An Elementary Introduction to Resource Description Framework

AI Summer School, IIT Mandi, 19-28 July 2017

Baskaran Sankaranarayanan

Artificial Intelligence and Database Lab
Department of Computer Science and Engineering
Indian Institute of Technology Madras

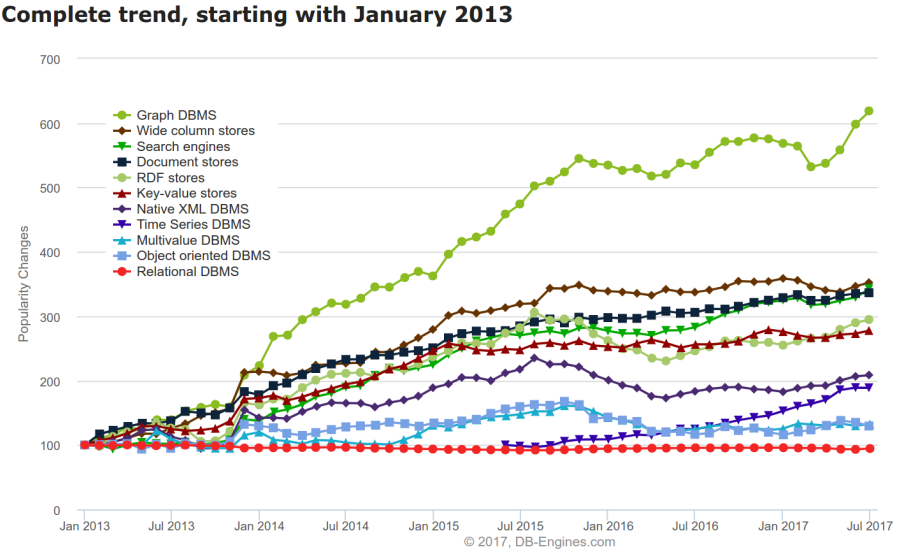
Abstract

This note, for beginners, provides the necessary background to develop a perspective on Resource Description Framework (RDF). Here, we present a motivation for RDF, and describe the logical foundation of RDF and RDFS, and point-out some limitations. Then we introduce SPARQL query language for RDF through a set of examples.

Some Perspective

The word cloud shows a list of database management systems[1] ranked by the number of mentions on the net[2]: in websites, technical forums, tweets, job offers, LinkedIn profiles, Google Trends, etc. This ranking gives one perspective, and http://www.tpc.org gives another perspective — a performance based comparison of database systems, mainly relational systems.

In the word cloud there are over 300 database systems. Not all are relational systems, by far the most popular ones are relational systems. NoSql systems are also popular. Graph stores and RDF stores are trending now.[3]



Back in the days, flat-files acted as databases, then database systems evolved and adopted a variety of data models to meet the needs of storage, availability, consistency, recency, distributed processing, analysis, reporting, etc.

We invented a data model for every need.

[1] DB systems from https://db-engines.com/en/ranking, as of July 2017.
[2] Ranking method: https://db-engines.com/en/ranking_definition.
[3] Rank trend: https://db-engines.com/en/ranking_categories.

Some popular data models are: hierarchical model, relational model, object model, object-relational model, multidimensional model, document model, columnar model, key-value model, graph based model, triples based model, etc.

> The Entity-Relationship (ER) model, UML model, and Star-Schema are logical-modeling frameworks; they help us to organize data for a particular usage.

Some database systems are multi-modal in that they support more than one data type, say, relational, XML, RDF, etc. Each data model serves a purpose like transaction processing (hierarchical and relational model), reporting (multidimensional and columnar model), ETL processing (key-value and columnar model), networked data processing (graph and triples model), etc.

A recent trend in databases is the graph data model (property graph model), where data is stored in native graph format as nodes and edges. A list of graph stores is shown below:[4]

In an increasingly open world — where the data organization is not known in advance or the data organization rapidly evolves or the connectivity (edges and paths) is the primary data like the city maps — a graph model is a good fit.

[4] Graph stores from https://db-engines.com/en/ranking, as of July 2017.

Neo4J is a graph store (with ACID properties) that supports large graphs. Titan is a distributed graph store.

Triple stores are going mainstream or at least there is a strong trend that we can observe from the level of tool support from the community and vendors. Some popular RDF stores are:[5]

AllegroGraph is a triple store (with ACID properties) that supports large RDF datasets. It comes with a SPARQL query interface and a graph visualization tool, runs in client-server mode, supports multiple users, multiple connections, allows bulk data import and export, and more.

AllegroGraph is a commercial product, it is free for data stores of up to 5 million triples.

[5] RDF stores from https://db-engines.com/en/ranking, as of July 2017.

Why RDF?

I will discuss three scenarios that will help us see the need for RDF.

Scenario 1: We are attending a summer school (this AI Summer School at IIT Mandi, 19-28 Jul 2017). The task is to model the information about this event. Create a database and populate it.

1. What information will you collect? (Identify)
2. How will you collect it? (Acquire)
3. How will you model it? The schema. (Store)

Scenario 2: The task is same as in scenario 1 but with one difference. Each one of you here today, independently, prepare a schema and collect the data, in effect creating one database per person. No collaboration with each other. Finally, integrate all the individual databases into a single database.

1. What information will you collect?
2. What schema will your choose for data collection?
3. What schema will you choose for the integrated database?
4. List some data integration challenges.

Scenario 3: Same as scenario 2 but the problem is open. You can collect any data about anything around you. Independently, each one of you go around Mandi and note down your observations, any observation. Finally, integrate all the data into a single dataset.

1. What information will you collect?
2. Data collection schema?
3. Integrated central database schema?
4. List some data integration challenges.

Try to solve these cases before reading further. Make an attempt.

Scenario 1

At the very least, we can think of collecting information about sponsors, organizers, speakers, participants, schedule, etc.

A simple model is shown below, there is symmetry between speaker and participant, both carry the same attributes.

Model 1

```
Sponsor     (Name, Amount, Affiliation)
Organizer   (Name, Task, Affiliation)
Speaker     (Name, Topic, Slot, Affiliation)
Participant (Name, Topic, Slot, Affiliation)
Calendar    (Slot, Hall, Date, Time)
```

For illustration we show a simple schema. In practice, there will be entities, dependent entities, relationships, a rich set of attributes and keys (primary, unique, and foreign keys).

This model is good for data collection, but not the best one for data storage because there are redundancies like name-affiliation and topic-slot which we can normalize.

A normalized version is Model 2.
The symmetry between speaker and participant is still present.

Model 2

```
Person      (Name, Affiliation)
Sponsor     (Name, Amount)
Organizer   (Name, Task)
Speaker     (Name, Slot)
Participant (Name, Slot)
Calendar    (Slot, Hall, Date, Time)
Schedule    (Topic, Slot)
```

Model 1

```
Sponsor     (Name, Amount, Affiliation)
Organizer   (Name, Task, Affiliation)
Speaker     (Name, Topic, Slot, Affiliation)
Participant (Name, Topic, Slot, Affiliation)
Calendar    (Slot, Hall, Date, Time)
```

We can collect the same data in several ways (*i.*) by entity: collect data for each table or relation separately; (*ii.*) by day: collect data about activities in day 1, day 2, etc.; (*iii.*) by topic: for each topic collect data about speakers, participants, slots, etc.

And we can find other strategies for collecting data.

Observe that collecting data by entity has the least amount of redundancy. Other methods would bring in duplicate data.

Scenario 2

We have a known problem at hand (modeling the information about the AI Summer School) with multiple representations and a single central database. The individual databases would have a large overlap in content but differ in layout and data encoding.

For a known problem an XML or columnar model would be good for data collection. Depending on the collection strategy both models would bring in duplicate data.

In a columnar model the division of labor is clear, we can integrate one column at a time. Still the entity ids have to be resolved manually. A relational model is also a good choice, but adding new attributes and new entities on the fly is cumbersome.

For the integrated schema, a relational or columnar model is a good choice. A relational model will provide a good balance between data management and reporting. A columnar model is excellent for data integration; narrow tables, each with a key and an attribute. Still the entity ids have to be resolved manually.

In this scenario, we are dealing with a single problem with multiple representations, naturally, it leads to data integration challenges.

- Schema, data and encoding mismatch.
- Schema and data redundancy.
- Synonyms and Homonyms.
- Coverage (width) and Richness (depth) of schema and data.

Scenario 3

This scenario tries to emulate the data on the web.

The problem is unbounded (it is an open data problem) so there will be a variety of data, and severe disparity in representation. We cannot fix any aspect of the data, say attributes, entities, relationships, tuples, granularity, completeness, correctness, etc., etc.

One can collect data about anything, say flora and fauna of Mandi, geology of the terrain, people, businesses, ... There is no limit on the type and kind of data one can collect.

A graph model (nodes and labeled edges) is good for data collection. A columnar model will be rigid because we have to create a thin table for each new column and we do not know the columns in advance. Whereas, in a graph model one can add new nodes and edges easily using primitive operations.

