ontotext

LPG to RDF Guide

Turning Your Property Graph into a Robust Knowledge Graph

A step-by-step guide to transitioning from a proprietary labeled property graph (LPG) technology towards a standards based, semantics powered RDF knowledge graph

Borislav Iordanov

EXECUTIVE SUMMARY

Starting with the observation that enterprises embarking on the knowledge graph journey via a labeled property graph backend quickly realize its shortcomings and face the necessity to migrate to a standardized semantics-based tech stack, this paper offers a step-by-step guide for performing precisely such an upgrade towards a true EKG (Enterprise Knowledge Graph).

Each step is detailed with supporting examples and some theoretical background. First, we cover the data modeling effort - how the implicit schema in a property graph leads to an explicit semantic model. We address some of the fundamental differences between the two technologies, but, as the Semantic Web stack is more expressive, the ride is relatively smooth. Then we deep dive into converting the raw graph data into an actual knowledge graph following an ontology. Every property graph data construct is directly mapped to an equivalent RDF construct. We also demonstrate some tooling for the automation of that ETL process. Lastly, we show how queries in the Cypher query language can be systematically converted to equivalent SPARQL queries, offering some examples and discussing philosophical differences between the two languages.

While we assume only minimal exposure to the Semantic Web stack, this guide does not aim to be a tutorial on this technology. It can be read with only a superficial understanding of the Semantic Web to get a feel of what is possible and the scope of the effort, or it can be read as a practical guide to follow.

TABLE OF CONTENTS

INTRODUCTION

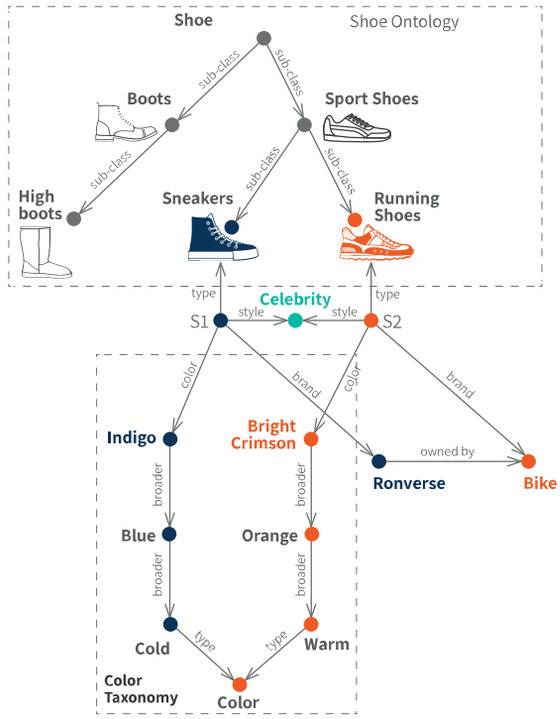
As graph technology is penetrating the mainstream rapidly, the confusion between property graphs and knowledge graphs persists, despite multiple attempts to dispel it. Due in part to the similarity of the underlying technologies and to a large extent to marketing efforts from vendors trying to co-opt the buzz around knowledge graphs, the result of this confusion is that, frequently, engineers turn to a property graph database when what they really need is a knowledge graph solution. Time and again in my consulting engagements I've run across precisely this scenario where a proof-of-concept was accomplished with a property graph - but attempts to take it to the next level run into its well-known limitations. Ultimately, a decision is made to migrate to a proper knowledge graph.

This kind of situation occurs in the typical business environment where either a complex domain model is required, or there is a large-scale effort to extract value from data across the enterprise or "simply" consolidate data silos in a meaningful way. In fact, most business cases are much better served by a knowledge graph than a property graph. Why is that? There are many reasons - from well-established standards, a whole industry of competing vendors, and a rich tooling ecosystem to a vast body of academic research and proven best practices, publicly available data models and the ability to define a schema. But ultimately the biggest reason is the fundamental idea that the value of data is in the knowledge it begets.

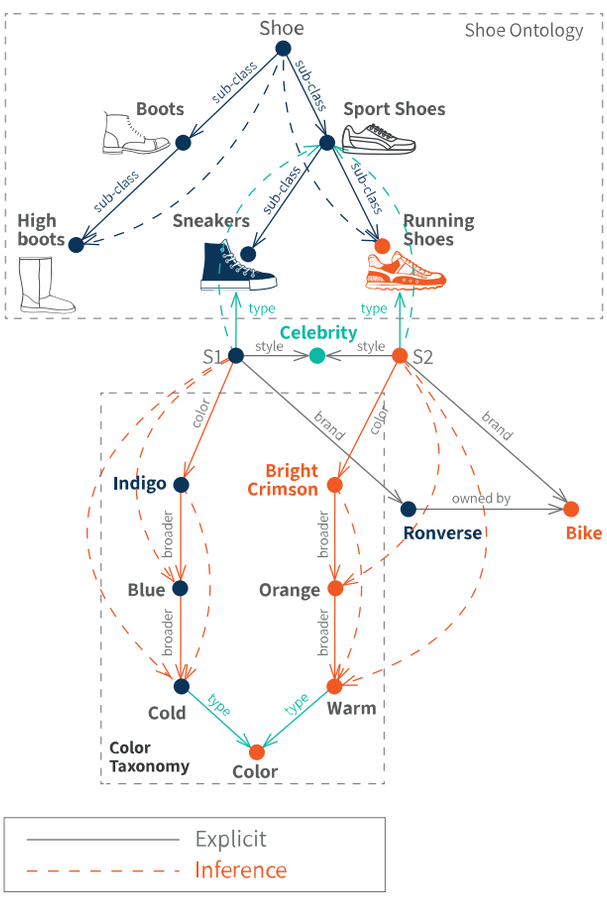
And folks are surprised to discover that property graphs are not simply another knowledge graph solution. No need to go deep into epistemological theory and try to answer the question "what is knowledge?" to understand why that is the case. On a much more practical level, there is a long tradition in knowledge representation and reasoning in computer science, a sub-field commonly understood as AI in the last century (today sometimes referred to as GOFAI for Good Old Fashioned AI) and the whole Semantic Web tech stack comes from that tradition. As a result, the tools at one's disposal are more mature in many substantive ways and enterprise data projects that aim at a more solid foundation chose to make the transition from their LPG (Labeled Property Graph) prototype to a triple store (a.k.a. semantic database engine) built on RDF (Resource Description Framework) and related W3C semantic standards.

