CS520: KNOWLEDGE GRAPHS
How Should AI Organize Knowledge?
Searching, Learning, Reading, Explaining
Panel Discussion Format
Spring 2020, Tu 4:30-6:20, cs520.Stanford.edu
Learn about the latest
Research & Applications

CS520

Knowledge Graphs

What should AI Know ?

How to Create a Knowledge Graph?

1. Introduction

It is possible to get started with a knowledge graph with no upfront design of its schema, and populate both its schema and instances during the creation process. To the degree an upfront design of a knowledge graph is practical, it can significantly improve its usefulness. Such a design involves making a suitable choice of the nodes, node labels, node properties, relations and relation properties.

The input to knowledge graph population can come from one or more sources consisting of structured data, semi structured data, free text or images, or direct authoring by human input. When we are working with structured and semi structured data sources, we have to perform schema mapping task (i.e., relating the schema in the input source with the schema of the knowledge graph) and record linkage task (i.e., relating new instances with the pre-existing instances in the knowledge graph). These exact same tasks are also faced during data integration with the only difference that the integrated data is expressed in a graph data model. When we are working with the unstructured sources, we have to solve the information extraction problems of entity extraction and relation extraction.

The choice of methods used in knowledge graph population depend on the scale of the problem and the desired accuracy. //If a knowledge graph is to be used on the web scale for information retrieval, the accuracy need not be perfect, and it is infeasible to use human verification for every triple of the graph. /If a knowledge graph is to be used within an enterprise where the accuracy needs to be nearly perfect, the human verification is essential even if it is performed just before the information is to be used. As accuracy is always desired regardless of the enterprise or the WWW settings, to ensure cost effectiveness and scalability, there is an empahsis on crowdsourcing and other low-cost methods of obtaining human input.

In this chapter, we will focus on knowledge graph schema design. In the next two chapters we will discuss the problems that arise in populating a knowledge graph from structured data, i.e., the problems of record linkage and schema mapping, and the problems that arise while populating from text, i.e., entity extraction and relation extraction.

2. Knowledge Graph Design

Both property graph and RDF data models have a set of design issues some of which are common across the two, while others are unique. For example, both models need to use reification for situations that cannot be directly modeled using triples. An **RDF model** needs to adopt a scheme for IRIs which is not necessary for property graphs. In a **property graph model,** we need to decide whether a value should be represented as a property or as a node, while this distinction is unnecesary in an RDF model. In this section, we will present an overview of such design issues that are faced in each of these two models.

2.1 Design of an RDF Graph

The knowledge graph authoring guidelines for RDF data on the WWW are known as the linked data principles as outlined below.

1. Use URIs as names for things.
2. Use HTTP URIs so that people can look up those names.
3. When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL).
4. Include links to other URIs, so that they can discover more things.

We will consider each of these guidelines in greater detail.

2.1.1 Use URIs as names for things

To publish a knowledge graph on the WWW, we first have to identify the items of interest in our domain. They are the things whose properties and relationships, we want to describe in the graph. In WWW terminology, all items of interest are called resources. **The resources are of two kinds:** information resources and non-information resources. // All the resources we find on the traditional WWW, such as documents, images, and other media files, are information resources. But many of the things we want in our knowledge graph are not: People, physical products, places, proteins, scientific concepts, etc. As a rule of thumb, all "real-world objects" that exist outside of the WWW are non-information resources.

The publishers of knowledge graphs should construct the URIs to be shared in a way that they are simple, stable and manageable. Short, mnemonic URIs will not break as easily when sent in emails and are in general easier to remember. Once we setup a URI to identify a certain resource, it should remain this way as long as possible. To ensure long-term persistence, it is best to keep implementation-specific bits and pieces such as ".php" and ".asp" out of the URIs. Finally, the URIs should be defined in a way that they can be fully managed by the publisher.

2.1.2 Use HTTP URIs so that people can look up those names

We identify resources using Uniform Resource Identifiers (URIs). We restrict ourselves to using HTTP URIs only and avoid other URI schemes such as Uniform Resource Names (URNs) and Digital Object Identifiers (DOIs).

The process of looking up names is referred to as URI dereferencing. When we dereference a URI for an information object, we expect to get the representation of its current state (e.g., a text document, an image, a video, etc.) But, when we dereference a non-information resource, we can obtain its description in RDF expressed in an XML notation.

