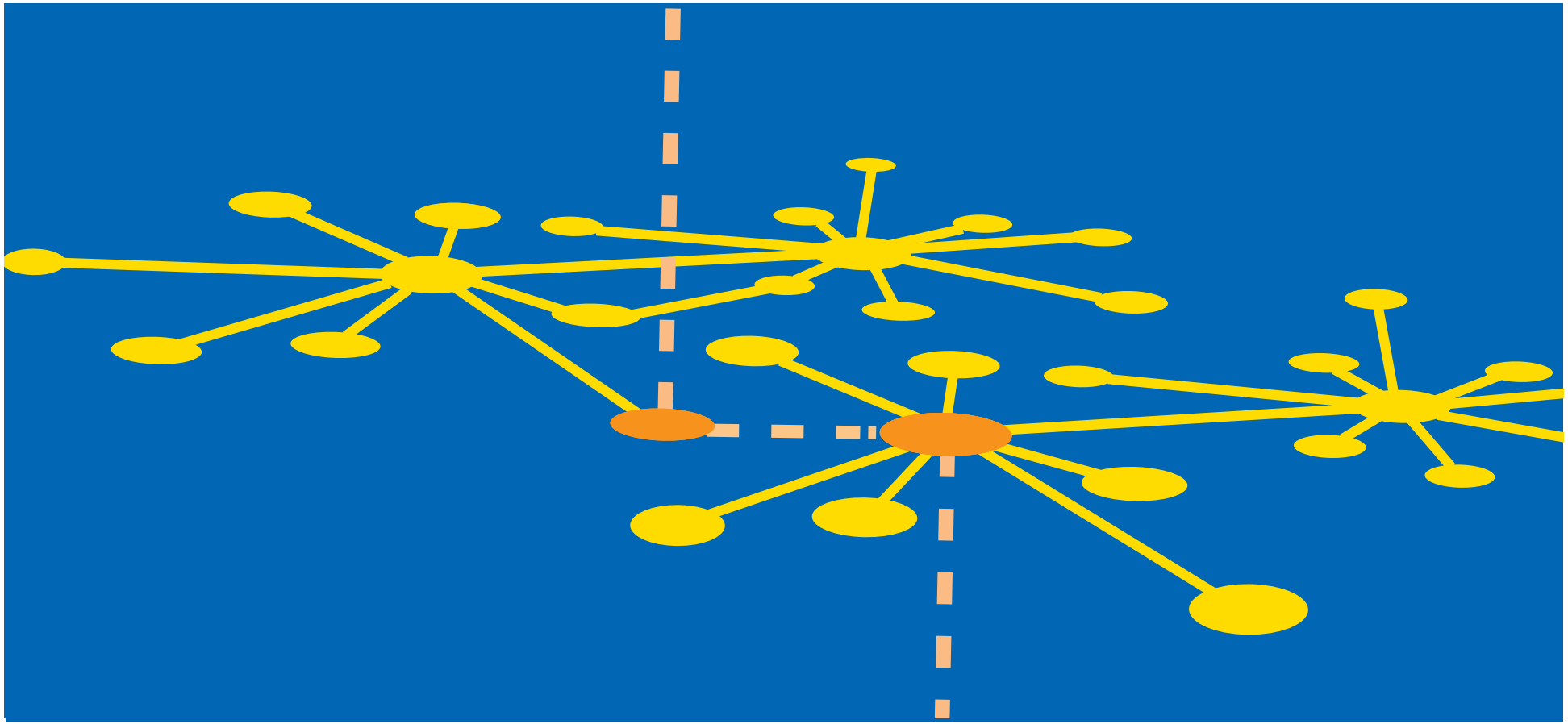
TopQuadrant™

Knowledge Graphs versus Property Graphs:

Similarities, Differences and Some Guidance on Capabilities



We are in the era of graphs. Graphs are hot. Why? *Flexibility* is one strong driver: heterogeneous data, integrating new data sources, and analytics all require flexibility. Graphs deliver it in spades.

Over the last few years, a number of new graph databases came to market. As we start the next decade, dare we say "the semantic twenties," we also see vendors that never before mentioned graphs starting to position their products and solutions as graphs or graph-based.

Graph databases are one thing, but "Knowledge Graphs" are an even hotter topic. TopBraid EDG is a solution for creating Knowledge Graphs and putting them to work. *(See page 10 for more information on TopBraid EDG.)* As a result, we are often asked to explain Knowledge Graphs.

- ***What are they?***
- ***Why and where are they useful?***
- ***How are they different from "just graphs?"***

At the recent Data Governance Vision conference, we gave a talk on the topic of supporting Data Governance using Knowledge Graphs. One of the questions asked at the end of the talk was whether we were using Microsoft's SQL Graph, and if not, then why not. After answering the question there on the fly, we decided that it was time to write a short paper explaining the differences between distinct implementations of graphs.

Today, there are two main graph data models:

- **Property Graphs**
  (also known as Labeled Property Graphs)
- **RDF Graphs**
  (Resource Description Framework)

Other graph data models are possible as well, but over 90% of the implementations use one of these two models. We will start by describing each of them.

Graph Data Models: Property Graphs and RDF Graphs

When we say that over 90% of implementations use either *Property Graphs* or *RDG Graphs*, we mean implementations that use some kind of an industry recognized graph data model. Due to the current expansive popularity of graphs, many vendors are starting to represent their technology as graph based, when in reality they use a home-grown object repository that can resemble certain aspects of graphs.

This white paper is not intended to cover such implementations since they do not use a recognized data model and, thus, there is no basis for comparison. If you are considering a technology that claims to be graph based, our recommendation is to always ask what graph data model it uses.

Property Graphs

While there are core commonalities in property graph implementations, there is no true standard property graph data model.

Each implementation of a Property Graph is, therefore, somewhat different. In the following, we will focus our discussion on the characteristics that are common for any property graph database.

The most well-known implementation, which popularized property graphs as a concept, is the Neo4J graph database. At minimum, everything stated here is true for Neo4J.

Other examples of property graph implementations are TigerGraph and Titan. MS SQL Graph is based on the same underlying concept, but it currently offers more limited capabilities than either Neo4J or some of the other products that are using the property graph data model.

Generally, the property graph data model consists of three elements:

- **Nodes** are the entities in the graph. Nodes can be tagged with zero to many text labels representing their type. Nodes are also called vertices.
- **Edges** are the directed links between nodes. Edges are also called relationships. The *"from node"* of a relationship is called the source node. The *"to node"* is called the target node. Each edge has a type. While edges are directed, they can be navigated and queried in either direction.
- **Properties** are the key-value pairs associated with a node or with an edge.

If you have worked with object databases, you will find it easy to understand the Property Graph data model. It is really more of an object data model than a graph data model.

- **Nodes are entities**
- **Edges are relationships**
- **Properties are attributes**

Both, entities and relationships can have attributes.

Property values can have data types. Supported data types depend on the vendor. For example, Neo4j data types are similar, but not identical, to Java language data types.

Figure 1 shows a fragment of a property graph with data about actors, directors and films or TV programs they worked on. Nodes are represented as ovals. For example, the node with ID 123, as we can see from its properties, represents Tom Hanks. Node labels are shown in dark blue. Node 123's labels are Person, Actor and Director.