We choose a graph model for the integrated database as well. If we choose the node and edge identities carefully then the data linkage problem solves by itself. Without collaboration we cannot choose the ids correctly. So we must have a way of saying that certain nodes (and edges) are the same, that they are synonymous.

Data redundancy is not an issue in a graph model, but other data integration challenges exist, that are independent of the data model and the domain of interest.

- Data encoding.
- Synonyms and Homonyms.
- Coverage (width) and Richness (depth) of schema and data.

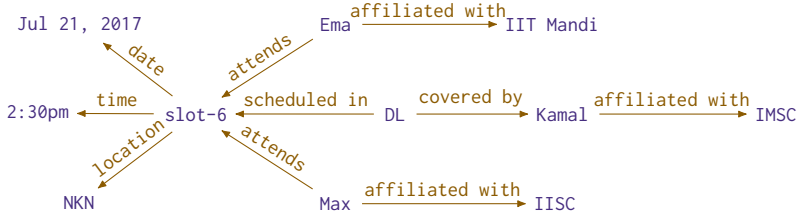
Scenario 3: A Graph Model

A graph model (a relational view) is presented below, each labeled edge in the graph is collected in a two column table that is named after the edge label, you can imagine these tables to be binary relations. Observe how the symmetry between speaker and participant seen in the earlier model is lost in this particular graph model.

Graph Model	
affiliatedWith	(Name, Org)
contributed	(Name, Amount)
assignedTo	(Task, Name)
coveredBy	(Topic, Name)
attends	(Name, Slot)
scheduledIn	(Topic, Slot)
date	(Slot, Date)
time	(Slot, Time)
location	(Slot, Hall)

Model 1	
Sponsor	(Name, Amount, Affiliation)
Organizer	(Name, Task, Affiliation)
Speaker	(Name, Topic, Slot, Affiliation)
Participant	(Name, Topic, Slot, Affiliation)
Calendar	(Slot, Hall, Date, Time)

A graph model solves a good part of the data integration problem. We can easily scale a graph model both horizontally (operationally) and vertically (functionally).



By choosing a graph model we moved away from structured information that is readily useful for an application, to a mere collection of simple facts or titbits of information.

We lost some structure and some classification information, from our graph model we cannot readily pick-out the sponsors, participants, speakers, etc. But we have captured the activities and associations of such entities in fine detail. There is opportunity to recover the lost structure and classification information. How? Can you think of some ideas?

Data Integration: A use case for RDF

We looked at three scenarios with increasing complexity (as summarized in the table below).

1. For a known problem with a known schema, the data collection and data integration is an easy problem. Conceptually trivial, we can plan and execute the collection and integration process, but there may be operational difficulties due to size and quality of data.

2. For a known problem with multiple schemas, the data collection is easy and data integration is difficult, but doable.

3. For an unbounded problem with many heterogeneous schemas, the data collection may or may not be easy, but definitely, the data integration is orders of magnitude harder than in the previous two cases.

   For an unbounded problem and a graph schema (*with a suitable node and edge identification scheme*), the data collection and data integration problems become manageable.

Case	Problem	Schema	Data Integration
1.	Known	Known	Easy
2.	Known	Many	Difficult
3.	Unbounded	Many	Hard
3b.	Unbounded	Graph Based	Manageable

A graph based model is good for data collection as well as data integration and more so for data exchange.

Triples "uRv" that form labeled edges "$u \xrightarrow{R} v$" in a directed graph is a good data representation model. Triples "uRv" can also be seen as a binary relation "$R(u, v)$".

RDF is such a triple based data representation model.

Resource Description Framework (RDF)

RDF is a data model. RDFS is a language for creating RDF data.

> Like other data models it has a data representation scheme and a purpose. For example, ER world contains attributes, entities, relationships, and constraints. Relational world contains attributes, relations, and uniqueness and referential constraints. Object oriented world contains attributes, objects, and object references. And XML world contains ordered list of name-value pairs (JSON), where a value is in turn an ordered list of name-value pairs, so the XML world is nested. ....
>
> ER model is used for conceptual, logical and physical design; and relational, object and XML models are used for physical design. See, stackoverflow (ans 1, 2) and data modeling levels.
>
> A given information can be represented in many data models.

The RDF world contains resources and statements.

A statement is a triple of the form "`<resource>` `<property>` `<value>`." And properties and values are also resources. A property relates a resource to another resource, the name of the property indicates the nature of the relationship.

Each statement (triple) is a labeled directed edge, and a collection of statements (triples) form a graph.

A single statement (triple) by itself is a graph, a simple graph. When you collect triples to form a graph, in fact, you are collecting small graphs to form a bigger graph.

In the RDF world everything is a resource, including physical objects, abstract concepts, data values, data types, etc., even the terms in the RDF framework are treated as resources.

If one can make a statement about something then that thing is a resource. In that sense properties are also resources. We can talk about the domain and range of a property.

Statements and Graphs

An RDF statement (an RDF triple) has different appearances.

```
<resource>  <resource>   <resource>.        (lexical view)
<resource>  <property>   <value>.          (data view)
<subject>   <predicate>  <object>.         (logic view)
```

Each view adds meaning to the triple and makes it human friendly.

Example 1:

Mr. Gandhi was born in Porbandar, he led the Dandi March.

A straight translation will produce two RDF triples that form a graph of 3 nodes and 2 edges:

```
<Mr. Gandhi>  <was born in>  <Porbandar>.
<Mr. Gandhi>  <led>  <the Dandi March>.
```

We can simplify the nodes to get a graph of 4 nodes and 3 edges:

```
<Gandhi>  <type>     <Man>.
<Gandhi>  <born-in>  <Porbandar>.
<Gandhi>  <led>      <Dandi March>.
```

Example 2: A syntactic variant of example 1.

Dandi March was led by Mr. Gandhi; Porbandar is his birth place.

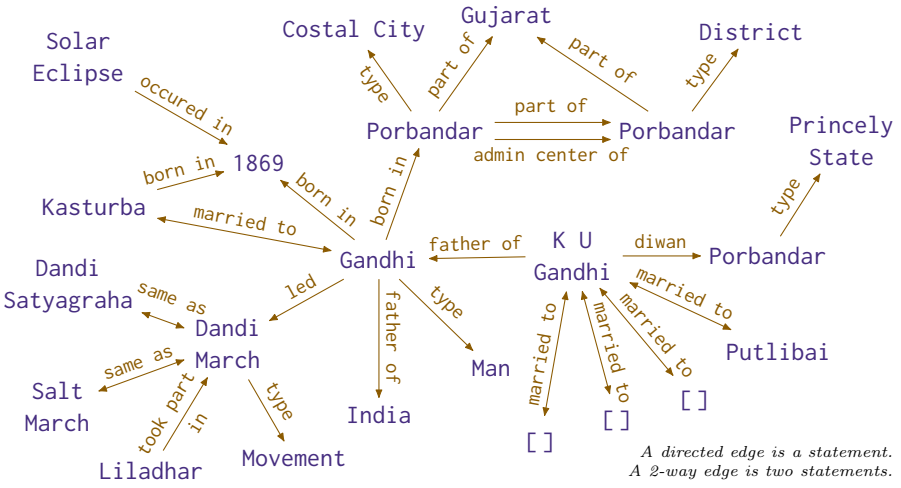
A simple RDF model would produce a graph of 4 nodes and 3 edges:

```
<Gandhi>       <type>            <Man>.
<Dandi March>  <led-by>          <Gandhi>.
<Porbandar>    <birth-place-of>  <Gandhi>.
```

These two examples capture the same information using active and passive sentences, respectively.

To establish equivalence between these examples, we have to capture the relationship between the properties <led> and <led-by> and <born-in> and <birth-place-of>. Later we will see how such relationships are defined.

RDF statements (triples) form graphs. The following graph with human readable labels is made of 32 statements that makeup 32 directed edges and 23 nodes, three of which are blank nodes "[ ]".



Several interesting observations can be made from this graph. Nodes refer to things we care about in the domain, or refer to literals (numbers, strings, date, XML string, HTML string, ...).

Nodes can have multiple edges of the same kind ("born in", "married to"). A pair of nodes can have multiple edges between them.

Porbandar occurs thrice: once as city, then as district, and as state; such terms are homonyms — same word with multiple meanings. Also, "father of" and "born in" are used in multiple senses.