Plain Graph

Knowledge Graph

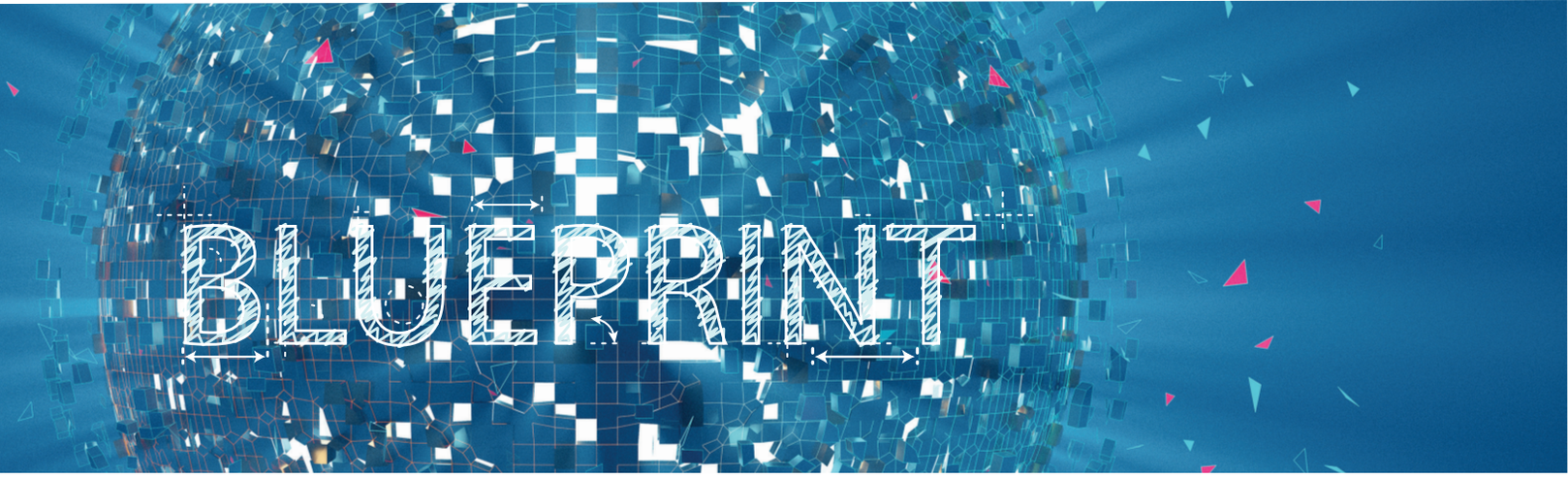


Knowledge Graph with Inference



As illustrated in the diagram above, the fundamental difference between an arbitrary graph data structure and a knowledge graph is that the latter includes a knowledge model, which then affords automated reasoning capabilities one may decide to take advantage of. Turning data into knowledge requires an explicit knowledge model, an ontology, that can combine a conventional data schema with other types of topical or terminological knowledge: taxonomies, controlled vocabularies, domain models and business rules. It's critical for such a knowledge model to come with well-defined semantics that allows reasoning and unambiguous interpretation by both machines and humans. This is everything that the labeled property graph technology stack is short of.

In what follows, I will show what it takes to make that jump from a property graph to a semantic knowledge graph. If you are wondering whether you would be losing something, how hard it would be, if it's really worth the effort, I hope to answer those questions at a technical enough level.



Migrating an LPG to a knowledge graph is a data migration and a technology migration project. When stated like this it sounds like a major endeavor, but it is probably the most straightforward project in that broad category you will ever undertake.

The steps are these:

- Create a semantic model for your data: the initial model doesn't need to be too deep, but it will get you on the right track. A short tutorial on RDF/OWL basics will teach you enough to get started.
- Translate the LPG data model to the semantic one: we will go into the subtleties of that below.
- Move data from LPG to the semantic store: there are tools to help - I will show you some.
- Translate your LPG queries to SPARQL: while not that difficult, full coverage of all language constructs is a separate document so I will cover only the critical aspects on a query of moderate complexity.
- Translate your code based on the LPG DML/Query language to SPARQL: this is outside the scope of this guide.

I assume the readers are fairly familiar with LPGs (e.g., used Neo4j). Superficial familiarity with RDF will ease following the guide, but is not a prerequisite.

When giving concrete examples, I will use the domain of movies, actors and directors as it has served as the example of choice in many graph tutorials, including as the main out-of-the-box example of the LPG solution with the largest market share - Neo4j.

I use the terms property graph and LPG as synonymous referring to the technology at large. Similarly, I will refer to the whole Semantic Web tech stack, which includes RDF, RDF Schema, RDF Star (RDF*), OWL, SHACL, SPARQL, and triple store/linked data endpoints as just "RDF" or knowledge graph. All of these specs make up the full Semantic Web technology stack, they work well together and are typically supported by all major vendors, often with a powerful reasoning engine and if-then rules.

Data Modeling - from LPG to RDF

A data model in an LPG database is implicit as there is no schema language that allows you to specify one. The graph-oriented data structure that property graphs implement is by itself a powerful tool for storing complex data, but the technology offers poor, if any, data modeling capabilities. And while there are some vendor-specific constructs to express some limited constraints, data integrity is one of the significant weaknesses of this type of graph technology. For example, a node carrying a particular label does not guarantee anything about its properties.