TopQuadrant™

A PROPERTY GRAPH FRAGMENT WITH DATA ABOUT ACTORS, DIRECTORS, AND FILMS OR TV PROGRAMS

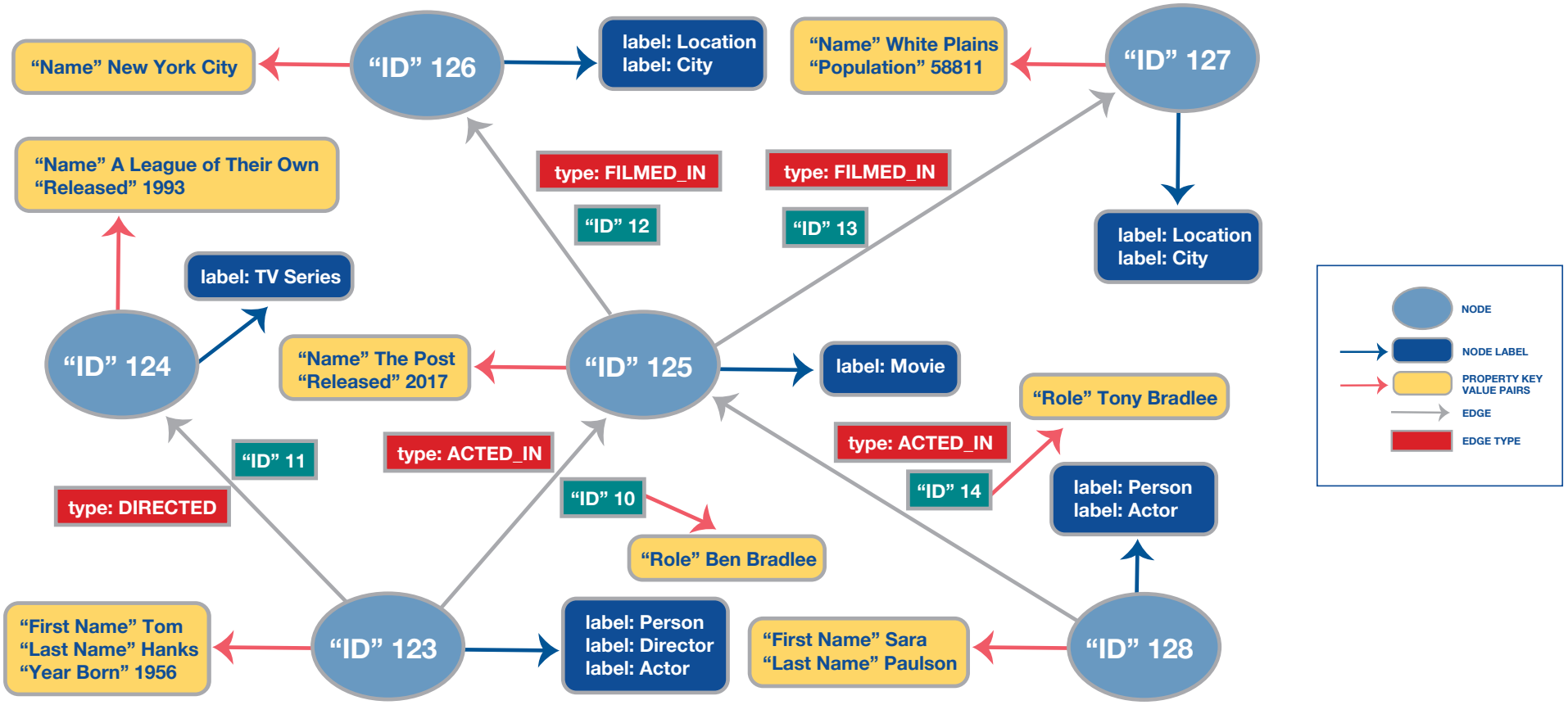


Figure 1: Simple Property graph excerpt with information about people and works of art

Relationships are depicted as grey arrows. Each relationship has a single type that is shown in red. Properties are shown in the rounded rectangles with the gold background. Properties are connected to nodes and relationships that they belong to using red arrows.

A key part of any data model is having a query language available for working with it. After all, users need to have a way to access and manipulate the data in the graph. No industry standard query language exists for property graphs. Instead, each database offers their own, unique query language that is incompatible with others:

- **Neo4J offers Cypher also known as CQL** — its own query language that, to some extent, took SQL as an inspiration;
- **TigerGraph offers GSQL** — its own query language that also took SQL as an inspiration;
- **MS SQL Graph** has their own extension to SQL to support graph query;
- Some vendors, in addition to their own query language, also implement some subset of Cypher. For example, SAP Hana offers its own extensions to SQL and its own GraphScript language plus they support a subset of Cypher

There is also Apache TinkerPop — an open source graph computing framework that is integrated with some property graph and RDF graph databases. It offers the Gremlin language which is more of an API language than a query language.

A key requirement for working with any data model is the ability to reference nodes, properties and relationships (edges). In the case of property graphs, internally, nodes and edges have IDs. IDs are assigned by a database and are internal to a database. Referencing is done by using text strings — node labels, relationship types, and property names.

The fastest way to load bulk data is by importing a text file. For property graph data, there is no standard serialization (a way to represent graph data as a text file). It is typical for a property graph vendor to define a CSV format that users should follow in order to prepare files for bulk load.

RDF Graphs

RDF graphs use a standard graph data model. The standard for the RDF technology stack is managed by the World Wide Web consortium (W3C), the same standards body that manages HTML, XML and many other web standards. Every database that supports RDF is expected to support the model in the same way.

The RDF graph data model basically consists of two elements:

- **Nodes**, the vertices in a graph. Nodes can be resources with unique identifiers or they can be "literals" with values that are strings, integers, etc.
- **Edges**, the directed links between nodes. Edges are also called predicates and/or properties. The "from node" of an edge is called the subject. The "to node" is called the object. Two nodes connected by an edge form a subject-predicate-object statement, also known as a **Triple** or a **Triple Statement**. While edges are directed, they can be navigated and queried in either direction.

Everything in an RDF graph is called a resource. "Edge" and "Node" are just the roles played by a resource in a given statement. Fundamentally in RDF, there is no difference between resources playing an edge role and resources playing a node role. An edge in one statement can be a node in another. We will give examples of this in the diagrams that follow that will make this core idea clearer.

There is a standard query language for RDF Graphs called SPARQL. It is both, a full featured query language and an HTTP protocol making it possible to send query requests to endpoints over HTTP.

A key part of the RDF standard is the definition of serializations. The most commonly used serialization format is called Turtle. There is also a JSON serialization called JSON-LD as well as an XML serialization. All RDF databases are able to export and import graph content in standard serializations making it easy and seamless to interchange data.

Built-in Semantics

The RDF Data Model provides a richer, semantically consistent foundation over property graphs. Let's see how a graph we showed earlier *(Figure 1)* is represented as an RDF Graph *(Figure 2)*.

Note that the diagrams depict relationships using the recommended conventions of the property graph and RDF graph communities. Relationships in Property Graphs are typically capitalized with multiple words joined together by an underscore as in ACTED_IN. Relationships (or any property) in RDF graphs are typically identified using the lower camel case convention as in ex:actedIn. In both cases, these are simply recommended practices, not a "must have."

The graph in *Figure 2* appears larger than the property graph in *Figure 1* because all literal values are also depicted as nodes in the graph. All nodes are depicted as rounded rectangles with the light yellow background.

When visualizing RDF Graph data, it is common not to show literal values as nodes in order to make a cleaner and simpler looking diagram. That said, from the data structure perspective, they are part of the graph just like any other node. The only difference is that they can't serve as a source node i.e., a subject of a statement. They can only be targets or objects. Throughout this paper, we will continue to show them in the diagrams as nodes.

Although this makes the diagrams larger and busier, we believe it helps to illustrate the differences between the two data models and the implications of these differences on knowledge capture, graph design and graph evolution.

Literal values in an RDF Graph can have datatypes. The datatypes are taken from the XML Schema (e.g., xsd:string, xsd:integer, etc.) Text values can also have language tags to support internationalization of data. For example, instead of a single value for rdfs:label for New York City we could have multiple values such as:

- **"New York City" xsd:string @en**
- **"Nueva York" xsd:string @sp**

Identifier is a very important concept for RDF graphs. Every non-literal node is assigned an identifier — typically, a URI/IRI. Local, non-URI identifiers are possible, but rarely used because they are not interoperable. Globally unique identifiers bring many benefits to graph data models. An RDF-based solution can auto-generate URIs based on selected URI construction rules. Alternatively, when adding data (e.g., loading a serialized file), users can provide URIs that they want to use.

The URIs identifying nodes are displayed in the diagram using qualified names, commonly called Qname notation. To form a Qname, the namespace part of the URI is abbreviated using a prefix. For example, "rdf:" and "rdfs:" represent the built-in standard namespaces **w3.org/1999/02/22-rdf-syntax-ns#** and **w3.org/2000/01/rdf-schema#**, respectively.

These namespaces define the semantics (the model behind) the RDF Data model. The built-in resources such as **rdf:type** carry semantics that are defined in the standard. The built-in resources can be used as either nodes or edges in a graph. For an example of such semantics in edges, see the predicates (aka properties) **rdf:type** and **rdfs:label** in the RDF graph diagram in *Figure 2*. For an example of such semantics in nodes, see the node **rdfs:Class** that is the object of the **rdf:type** predicate in the diagram shown in *Figure 3*.

