TECHNISCHE UNIVERSITÄT DRESDEN

KNOWLEDGE GRAPHS

Lecture 9: Limits of SPARQL / Introduction to Property Graphs

Markus Krötzsch
Knowledge-Based Systems

TU Dresden, 11th Dec 2018

Expressive power

Review

SPARQL is:

- PSpace-complete for combined and query complexity
- NL-complete for data complexity

⇝ scalable in the size of RDF graphs, not really in the size of query
⇝ similar situation to other query languages

Hardness is shown by reducing from known hard problems

- Truth of quantified boolean formulae (QBF)
- Reachability in a directed graph

Membership is shown by (sketching) appropriate algorithms

- Naive, iteration-based solution finding procedure runs in polynomial space
- For fixed queries, the complexity drop to nondeterministic logspace

Expressive power and the limits of SPARQL

The expressive power of a query language is described by the question:
"Which sets of RDF graphs can I distinguish using a query of that language?"

More formally:

- Every query defines a set of RDF graphs: the set of graph that it returns at least one result for
- However, not every set of RDF graphs corresponds to a query (exercise: why?)

Note: Whether a query has any results at all is not what we usually ask for, but it helps us here to create a simpler classification. One could also compare query results over a graph and obtain similar insights overall.

Definition 9.1: We say that a query language $\mathbf{Q_1}$ is more expressive than another query language $\mathbf{Q_2}$ if it can characterise strictly more sets of graphs.

Complexity limits expressivity

Intuition: The lower the complexity of query answering, the lower its expressivity.

Question: which complexity are we talking about here? — data complexity!

- Given a set of RDF graphs that we would like to classify,
- we ask if there is one (fixed) query that accomplishes this.

If classifying the set of graphs encodes a computationally difficult problem, then the query evaluation has to be at least as hard as this problem with respect to data complexity.

Example 9.2: We have argued that SPARQL queries can evaluate QBF, and we could encode QBF in RDF graphs (in many reasonable ways). However, there cannot be a SPARQL query that recognises all RDF graphs that encode a true QBF, since this problem is PSpace-complete, which is known to be not in NL.

Complexity is not the same as expressivity

Complexity-based arguments are often quite limited:

- They only apply to significantly harder problems
- Additional assumptions are often needed (e.g., it is assumed that NL ≠ NP, but it was not proven yet)
- Typically, query language cannot even solve all problems in their own complexity class (i.e., they do not "capture" this class)

Example 9.3: SPARQL cannot recognise all graphs in which two given resources (say s and t) are reachable by an arbitrarily long **parallel path**, where each pair of elements is connected by properties p and q:

- the only SPARQL feature that can check for paths are property path patterns
- a match to a property path pattern is always possible using only vertices of degree 2 on the path; higher degrees can only be enforced for a limited number of nodes that are matched to query variables
- the query requires an arbitrary number of nodes of degree 4 on the path

However, this check can be done in NL by a similar algorithm as for reachability.

Limits by design

Besides mere expressivity, SPARQL also has some fundamental limits since it simply has no support for some query or analysis tasks:

- SPARQL is lacking some datatypes and matching filter conditions, most notably geographic coordinates (major RDF databases add this)
- SPARQL cannot talk about path lengths, e.g., one cannot retrieve the length of the shortest connecting path between two elements
- SPARQL cannot return paths (of a priori unknown length) in results
- SPARQL has no support for recursive/iterative computation, e.g., for page rank or other graph algorithms

Potential reasons: performance concerns (e.g., page rank computation would mostly take too long; longest path detection is NP-complete [in data complexity!]), historic coincidence (geo coordinates not in XML Schema datatypes); design issues (handling paths in query results would require many different constructs)