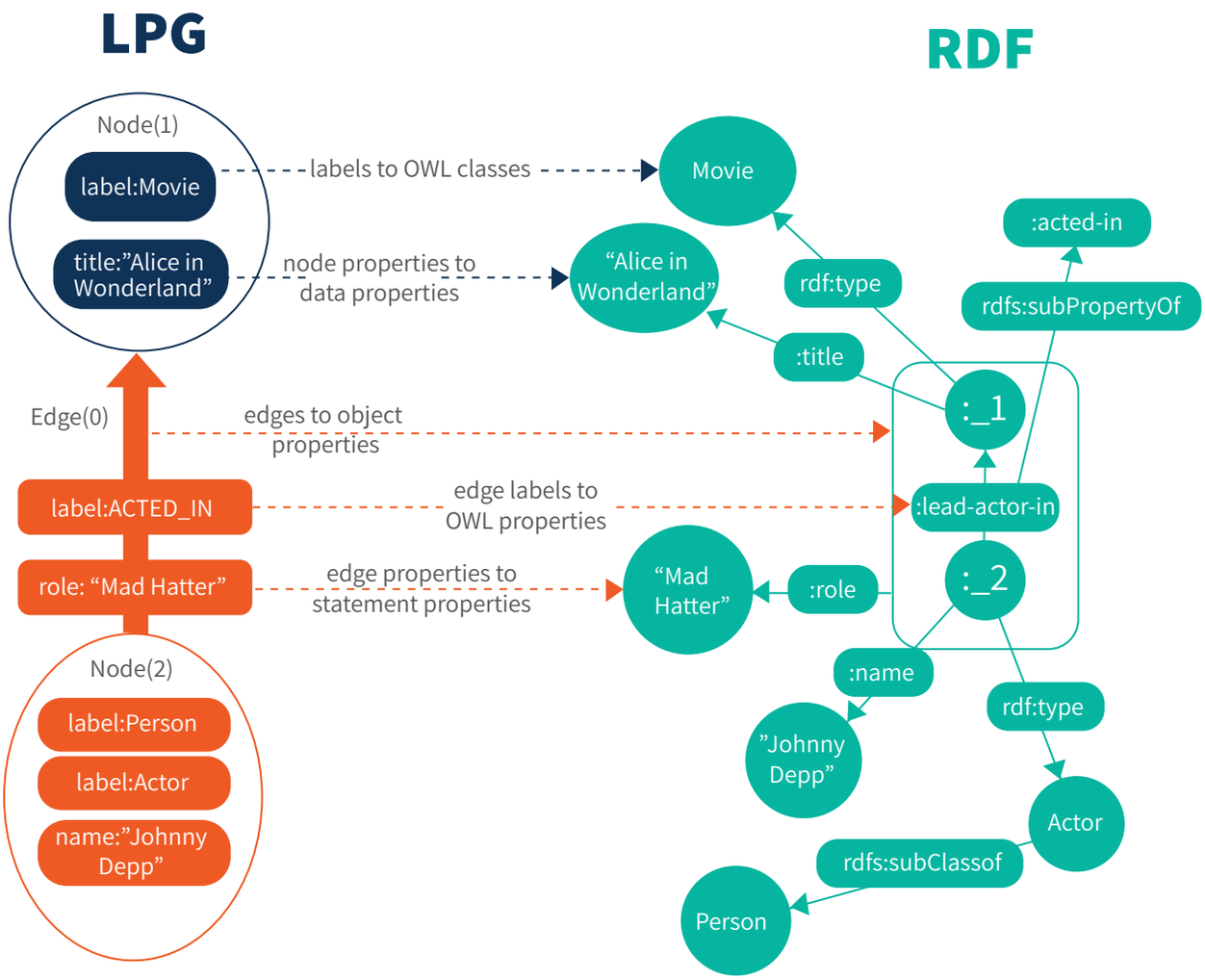
That said, there are some reasonable expectations of the data in an LPG and having specific properties associated with a label is, for example, one of them. In general, in a typical LPG data model:

- There is a set of predefined labels for nodes and edges roughly thought of as "types".
- The node labels typically correspond to entity types while the edge labels model relationship types and those two do not overlap: a given label is either for nodes or edges, but not both.
- When a node has some label L, one anticipates it will also have certain properties. For example, a node with the label Movie would be expected to have properties like title, release-date, etc.
- Similarly, when an edge has some label E, one expects the edge to carry a number of properties and, in addition, that its endpoints are of a particular type (e.g., have some labels). For example, an edge labeled "DIRECTED" would entail (again, this is not enforced by the database) that the source node is a Person and the target a Movie.
- Properties of nodes and edges can have values of any number of standard primitive types (in LPG they are not standardized, but in RDF they are following the XML Schema standard)[1].

On the RDF side, the plain RDF statement (a.k.a. triple) of Subject-Predicate-Object is just a deceptively simple foundation. To create a meaningful model, one needs to go beyond that basic construct of the triple, but luckily one quickly gets into familiar territory. While the RDF triple is just a graph edge with endpoints Subject and Object, the Predicate here is more than a mere label. Labels in RDF are human-readable names attached to various things while the predicate is a universally identified "edge type" of sorts to which one can associate constraints or attach several multilingual labels. One such constraint is whether the expected Object part of the triple should be a primitive value (a "literal" in RDF terminology) or some other Subject. And thus you get into the world of building a model of what the graph should look like.

Naturally, one can also avoid creating a data model in advance, but it is a good idea and there are several tools that make it easy. To take full advantage of the technology, one should go up the stack with OWL and SHACL. You don't need a full-fledged OWL ontology to begin with, but you can at least state in advance what types of entities and what types of relationships make up your domain. And you can do it in a text file - it is simply an RDF file and follows the same RDF language you'd use to store and query your data. That is, RDF data is self-describing, a concept rather foreign to the LPG world[2].

Even without using the OWL language in its full description logic glory, I'd recommend building your model in OWL[3]. At a minimum, you will get the familiar vocabulary of classes, inheritance, instances and properties. In the semantic knowledge graphs world, we call ***properties*** both the data attributes associated with an entity and the relationships it has with other entities. We refer to them as ***data properties*** and ***object properties*** respectively. You'd probably also want to take advantage of data constraints, which have their own special language called SHACL (Shapes Constraint Language). SHACL is what will give you data integrity and what you typically think of as a schema language. OWL is for representing knowledge about domains of interest and less about data properties. There are some silly fights in the knowledge graph community SHACL vs OWL, but don't pay attention to that. They complement each other and the combination of the two is incredibly powerful as long as one keeps in mind that OWL is about representing domain knowledge while SHACL is for data modeling. Before moving into a detailed explanation of how the different LPG elements translate into RDF, let us get a visual impression of it. Consider a small fragment from the movies domain where we have a particular movie, say "Alice In Wonderland" - with a specific actor, say Johnny Depp, playing the role of the Mad Hatter. The following diagram illustrates what this little fragment looks like in an LPG (left-hand side), what it looks like in an RDF knowledge graph (right-hand side) and how the mapping from LPG to RDF goes:



This diagram demonstrates a few key modeling differences between LPG and RDF:

1. **RDF graphs are more fine-grained:** two nodes and a single edge in LPG are represented by five nodes and six edges in RDF, even excluding the schema part.
2. **In RDF, the data model is represented in the graph in the same way as entity data:** classes and properties appear as nodes in the graph. In LPG there is no formal schema - node and property labels are just a bag of strings with no semantics.

At first glance, the verbosity of RDF may look like a high price to pay for an overly abstract data model. However, this characteristic of RDF allows one to have schema and metadata definitions, which are as complex as necessary, and to be able to access those via the same query language used to manage the data. Anyone who has had to build and operate knowledge graphs, which combine diverse data from multiple sources, knows that this capability is priceless.