TopQuadrant™

AN RDG GRAPH WITH THE SAME DATA ABOUT ACTORS, DIRECTORS, FILMS OR TV PROGRAMS

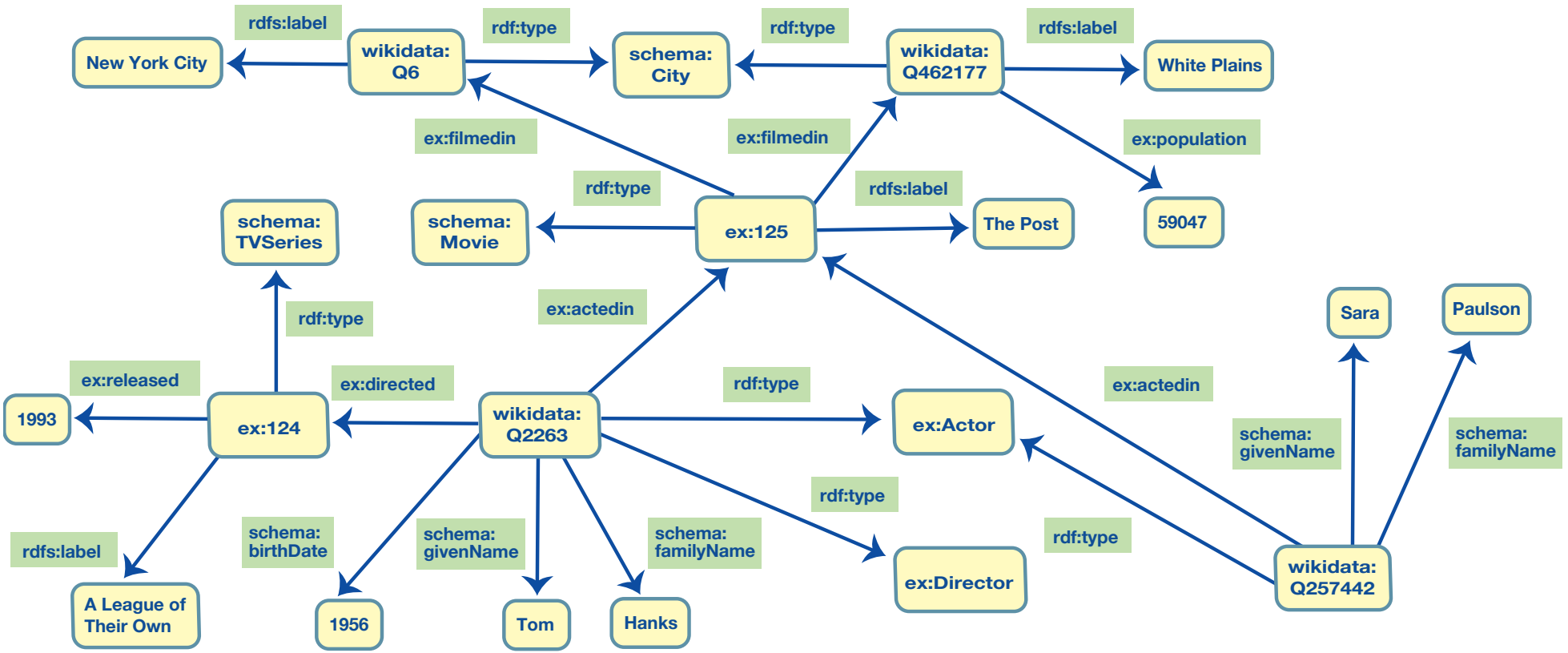


Figure 2: An RDF graph representing the information in the Property graph in Figure 1

A key differentiator that we will be introducing is how the underlying model (schema) is represented in the same way as the data. Just to serve as a primer, "rdf:type" is a predicate used to connect a resource with a class it belongs to; "rdfs:label" is used to provide a display name for a resource.

The uniformity of the data model makes RDF Graphs more easily evolvable and gives them more flexibility compared to Property Graphs. We will see examples of this later in the white paper.

Enrichment through Composition

With the inherent composability of RDF Graphs, when two nodes have the same URI, they are automatically merged. This means that you can load different files and their content will be joined together forming a larger and more interesting graph.

Examples of composability, can be found in the use of schema.org, and wikidata. Schema.org is a namespace jointly setup by Google, Bing and Yahoo to create and support a common set of schemas for structured data markup. The prefix 'schema:' stands for **schema.org.** Similarly, 'wikidata:' is a namespace used to provide DBPedia data in a structured, knowledge graph format. It provides a number of predicates and classes with commonly agreed and understood semantics. In the example, we are using schema:givenName, schema:familyName and schema:City. In this way, graphs developed by different organizations can link and share common semantics.

When organizations create their own knowledge graphs, they may use URIs of community defined resources as well as create resources for which they "mint" their own URIs. In the latter case, they would normally use a web domain they own as a namespace because a reference to a resource in an RDF Graph is expected to resolve and return information about it. In our example, in addition to using URIs from RDF, RDFS, Wikidata and Schema.org, we are also demonstrating the use of our own URIs. These URIs have 'ex:' prefix — to illustrate that they are provided as an example.

For human users browsing data, a reference to a resource URI will typically return information about a resource presented as a web page. For APIs making a call, information can be returned in JSON, any standard serialization of RDF or any other machine processable format.

The part of the Qname after the prefix is called a local name. A local name could be formed by using a display label if it can uniquely identify a resource within a namespace and is considered immutable. It could also be formed using a counter; much like in relational databases a record gets the next sequential number as its ID. It could also be formed using a machine-generated random ID or be based on the value of one or more predicates that can establish a locally unique identity.

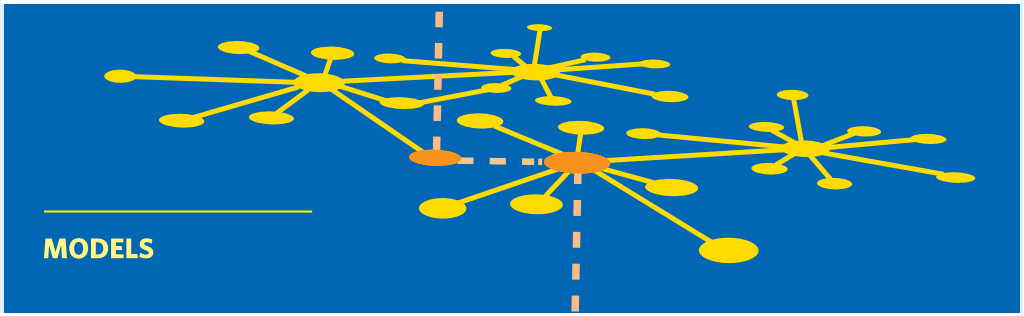
TopQuadrant™

TopBraid EDG: An Enterprise Knowledge Graph Infrastructure for Data Governance

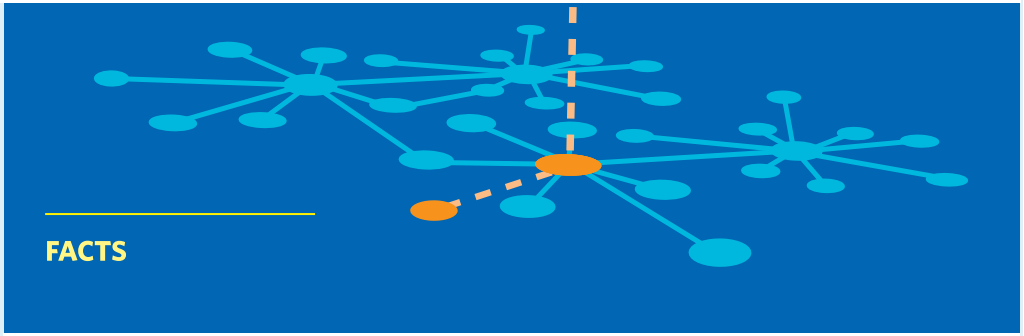
- TopBraid EDG, is a rich set of interconnected Knowledge Graphs expressing knowledge about how data is used and managed in the enterprise ecosystem.
- These integrated Knowledge Graphs are ready to be enriched with your enterprise specific knowledge.
- When this enrichment takes place, your enterprise is ready for implementing comprehensive Data Governance.

A knowledge graph contains facts about entities in the world together with the meaning of those facts expressed as models and rules.

RULES: If both of a person's parents have blue eyes, they will also have blue eyes



MODELS : A person has eye color. A person has two parents.
A person's father is also a person and he is male.



FACTS: James has blue eyes. James' father is Andrew. James is a person.