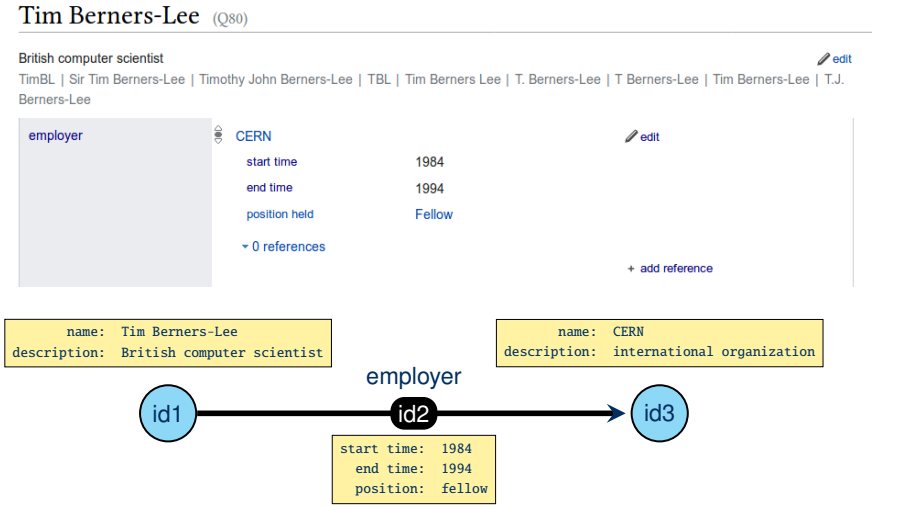
Property Graphs

What is a Property Graph?

Property Graph is a type of graph, that can be described as follows:

- directed (edges have source and target vertices)
- vertex-labelled (for some kind of “label”)
- edge-labelled (for some kind of “label”)
- multi-graph (several versions of the exact same edge may exists)
- with self-loops (vertices can have edges to themselves), and
- with sets of attribute-value pairs associated with any vertex or edge

Example



Obvious questions

Many issues require further specification:

- What are the ids for vertices and edges?
- What are “attributes”?
- What are “values”? Which datatypes are supported? How are they defined?
- What are those “labels” that one can use for edges?
- What are those “labels” that one can use for vertices?
- If vertices and edge can have arbitrary attribute-value pairs, why do we also need labels?

Unfortunately, Property Graph as such is not an answer to all such questions:

The name “Property Graph” refers to a broad class of enriched graph structures, allowing for many technical interpretations in different software systems. These interpretations are often incompatible and based on different assumptions.

Types of “Property Graphs”: Object Model

The name “Property Graph” primarily hints at the attribute-value pairs (called “properties”) that can be associated with nodes and edges.
There are different ways to interpret this model when designing actual data structures.

View 1: Property Graph as an Object Model

“If you have ever worked with an object model or an entity-relationship diagram, the labeled property graph model will seem familiar.”
– Neo4j, `https://neo4j.com/developer/guide-data-modeling/`

- Graphs viewed as data-modelling API in a programming language (often Java)
- “Values” are could be any other objects that represent data in programming
- Programmatic data access approaches are preferred over query language services
- Examples: Apache TinkerPop/Gremlin[1], Neo4j/Cypher, multi-model object databases (e.g., Azure Cosmos DB, OrientDB, Oracle Spatial and Graph),

[1] Many graph DBMS have TinkerPop bindings, but TinkerPop’s native view is object-based.

Types of “Property Graphs”: Relational Database Extension

The name “Property Graph” primarily hints at the attribute-value pairs (called “properties”) that can be associated with nodes and edges.
There are different ways to interpret this model when designing actual data structures.

View 2: Property Graph as Extended Relational Model

> “Vertex attributes match to columns of the vertex table. Edge attributes match to columns of the edge table. The maximum number of attributes is bound by the maximum number of columns for the underlying tables”
> – SAP HANA Graph Reference, v1.0 – 2016-11-30

- Graphs viewed as indexing and access layer on top of RDBMS
- “Values” can be any values in standard or proprietary SQL datatypes
- Various data access paradigms: in-DB-code (e.g., SAP Hana GraphScript) or query language (e.g., Tigergraph GSQL)
- Examples: SAP Hanah Graph, Tigergraph[1]

[1] Not based on a full RDBMS, but strongly using RDB concepts, rigid schema.