Now, looking at the diagram, we can read out the mapping between a property database to an RDF database. Using the OWL vocabulary of classes, instances, object properties and data properties:

- Nodes become OWL instance entities (also known as ***individuals***), ultimately just nodes in the underlying RDF graph.
- Node labels become OWL classes.
- Node properties become data properties. At a low level, those are RDF triples where the Object part is a literal value of some primitive type.
- Edges become object properties. Those are triples where the Object part is another individual entity.
- Edge labels become OWL properties definitions.
- Edge properties become statement properties. Those are RDF Star triples where the Subject part is the triple representing the main edge.

We will go deeper into each of those mappings, but before that a quick note about

IDENTIFIERS

In the LPG world, identifiers of nodes and edges are something the database will be very possessive about - it's an internal thing and it will look differently from database to database. If you want something more universal, you must roll your own "property with a uniqueness constraint." In the knowledge graph world, you construct your own identifiers and ensure their universality following standard URI (Unified Resource Identifier) syntax or IRI to include non-latin characters. URIs are either the well-known URLs, the standard locators for web pages, or URNs - globally unique names without means for locating a resource such as the ISBN book identifiers (e.g., urn:ISBN:0-395-36341-1). These URIs are used for everything, and I mean ***everything*** that is not

primitive value: class names, property names, instances. It's a decision point always leading to fun debates about how opaque these URIs should be - do we include the node type, do we use some identifying property or just append a UUID to a company namespace prefix and be done with it? When importing an LPG graph you could always use the internal node ID as the unique portion of a URI, but I would recommend devising a new sensible long-term schema altogether.

URIs tend to be lengthy and in the various textual representations of RDF, XML style prefixes are used, including often the empty prefix so you see things like :Person instead of http://www.example.com/ontology/Person.

Now, note that these identifiers apply to the entities of your data and the entities of your model (which is also data). Relationships don't have their own identifiers since they are fundamentally a statement and their essence is really the whole tuple (subject, predicate, object). You can't enter multiple copies of the same statement - it doesn't make sense logically. And to refer to a statement in RDF Star, you just spell it out completely. In the example below, "Bob lives in Miami" is called ***a quoted triple*** and it looks like this:

```
<< :Bob :lives-in :Miami >> :since "2000-12-08"^^xsd:date
```

If you find the need to have duplicate edges because you want different properties for each, it probably means you are semantically treating relationships as entities, which is perfectly valid, but then you might as well model them as such instead. For example, to state that Bob has worked for Carnival Cruise Line on multiple occasions, one cannot write:

```
<<:Bob :worked-at :Carnival-Cruise-Line>>
    :from-date "2001-05-04"^^xsd:date;
    :to-date "2010-11-01"^^xsd:date .

<<:Bob :worked-at :Carnival-Cruise-Line>>
    :from-date "2015-01-01"^^xsd:date;
    :to-date "2019-12-15"^^xsd:date .
```

Because all of those dates will be associated with a single <:Bob :worked-at :Carnival-Cruise-Line> relationship, not with two separate copies of it. Instead, you are probably thinking about an entity of type Employment-Event that has a certain time span and a certain set of participants:

```
:bob-employment-1 a :Employment-Event ;
  :from-date "2001-05-04"^^xsd:date;
  :to-date "2010-11-01"^^xsd:date ;
  :employee :Bob ;
  :employer :Carnival-Cruise-Line .

:bob-employment-2 a :Employment-Event ;
  :from-date "2015-01-01"^^xsd:date;
  :to-date "2019-12-15"^^xsd:date ;
  :employee :Bob ;
  :employer :Carnival-Cruise-Line .
```

NODE LABELS TO CLASSES

A node label in an LPG almost certainly represents the type of the node and can simply be translated into an OWL class. For example, for the label Person in the movie data, we could declare:

```
:Person a owl:Class
```

Since labels don't have any inherent semantics, it is theoretically possible to use them with a different intent, e.g., as facets when tagging content. Cases like this arise in practice when end-user input gets translated to a node label. It would be unwise to still map such labels to classes for these situations. Instead, one should create a dedicated OWL property, say :facet, and store the open-ended labels as properties - either plain text like the LPG labels or strongly typed entities like categories in a taxonomy. Or pick up an existing industry standard like SKOS[4], which is widely used to model taxonomies and controlled vocabularies in RDF.

Note that an entity in OWL can belong to any number of classes and can also contain any number of values of a given property. On the other hand, if you have multiple labels in your LPG model that form a conceptual hierarchy, in OWL you only declare the most specific class; the rest will be automatically inferred. For example, if you have the additional label Actor for people nodes, in a semantic graph you can declare:

```
:Actor rdfs:subClassOf :Person
```

You can declare a given person to be an actor and it will automatically be a person as well (no offense to non-person actors such as Pal, the dog who played Lassie and reportedly earned more money than co-star Elizabeth Taylor).

NODE PROPERTIES

A key aspect of RDF properties is that they are global. Typically, this is not an issue as property names in a single LPG database will tend to share datatype and intent. If you want to take advantage of validation and data integrity capabilities, you'd want to be mindful of defining constraints in the appropriate classes. You can always leave the property values unconstrained, but it's better to maintain unambiguous semantics for a given property. If your LPG data set uses the same property name for widely different things in different places, create separate RDF properties for each type of usage. For example, say you have the property name "favorite", which in some cases contains the name of a favorite character, but in others, it's a number indicating how many people marked a movie as a favorite. A semantically clean approach would be to create two properties, e.g., "favorite-character" and "favorite-count".

EDGES TO OBJECT PROPERTIES

Edges between LPG nodes are translated into ***object properties*** in RDF, i.e., properties whose values are entities, properly identified with a URI, instead of plain values (known as "literals" in the RDF world).

Note that there is a bit of terminological ambiguity. When we speak about object properties in OWL, we may be referring to the schema level construct where we'd express constraints about things like domain and range. While in other cases we mean the data level properties, the predicate-value portion of an RDF statement about an individual subject.

An edge in RDF, just as in LPG, has only one label - the URI of the property. However, OWL, being a knowledge representation language, has the extra ability to organize properties hierarchically. For instance, in OWL, you could declare that:

```
:lead-actor-in rdfs:isSubPropertyOf :acted-in
```

to easily separate lead actors in a movie from supporting ones. Just like with classes, this is a domain modeling statement - you only do it once and you don't need to create two separate edges, one for actor another for lead actor in order to be able to query for one or the other. In other words, querying for people who "acted-in" some movie will return both lead actors and actors while querying for those who "lead-actor-in" will return only the lead actors.