Differences in Terminology and Capability

Certain key terms used when describing graphs actually mean very different things depending on the graph data model one talks about. This is important to understand to avoid confusion. It is also important to understand in order to appreciate differences in the capabilities that these two graph data models provide.

We will now describe the differences in the meaning and use of some key concepts —

- **LABELS**
- **TYPES**
- **PROPERTIES.**

What are Labels and Types

In RDF Graphs, a label is a standard predicate defined in the RDFS namespace — rdfs:label. It is used to point to the value of a display name for any resource. For example, the label for resource wikidata:Q6 in the graph shown in *Figure 2* is "New York City." You could also use another predicate for this purpose, but rdfs:label is widely accepted as a unique identifier of a property that connects a resource to its display name. In Property Graphs it is typical to create a property called "name" and use it to hold a display name for a node. You could also use a differently named property.

In Property Graphs, the term "label" is used to identify the type of a node. It is called a label rather than a type because it is simply a string — a textual tag. It has no meaning beyond the text. No information about it can be captured in a graph. ***Edges in a Property Graph also have a tag that identifies the type of an edge.*** It is called a "type" or, sometimes, "relationship type". It is used in queries when matching relationships, and it is also used as a display name for edges when graphs are shown visually.

Contrastingly, in RDF Graphs, the type of a node or property's type is a resource i.e., another node in the graph — typically, with additional information associated with it to define its intended use and semantics. A node is connected to its type using the rdf:type predicate.

Note that some Property Graph databases (e.g., SAP Hana) do not use the term "label" at all and, instead use the term "type" or "node type." The underlying implementation, however, is the same — type is a tag for a node or a tag for a property. It is not a node itself.

Let's take a look in *Figure 3*, at a fragment of the same RDF graph we showed in *Figure 2*, now expanded with more information about types or classes and other schema elements.

The green border around nodes or edges indicates graph elements that describe the data model. In RDF, as in Property Graphs, nodes can belong to more than one set (class). We see this with Actor and Director. Tom Hanks is both. However, if one of the classes is a subclass of another, there is no need in RDF to specify a "parent type." Instead, this information is provided at the class level for all resources that belong to a class — because class information is also a part of the RDF graph.

MODELING INFORMATION, REPRESENTED THE SAME WAY AS FACTS, CAN EXPAND AN RDF GRAPH

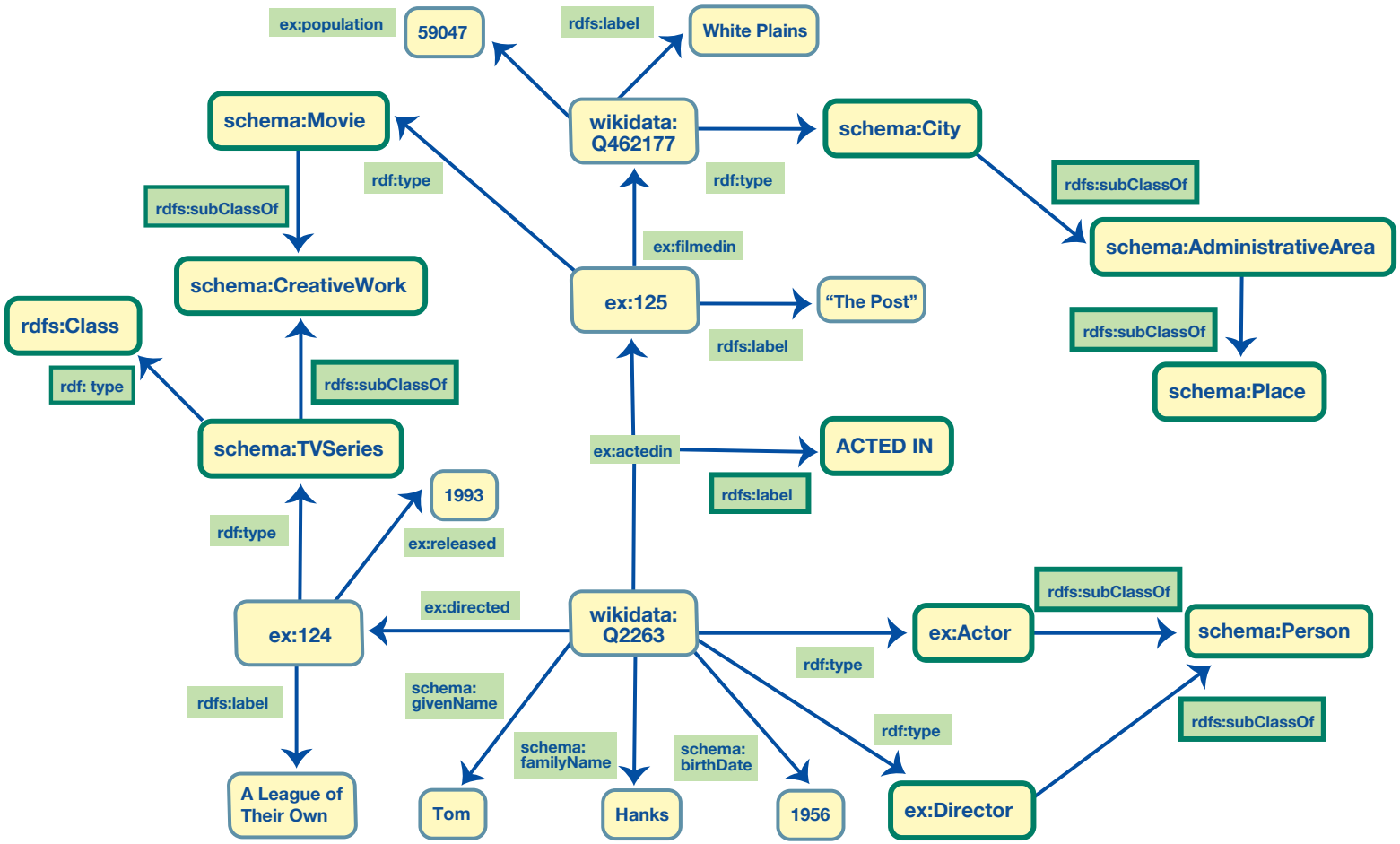


Figure 3: Part of the RDF graph diagram of Figure 2 expanded with modeling information

For example, unlike the Property Graph in *Figure 1*, we do not say in *Figure 2* that Tom Hanks is a person in addition to being an actor and a director or that Sara Paulson is a person in addition to being an actor. We simply say that there is a rdfs:subClassOf relationship between the class of Actors and the class of People. And the same for the class of Directors. The semantics of rdf:type and rdfs:subClassOf are defined in the standard — the graph depicted in *Figure 3* says that every resource of type Actor is also of type Person.

We also do not say that the type of New York City or White Plains is a place (location) in addition to a city. We do not need to repeat this fact for each city. We already said it in the model — each city is also a place and what is defined for a place will apply to a city.

In an RDF Graph, we can capture any information about the model of the data that is stored in a graph. This information will be stored, accessed and processed the same way as any other data. For example, the graph diagram in *Figure 3* shows that we can add a label to the predicate ex:actedIn. Similarly, we could also say that when the relationship ex:actedIn is used to navigate in the opposite direction (from a movie to an actor), the display name of the relationship should be shown as 'actors'. In an RDF Graph, a resource that is used as a predicate in one statement can be used as a subject or object in another statement. This is an example of the additional flexibility that, among other things, lets us store information about predicates and their usage. The edges in Property Graphs offer nothing comparable.

We can extend the RDF graph further to explicitly define how a predicate should be used. For example, we could say that any resource of type schema:CreativeWork can have a property ex:released and the value of that property must be a date. This would apply to a Movie or a TVSeries since they both are subclasses of schema:CreativeWork. The diagram in *Figure 4* shows what this looks like in a graph.

In *Figure 4*, the sh: prefix (e.g. in sh:property) stands for **w3.org/ns/shacl#**, the standard namespace that is used for ***SHACL*** — a language for defining rules and constraints for RDF Graphs, turning them into fully fledged Knowledge Graphs. SHACL offers a very strong approach to ensuring the integrity of RDF data and more.

For instance, we can:

- Consult a graph to find out what properties are appropriate for, let's say, a movie and what are the valid values for these properties.
- Define constraints also known as rich data quality/validity rules. For example, as shown in *Figure 4,* we have defined a min range of allowed date values for the 'released' property of a creative work (e.g., a movie or a TV Series). Now, if a movie released prior to 1900 is added to a graph, the graph can identify it as a problem. While this example is simple, we can add to the graph much more sophisticated rules. For instance, we could specify copyright regulations that must be in place for resources released or published after a certain date.

