported.

## Property Pair Constraints

**OWL:**

- **owl:equivalentProperty** can be used to state that two properties have the same property extension, which means that they have the same “values”. Not to be confused with owl:sameAs which can be used to specify that two properties are the same even though they have different names.

**SHACL:** pair check constraints

- sh:equals - it specifies the condition that the set of all value nodes is equal to the set of objects
- sh:disjoint - it specifies the condition that the set of value nodes is disjoint with the set of objects
- sh:lessThan - it specifies the condition that each value node is smaller than all the objects
- sh:lessThanOrEquals - it specifies the condition that each value node is smaller than or equal to all the objects

# Multiple values

**XML XSD:** An element (within a *sequence* or *all*) can have a *maxOccurs* attribute other than the default 1, including *unbounded*. In an instance document, there is no difference between an element specified with *maxOccurs=1* and a single occurrence of an element that could have been multiple (i.e. there is no special structure for an array)

**JSON Schema**: Validation Keywords for Arrays
- items
- additionalItems
- maxItems
- minItems
- uniqueItems
- contains

**GraphQL**:
GraphQL supports arrays and enumerations.

**SHACL**:
SHACL supports RDF lists.

# Edges (Relationships)

**OWL:**
In OWL, links between instances of classes are defined by “object properties”. Note that OWL also has the concept of “datatype properties”, see above. Like datatype properties, object properties are defined independent of any class type. After having defined them, they can be assigned to an instance of any class, to relate it to any instance of another class.

**SHACL:**
As RDF, SHACL does not distinguish between properties and relationships. Both are treat in similar fashion. Nethertheless, SHACL allow to constraint the endpoints of a

property/relationship as well as cardinalities. A predicate can be constraint into representing a property or relationship by the class constraint and the node kind constraint.

# Source and destination types for Edges (Relationships)

**OWL:**

- **rdfs:domain** allows you to specify the permissible source classes of an object (or datatype) property

  Multiple rdfs:domain axioms are allowed and should be interpreted as a conjunction: these restrict the domain of the property to those individuals that belong to the intersection of the class descriptions. If one would want to say that multiple classes can act as domain, one should use a class description of the owl:unionOf form.

- **rdfs:range** allows you to specify the permissible destination classes of an object property (or the permissible datatype property values for a datatype property).

  Multiple range restrictions are interpreted as stating that the range of the property is the *intersection* of all ranges (i.e., the intersection of the class extension of the class descriptions c.q. the intersection of the data ranges). Similar to rdfs:domain, multiple alternative ranges can be specified by using a class description of the owl:unionOf form (see the previous subsection).

  Note that, unlike any of the value constraints described in the section on class descriptions, rdfs:range restrictions are global. Value constraints such as owl:allValuesFrom are used in a class description and are only enforced on the property when applied to that class. In contrast, rdfs:range restrictions apply to the property irrespective of the class to which it is applied. Thus, rdfs:range should be used with care.

- **owl:disjointWith** allows you to state that an instance of this class cannot be an instance of another
  – E.g. Man and Woman could be stated as disjoint classes

- **owl:unionOf** allows you specify that a class contains things that are from more than one class
  – E.g. Restroom could be defined as a union of MensRoom and LadiesRoom

- **owl:intersectionOf** allows you to specify that a class contains things that are both in one and the other

- **owl:complementOf** allows you specify that a class contains things that are not other things
  – E.g. Children are not SeniorCitizens

**SHACL:**
SHACL allow to constraint the endpoints of a relationship.

**GSQL:**
- An edge type definition specifies the types of vertices that it connects. The definition can specify the exact vertex types, e.g.,
  CREATE DIRECTED EDGE enrolledIn (FROM Student, TO Course, …)
- An edge type definition can use a wildcard for either the source vertex type, the destination vertex type, or both, e.g.,
  CREATE DIRECTED EDGE likes (FROM Person, TO *, …)

**Cypher for Apache Spark:**
- An edge type specifies the types of nodes that it connects. See here for more details.
- An edge type definition can use blank brackets to denote a wildcard for either the tail or head or an edge type. E.g. `()-[LIKES]->()`
- An edge type definition can also indicate that there is no direction to the edge. E.g. `(Person)-[LIKES]-(Person)`