Types of “Property Graphs”: RDF-Database Extension

The name “Property Graph” primarily hints at the attribute-value pairs (called “properties”) that can be associated with nodes and edges.
There are different ways to interpret this model when designing actual data structures.

View 3: Property Graph as Access Layer on Top of RDF

> “property graph data can be loaded and accessed via the TinkerPop3 API, but underneath the hood the data will be stored as RDF” – Blazegraph TinkerPop3

- Property Graphs stored internally as RDF graphs
- “Values” can be any values supported in RDF (XML Schema + proprietary extensions)
- Multiple access paradigms: Apache Tinkerpop Gremlin or SPARQL
- Examples: BlazeGraph, Stardog

Types of “Property Graphs”: Summary

Main approaches:

- Object databases: flexible, schema-less; often multi-model; includes many graph extensions of noSQL DBMS; varying datatypes and formats (e.g., JSON for many object DBs)
- Relational databases: rather rigid, schema-based; graph extensions of classical RDBMS; SQL datatypes
- RDF databases: flexible, schema-less, highly normalised (data atomised into triples); property graph extensions of RDFDBs; RDF datatypes

Other types of graph databases:

- Simpler graph models (neither RDF nor property graph); mostly for network analysis; e.g., Apache Giraph
- Based on other paradigms; e.g., AllegroGraph (RDF database with Prolog support)
- Combinations and specialised components/frameworks; e.g., some data stored in Lucene or Solr (for text search), exchangeable storage back-end

Property graph: data access

Methods for data access are as diverse as the data structures that are used.

Programmatic access:

- Proprietary or common APIs (mostlty Tinkerpop)
- Scripting and processing languages (e.g., Apache Gremlin, SAP GraphScript)
- MapReduce, Spark, and other processing frameworks

Query languages:

- Neo4j Cypher
- Oracle PGQL
- Tigergraph GSQL
- . . .

Property Graph: What to teach?

The current implementation chaos has prompted some activities:

- OpenCypher publishes a specification for some parts of Neo4j's Cypher query language
- The Linked Data Benchmark Council has proposed a merger of PGQL, Cypher, and some Gremlin features, called G-CORE (not implemented yet)
- A current push towards a unified "(Property) Graph Query Language" GQL is underway

⇝ In this course: focus on (Open) Cypher + discussion of proposed changes

A note on terminology

Unfortunately, the OpenCypher/Neo4j world uses completely different names for concepts than the RDF world:

Graph-theoretic concept	OpenCypher terminology
vertex	node
edge	relationship
vertex label	a set of strings, each of which is called label
edge label	relationship type
key-value pair	property

Terms are shown in a distinct style to clarify this special meaning

Identity and labelling

Nodes and relationships have independent identity

- Especially: two relationships can be indistinguishable in terms of data (same source, target, label, properties) and yet be different
- Identity conferred by identifiers, which are implementation specific

Labels are based on strings:

- Vertex labels are sets of unicode strings, called labels
- Edge labels are single unicode strings, called relationship types
- Both vertices and edges may have no labelling

In practice, labels are used to take the role of types or classes, e.g., one may have a label `person` used for all nodes representing people.

Relationship types play the role of RDF properties, denoting the type of relationship that an edge expresses.

Properties

Sets of properties can be assigned to vertices and edges.

Properties are key-value pairs:

- Property keys are unicode strings
- Property values are either concrete values of some datatype, or lists of values of the same datatype

Property keys must be unique in a set of properties used on some node or relationship, but lists can be used to encode several values (not quite the same, e.g., in query answering).