- Define rich inference rules. Inference rules generate new facts from the facts in the graph.

These key capabilities turn RDF Graphs into Knowledge Graphs.

What are Properties

In RDF Graphs, an edge is called a property (predicate) and an object that a property points to may be called a property value.

All property values (literals and URIs alike) are stored as nodes. For example, as shown in *Figure 2*:

- The rdfs:label for the resource ex:125 is "The Post." In this example, rdfs:label is a property and "*The Post*" is a value.
- The edge ex:filmedIn is also a property. Its values for ex:125 are wikidata:Q6 and wikidata:Q462177.
- In "data modeling speak," in an RDF Graph properties can be either attributes or relationships.

In Property Graphs, properties can only have literal values. These are stored and treated differently from the nodes in a graph. In data modeling speak, properties in a Property Graph are always attributes. This is why property graphs are formally described as directed, edge labeled, attributed graphs.

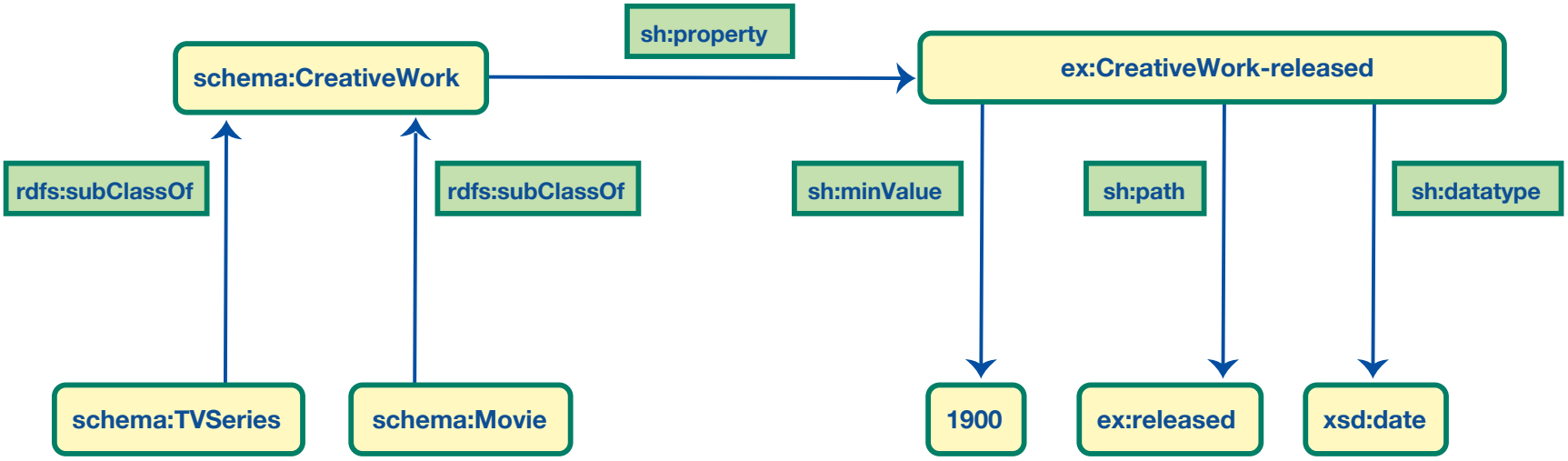


Figure 4: Extending an RDF graph with more modeling information about the ex:released property

A property structure is that of key-value pairs. This means that a property key can only have a single value. If it has more than one value, then the single value is turned into an array of comma separated values. For an example, see *Figure 5.*

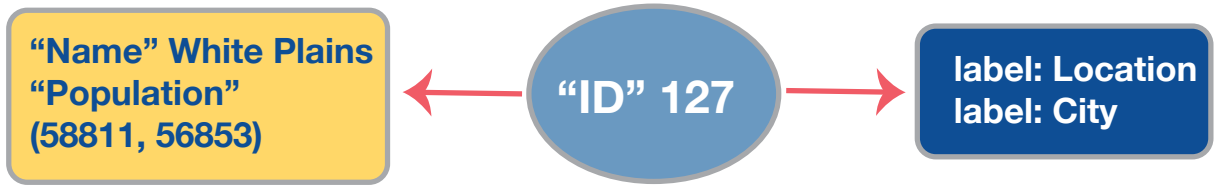


Figure 5: In Property graphs, the property structure is that of key-value pairs — multiple values must be turned into an array of comma separated values

Turning multi-valued properties into arrays makes it harder to efficiently answer queries such as "all cities with population over 58,000." The first value in the array is the population of White Plains in 2018. The second value is the population of White Plains in 2010. There is no way in a Property Graph to capture what each of these values represents beyond the fact that the key part of the key-value pair is Population. This brings us to the next important difference — how to capture additional information about a property value. In saying this, we mean any property — whether it is an attribute or a relationship. As we see with the population example, it may be important to qualify a measurement by the date it was measured on. There are also other important information qualifiers — including source and confidence.

For example, Wikidata captures many details about the source of the information about Tom Hanks' birth date in order to give users confidence in the reliability of the data. As shown in *Figure 6*, it got the information from 9 sources which all agree on the date. The sources include the Encyclopedia Britannica, Internet Broadway Database and others.

Differences in Attaching Information about an Edge

In RDF Graphs, unlike in Property Graphs, edges are typically re-used:

- In the Property Graph shown in *Figure 1*, there are two **ACTED_IN** edges with different IDs: an edge connecting the node representing Tom Hanks to the node representing the movie *The Post* and an edge connecting Sarah Paulson to this movie. The two edges have the same type, but different identity.
- In RDF, it is the same edge. This means that if you need to say something about a relationship between Tom Hanks and *The Post* (e.g., the role he played in the movie), you can't simply add a statement to the **ex:actedIn** property. If you do this, it will apply everywhere this property is used.

In other words, in the Property Graph data model, edges uniquely identify the ***source-node — edge — target-node*** combination. In the RDF data model, they tend not to. Of course, one could create a unique edge and

simply give it the type ex:actedIn. However, this is normally not done because RDF databases are optimized for working with edges that represent types instead of occurrences of types.

To support the need to attach information on an edge between two specific nodes, RDF provides a way to create a new node that uniquely identifies the source-edge-target triple (or the subject-predicate-object in RDF speak) combination. With that in place, we can make statements about the new node using the regular approach — it can be a subject or an object of any statement. This is shown in *Figure 7* where we created a new node ex:126 to represent the statement (triple) of Tom Hanks' acting in *The Post*. The new node is connected to the statement about Tom's acting in *The Post* using rdf:subject, rdf:predicate, rdf:object and rdf:Statement, built-in elements of the RDF data model that support this use case.