Edges in RDF do not have identifiers (the edge labels are URIs, yes, but the edges themselves are id-less). Instead, to refer to an edge in a statement or a query, one has to spell out the

whole relationship with both of its ends. This may sound a bit too verbose, but it neatly preserves the intuition that the essence of a relationship is the whole of: subject, predicate and object, an irreducible construct. Naturally, there are syntactic shortcuts to alleviate the verbosity.

THE STAR IN RDF* - PROPERTIES ON EDGES

But where would you want to refer to an RDF edge you might ask? When you want to say something about it. In other words, whenever you want to attach properties to an edge. A standard feature of LPGs, this capability is a relatively new addition to the Semantic Web standards (RDF Star[5]), yet a very powerful one, going way beyond property graphs. Which highlights how much more solid and versatile the Semantic Web foundation is.

Unlike property graphs, attaching metadata to relationships in RDF is not limited to primitive types - one can attach any property whatever, including other entities. For instance, there could be an edge property "introduced by", the value of which is the node in the graph representing a user who asserted the relationship. An RDF relationship can freely participate in other relationships and do so recursively at any nesting depth. A primary use case is the formidable problem of provenance that nearly all modern enterprise systems face and are barely scratching the surface of solving.

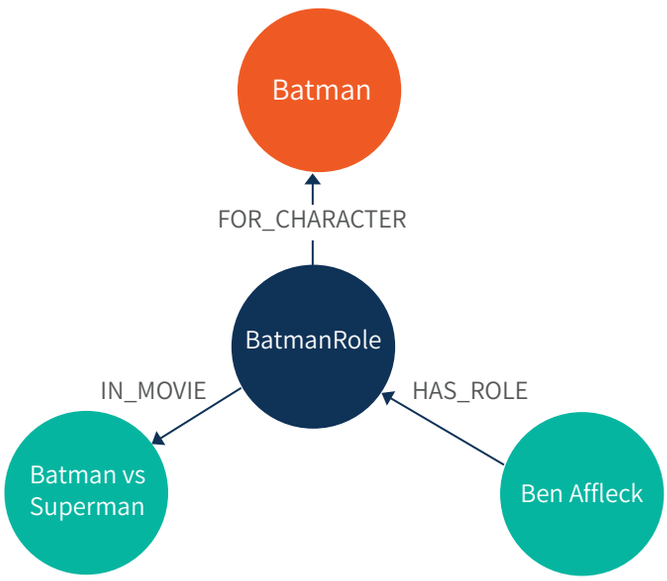
Furthermore, one can add information about a relationship without adding the relationship itself. That is, in an LPG database, any relationship you want to annotate with properties will have to be part of the database and it will be found by the query engine. By contrast, in RDF there is a distinction between quoted and asserted properties. Thus it is possible to express that "The Pope thinks that Bruce Willis acts in Dirty Dancing" without having to assert that Bruce Willis acts in Dirty Dancing (the quotation delimiters are << and >>):

```
:ThePope :thinks << :BruceWillis :acted-in :DirtyDancing >>
```

But as a more practical example, let's consider adding some information about the role an actor plays in a movie such as what character is being played and what the remuneration was. The remuneration part is easy. One would expect it to be a number valued property of the acts_in edge in LPG. And in RDF this translates to a number valued relationship property like this:

```
<< :BenAffleck :acted-in :BatmanVsSuperman>> :has-salary "10000000"
```

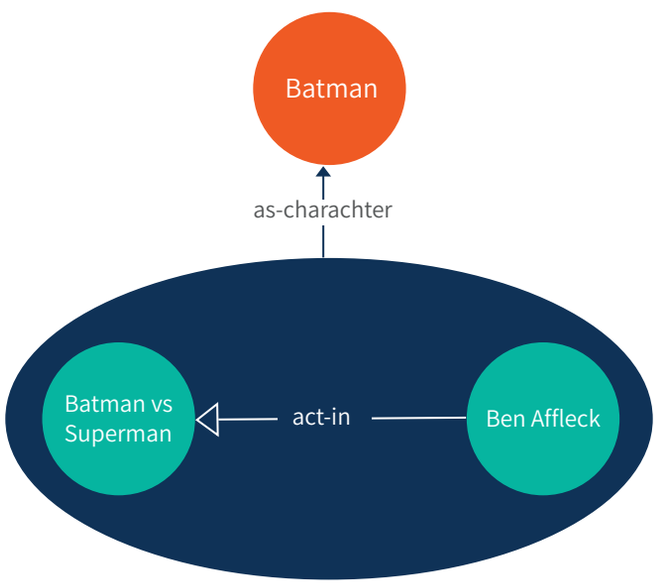
To capture the character played, we'd need something representing a character, say of class FictionalCharacter, and of course we imagine the same character being possibly portrayed in several movies, played by different actors. Then, one option would be to create a separate node for the role, say of class MovieRole and have the role be the connection point between actor, movie and character. In the LPG world, this would be arguably the only option. The graph motif capturing the whole thing would be like this:



As shown in the section on Identifiers above, this modeling pattern can be employed in RDF as well. However, since in RDF relationships can participate in other relationships, we can think of the role as a property of the acts_in relationship, which it kind of is. The same information would then be expressed like this:

```
<< :BenAffleck :acted-in :BatmanVsSuperman>> :as-character :Batman
```

That is, the relationship becomes a node in a "meta graph" so to speak:



Ironically, in some LPG vs RDF debates of years past, it has been argued that the inability to create edge properties puts RDF at a disadvantage. This was true - one had to use a reification modeling pattern where the relationship is represented as an entity. But in this case, with the advent of RDF Star, the opposite is happening - an LPG based model is forced to blow up in size due to the inability to have entities as metadata values of relationships.