Datatypes

Supported datatypes for property values in OpenCypher:

- INTEGER: “exact numbers without decimals” (apparently of arbitrary magnitude)
- FLOAT: double precision (64bit) floating point numbers
- STRING: unicode strings
- BOOLEAN: true or false
- lists of values of the above

Missing datatypes of much practical importance

- dates and times
- geographic coordinates
- fine-grained numeric types

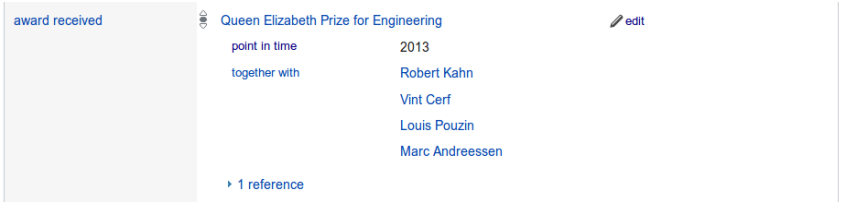
⇝ might be supported as proprietary extensions in implementations

Anything missing?

Another major omission and one of the biggest shortcomings or Property Graph models:

Property values cannot be references to vertices or edges.

Property Graph in this form is therefore not suitable to model, e.g., Wikidata statements:



This information could be captured in a Property Graph that looks like the RDF graph we used before, but not using any properties at all.

From Property Graph to RDF

As expected, every Property Graph can be expressed as an RDF graph:

- Use reification to represent edges by auxiliary nodes
- Use special RDF properties to encode source and target of edges, and the node-label and relationship-relationship type association
- Use application-based RDF properties to encode property keys
- Depending on the exact Property graph implementation, use some appropriate datatype-to-RDF mapping
  (e.g., based on RDF2RDB mappings from SQL datatypes to RDF)

Remarks:

- This scheme is implemented natively in existing DBMS (Blazegraph, Amazon Neptune), and scales despite of the increase size of the graph data that the system has to work with
- The use of reification can be avoided for relationships without properties
- One can also use both (reified and direct statements) for flexibility (similar to Wikidata’s RDF encoding)

From RDF to Property Graph

As expected, every RDF Graph can be expressed as a Property Graph:

- Represent all RDF resources by nodes
- Use property keys to associate resources with IRIs and/or datatype literanl information
- Represent all RDF triples with auxiliary nodes
- Use relationships with special relationship types to associate auxiliary nodes with triple subject, predicate, and object

Remarks:

- There does not seem to be any simpler way of capturing the full power of RDF in Property Graph, due to the restrictions of the latter (no reference in relationship types or property values to nodes)
- In some cases, certain triples could be represented as properties (if their datatype is supported in Property Graph and we do not need their RDF property to be addressable in the graph)
- Many Property Graph implementations will have performance problems in handling graphs with so many edges (part of the motivation for moving data into properties is to reduce the graph size)

(No) Schema Modelling

In RDF, properties were identified by IRIs and could be subject of triples

- to define labels and descriptions in several languages
- to specify the dataype for the property
- to related it to other properties, e.g., to their inverse

Example 9.4: Wikidata describes properties on own pages, and allows them to be used in statements, references, or statement qualifiers.

In Property Graph, labels, relationship types, and property keys are plain strings

- They cannot occur in the graph
- They can have neither relationships nor properties

Workarounds:

- One can create nodes that refer to a string token through a property value, and encode the knowledge that this is meant as a reference in application software
- Some database management system may support the declaration of constraints that restrict the usage of labels, relationship types, or property keys

Summary

SPARQL expressivity is still limited, partly by design

Property Graph is a general concept for organising graph data in two layers: a primary graph layer and a sub-ordinate key-value-set layer

Property graph has many different, incompatible implementations, based on several database paradigms (object, relational, RDF graph)

What's next?

- The Cypher query language
- Quality assurance in knowledge graphs
- More graph analysis