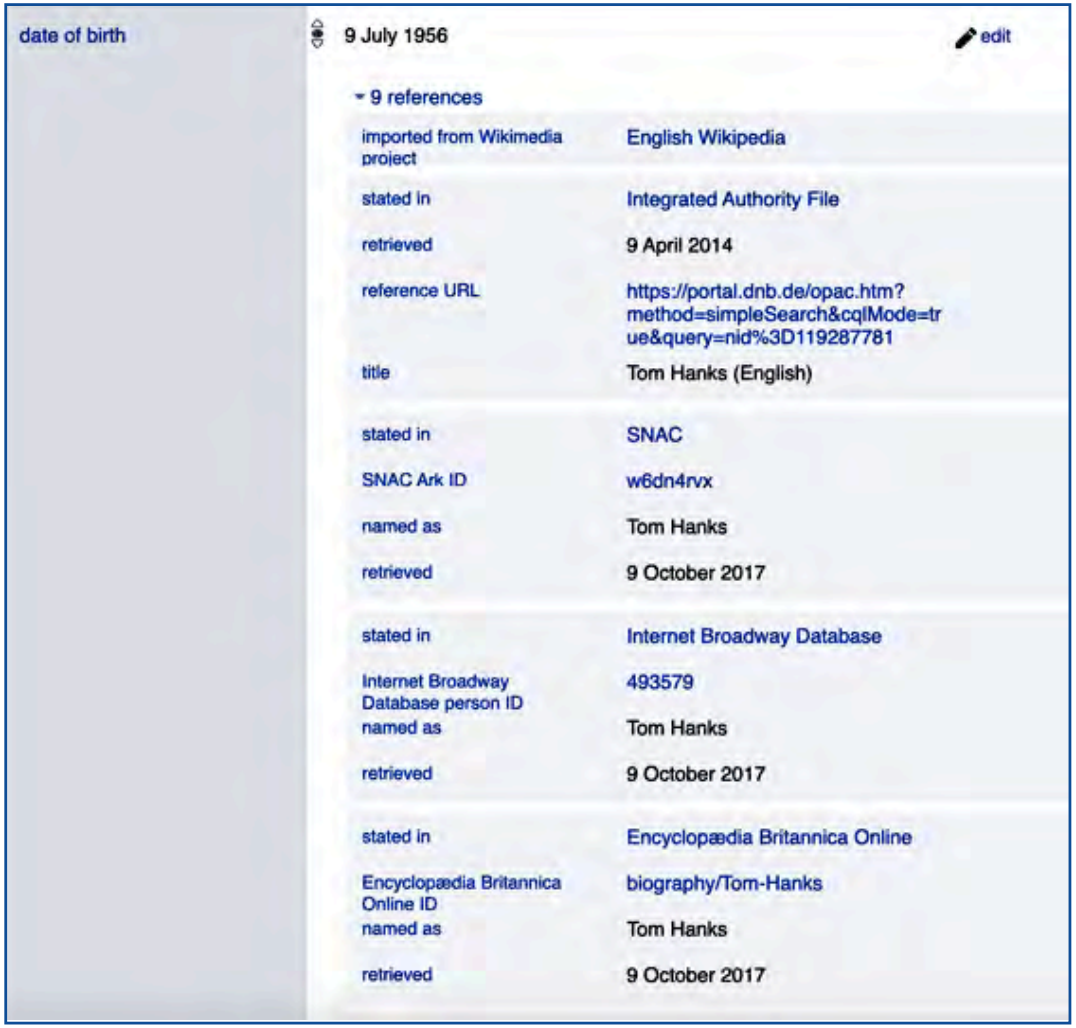


Figure 6: A screenshot from WIkidata showing the sources of information about Tom Hanks' birth date

Compared to Property Graphs, this approach is more powerful and flexible because it supports:

- Adding other edges (relationships) to edges. For example, instead of having a role as a string, we may want to have a connection to a node representing Ben Bradlee, a person. This is fundamentally not possible with Property Graphs without changing (restructuring) the original graph.
- Adding more information to any property, not just a relationship. For example, we can use it to specify the effective date of each population measurement for White Plains. This is also not possible with Property Graphs.

For Property Graphs, the solution to the need to add edges to other edges is to create intermediate nodes — as shown in *Figure 8.*

This requires restructuring of a graph and changing all queries and logic because the path between actors and movies is now different *(compare with the original graph in Figure 1)*.

With RDF, you do not need to make changes to the graph structure to make a link to the

RDF GRAPH WITH AN EXAMPLE OF MAKING A STATEMENT ABOUT ANOTHER STATEMENT

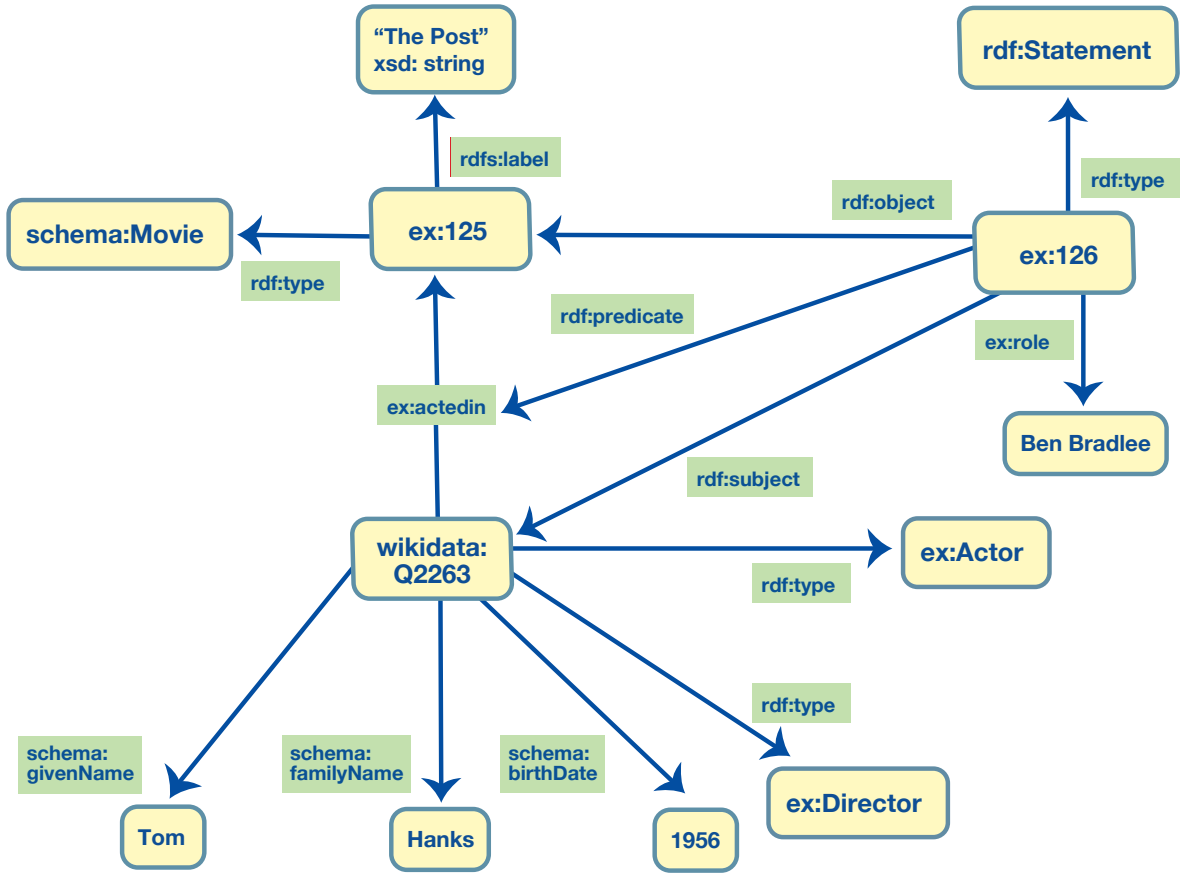


Figure 7: RDF graph showing making a statement about another statement — to attach information on an edge between two specific nodes.

resource representing Ben Bradlee. You simply change the node at the end of the ex:role relationship from a string to a URI. This is demonstrated in *Figure 9.* The approach is evolutionary and does not require any refactoring other than the change of the value itself.

There may, however, be some other situations where you would want to introduce new intermediate nodes. If you do so, SHACL rules can be used to deliver the original relationship path inferring its value from the new, more complex path. In this way, your existing queries and programs can remain the same.

The Property Graph solution to adding more information to a property (e.g., population) is to change the structure of the graph to turn a property into an edge and a value to a node. This requires restructuring of a graph and change to all queries and logic for its processing because the storage and access of properties is fundamentally different and separate from the graph traversal. This makes Property Graphs less evolvable or flexible than RDF Graphs.