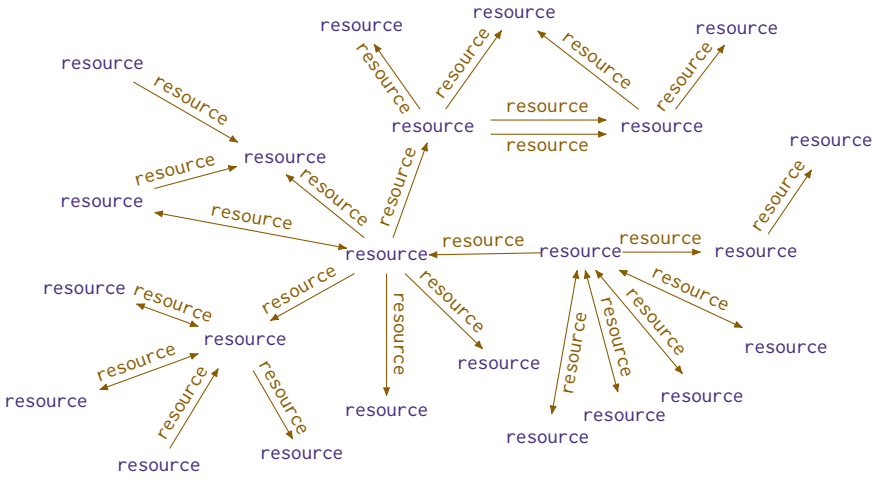
Dandi March, Dandi Satyagraha, and Salt March refer to one event; such terms are synonyms — multiple words with same meaning.

Homonyms are disambiguated using unique identifiers (not shown in this graph). And synonyms are merged using "same as" predicate, as shown in the graph. Note that, RDF processors do not recognize "same as" predicate, but OWL does.

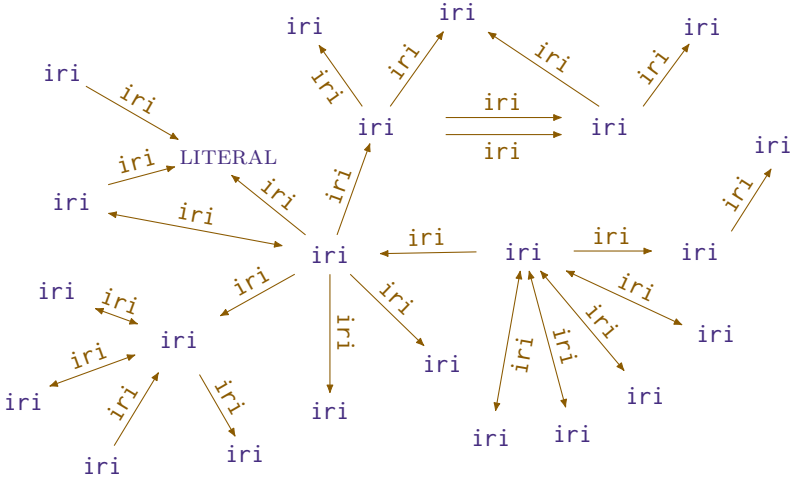
Properties are binary relations. In this graph, "married to" is a symmetric property, "same as" is a symmetric and transitive property, and "part of" is a transitive property.

RDF Graph

In the RDF world everything is a resource and we can make statements about resources using other resources called properties.
An RDF graph is a (linked) resource graph. Thus:



RDF resource are of three types: IRI's (resource identifiers), literals (plain or typed), and blank nodes. Therefore, at the encoding level an RDF graph looks like this:



Resource Identifiers and Namespaces

Identifiers (aadhar no., passport no., PAN no., license no., voter id, credit card number, employee no., facebook id, twitter id, email id, etc.) have a purpose, a format, and an allowable character set.

On the web, there are four kinds of resource identifiers, each with a purpose, a format, and an allowable character set.

1. URL (Uniform Resource Locator) points to a location on a computer network. `https://www.iitm.ac.in`
2. URN (Uniform Resource Name) is a persistent *location independent identifier*. `urn:isbn:9781259029981`
3. URI (Uniform Resource Identifier) is a URL or a URN. `http://www.w3.org/1999/02/22-rdf-syntax-ns#type`
4. IRI (International Resource Identifier) is the `i18n`[6] version of URI. It allows a bigger character set than URI. IRIs are converted to URIs at some point in the processing pipeline.

https://sa.wikipedia.org/wiki/मुख्यपृष्ठम् https://ta.wikipedia.org/wiki/விக்கிப்பீடியா

So an identifier may be a name (location independent) or a locator. And the identifier may be an internationalized version.

Semantic web frameworks (RDF/RDFS, OWL) use URIs and IRIs to identify resources.

A resource identifier has two parts: an XML namespace (prefix) and a resource name. The XML namespace is a URI; several resource names can be placed under it. See XML Namespaces for details.

Namespaces can nest; the nesting forms a tree, and a path from the root to a node forms a fully qualified name.

XML namespaces and Java namespaces (package names) serve the same purpose, i.e., to collect related items together and to keep unrelated items apart, and to disambiguate names.

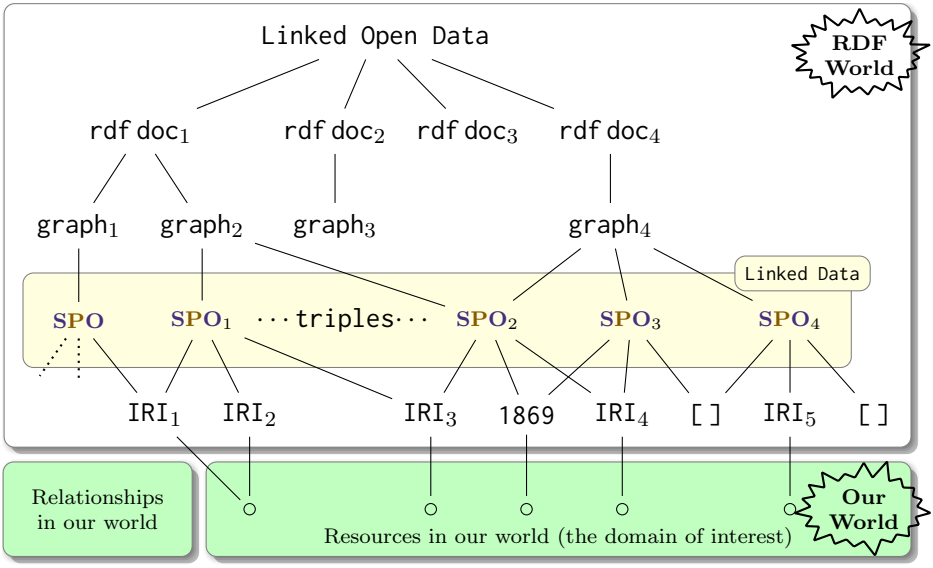
[6] `i18n` stands for "internationalization"; "i" 18 characters "n".

RDF Document

RDF document is a collection of graphs: a default graph (it has no name) and zero or more named graphs. Graph names are IRIs, we can use the IRIs to make statements about named graphs.

$\{\ G_0\ \ (N_1, G_1)\ \ (N_2, G_2)\ \ \ldots\ \ (N_k, G_k)\ \ \}$ where $k \geq 0$

An RDF graph is a collection of statements, i.e., triples of the form "<subject> <predicate> <object>."



RDF triples are encoded as:

- <iri> | blank-node <iri> <iri> | blank-node.
- <iri> | blank-node <iri> "plain literal".
- <iri> | blank-node <iri> "typed literal"^^<type-iri>.

IRIs occur in any position, blank nodes occur only as subjects and objects, and literals occur only as objects. Often IRIs are written as prefixed names, for example, <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> is written as rdf:type. See turtle.

There are two worlds: our world (the domain of interest) which contains resources, and a model of our world, i.e., the RDF world which contains symbols (IRIs, blank nodes, and literals) that denote (stand for) resources in our world.

RDF world is mapped to our world.

IRIs and literals map to resources in our world, for every IRI and every literal there is a corresponding resource in our world.

Blank nodes (shown as `[]`) are place holders, they guarantee the existence of resources, but do not identify them directly.

IRIs, blank nodes and literals are mutually disjoint.

Each IRI maps to one resource and a resource may have many IRIs. Each blank node stands for one resource.

IRIs have global (WWW) scope, all occurrences of a given IRI from all RDF documents denote the same resource. IRI equality is based on simple string comparison (see section 5, Normalization and Comparison, in RFC3987).

Blank nodes have local scope that is limited to an RDF document. Blank nodes are visible to (and are shared between) all graphs in an RDF document. The scope may expand to a triple store (created from multiple RDF documents), this scoping is vendor specific.

Blank nodes' scope is also local in time, i.e., two accesses to the same document via different interfaces will return two sets of blank node identifiers, and these sets are *not comparable*, of course, within a set the blank node identifiers are comparable.

Blank node identifiers are runtime artifacts that vary for one session to another. Blank nodes in two query results (from the same query run at different times on the same document) are not comparable.

If you permit only one operation per assignment then $y = x * x + x$ will decompose into $t = x * x$, $y = t + x$, and t is an intermediate variable. Blank nodes indicate (*i.*) intermediate resources in a triple based representation or (*ii.*) unknown but existing resources.