2.1.3 When someone looks up a URI, provide useful information using RDF and SPARQL

When someone looks up a URI, the provider should return a knowledge graph in RDF. The data should reuse standardized vocabularies to name the IRIs used in describing the RDF data. Several useful vocabularies are available for describing data catalogs, organizations, and multidimensional data, such as statistics on the Web. An open source effort called Schema.Org publishes community created open source vocabularies for open use over the web. We consider a few examples of such vocabularies.

The following RDF data describes a snippet of the organizational structure of the UK Cabinet office.

```
@prefix uk_cabinet: <http://reference.data.gov.uk/id/department/>
uk_cabinet:co rdf:type org:Organization
uk_cabinet:co skos:prefLabel "Cabinet Office"
uk_cabinet:co org:hasUnit uk_cabinet:cabinet-office-communications
uk_cabinet:cabinet-office-communications rdf:type org:OrganizationUnit
uk_cabinet:cabinet-office-communications skos:prefLabel "Cabinet Office
Communications"
```

```
uk_cabinet:cabinet-office-communications org:hasPost uk_cabinet:post_246
uk_cabinet:post_246 skos:prefLabel "Deputy Director, Deputy Prime
Minister's Spokesperson"
```

In the data above, the first triple uses the class *org:Organization* from the Organization ontology. The second triple uses the relation *skos:prefLabel* drawn from the SKOS ontology. SKOS stands for a Simple Knowledge Organization System, and provides a few commonly useful relations such as *skos:prefLabel* for describing data. In this case, *skos:prefLabel* simply allows us to associate a text label with *uk_cabinet:co*. The third triple uses the relation *org:hasUnit* from the Organization ontology to describe a unit within the UK Cabinet office. Next two triples make additional assertions about this unit. The sixth triple uses the *org:hasPost* relation to describe a position with in a department, and the final two triples give additional information about that position.

It may not be always possible to find pre-existing vocabularies that can be used in creating an RDF data set. If creating a new vocabulary becomes necessary, one should ensure that it is documented, self-describing, has a versioning policy, is defined in multiple languages, and is published by a trusted source so that the URIs used in it persist for a long period of time. We say that a vocabulary is self-describing if each property or term has a label, definition and comment defined.

2.1.4 Include links to other URIs, so that they can discover more things

While publishing data using RDF one should provide links to other objects so that its usefulness increases. **There can be three kinds of links:** *relationship links*, *identity links*, and *vocabulary links*. We will consider an example of each of these kinds of links.

Relationship links point at related things in other data sources such as other people, places or genes. For example, relationship links enable people to point to background information about the place they live, or to bibliographic data about the publications they have written. In the triple below, we show a link in which a person in one data set is asserted to be based near a geographical location that is specified using a URI in another data set.

```
@prefix big: <http://biglynx.co.uk/people/>
@prefix dbpedia: <http://dbpedia.org/resource/>
big:dave-smith foaf:based_near dbpedia:Birmingham
```

Identity Links point at URI aliases used by other data sources to identify the same real-world object or abstract concept. Identity links enable clients to retrieve further descriptions about an entity, and serve an important social function as they enable different views of the world to be expressed on the WWW of Data. It is a standard practice to use the link type *http://www.w3.org/2002/07 /owl#sameAs* to state that two URI aliases refer to the same resource. For example, if *Dave Smith* would also maintain a private data homepage besides the data that *Big Lynx* publishes about him, he could add a *http://www.w3.org/2002/07/owl#sameAs* link to his private data homepage, stating that the URI used to refer to him in this document and the URI used by *Big Lynx* both refer to the same real-world entity. A triple capturing this information is shown below.

```
@prefix ds: <http://www.dave-smith.eg.uk>
@prefix owl: <http://www.w3.org/2002/07/owl>
@prefix big: <http://biglynx.co.uk/people/>
ds:me owl:sameAs big:dave-smith
```