IN PROPERTY GRAPHS ADDING EDGES TO OTHER EDGES REQUIRES REFACTORING THE GRAPH

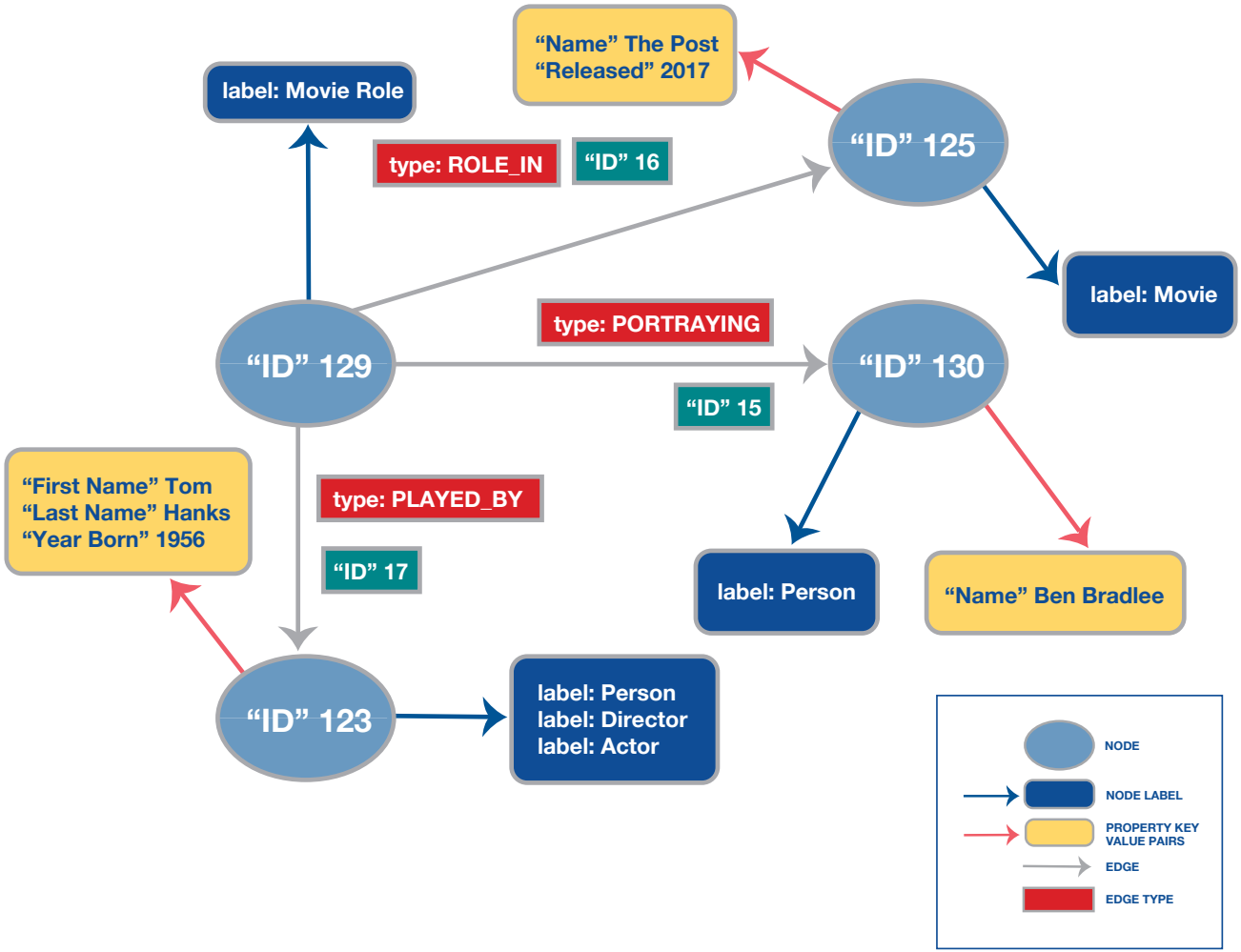


Figure 8: Refactored Property Graph with Ben Bradlee as a Person

Flexibility is acknowledged as the key differentiating advantage of graph databases. For example, the leading vendor of property graph databases says, "With graph databases, IT and data architect teams move at the speed of business because the structure and schema of a graph model flexes as applications and industries change. Rather than exhaustively modeling a domain ahead of time, data teams can add to the existing graph structure without endangering current functionality." We agree that this would be a very important and desired advantage. However, as we describe in this paper, changes in the model of the Property Graph data will require refactoring and changes to queries. In a Property Graph edges and properties are different data structures and their handling in queries is fundamentally different.

As you can see, compared to an RDF Graph, it is harder to organically grow a Property Graph in response to changes in your information requirements.

A current downside of the RDF Statement approach to capturing information about edges is what is sometimes called "graph bloat." To capture a role that Tom Hanks had in *The Post*, we need to add at least three extra statements (rdf:subject, rdf:predicate and rdf:object) in addition to the role information — four if you also add a type link to rdf:Statement. Quite a lot of overhead for just one fact. If, however, you need to capture several facts about Tom's acting in this movie, then this approach has less overhead.

A new extension to the RDF data model called RDF* (RDF Star) and its variation called RDF Plus address this issue. It is currently in the process of being added to the standard. In the meantime, TopBraid EDG can create a new node with the URI composed from the subject-predicate-object nodes of the statement you need to add information to. The new node uniquely identifies the original statement and can be used as a subject of other statements, avoiding graph bloat. For standard-compliant information exchange, EDG serializes such nodes as RDF Statements.

Graph Analytics, Named Graphs and Other Topics

This white paper is not intended to completely cover all capabilities of *Property Graphs* or *Knowledge Graphs*. We have focused only on critical differentiators.

With this, we need to at least mention two important topics:

- **Algorithms for Graph Analytics**
- **Named Graphs**

Graph analytics is a key application for property graphs. By analytics, we mean node centrality, node similarity, shortest paths, clustering and other algorithms. Property Graphs are known for offering these algorithms and many applications of property graphs rely on such algorithms. Having said this, there isn't anything special in a property graph data model that makes these algorithms possible. They can be applied equally well over RDF Graphs. In fact, many RDF-based solutions are also offering similar algorithms.

TopQuadrant™

IN AN RDF GRAPH, YOU DO NOT NEED TO CHANGE THE GRAPH STRUCTURE TO MAKE A LINK TO A RESOURCE

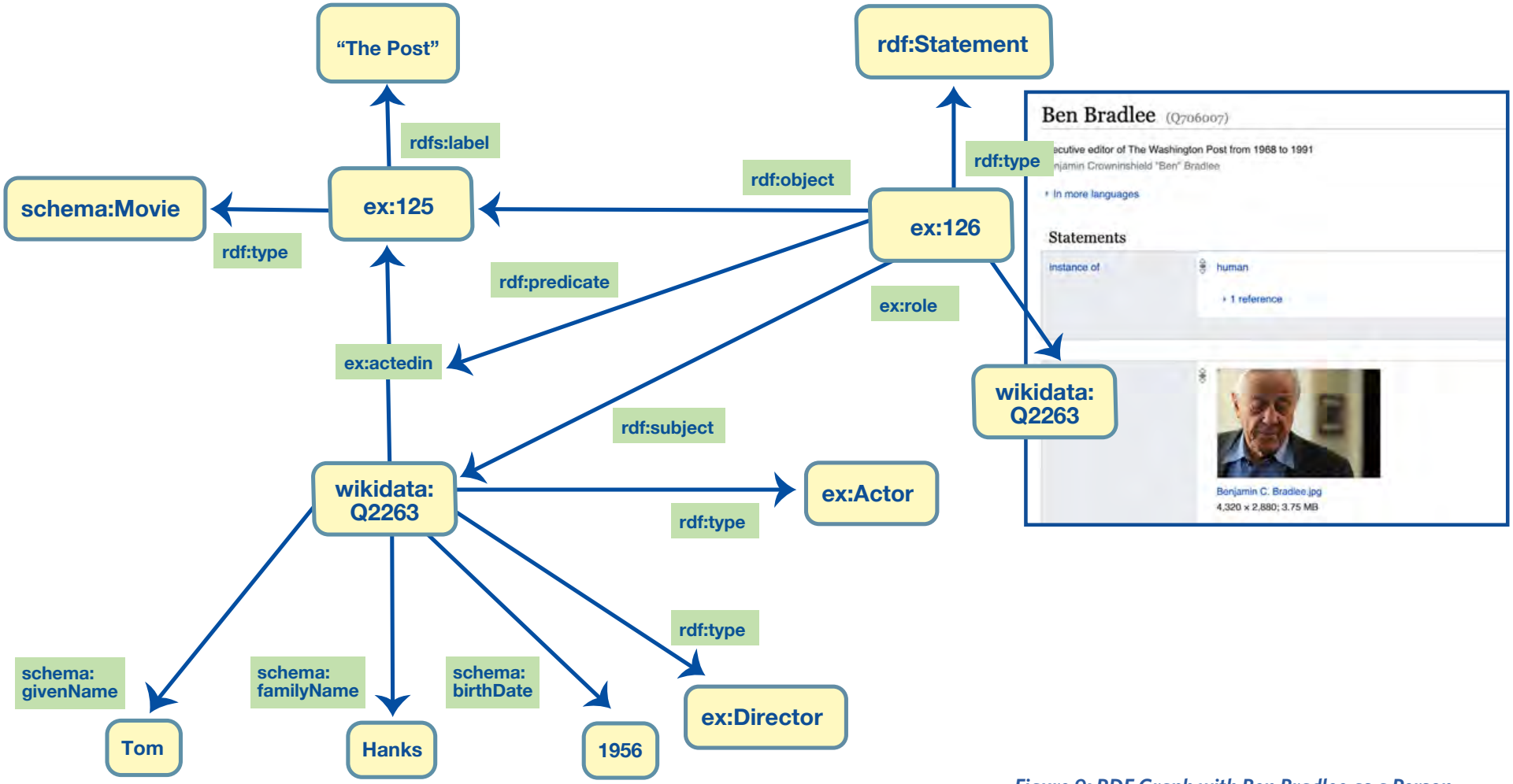


Figure 9: RDF Graph with Ben Bradlee as a Person

The ability to partition data is important. Relational databases partition data using tables and views. Both Property Graphs and RDF Graphs let users work with sets of nodes of a specific type (in the case of Property Graphs, nodes carrying a specific label), e.g., a query can be limited to only work with actors or to only work with directors. This provides a very basic, limited partitioning.