RDF Logical Foundation

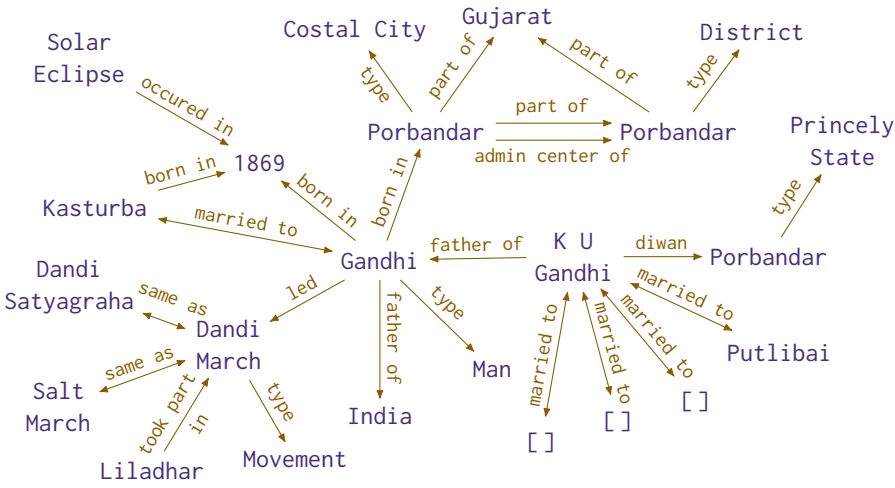
We can interpret RDF documents in a logic setting:

1. Map the property IRIs to binary relation names.
2. Map the triples to binary relations.
3. Map the IRIs and literals to resources in the world.
4. Treat the blank nodes "[ ]" as existential variables or as Skolem constants (sk-1, sk-2, . . .).

Now, in this simple interpretation:

1. A triple is true if the corresponding binary relationship is true in the domain of interest.
2. A graph is true if all its triples are true.
3. An empty graph is always true.
4. An RDF document is true if all its graphs are true.

The logical formulation of the following RDF graph is shown next.



same-as($\text{IRI}_{\text{Dandi March}}$, $\text{IRI}_{\text{Salt March}}$)
same-as($\text{IRI}_{\text{Dandi March}}$, $\text{IRI}_{\text{Dandi Satyagraha}}$)
and two inverses ...

born-in($\text{IRI}_{\text{Gandhi}}$, $\text{IRI}_{\text{Porbandar}}$)
born-in($\text{IRI}_{\text{Gandhi}}$, 1869)
born-in($\text{IRI}_{\text{Kasturba}}$, 1869)

married-to($\text{IRI}_{\text{Kasturba}}$, $\text{IRI}_{\text{Gandhi}}$)
married-to($\text{IRI}_{\text{K U Gandhi}}$, $\text{IRI}_{\text{Putlibai}}$)
married-to($\text{IRI}_{\text{K U Gandhi}}$, sk-1)
married-to($\text{IRI}_{\text{K U Gandhi}}$, sk-2)
married-to($\text{IRI}_{\text{K U Gandhi}}$, sk-3)
and five inverses ...

led($\text{IRI}_{\text{Gandhi}}$, $\text{IRI}_{\text{Dandi March}}$)
took-part($\text{IRI}_{\text{Liladhar}}$, $\text{IRI}_{\text{Dandi March}}$)

type($\text{IRI}_{\text{Gandhi}}$, IRI_{Man})
type($\text{IRI}_{\text{Dandi March}}$, $\text{IRI}_{\text{Movement}}$)
type($\text{IRI}_{\text{Porbandar}}$, $\text{IRI}_{\text{Costal City}}$)
type($\text{IRI}_{\text{Porbandar-2}}$, $\text{IRI}_{\text{District}}$)
type($\text{IRI}_{\text{Porbandar-3}}$, $\text{IRI}_{\text{Princely State}}$)

part-of($\text{IRI}_{\text{Porbandar}}$, $\text{IRI}_{\text{Gujarat}}$)
part-of($\text{IRI}_{\text{Porbandar}}$, $\text{IRI}_{\text{Porbandar-2}}$)
part-of($\text{IRI}_{\text{Porbandar-2}}$, $\text{IRI}_{\text{Gujarat}}$)

admin-center($\text{IRI}_{\text{Porbandar}}$, $\text{IRI}_{\text{Porbandar-2}}$)

diwan($\text{IRI}_{\text{K U Gandhi}}$, $\text{IRI}_{\text{Porbandar-3}}$)

father-of($\text{IRI}_{\text{K U Gandhi}}$, $\text{IRI}_{\text{Gandhi}}$)
father-of($\text{IRI}_{\text{Gandhi}}$, $\text{IRI}_{\text{India}}$)

occured-in($\text{IRI}_{\text{Solar Eclipse}}$, 1869)

The RDF graph is true if each of these predicates (consider IRIs as variables that stand for resources) are true in our world and also their conjunction is true in our world.

IRIs denote resources in our world, there is a mapping between IRIs and resources, and hence the binary predicates apply to resources denoted by the IRIs and not to the IRI strings.

So, same-as($\text{IRI}_{\text{Dandi March}}$, $\text{IRI}_{\text{Salt March}}$) is true when the resources denoted by $\text{IRI}_{\text{Dandi March}}$ and $\text{IRI}_{\text{Salt March}}$ are the same in our world.

> same-as(...) is not built-in RDF/RDFS, but applications can interpret it as a symmetric and transitive relation.

Similarly, type($\text{IRI}_{\text{Gandhi}}$, IRI_{Man}) is true if the resource denoted by $\text{IRI}_{\text{Gandhi}}$ is a type of the resource denoted by IRI_{Man}.

> The type relationship is between resources in our world and not between IRI strings.

Similarly, every property/predicate must be true in our world.

RDF Vocabulary

RDF allows us to define properties, reified statements, lists, and containers like bags, sequences, and alternatives.

The RDF vocabulary (a set of keywords with associated meanings) is divided into Classes, Instances, and Properties. Refer RDF Schema 1.1 for a full account.

Classes	Instances	Properties
`rdf:Property`	`rdf:type`	
`rdf:Statement`		`rdf:subject`, `rdf:predicate`, `rdf:object`.
`rdf:List`	`rdf:nil`	`rdf:first`, `rdf:rest`.
`rdf:Bag`, `rdf:Seq`, `rdf:Alt`.		`rdf:_1`, `rdf:_2`, ..., `rdf:_nnn`, `rdf:li`.

`rdf:type` is a fundamental property, it maps a resource to its class. We can make type-statements or infer type statements from other statements. We will see the latter later.

```
:Gandhi  rdf:type  :Man.
rdf:nil  rdf:type  rdf:List.
```

Compare the above type-statements to that in other languages.

Set Theory	SWI Prolog	C++
`Gandhi ∈ Man`	`man(gandhi).`	`Man Gandhi;`
`nil ∈ List`	`list(nil).`	`List nil;`

Man and List are given, we are not defining those classes, we are merely defining instances of those classes.

Observe the syntax, for this example, the statements in set theory resemble RDF triples, and C++ resembles Prolog (ignore case).

In RDF, properties are first class citizens, they exist independent of classes. In Java, member variables and methods cannot exist outside a class, they are lost when the enclosing class is deleted.

In RDF, all properties are instances of the `rdf:Property` class. In every RDF statement, the middle element is taken to be a property, regardless of whether that property has been defined or not.

Thus, from $S \quad P \quad O.$

we conclude P `rdf:type` `rdf:Property`.

And we can define new properties using a type-statement:

```
:born-in  rdf:type  rdf:Property.
:part-of  rdf:type  rdf:Property.
```

Hence by usage, `rdf:type` is a property. It has a circular definition.

```
rdf:type  rdf:type  rdf:Property.
```

Other built-in properties are easily defined as below.

```
rdf:subject    rdf:type  rdf:Property.
rdf:predicate  rdf:type  rdf:Property.
rdf:object     rdf:type  rdf:Property.
rdf:first      rdf:type  rdf:Property.
rdf:rest       rdf:type  rdf:Property.
```

and more ...

In RDF, a property is a mapping that associates elements from one set (domain) to elements in another set (range).

The domain of `rdf:type` is the set of all resources, and its range is the set of all RDF types and classes. The need to express this condition along with the notion that all things are resources contribute to circular definitions in RDF/RDFS. Discussed in the next section.

The domain of `rdf:first` is set of all lists (`rdf:List` class), and the range is the set of all resources.

The domain and range of `rdf:rest` is `rdf:List` class.

RDF allows us to capture beliefs and assumptions.

Bob believes that Mary likes John.

The statement `<Mary> <likes> <John>` may or may not be true in the domain of interest. The terms "Mary", "John" and "likes" may not even exist in the domain. But, Bob believes in that statement.

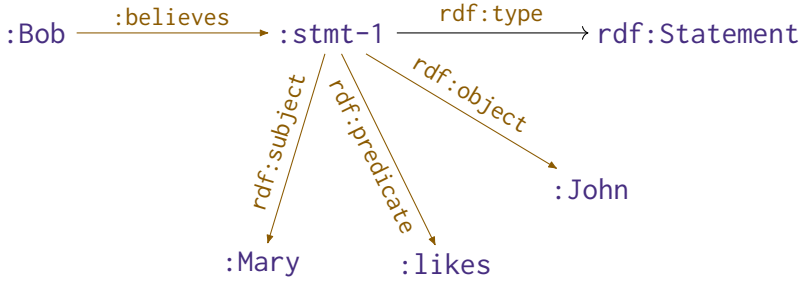
We can express Bob's belief by converting the belief statement into a resource and linking Bob to that resource.

```
:stmt-1   rdf:type       rdf:Statement.

:stmt-1   rdf:subject    :Mary.
:stmt-1   rdf:predicate  :likes.
:stmt-1   rdf:object     :John.

:Bob      :believes      :stmt-1.
```

Now, `:stmt-1` is a resource, it is a reified statement, a statement that is represented as data. Its truth value is not known.