# Structural Constraints

## Cardinality constraints

**OWL:**
- owl:cardinality
- owl:qualifiedCardinality
- owl:maxCardinality
- owl:maxQualifiedCardinality
- owl:minCardinality
- owl:minQualifiedCardinality

**SHACL:** instance cardinalities and endpoint cardinalities
- sh:qualifiedMinCount
- sh:qualifiedMaxCount
- sh:minCount - it specifies the minimum number of value nodes that satisfy the condition
- sh:maxCount - it specifies the maximum number of value nodes that satisfy the condition

**XML Schema:**

- xs:minOccurs
- xs:maxOccurs

**JSON Schema**:

- maxProperties
- minProperties

**GSQL:**
A side effect of defining an edge's primary key as (source_vertex_id, dest_vertex_id) is that there is an implicit 1:1 constraint, for a given edge type. Future versions of GSQL will support a user-defined or user-expanded primary key, to allow many:many relationships.

**Cypher for Apache Spark:**
Node and edge keys

## Logical Constraints

**SHACL:** logical combination of these

- sh:not - it specifies the condition that each value node cannot conform to a given shape
- sh:and - it specifies the condition that each value node conforms to all provided shapes
- sh:or - it specifies the condition that each value node conforms to at least one of the provided shapes
- sh:xone - it specifies the condition that each value node conforms to exactly one of the provided shapes