EDGE PROPERTIES TO DATA PROPERTIES

Edge properties are converted in a very straightforward manner to data properties, which are then attached to complete RDF statements like we showed just above. Going back to the actor salary example as an edge property, the Cypher version would look like this:

```
CREATE (actor)-[r:ACTED_IN]->(movie) SET r.salary 10000000
```

In SPARQL, which is the standard query and manipulation language for RDF graphs, it goes like:

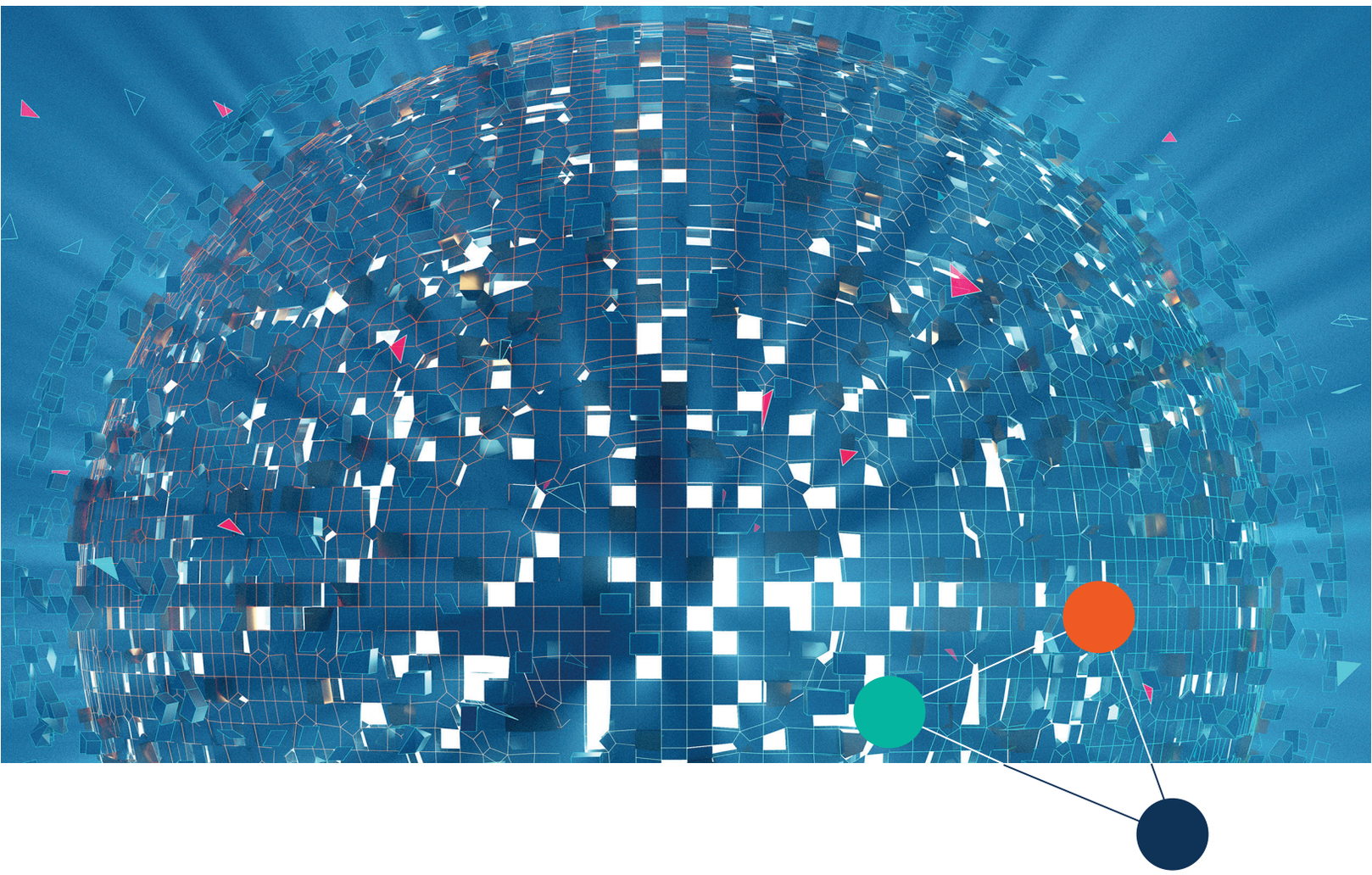
```
INSERT DATA { << ?actor :acted-in ?movie >> :salary "1000000"  }
```

Much of the details elaborated on above about how to translate information from LPG to RDF stem from the logical foundation of RDF - every property, every edge is a logical statement about something. You can assert such a statement, provided you have a way to "refer" to that something, and you make that assertion only once. Hence LPG node properties and labels, LPG edge properties and labels as well as LPG edges are ultimately simple assertions about things. This foundational aspect is what allows you to make assertions about edges-as-statements even if those edges are hypothetical, not actually stored in the database: whether a statement is asserted in the database or not, it's its own identifier - there can't be two like it.

Consequently, one can attach metadata to node properties and meta-metadata about them, etc. One can pretend the need for this sort of thing rarely occurs in practice and one would be almost right. Yet, when aspects like temporality or provenance become important, as they invariably do in an enterprise context, the need is there and screaming.

One other benefit of this declarative, logical aspect of RDF statements is that you never have to worry about inserting duplicate information in your knowledge graph. If you insert the same triple again in a triplestore, the system will do strictly nothing. It does not make

sense to do a “insert if not already there” in a knowledge graph - either something is known or not. And if not known, that does not make it not true .... but that’s another subject.



GETTING PROPERTY GRAPH DATA INTO A TRIPLESTORE

Different triplestore vendors might offer some tools for mapping and importing data into a knowledge graph. For example, the Ontotext Refine tool provides a GUI wizard[6], which includes a preview facility similar to the import function in Microsoft Excel and it can automatically generate SPARQL query for loading the data, wich you can then easily tweak.

To illustrate the mechanics of a typical import, let us look at something less advanced than Ontotext Refine - the open source command line tool Tarql[7], which also leverages SPARQL. More on SPARQL below, but the gist of it is that it’s a pattern-oriented language where each pattern match instantiates some variables. So what Tarql does is treat each row in the CSV input file as a pattern match. The first line of the file gives the names of variables and then each row instantiates their values. Then one uses those variables to construct the graph.

Imagine you have a CSV file of nodes with their labels and properties similar to the Neo4j's Movies sample database that we refer to as popular example for illustrative purposes only:

```
"id","labels","born","name","released","tagline","title","_start","_end","_type"
"1",":Person","1964","Keanu Reeves","","","",,, "2",":Person","1967","Carrie-Anne
Moss","","","",,, "3",":Person","1961","Laurence Fishburne","","","",,, ... "128",":Mov-
ie","","","1999","Walk a mile you'll never forget.","The Green Mile",,, "92",":Mov-
ie","","","2006","Based on the extraordinary true story of one man's fight for
freedom","RescueDawn",,, ...
```

There will be nodes with various labels there. And we can convert them to instances of the respective OWL classes like this:

```
# movies.tarql file
prefix movie: <http://www.example.com/ontologies/movie/>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>

# Turn nodes with label :Person into instances of the class Person
construct {
  ?thing a movie:Person ;
        rdfs:label ?name .
}
where {
  filter  (?labels = ":Person") .
  BIND(tarql:expandPrefixedName(CONCAT('movie:', ?id)) as ?thing) .
}

# Turn nodes with label :Movie into instances of the class Movie
construct {
  ?thing a movie:Movie ;
        rdfs:label ?title .
}
where {
  filter  (?labels = ":Movie") .
  BIND(tarql:expandPrefixedName(CONCAT('movie:', ?id)) as ?thing) .
}
```

The code above is given as the “query” input file to the Tarql tool, which also takes the CSV file as its second argument:

```
tarql movies.tarql data.csv
```

The construct portion is where the RDF graph structure is specified. It’s also a pattern that is instantiated as part of the output for every row in the CSV. Some variables (the stuff prefixed with ?) such as ?name and ?labels are taken from the first row of the CSV file while others such as ?thing are created in the where clause. The complicated looking BIND is constructing a full URI from the id column in the input by using the XML style prefixes declared at the beginning of the file.