RDF lists are instances of `rdf:List` class. Lists are encoded as first-rest pairs (like head-tail pairs in Haskell); `rdf:first` points to a list element and `rdf:rest` points to the remainder of the list. Thus, the list `(:Bob, :Bill)` is encoded as:

```
:list-1   rdf:first   :Bob.
:list-1   rdf:rest    _:tmp-2.
_:tmp-2   rdf:first   :Bill.
_:tmp-2   rdf:rest    rdf:nil.
```

Here, `_:tmp-2` is a blank node; and `:list-1`, `:Bob`, `:Bill` are IRIs from default namespace.

RDF Schema (RDFS) Vocabulary

Using only RDF vocabulary (keywords from RDF namespace) we cannot express much; to mention a few, we cannot create classes (Man, Woman, Person, List) or express subclass relationship or express domain or range constraints. RDF Schema (RDFS) provides an extended vocabulary that is good for developing real world models. RDF Schema 1.1 gives a full account.

In Java, the keywords "`class`" and "`extends`" are used to create classes and subclasses, respectively. The corresponding keywords in RDF are `rdfs:Class` and `rdfs:subClassOf`, respectively.

```
class Person {...}                          (Java class)
class Man extends Person {...}              (Java subclass)
```

```
:Person  rdf:type         rdfs:Class.       (RDF class)
:Man     rdf:type         rdfs:Class.       (RDF class)
:Man     rdfs:subClassOf  :Person.          (RDF subclass)
```

`rdfs:Class` is a meta-class, all its members are classes. The domain and range of `rdfs:subClassOf` is `rdfs:Class`, therefore it is a mapping between the members of `rdfs:Class` meta-class.

```
rdfs:subClassOf  rdfs:domain  rdfs:Class.
rdfs:subClassOf  rdfs:range   rdfs:Class.
```

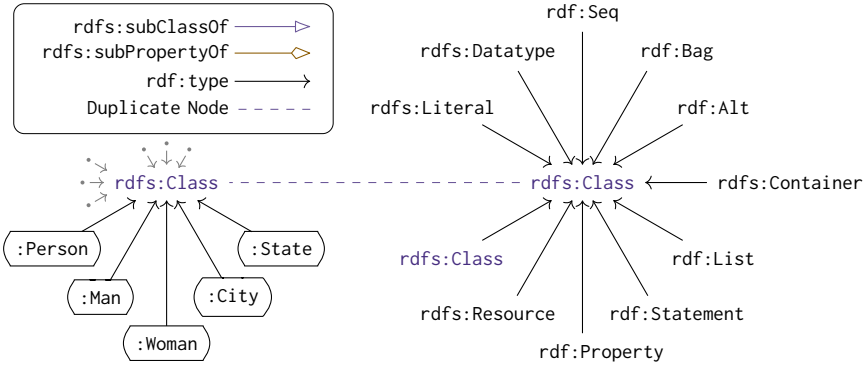
Now, we can simplify the Man-Person example as

```
:Person  rdf:type         rdfs:Class.
:Man     rdfs:subClassOf  :Person.
```

And infer "`<:Man> <rdf:type> <rdfs:Class>`" from the domain of the subclass statement.

In Java, each object of a class carry same variables and same methods, nothing more, nothing less, whereas, in RDF, individuals in a class carry properties that are affiliated with that class and in addition they may carry other properties not affiliated with that class.

The RDFS classes and their type-statements are shown in the RDF graph below. For readability, rdfs:Class is shown multiple times.



User defined classes are shown on the left and the built-in classes on the right. Both kinds are defined in the same way.

rdfs:Class is a class of all classes. By itself it is a class, therefore, it is a member of itself. This is another recursive definition:

rdfs:Class rdf:type rdfs:Class.

Other built-in classes as well as user-defined classes are created as instances of rdfs:Class. Now, go around the above graph to get:

rdf:Property rdf:type rdfs:Class.
rdf:Statement rdf:type rdfs:Class.
and more . . .

The big one is the class that represents all resources.

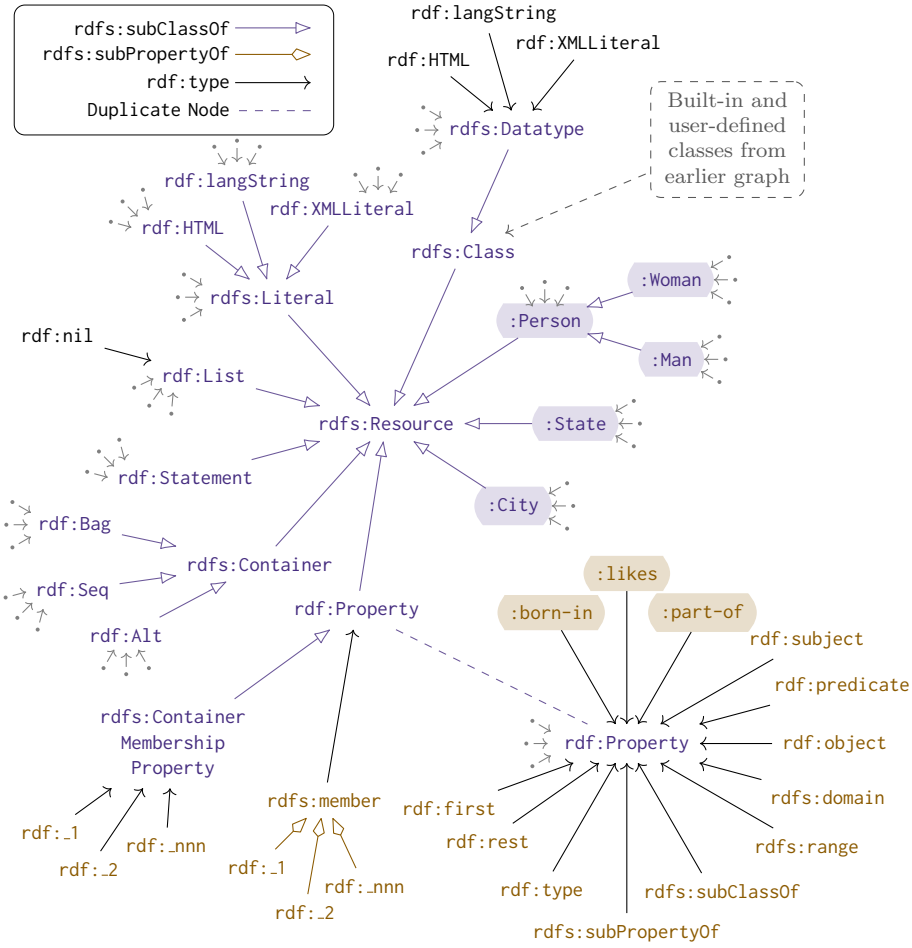
rdfs:Resource rdf:type rdfs:Class.

In RDF, all artifacts are resources — artifacts from the domain of interest, and artifacts from the RDF framework: such as the various classes, data types, properties, containers, etc., which are types that represent a space of values.

If rdfs:Resource contains everything, then the three levels: the rdfs:Class, its members which are classes, and their members belong in the rdfs:Resource class. Yet another recursion.

The above graph (a fragment of the following RDF graph) shows the built-in and user-defined classes in RDFS vocabulary.

The following RDF graph depicts the "all are resources" view. It also shows the world view — how the user sees his world.



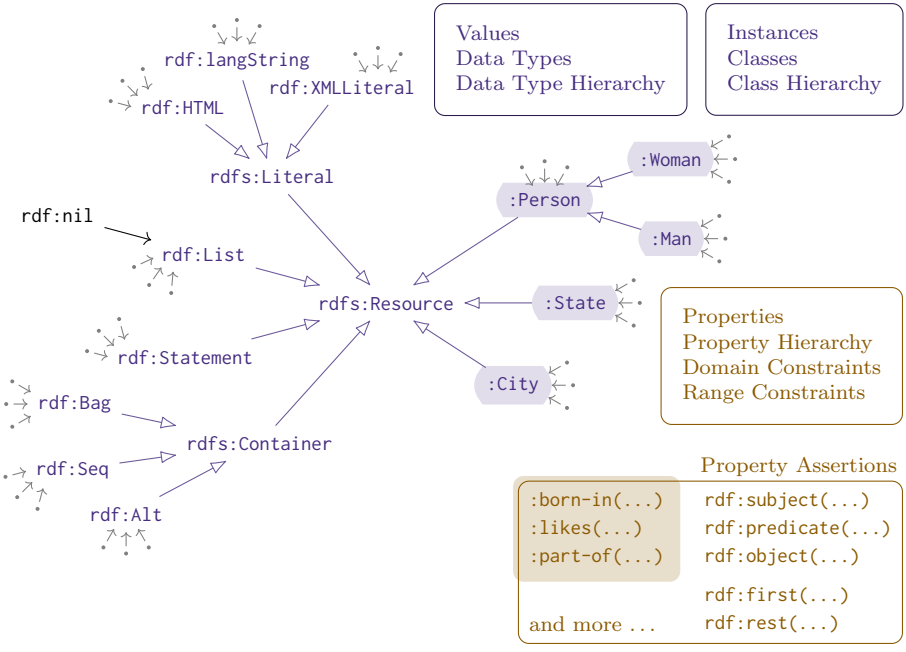
Like in other languages, RDFS supports classes, data types, and instances. `rdfs:Class` helps us to define classes and data types, and allows us to treat those types as resources, and make statements about them. There is a single type-space and several value-spaces.