# Example schemas

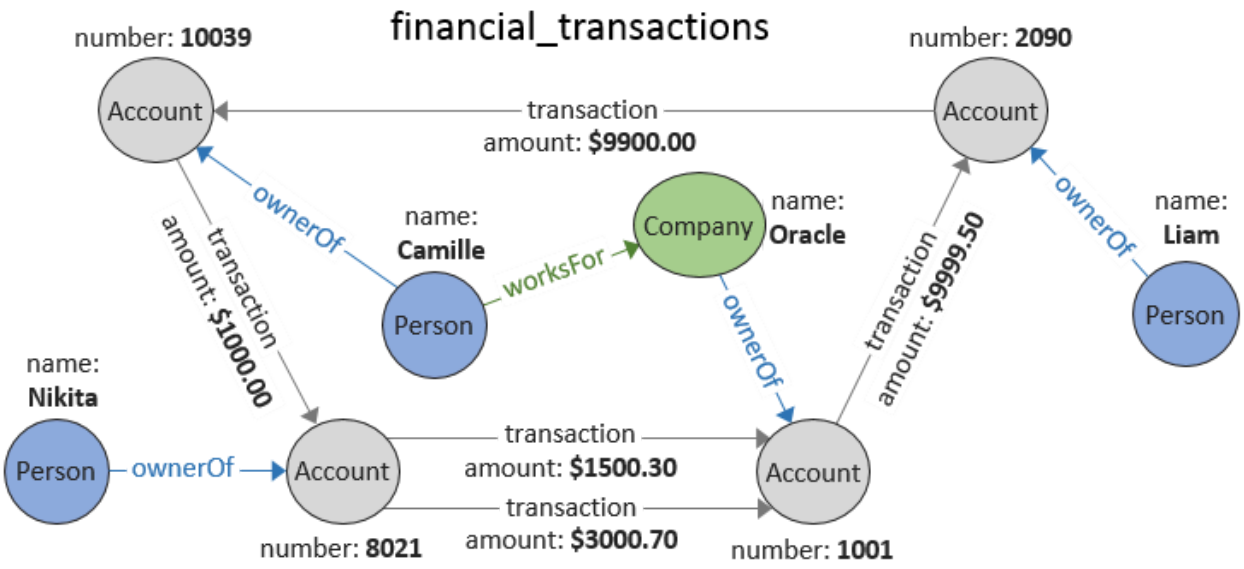
## OWL

From http://www.linkeddatatools.com/introducing-rdfs-owl

```
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:dc="http://purl.org/dc/elements/1.1/"
    xmlns:plants="http://www.linkeddatatools.com/plants#">

    <!-- OWL Header Omitted For Brevity -->

    <!-- OWL Class Definition - Plant Type -->
    <owl:Class rdf:about="http://www.linkeddatatools.com/plants#planttype">

        <rdfs:label>The plant type</rdfs:label>
        <rdfs:comment>The class of all plant types.</rdfs:comment>

    </owl:Class>

    <!-- OWL Subclass Definition - Flower -->
    <owl:Class rdf:about="http://www.linkeddatatools.com/plants#flowers">

        <!-- Flowers is a subclassification of planttype -->
        <rdfs:subClassOf
        rdf:resource="http://www.linkeddatatools.com/plants#planttype"/>

        <rdfs:label>Flowering plants</rdfs:label>
        <rdfs:comment>Flowering plants, also known as angiosperms.</rdfs:comment>

    </owl:Class>

    <!-- OWL Subclass Definition - Shrub -->
    <owl:Class rdf:about="http://www.linkeddatatools.com/plants#shrubs">

        <!-- Shrubs is a subclassification of planttype -->
        <rdfs:subClassOf
        rdf:resource="http://www.linkeddatatools.com/plants#planttype"/>

        <rdfs:label>Shrubbery</rdfs:label>
        <rdfs:comment>Shrubs, a type of plant which branches from the base.
        </rdfs:comment>

    </owl:Class>

    <!-- Individual (Instance) Example RDF Statement -->
    <rdf:Description rdf:about="http://www.linkeddatatools.com/plants#magnolia">

        <!-- Magnolia is a type (instance) of the flowers classification -->
        <rdf:type rdf:resource="http://www.linkeddatatools.com/plants#flowers"/>

    </rdf:Description>

</rdf:RDF>
```

# Oracle PGX “heterogeneous graph” schema

Example graph:




Corresponding schema definition:

```
{
  "name": "financial_transactions",
  "vertex_id_type" : "long",
  "edge_id_type" : "long",
  "edge_id_strategy" : "keys_as_ids",
  "vertex_providers": [
    {
      "format": "csv",
      "name": "Account",
      "key_column" : 1,
      "key_type" : "long",
      "props": [
        {
          "name": "number",
          "type": "long",
          "column": 2
        }
      ],
      "separator": " ",
      "uris": ["accounts.csv"]
    },
    {
      "format": "csv",
      "name": "Person",
      "key_column" : 1,
```

```
    "key_type" : "long",
    "props": [
      {
        "name": "name",
        "type": "string",
        "column": 2
      }
    ],
    "separator": " ",
    "uris": ["persons.csv"]
  },
  {
    "format": "csv",
    "name": "Company",
    "key_column" : 1,
    "key_type" : "long",
    "props": [
      {
        "name": "name",
        "type": "string",
        "column": 2
      }
    ],
    "separator": " ",
    "uris": ["companies.csv"]
  }
],
"edge_providers": [
  {
    "format": "csv",
    "name": "transaction",
    "source_vertex_provider": "Account",
    "destination_vertex_provider": "Account",
    "key_column" : 1,
    "key_type" : "long",
    "source_column" : 2,
    "destination_column" : 3,
    "props": [
      {
        "name": "key",
        "type": "long",
        "column": 1
      },
      {
        "name": "amount",
        "type": "double",
        "column": 4
      }
    ],
    "loading": {
      "create_key_mapping": true
    },
    "separator": " ",
    "uris": ["transactions.csv"]
  },
  {
    "format": "csv",
```

```
  "name": "ownerOf1",
  "source_vertex_provider": "Person",
  "destination_vertex_provider": "Account",
  "key_column" : 1,
  "key_type" : "long",
  "source_column" : 2,
  "destination_column" : 3,
  "props": [
    {
      "name": "key",
      "type": "long",
      "column": 1
    }
  ],
  "loading": {
    "create_key_mapping": true
  },
  "separator": " ",
  "uris": ["person_ownerOf_account.csv"]
},
{
  "format": "csv",
  "name": "ownerOf2",
  "source_vertex_provider": "Company",
  "destination_vertex_provider": "Account",
  "key_column" : 1,
  "key_type" : "long",
  "source_column" : 2,
  "destination_column" : 3,
  "props": [
    {
      "name": "key",
      "type": "long",
      "column": 1
    }
  ],
  "loading": {
    "create_key_mapping": true
  },
  "separator": " ",
  "uris": ["company_ownerOf_account.csv"]
},
{
  "format": "csv",
  "name": "worksFor",
  "source_vertex_provider": "Person",
  "destination_vertex_provider": "Company",
  "key_column" : 1,
  "key_type" : "long",
  "source_column" : 2,
  "destination_column" : 3,
  "props": [
    {
      "name": "key",
      "type": "long",
      "column": 1
    }
```

```
      ],
      "loading": {
        "create_key_mapping": true
      },
      "separator": " ",
      "uris": ["person_worksFor_company.csv"]
    }
  ]
}
```