You can consider implementing a translation of your own LP graph into RDF graph using such queries. That would require first to get your LPG extracted into a CSV file. The Cypher queries below are meant only to illustrate how such export can take place:

```
MATCH (left)-[rel]->(right)
WITH collect(DISTINCT left) AS origin, collect(DISTINCT right) AS destination,
collect(rel) AS relation
CALL apoc.export.csv.data(origin + destination, [], "movies-nodes.csv", {})
YIELD file, source, format, nodes, relationships, properties, time, rows, batchSize,
batches, done, data
RETURN file, source, format, nodes, relationships, properties, time, rows, batch-
Size, batches, done, data
```

A Cypher query analogous to the one above can export the edges, which you can then convert to object properties again using Tarql. I am omitting the details for brevity.

Now, let’s take a closer look at SPARQL and in particular at how to convert LPG queries to SPARQL.

CONVERTING LPG QUERIES TO SPARQL

The first LPG language was Gremlin, which was inspired from XPath and was all about traversing a graph. In fact the Gremlin/TinkerPop folks came up with the term “property graph” and postulated as its abstract definition for what was at the time a single property graph product - Neo4J. Gremlin still exists, but the main language now familiar to LPG developers is Neo4J’s Cypher and its derivatives, including the upcoming GQL standardization effort.

Cypher is fundamentally a pattern-oriented language just like SPARQL - it was in fact to a large extent inspired by SPARQL! Consequently queries will have a straightforward translation.

In both languages, one specifies a pattern, a graph structure with some variables that are instantiated by a match to the pattern and one specifies which variables to return. In Cypher it looks like this:

```
MATCH <graph pattern>
RETURN <result>
```

And the equivalent form in SPARQL:

```
SELECT <result> WHERE {
  <graph pattern>
}
```

The meat of the query will be the graph pattern - what structure are we looking for? A graph pattern is a subgraph where some of the nodes or edges are unknowns, represented as variables that we can then refer to in the result portion.

To formulate a graph pattern, both languages offer various syntactic constructs. SPARQL has a simpler, but sometimes more verbose syntax due to the simplicity of the underlying graph model. I will spare you an exhaustive list of Cypher constructs with their SPARQL counterparts (it's been done elsewhere - see ref). Suffice it to know that everything you can do in Cypher, you can do in SPARQL. To get a sense of the query migration process, let's parse a single moderately complex Cypher query and build an equivalent one in SPARQL.

First, here is the query in plain English:
Find all male actors who have played the same character in a movie and who have also worked with the same director while earning at least a seven figure salary.

Admittedly this is a bit contrived, but it will serve our purpose. Before looking at the Cypher and at the SPARQL versions, a few observations:

- We are searching for actors as the main entity type of interest and we are constraining them by their relationships with other things in the graph and by their properties.
- We are trying to match pairs of actors by some aspects in common. So our pattern will need two variables referring to two distinct entities of type actor.
- Since the fact that an actor is playing a given character in a movie is expressed in a fundamentally different way in the RDF model proposed above, we take advantage of the meta-level capabilities of RDF, the part of the queries constraining the actor roles will look very different.

In Cypher, this looks like

```
match (a1:Actor {gender:"male"})->[:HAS_ROLE]->
            (r1:Role)->[:FOR_CHARACTER]->(c:Character)<-[:FOR_CHARACTER]
-(r2:Role)<-[:HAS_ROLE]<-(a2:Actor {gender:"male"}),
  (a1)-[:ACTED_IN]->()<-[:DIRECTED_BY]-(d),
  (a2)-[:ACTED_IN]->()<-[:DIRECTED_BY]-(d),
where r1.salary > 1000000 and r2.salary > 1000000
```

We are using the variables a1 and a2 for the two actors, r1 and r2 for their respective roles, c for the common character and d for the common director. The first portion of the pattern is a long path connecting actor a1 to actor a2 via their roles of playing c. Then we list the linkage to a common director separately for each actor.

And here is the SPARQL version. While parsing through it, note that whereas in Cypher variables appear on the left-hand side of a colon with their label on the right-hand-side, in SPARQL variables are syntactically recognized from the ? prefix and labels don't have any special status. Instead, we have actual types (the labels are just a standardized property for user-friendly display…which is what labels really are when you think about it). The colon in front of various names is just the empty/default namespace. In a query with multiple, equally important namespaces, for example, if we have a knowledge graph integrating many domains, one would be spelling the namespaces - ***movies:Actor*** instead of ***:Actor,*** ***movies:as-character*** instead of ***:as-character*** and so forth.

```
select ?a1 ?a2 where {
            ?a1 a :Actor; :gender "male" .
            ?a2 a :Actor; :gender "male" .
  << ?a1 :acted-in ?m1 >> :as-character ?character ; :salary ?a1-salary .
  << ?a2 :acted-in ?m2 >> :as-character ?character ; :salary ?a2-salary .
            ?d ^:directed-by/^:acted-in ?a1, ?a2 .
            filter (?a1-salary > 1000000 && ?a2-salary > 1000000)
```

Let us now see how the two versions compare. In both, a graph pattern is a sequence of constructs "and-ed" together, each describing the form of a graph edge or a node or a longer path. In Cypher, they are separated by a comma and in SPARQL by a dot.

Where in Cypher you specify node properties in curly braces:

```
(a1:Actor {gender:"male"})
```

in SPARQL you specify them as a statement, the same way you specify an edge. As you may recall, conceptually RDF unifies attributes and relationships at some level - both are just properties stated as subject-predicate-object statements.

The semicolon ';' in SPARQL lets you list multiple properties for a given subject. For instance, the following constrains the variable ?a1 to be something of type Actor and also to have "male" as the value of the property gender:

```
?a1 a :Actor; :gender "male"
```

Notice how the node types as well as properties and outgoing edges are all specified with the same syntax in SPARQL. This sort of universality of syntax makes it both much easier to learn, more natural and more powerful.