In the RDF graph above, `rdfs:Datatype` and `rdfs:Class` represent types. And `rdfs:Literal` and `rdfs:Resource` represent their values (instances or members).

For example, `:Person`, `:Woman`, `rdf:langString`, `rdf:XMLLiteral`, etc., occur as types and also as a collection of values (or instances).

Observe that unary predicates and their members are resources, the properties (binary relations) are resources, but the tuples in the binary relations are not treated as resources, but the components of the tuples in the binary relation are resources.

In the above RDF graph, `rdfs:Class` and `rdf:Property`, and the descendants of those nodes are RDF artifacts. Hide those artifacts and the remaining nodes show the world as seen by the user:



A user or a domain expert sees the domain as containing values, data types, data type hierarchy, instances, classes, class hierarchy, property assertions, properties, property hierarchy, domain and range constraints, etc.

RDF is flexible. It allows a "Person" to be simultaneously defined as a class, a property, a statement, a container, etc., and it can be a member of itself. RDF does not complain or prevent us from doing so. The disjointness of the various categories (like classes, properties, etc.) are not enforced in RDFS.

```
:Person  rdf:type  rdfs:Class.
:Person  rdf:type  rdf:Property.
:Person  rdf:type  rdf:Statement.
:Person  rdf:type  rdf:Bag.
:Person  rdf:type  :Person.
```

Yes, RDF has circular definitions, it allows classes to be members of itself, and disjointness cannot be expressed, and more.

Nonetheless, RDF as a data model solves a big slice of the data integration and data exchange problem. RDF as a graph store (a triple store) is good for modeling complex relationships. The limitations of RDF, for the most part, do not impact the data management functions (create, read, update, delete) in a triple store.

> Note: RDF framework includes both RDF vocabulary and RDFS vocabulary. When we refer to RDF it may mean either the RDF framework or the RDF vocabulary, the surrounding context will make the meaning clear.
>
> "RDF is flexible" means that "RDF framework is flexible".

RDFS Logical Foundation

RDFS extends RDF. The RDFS interpretation, below, builds on top of the simple interpretation of RDF vocabulary presented before.

1. `rdfs:Resource` denotes the space of all resources.

2. Each C in a statement like "C `rdf:type rdfs:Class`" represents a class that is a subset of `rdfs:Resource`.

   Every class is a member of `rdfs:Class` meta-class. `rdfs:Class` is a class and it is a member of itself: `rdfs:Class rdf:type rdfs:Class`.

3. Each P in a statement like "P `rdf:type rdf:Property`" represents a binary predicate (or a binary relation).

4. Each V in a statement like "V `rdf:type rdfs:Literal`" represents a literal value.

5. Each T in a statement like "T `rdf:type rdfs:Datatype`" represents (a space) a space of values.

   Each member of `rdfs:Datatype` is a subclass of `rdfs:Literal`.

   Thus, T `rdf:type` `rdfs:Datatype`.
   gives T `rdfs:subClassOf` `rdfs:Literal`.

6. Each statement "P `rdfs:domain` A" restricts the domain of property P. The domain of P is now a subset of class A.

   Thus, P `rdfs:domain` A.
   and x P y.
   gives x `rdf:type` A.

7. Each statement "P `rdfs:range` B" restricts the range of property P. The range of P is now a subset of class B.

   Thus, P `rdfs:range` B.
   and x P y.
   gives y `rdf:type` B.

8. `rdfs:subPropertyOf` is transitive and reflexive on the set of properties (members of `rdf:Property`).

   Each statement "P `rdfs:subPropertyOf` R" restricts the property P to be a subset of the property R, i.e., all tuples of binary relation P are tuples of R.

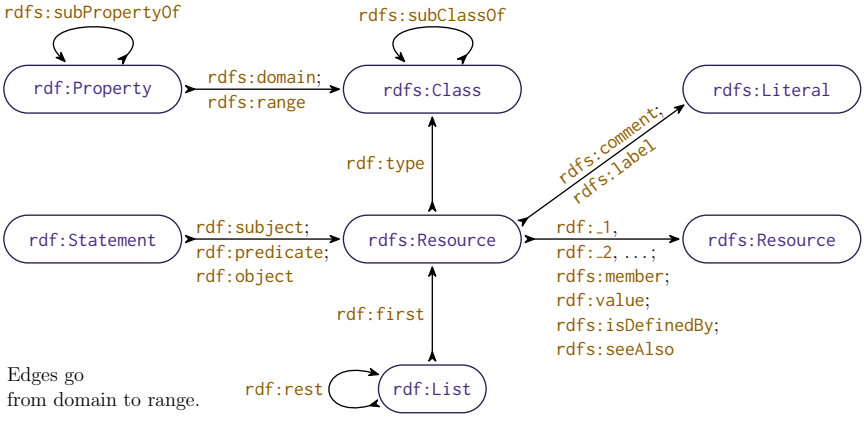
Thus,	P	`rdfs:subPropertyOf`	R.
and	x	P	y.
gives	x	R	y.

9. `rdfs:subClassOf` is transitive and reflexive on the set of classes (members of `rdfs:Class`).

   Each statement A `rdfs:subClassOf` B restricts the class A to be a subclass of the class B, i.e., the members of class A are members of class B.

Thus,	A	`rdfs:subClassOf`	B.
and	x	`rdf:type`	A.
gives	x	`rdf:type`	B.

10. A property as a binary relation has a domain and a range. The following domain and range constraints are enforced in RDFS:

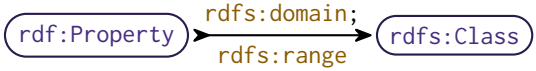


   Each directed edge in this graph states two constraints: a domain constraint and a range constraint.

For example, 

states:

```
rdf:type   rdfs:domain   rdfs:Resource.
rdf:type   rdfs:range    rdfs:Class.
```

And, 

makes 4 statements:

```
rdfs:domain   rdfs:domain   rdf:Property.    (circular ref.)
rdfs:domain   rdfs:range    rdfs:Class.

rdfs:range    rdfs:domain   rdf:Property.
rdfs:range    rdfs:range    rdfs:Class.      (circular ref.)
```

Similarly, for other edges.

RDF allows multiple type statements for a subject. Such statements are interpreted conjunctively.

```
:Indira   rdf:type   :Woman.
:Indira   rdf:type   :Parent.
```

Now, Indira is a woman and a parent; `:Indira` has a type that is an intersection of `:Woman` and `:Parent`.

Multiple domains statements are interpreted conjunctively:

```
:father-of   rdfs:domain   :Man.
:father-of   rdfs:domain   :Parent.
```

now, the domain is the intersection of `:Man` and `:Parent`.

In RDF, multiple statements (for type, domain, range, subclass, and sub-property constraints) produce an intersection of classes (or intersection of properties for sub-property constraint).

From Data to Logic Program

Given the logical foundation, now an RDF triple makes a logical statement about the world. And an RDF graph becomes a logic program. Now, we can talk about inferences. This is a big improvement, we moved from data to logic program.

Let us understand the implications through some graphs.

G_1 : *Priyanka is a grandchild of Indira.*

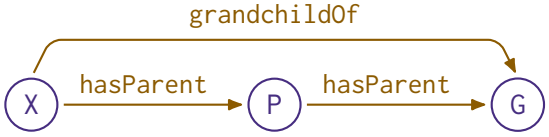
```
:Priyanka  :grandchild-of  :Indira.
```

G_2 : *Priyanka's parent is Rajiv who's parent is Indira.*

```
:Priyanka  :has-parent  :Rajiv.
:Rajiv     :has-parent  :Indira.
```

G_2 contains G_1. We need role composition to establish the containment. RDF cannot compose roles but extensions of RDF such as OWL can express compositions like:

```
grandchildOf(X,G) :- hasParent(X,P), hasParent(P,G).
```



G_3 : *Priyanka's father is Rajiv who's mother is Indira.*

```
:Priyanka  :has-father  :Rajiv.
:Rajiv     :has-mother  :Indira.
```

G_4 : *Priyanka is the daughter of Rajiv who is a son of Indira.*

```
:Priyanka  :daughter-of  :Rajiv.
:Rajiv     :son-of       :Indira.
```

G_5 : *Indira has a son Rajiv who has a daughter Priyanka.*

```
:Rajiv   :has-daughter  :Priyanka.
:Indira  :has-son       :Rajiv.
```

I hope you got the point? The same information can be modeled in several ways, some models are richer than others. And some models contain other models. And models may have a non empty intersection, and more.