Vocabulary links point from data to the definitions of the vocabulary terms that are used to represent the data, as well as from these definitions to the definitions of related terms in other vocabularies. Vocabulary links make data self-descriptive and enable Linked Data applications to understand and integrate data across vocabularies. In the vocabulary link shown below, the class *SmallMediumEnterprise* defined by BigLynx is defined to be a subclass of the class *Company* in the DBpedia ontology. By making such a link, it is possible to retrieve various assertions about the class Company from the DBPedia, and use them with the class *SmallMediumEnterprise*.

```
@prefix dbpedia: <http://dbpedia.org/ontology/>
big:sme#SmallMediumEnterprise rdfs:subClassOf dbpedia:Company
```

2.2 Design of a Property Graph

The design of a property graph involves choosing nodes, node labels, node properties, edges and edge properties. The basic design questions are whether to model a piece of information as a property, label or as a separate object; when to introduce relation properties; and how to to handle higher arity relationships. We will illustrate the process of making these choices using examples.

2.2.1 Choosing Nodes, Labels and Properties

In a property graph model, the nodes usually represent entities in the domain. If we were interested in representing information about people, we will create a node for each individual person (e.g., John), and associate the label Person with that node.

There are **several considerations** in making further choices of node labels, node properties and edges. These considerations include: naturalness of labels, whether the labels might change over a period of time, runtime query performance, and the cardinality of values.

To illustrate the choice of whether to model a piece of information as a label, property, or as a separate object, consider the task of representing the gender of a person. We have three potential ways to capture this information: we can create *:Male* and *:Female* as labels and associate them with the *Person* nodes; (2) we can create a property called "gender", and associate it with *Person* nodes and allow it to have the values "male" and "female"; (3) we can create a *Gender* object, associate it with *Person* using a *has_gender* relationship, and give it a property called "name" that can take "male" and "female" as values.

The labels in a property graph model are used to group nodes into sets. All nodes labeled with the same label belong to the same set. Queries can work with these sets instead of the whole graph, making queries easier to write and more efficient. A node may be labeled with any number of labels, including none, making labels an optional addition to the graph. As a label groups nodes into a set, it can be viewed as a class. The question of whether to introduce a new label can be restated as whether to introduce a new class?

Creating new classes *Male* and *Female* vs introducing a node property "gender" that can take two values of "male" and "female" captures the same information. In general, whenever a phrase naturally occurring in language is frequently used in a domain, it is a candidate to be introduced as a class as long as the membership in the class does not change with time. As some implementations optimize the retrieval based on the use of labels, the use of labels can result in fast performance on queries that need to filter the results based on the membership in the class. If class membership changes with time, neither a label nor a node property value is an appropriate choice, and we need to use a relation. We will consider this in the next section.

2.2.2 When to introduce Relationships between Objects

For situations that could be modeled either by using a node property or by introducing a separate object and relationship, there are, at least, **two different considerations**. First of those considerations was introduced in the previous section: the membership in the class changes with time. The second consideration arises when we wish to achieve better query performance. We will consider these situations in greater detail next.

Continuing the example from the previous section, when the gender of a person could change over a period of time, then our only choice is to capture the information as a separate *Gender* object that is related to *Person* using the *has_gender* relationship. We can then associate a relationship property with the *has_gender* relationship that indicates the time duration for which that particular value of gender holds. Creating a separate *Gender* node would, however, lead to a huge number of edges which is wasteful as for most people the gender does not change. In such a situation, a combination of the two solutions might be desired where for most people the gender is represented as a node property value, but for a small fraction of people, it is represented as a relation property value on a relation to a separate *Gender* node.

Let us consider a situation where better query performance is a key consideration. Suppose we wish to model movies, and their genres. In one design, for a node of type *Movie*, we can introduce a property "genre" that can take values such as "Action", "SciFci", etc. In another design, we can introduce a new node type *Genre* that has a node a property "name" that can take values such as s "Action", "SciFci". We will then relate a node of type *Movie* with a node of type *Genre* using the *has_genre* relationship. In general, we can associate more than one genre with a movie. Suppose we wish to query for those movies that have at least one common genre. In the first solution in which we **use the node property "genre"**, this query would be stated in Cypher as follows:

```
MATCH (m1:Movie), (m2:Movie)
WHERE any(x IN m1.genre WHERE x IN m2.genre)
AND m1 <> m2
RETURN m1, m2
```

When we **model genre as a separate object**, the same query can be stated as follows:

```
MATCH (m1:Movie)-[:has_genre]->(g:Genre),
      (m2:Movie)-[:has_genre]->(g)
WHERE m1 <> m2
RETURN m1, m2
```

In the second query above, we are able to more directly make use of graph patterns, and in some graph engines, this query has a faster runtime performance because of indexing on relations. Hence, in this case, one has to choose between the two designs depending on the kind of queries that will be expected.