RDF data can also be partitioned in named graphs. A named graph offers us a way to say that some group of triple statements belong to a "sub-graph." We can then give it a uniquely identifying name (hence, the term "named graph") and associate any other information with it that we see as important. The idea is somewhat similar to views in relational databases. A single statement can belong to many named graphs. Thus, it is a different concept from physically partitioning distinct graphs across different machines.

We can query a named graph individually, or we can query all available graphs, or a subset of available graphs. We can load a named graph, clear it and perform any other manipulations with it. This again follows the idea of "separate, but connectable."

For example, in TopBraid EDG, a given business glossary or a taxonomy is a named graph. Resources in it can be connected to resources in other graphs, but it can also be manipulated as a distinct set of statements. For example, there could be a purpose associated with a glossary as a whole e.g., its users and uses can be identified and so on.

There is no similar concept in the Property Graph world.

Limitations of Property Graphs

In this white paper, we describe some limitations of Property Graphs and their differences with Knowledge Graphs that are based on RDF.

The main vendor for property graph technology, Neo4J, offers a mature system with some attractive, easy to get started with capabilities. There are also a few other Property Graph databases on the market today.

However, we increasingly hear of customers hitting the wall with Property Graphs because as they start to use them, they recognize the need for one or more of the following capabilities:

- **Capture of Schema in a Graph**
- **Support for Validation and Data Integrity**
- **Capture of Rich Rules**
- **Support for Inheritance and Inference**
- **Globally Unique Identifiers**
- **Resolvable Identifiers**
- **Connectivity Across Graphs**
- **Better Solution to Graph Evolvability**

Note that these are fundamental limitations that are not addressed in the design of property graphs. In principle, it may be possible to add at least some of these capabilities to a Property Graph — *but not that easily or elegantly*. Some of you may have already started on the road to doing this.

However, it is a lot of effort, both conceptual (i.e., design and architecture) and implementation work. Even if you succeed in

accomplishing it, you will end up with a proprietary home-grown version of capabilities that already exist, are standardized and well proven.

Inherent Semantics make it easy for RDF Graphs to become Knowledge Graphs

As illustrated in the previous sections, RDF-based graphs capture more than just data. They capture the meaning or semantics of data, including rich constraints and highly expressive rules. All information is stored in a graph and is available for query and any other algorithms that can help us reason and discover new knowledge based on the available knowledge. And the amount of the available knowledge with Knowledge Graphs is practically unlimited — just as it is on the world wide web. We can reach out and take advantage of the information available in other graphs. Separate, but connectable is a key feature of the web — and of Knowledge Graphs.

With Property Graphs, data modeling happens on paper or on a white board, separate from the graph itself. ***Property Graphs are not self-describing and the meaning of the data they store is not a part of a graph.***

Some Guidance for Moving from a Property Graph to a Knowledge Graph

It is fairly easy to generate one of the RDF standard serializations from a property graph. In fact, Neo4J offers a library for doing this. You can readily get the data out, but you will not be able to get the semantics of the data; this is due to the fact that the data model only exists in your initial design sketches and, partially, within Cypher queries and programs.

Further, as we have discussed, the structure of the graph data may be influenced by the specific limitations of the property graph data model and optimizations that were required due to the architecture of a property graph database. We already demonstrated how a decision to use intermediate nodes in a property graph may be based on the need to add information to a property, which is only possible if a property is turned into an edge.

Further, in property graphs some property values such as dates or names are often turned into entities because there is no efficient way of querying literal values, especially if they are multi valued. As a result, you may have an entity for a number 58,811 or a year 1956. This, however, could result in having so-called “dense nodes” or nodes that participate in many relationships. Typically, nodes that are targets of thousands of relationships are considered to be dense in Neo4J with the potential of performance issues when such nodes are deleted. The design of the model may, therefore, be impacted by the density considerations. Similarly, you may have relationships that represent specific dates e.g., BORN_IN_1956, BORN_IN_1957, etc. This is a design pattern used in property graphs

because with a generic BORN_IN relationship, Cypher queries looking for people born in, let's say 1956, do not perform well. Once you move to RDF, you may decide to revisit some of these design decisions.

The simplest way forward is to export property graph data *as-is* and then create a data model in RDF that represents the structure of the data. For example, if you created intermediate nodes in order to link roles to people portrayed by roles, you would mirror this in your RDF model (often called an ontology) even if strictly speaking this is not necessary in the RDF-based implementation.

TopBraid EDG can use data to reverse engineer an ontology. This will speed up your migration efforts and will make the data model explicit. You can then decide if you want to adjust the model and change the data or move forward with it as-is, evolving it later if necessary.

Many applications today use GraphQL to read and write data. Neo4J and some other Property Graph offerings support GraphQL access to data. If you have used GraphQL to build your solution on top of a Property Graph, you will be able to keep much of your code as you move to an RDF platform like TopBraid EDG that also supports GraphQL.

For property graphs, GraphQL Schemas need to be manually created and then manually maintained as the graph structures get extended and changed. One of the advantages of a self-describing graph is that GraphQL Schemas can be automatically generated from the data model. This delivers on the promise of frictionless development and graceful systems maintenance by rendering unnecessary any manual effort for defining and maintaining schemas. For more information on how TopBraid EDG works with GraphQL, visit ***topquadrant.com/technology/graphql/***.

For the types of queries that can't be easily supported by GraphQL, you will typically use SPARQL. TopBraid EDG lets you use either of the query languages and it also lets you put SPARQL expressions into GraphQL.

Summary

Neo4J is a mature solution that popularized Property Graphs and made them easy to get started with. People tend to think that RDF based Knowledge Graphs are hard to understand, complex and hard to get started with. In the past, there was some truth to that characterization. Today, with products like TopBraid EDG, it is no longer the case.

Many users are discovering the limitations of property graphs. Even if you started your first graph project using a property graph, it is likely that sooner or later you will be hindered by limitations and will want to adopt or at least explore the feasibility of an RDF / Semantic Knowledge Graph based system. You will not be alone, as a number of organizations are graduating from property graphs to knowledge graphs. We hope that this paper has provided some insight and value in your decision making.

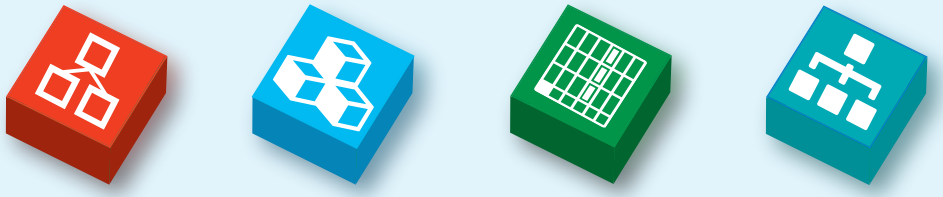
TopQuadrant™

About TopQuadrant

TopQuadrant helps organizations succeed in Data Governance. Its flagship product, TopBraid EDG, delivers easy and meaningful access for all data stakeholders to enterprise metadata, business terms, reference data, data and application catalogs, data lineage, requirements, policies, and processes.

TopQuadrant's customer list includes over 120 organizations in financial services, pharma, healthcare, digital media, government and other sectors.

GOVERNANCE PACKAGES AVAILABLE IN TOPBRAID EDG



Vocabulary Management | **Metadata Management** | **Reference Data Management** | **Business Glossaries**

In addition to the above, TopBraid Tagger and AutoClassifier is a popular additional module that is part of a comprehensive information management and governance environment where packages for other types of assets can be easily added if needed.

In ramping up a Data Governance program, different organizations may have different starting points. With TopBraid EDG, you can start incrementally and add capabilities as you go. For details on available EDG packages and additional modules visit ***topquadrant.com/products/topbraid-enterprise-data-governance/***

For more details or to schedule a demo, contact us at: edg-info@topquadrant.com