Here, G_1 is contained in each of G_2 thru G_5. And G_2 is contained in G_3, etc. By introducing the classes `:Man` and `:Woman` we can find more relationships between these examples. For now:

G_3 will entail G_2 if the following is given. This works in RDFS.

```
:hasFather  rdfs:subPropertyOf  :hasParent.
:hasMother  rdfs:subPropertyOf  :hasParent.
```

G_4 will entail G_2 in the presence of:

```
:daughter-of  rdfs:subPropertyOf  :hasParent.
:son-of       rdfs:subPropertyOf  :hasParent.
```

G_5 will entail G_4 and vice versa if inverses are defined.

```
:daughter-of  owl:inverseOf  :has-daughter.
:son-of       owl:inverseOf  :has-son.
```

We have to use OWL to define inverse properties. Given a property, its inverse is generated by simply reversing the arrows.

RDF documents represent data. By turning-on the logical interpretation, we are able to detect entailments and equivalences. And we can check for inconsistencies as well.

Can you infer the following from the above examples? What inputs do you need for such an inference?

Priyanka is the only granddaughter of Indira.

The answer is no, we can't. But OWL can, with additional inputs.

I ask the reader to investigate this problem.

RDF Entailments

An RDF triple encodes a statement about our world.

A triple is true when the relationship implied by the triple is true in our world. Otherwise it is false.

An RDF graph is true when all its triples are true.

An empty graph is always true.

Given a set of triples (a graph G) that is true, we may be able to infer other triples (another graph H) that must be true. In that case, we say that graph G entails graph H.

Thus,	`:bob`	`foaf:knows`	`:alice`.
and	`foaf:knows`	`rdfs:domain`	`foaf:Person`.
entails	`:bob`	`rdf:type`	`foaf:Person`.

Entailment:

> An RDF graph G entails another RDF graph H
> if every possible arrangement of the world
> that makes G true also makes H true.
>
> When G entails H,
> if the truth of G is demonstrated
> then the truth of H is established.

Equivalence:

> Two RDF graphs G and H are equivalent
> if they make the same claim about the world.
>
> G is equivalent to H, if and only if,
> G entails H, and H entails G.

Inconsistency:

> An RDF graph is inconsistent if it contains an internal contradiction. There is no possible arrangement of the world that would make the graph true.

SPARQL

SPARQL is a recursive acronym for *SPARQL Protocol and RDF Query Language*. In this section, we introduce SPARQL select-query, and we refer you to web resources for a full account.

> SPARQL Query Language for RDF,
>
> SPARQL by Example, and SPARQL by Example.

SPARQL is a pattern based query language.
A SPARQL query expresses a graph pattern.

Triples form graphs, and triple patterns form graph patterns.

The elements of a triple pattern are IRIs, literals, blank nodes, and variables (`?name`). Triple patterns are encoded as:

- `(iri | blank | var)  (iri | var)  (iri | blank | var).`
- `(iri | blank | var)  (iri | var)  "literal".`
- `(iri | blank | var)  (iri | var)  "literal"^^<type-iri>.`
- `var` is a name prefixed with "?", like `?x`, `?count`, etc.

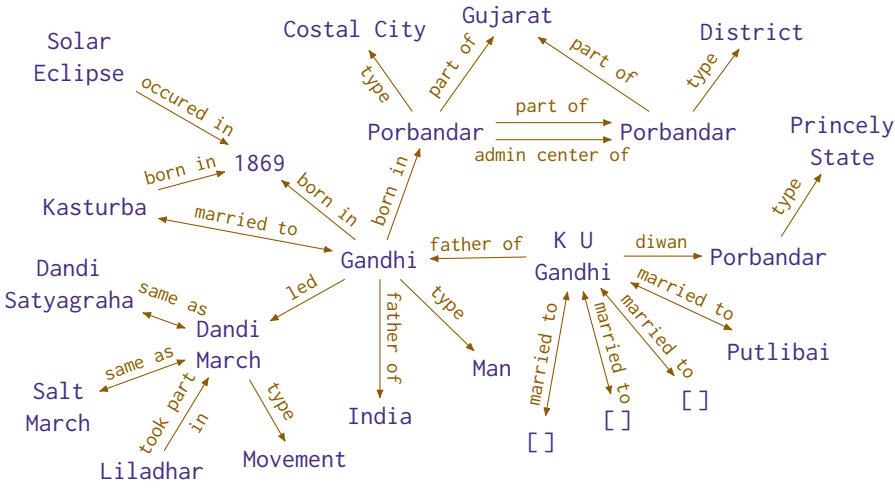
For example: `?x :born-in "1869"` is a triple pattern. The variable `?x` represents a wild card like the `"*"` in `"*.java"`. Variables are *named wild-cards* that can be referenced in other triple patterns.

A variable may bind/match to any of IRI, blank node, or literal. And depending on its position in the triple pattern, it may represent a subject, predicate or an object.

A variable may also derive its value from other variables.

The triple pattern `?x :born-in "1869"` will match the following triples in the example RDF graph shown below.

```
:Gandhi     :born-in  "1869".
:Kasturbai  :born-in  "1869".
```



A SPARQL select-query processes RDF triples and produces a relation as output. The where clause expresses a graph pattern along with a set of filter conditions.

A graph pattern is a directed graph that is constructed from one or more triple patterns.

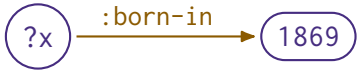
A single solution to a query is a subgraph that matches the graph pattern and also satisfies the filter conditions. The output of a single solution is a tuple prepared from the bindings of variables in the graph pattern — those listed in the select clause.

A query result is a sequence containing all solutions — all tuples (bindings) from all subgraph matches. Not a set but a sequence.

SPARQL queries operate at the data layer, there is no inherent reasoning involved. We have to externally turn-on the reasoner to include the inferred content in the query result.

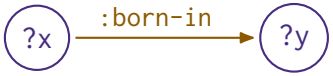
Example 1: Those who were born in 1869.

```
select ?x
where {
  ?x :born-in "1869".
}
```



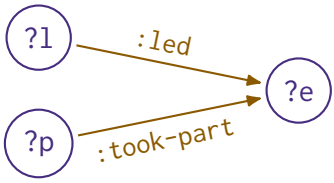
Example 2: Those who were born in 1860s.

```
select ?x
where {
  ?x :born-in ?y.
  filter (1859 < ?y && ?y <= 1869 )
}
```



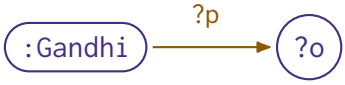
Example 3: Prepare a (leader, participant, event) relation.

```
select ?l ?p ?e
where {
  ?l :led       ?e.
  ?p :took-part ?e.
}
```



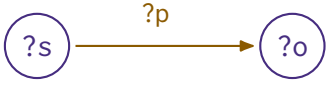
Example 4: Tell me everything about Gandhi.

```
select ?p ?o
where {
  :Gandhi ?p ?o.
}
```



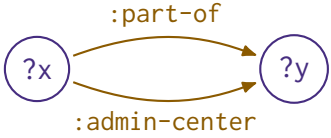
Example 5: Tell me everything about every resource.

```
select ?s ?p ?o
where {
  ?s ?p ?o.
}
```



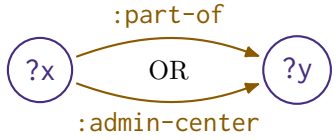
Example 6: List node pairs that are linked by part-of as well as admin-center edges.

```
select ?x ?y
where {
  ?x :part-of      ?y.
  ?x :admin-center ?y.
}
```

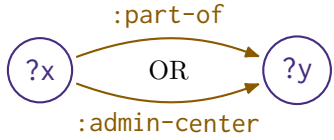


Example 7: List node pairs that are linked by either part-of or admin-center or both edges.

```
select ?x ?y  where {
  { ?x :part-of      ?y. }
  union
  { ?x :admin-center ?y. }
}
```

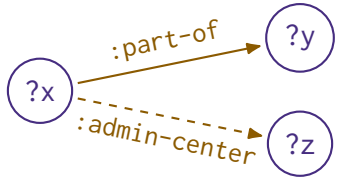



```
select ?x ?y
where {
  ?x :part-of|:admin-center ?y.
}
```



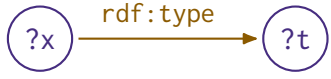
Example 8: Prepare a relation (x, y, z) where x is part-of y, and optionally, x is an admin-center of z.

```
select ?x ?y ?z
where {
  ?x :part-of ?y.
  optional {
    ?x :admin-center ?z.
  }
}
```



Example 9: List the various types used in an RDF document and the number of such type statements.

```
select ?t (count(?x) as ?count)
where {
  ?x rdf:type ?t.
}
group by ?t
```

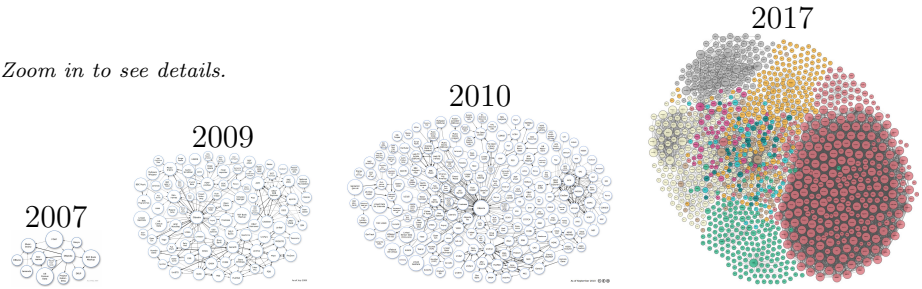


SPARQL provides a rich set of graph patterns, filter conditions, edge compositions, sub queries, derived variables, inline distributed access, etc. See SPARQL Query Language for RDF and SPARQL 1.1 Federated Query for details.