2.2.3 When to introduce Relationship Properties

We have already seen an example of a property associated with a relationship to deal with situations when the relationship changes with time. Other situations in which it makes sense to introduce properties with relationship include associating weights or confidence with a relationship or to associate provenance or other meta data with a relationship.

Some graph engines do not index based on relationship properties. If the use case is such that much of the query evaluation can be done without using the relationship properties, and they are required only for final filtering of the results, one may not pay significant performance panelty because of lack of indexing. If access to relationship properties is central to query performance, it is better to reify the relation as we will discuss in the next section.

2.2.4 Handling non-binary Relationships

We often need to model relationships that are not binary. A common example of such a relationship is the *between* relationship that given objects *A*, *B* and *C* captures that *C* is between *A* and *B*. A standard approach to capturing such higher arity relationships in a graph is *reification*. We have previously discussed *reification* in the context of RDF, but this technique is equally useful and desirable for property graphs. To capture the *between* relationship we introduce a new node type, *Between_Relationship* that has two properties: *has_object* (with values *A* and *B*) and *has_between_object* (with value *C*). We can use reification for relationships with any arity by creating a new node type for the relation, and by introducing node properties for the different arguments of that relation.

3. Summary

In this chapter, we considered the design of the graph data model for both RDF and property graphs. The data model design concerns such as whether to reify a relationship, handling non-binary relationships, etc., are common across the RDF and the property graph data models. The choice of whether to use a property vs a relation is unique to the property graph data model. The RDF model provides explicit guidelines on the use of IRIs, reuse of existing vocabularies, and making links across vocabularies. Even though the data linking considerations are not integral to the property graph model, but their use can make a property graph system more useful in data integration.

Exercises

Exercise 3.1.Which of the following statements about knowledge graph design are true?

(*a*) As knowledge graphs are schema free, no design of the schema is required.

(*b*) Knowledge graphs are always created using automatic techniques.

(*c*) For many knowledge graph applications, a perfect accuracy is not a hard requirement.

(*d*) Knowledge graphs can contain undirected relationships.

(*e*) Knowledge graphs do not use keys and foreign keys as defined for the relational database systems.

Exercise 3.2.Which of the following is a good choice of an IRI for an RDF knowledge graph?

(*a*) ISBN-13 : 978-1681737225

(*b*) http://fcvcz.abt.co/mckz/

(*c*) https://www.wikidata.org/wiki/Q6135847

(*d*) http://worksheets.stanford.edu/homepage/index.php

(*e*) http://dbpedia.org/resource/Frederick_Loewe

Exercise 3.3.What type of link is captured by each of the following RDF statements? (Assume the following prefixes have been defined.)

@prefix dbpedia: http://dbpedia.org/resource/
@prefix bbc: http://www.bbc.co.uk/nature/species/
@prefix umbel-rc: https://umbel.org/umbel/rc/Person
@prefix foaf: http://foaf.org/

(*a*) dbpedia:Aardvark owl:sameAs bbc:Aardvark

(*b*) dbpedia:Lady_Gaga skos:broader_of dbpedia:Lady_Gaga_audio_samples

(*c*) dbpedia:Tetris foaf:isPrimaryTopicOf wikipedia-en:Tetris

(*d*) dbpedia:Person rdf:subClassOf umbel-rc:Person

(*e*) dbpedia:Sky_Bank foaf:homepage http://www.skyebankng.com/

Exercise 3.4.Which of the following are good class labels in a knowledge graph?

(*a*) Customers with overdue accounts

(*b*) Australian Customers

(*c*) Customers with revenues between 5 to 10 million

(*d*) Customers who supply to recently funded startups

(*e*) High Networth Value Customers

Exercise 3.5.Which of the following requires reification for representing in a knowledge graph?

(*a*) John believes that life is good.

(*b*) John was referred to Peter by Mary.

(*c*) The effectiveness of a vaccine is 95%.

(*d*) Earth revolves around the Sun.

(*e*) On LinkedIN John rated Peter for being an expert in AI.