To specify an edge in Cypher, you use dashes and arrows to "draw correctly" the graph structure in the correct direction:

```
(a1)-[:HAS_ROLE]->(r1)->[:FOR_CHARACTER]->(c1)

or equivalently

(c1)<-[:FOR_CHARACTER]-(r1)<-[:HAS_ROLE]-(a1)
```

In SPARQL, chaining in this sort of way would look like this:

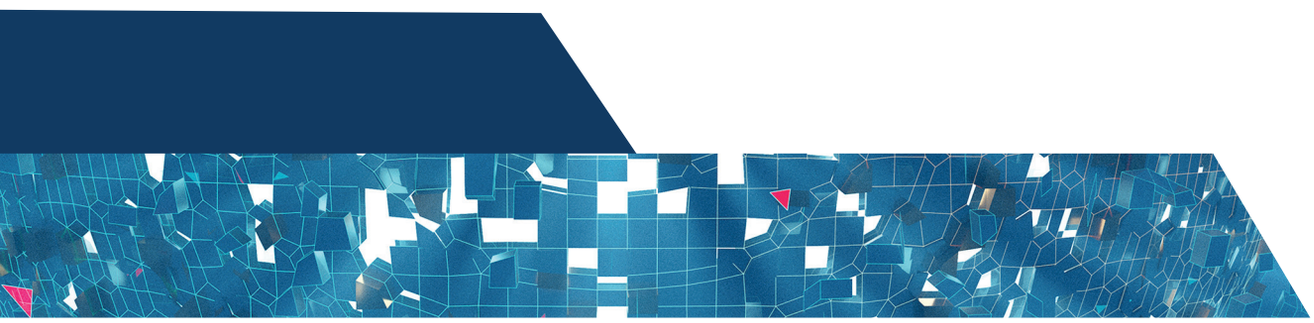
```
?a1 :has-role/:for-character ?c1

or equivalently, using ^ to reverse the direction of the edge

?c1 ^:for-character/^:has-role ?a1
```

Notice how in Cypher you would indicate the direction of the edge through the direction of the arrow <- or -> while in SPARQL you use the carat ^ to reverse the direction of a property.

If you are actually reading through the queries, you have probably noticed that in the SPARQL version the path ***?a1 :has-role/:for-character ?c1*** is absent. This is because our RDF-based model is different, more concise and it does not need the extra role node. We simply took advantage of the RDF Star meta-level capability to treat edges essentially as nodes and link them to anything we want. This illustrates an important aspect to migrating LPG data to RDF: since you will have the opportunity to make your RDF model richer, the queries may take advantage of that. In this case, to match the character that must be played by the two actors, in the LPG version the graph pattern requires two separate role nodes that link to the same character and the query is "ASCII drawing" the whole thing. In the SPARQL version, we are relying on the modeling variation described above where we don't need that intermediate role node because we can state it as a metadata on the acted-in edge itself.

The filtering conditions at the end of both queries look very similar and they behave the same way - filtering out matches that do not satisfy the logical expression. Notice however that in the SPARQL version, since everything is a graph pattern, separate variables have to be instantiated for the salaries.

Finally, it is worth mentioning that both Cypher and SPARQL support grouping and aggregation functions like SQL and with very similar syntax. No surprises there either.



CONCLUSION

Graph database technology is a mixed bag of tools, some standards based, others not, some oriented towards knowledge representation, others towards graph analytics. It turns out, in practice enterprises are still struggling mainly with data integration, data silos and just making sense of data in context. And there is only one long established, standards based technology stack that addresses these problems head on and that is the Semantic web stack. Also known as Linked Data or RDF, this is the technology that will enable seamless integration between data silos, rich semantic modeling, reuse of domain models and large scale interoperability.

In this guide, we have not elaborated on all advantages of the Semantic Web or the general trade-offs between the two technologies. That would be a subject for another document . There are some less obvious, unique to the RDF stack aspects that practitioners enjoy such as query federation - the ability to issue a query that spans multiple different end-points across the Internet or the fact that data schemas are as scalable as the data itself. But the main advantages that ultimately drive the migration towards RDF remain the much more solid conceptual foundation, the ability to have schemas in the first place as well as vendor-neutral standardization.

Once you have made the decision to go for a true knowledge graph, migrating from a labeled property graph to RDF is relatively straightforward. Create a domain model, potentially leveraging industry standards and prior work in the public domain or within your organization. This will establish a clean conceptual backbone of your data architecture and make sure there are no important business assumptions locked in the heads of a few experts. Then systematically convert the implicit data model, the data itself and your queries following the natural syntactic transformation we have outlined here. We skipped over some use cases such as interacting with APIs at the application level and updating the graph, but ultimately those are rather mundane and a matter of picking a convenient programming library.

When it comes to things like interoperability and meaning in the data economy, RDF knowledge graphs are the only game in town. Echoing Churchill's remark that democracy is the worst form of government except for all the others, RDF graphs are the worst Internet-scale data management technology except for all the others.

1. In GQL, they are standardized, but more aligned with the SQL standard.
2. For an overview of the whole technology stack in the form of short introductory articles, feel free to browse https://www.ontotext.com/knowledgehub/fundamentals/
3. In fact, using the DL profile of OWL 2 and everything more expressive than OWL 2 RL can be harmful to the performance of reasoning on top of big datasets
4. The W3C standard SKOS - Simple Knowledge Organization System - is one of the most popular standards for modeling taxonomies and controlled vocabularies. Formally, it represents an RDF schema, all sorts of attributes relevant to a term (e.g., preferred and alternative labels) as well as relationships (e.g., broader and narrower). https://www.w3.org/2004/02/skos/
5. Short intro to RDF Star is available at https://www.ontotext.com/knowledgehub/fundamentals/what-is-rdf-star/, full specification at https://www.w3.org/2021/12/rdf-star.html
6. See https://www.ontotext.com/products/ontotext-refine/
7. https://github.com/tarql/tarql

ontotext

LPG to RDF Guide

Europe: + 359 2 974 61 60
N.America: +1 412-638-7394

info@ontotext.com / www.ontotext.com

About the author:

Borislav Iordanov is a Knowledge Engineering Consultant at Semantic Arts, Inc. He has over 25 years experience in the software industry with recurring focus on knowledge representation, model-driven development, distributed architectures and programming languages. His career includes co-founding several startups, driving innovation projects in several large organizations and more recently helping clientsembrace a data-centric approach to information technology.