## Graph processing in SQL Server

From https://docs.microsoft.com/en-us/sql/relational-databases/graphs/sql-graph-overview?view=sql-server-2017

```
CREATE TABLE Person (ID INTEGER PRIMARY KEY, Name VARCHAR(100), Age INT) AS NODE;
CREATE TABLE friends (StartDate date) AS EDGE;
```

## SQL

```
CREATE TABLE Persons (
      PersonID int,
      LastName varchar(255),
      FirstName varchar(255),
      Address varchar(255),
      City varchar(255)
);

ALTER TABLE Persons
ADD Email varchar(255);
```

## Cypher for Apache Spark

```
CREATE GRAPH Census_1901 (

  -- Nodes
  LicensedDog (
    licence_number INTEGER
  ) KEY LicensedDog_NK (licence_number),

  Person (
    first_name STRING?,
```

```
  last_name STRING?
),

Visitor (
  date_of_entry STRING,
  sequence INTEGER,
  nationality STRING?,
  age INTEGER?
) KEY Visitor_NK (date_of_entry, sequence),

Resident (
  person_number STRING
) KEY Resident_NK (person_number),

Town (
  CITY_NAME STRING,
  REGION STRING
) KEY Town_NK (REGION, CITY_NAME),

-- Relationships
PRESENT_IN,
LICENSED_BY (
  date_of_licence STRING
),

-- Node mappings:
(Visitor, Person)
     FROM VIEW_VISITOR,

(LicensedDog)
     FROM VIEW_LICENSED_DOG,

(Town)
     FROM TOWN,

(Resident, Person)
     FROM VIEW_RESIDENT,

-- Relationship mappings:
(Person, Resident)-[PRESENT_IN]->(Town)
    FROM VIEW_RESIDENT_ENUMERATED_IN_TOWN edge
        START NODES (Person, Resident)
            FROM VIEW_RESIDENT start_nodes
                JOIN ON start_nodes.PERSON_NUMBER = edge.PERSON_NUMBER
        END NODES (Town)
            FROM TOWN end_nodes
                JOIN ON end_nodes.REGION = edge.REGION
                AND end_nodes.CITY_NAME = edge.CITY_NAME,
```

```
  (Person, Visitor)-[PRESENT_IN]->(Town)
      FROM VIEW_VISITOR_ENUMERATED_IN_TOWN edge
          START NODES (Person, Visitor)
              FROM VIEW_VISITOR start_nodes
                  JOIN ON start_nodes.NATIONALITY = edge.COUNTRYOFORIGIN
                  AND start_nodes.PASSPORT_NUMBER = edge.PASSPORT_NO
          END NODES (Town)
              FROM TOWN end_nodes
                  JOIN ON end_nodes.REGION = edge.REGION
                  AND end_nodes.CITY_NAME = edge.CITY_NAME,
  (LicensedDog)-[PRESENT_IN]->(Town)
      FROM VIEW_LICENSED_DOG edge
          START NODES (LicensedDog)
              FROM VIEW_LICENSED_DOG start_nodes
                  JOIN ON start_nodes.LICENCE_NUMBER = edge.LICENCE_NUMBER
          END NODES (Town)
              FROM TOWN end_nodes
                  JOIN ON end_nodes.REGION = edge.REGION
                  AND end_nodes.CITY_NAME = edge.CITY_NAME,

  (LicensedDog)-[LICENSED_BY]->(Person, Resident)
      FROM VIEW_LICENSED_DOG edge
          START NODES (LicensedDog)
              FROM VIEW_LICENSED_DOG start_nodes
                  JOIN ON start_nodes.LICENCE_NUMBER = edge.LICENCE_NUMBER
          END NODES (Person, Resident)
              FROM VIEW_RESIDENT end_nodes
                  JOIN ON end_nodes.PERSON_NUMBER = edge.PERSON_NUMBER
)
```