RDF in Practice

Linked Open Data (LOD) is a collection of open datasets that share and use resources from the collection. It is a large collection that is used for research and in the industry. LOD started out with 12 datasets in 2007 and gathered well over 1,000 datasets in 2017.

Zoom in to see details.



See http://lod-cloud.net for the latest version. Some datasets in the LOD collection are DBpedia, DBLP, Geonames, Wordnet, Freebase, etc. And this list is growing.

Linked data grows and evolves in a number of ways: by adding new datasets, by adding and updating triples, by adding cross references, by adding schema and constraints, by correcting errors and improving the quality of data.

Linked data may originate from structured sources sources (like relational or XML stores) or originate from information extraction systems like NELL, Open-IE projects, etc.

Linked data can be accessed in two ways: as data dumps, where a user can pull a data dump from a publisher, or through custom APIs, or through SPARQL end points.

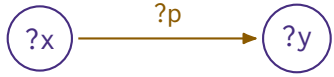
There are several public SPARQL end points, in general, they have resource limits (on query run time and result size) and are prone to traffic congestion. They are useful for exploration, research and education, and for asking lightweight queries.

http://dbpedia.org/sparql is a SPARQL end point, it is hosted on Virtuoso server. I encourage you to explore DBpedia through this end point, first, take a look at the namespace prefixes, then probe the dataset using queries. LIMIT the output to a small number.

Try the following in the DBpedia end point. Watch for timeouts.

Query 1: Properties used in DBpedia.

```
select distinct ?p
where {
  ?x ?p ?y.
} LIMIT 100
```

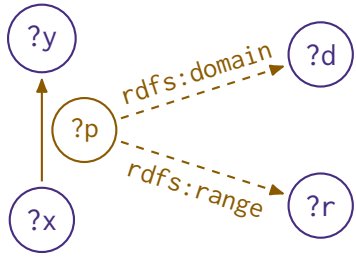


Query 2: Properties associated with the film *A Beautiful Mind*.

```
select distinct ?p
where {
  <http://dbpedia.org/resource/A_Beautiful_Mind_(film)> ?p ?y.
}
```

Query 3: List the properties, their domain and range.

```
select distinct ?p ?d ?r
where {
  ?x ?p ?y.
  optional { ?p rdfs:domain ?d. }
  optional { ?p rdfs:range  ?r. }
}
```



This query (which scans the entire triple store) may timeout, so restrict the triples to get a working query. Try the following.

```
select distinct ?p ?d ?r
where {
  dbr:Slumdog_Millionaire ?p ?y.
  optional { ?p rdfs:domain ?d. }
  optional { ?p rdfs:range  ?r. }
}
```

Query 4: Find the most specific classes (msc) of the resource *The Band Wagon*. A most specific class has no subclasses, therefore, it is a leaf in the class hierarchy (taxonomy).

```
select distinct ?msc
where {
  <http://dbpedia.org/resource/The_Band_Wagon> rdf:type ?msc.
  optional { ?sub rdfs:subClassOf ?msc. }
  filter ( !bound(?sub) )
} LIMIT 100
```

Because of resource limits, we cannot submit complex queries to end points. In such cases, we can use a dedicated end point or a local triple store or use RDF-3X to run complex queries.

Alternatively, we can use Apache Jena RDF API and OWL-API, these are programming interfaces that are good for creating, maintaining, and querying RDF datasets. These are in memory systems that are limited only by the system/server configuration.

Triple stores are a good choice for small to medium to large datasets. For example, AllegroGraph is a full scale client-server based triple store with ACID properties that supports a rich programming interface and SPARQL interface. It if free for up to 5 million triples.

Summary

RDF is here to stay and it has found its use case. RDF (a triples based data model) is good for representing data that is open, distributed, and multi-disciplinary — like the LOD datasets, NELL knowledge graph (NELL), etc.

Now, there is database and tool support, query language and APIs for managing RDF data. RDF has good standards and documentation, and a strong research and technical community.

RDF has limitations (or has too much expressive power depending on the perspective — data modeling or inference): RDF semantics has circular references, it lacks disjointness between the various RDF classes, and lacks constructs that are needed for modeling real world datasets. These limitations have less impact on the data management functions (create, read, update, delete).

In general, by choosing a triples representation, there is an apparent loss of (and lack of) structure and classification information. As discussed earlier, the structure and class information are inherent in the triples but it is not readily available to the user. That is why it is an apparent loss and not a real loss. Now, OWL and SWRL can help us to synthesize the structure and classes from RDF datasets. We will discuss this in the next session.

Acknowledgements

I thank Vinu and Rajeev (from AIDB Lab, CSE IITM) for putting together "RDF in Practice". I thank Prof. Sreenivasa Kumar, Subhashree, and Savitha (from AIDB Lab) for their review comments. And I thank Prof. Deepak Khemani for giving me the opportunity to present this material in the AI Summer School at IIT Mandi. And a million thanks to the summer school audience for their interest, questions and queries which shaped this notes.

References

[1] "DB-Engines Ranking." https://db-engines.com/en/ranking. Accessed 08-Aug-2017.

[2] "Database Benchmarks." http://www.tpc.org/. Accessed 08-Aug-2017.

[3] "Linked Data." http://linkeddata.org/. Accessed 08-Aug-2017.

[4] T. Mitchell and 24 others, "Never-Ending Learning," in *Proc. of the Twenty-Ninth AAAI Conf. on AI (AAAI-15)*, 2015.

[5] Y. Raimond and G. Schreiber, "RDF 1.1 Primer," W3C Note, June 2014. http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/. Accessed 08-Aug-2017.

[6] M. Lanthaler, D. Wood, and R. Cyganiak, "RDF 1.1 Concepts and Abstract Syntax," W3C Recommendation, Feb. 2014. http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/. Accessed 08-Aug-2017.

[7] R. Guha and D. Brickley, "RDF Schema 1.1," W3C Recommendation, Feb. 2014. http://www.w3.org/TR/2014/REC-rdf-schema-20140225/. Accessed 08-Aug-2017.

[8] R. Guha and D. Brickley, "RDF Vocabulary Description Language 1.0: RDF Schema," W3C Recommendation, Feb. 2004. http://www.w3.org/TR/2004/REC-rdf-schema-20040210/. Accessed 08-Aug-2017.

[9] W. Nejdl, M. Wolpers, and C. Capelle, "The RDF Schema Specification Revisited," 2000.

[10] P. Hayes and P. Patel-Schneider, "RDF 1.1 Semantics," W3C Recommendation, Feb. 2014. http://www.w3.org/TR/2014/REC-rdf11-mt-20140225/. Accessed 08-Aug-2017.

[11] A. van Kesteren, "URL Living Strandard." https://url.spec.whatwg.org. Accessed 08-Aug-2017.

[12] "Uniform Resource Names (URNs)." https://tools.ietf.org/html/rfc8141. Accessed 08-Aug-2017.

[13] "Uniform Resource Identifier (URI): Generic Syntax." https://tools.ietf.org/html/rfc3986. Accessed 08-Aug-2017.

[14] "Internationalized Resource Identifiers (IRIs)." https://www.ietf.org/rfc/rfc3987.txt. Accessed 08-Aug-2017.

[15] H. Thompson, T. Bray, R. Tobin, A. Layman, and D. Hollander, "Namespaces in XML 1.0 (third edition)," W3C recommendation, Dec. 2009. http://www.w3.org/TR/2009/REC-xml-names-20091208/. Accessed 08-Aug-2017.

[16] E. Prud'hommeaux and G. Carothers, "RDF 1.1 Turtle," W3C Recommendation, Feb. 2014. http://www.w3.org/TR/2014/REC-turtle-20140225/. Accessed 08-Aug-2017.

[17] G. Carothers and A. Seaborne, "RDF 1.1 N-Triples," W3C Recommendation, Feb. 2014. http://www.w3.org/TR/2014/REC-n-triples-20140225/. Accessed 08-Aug-2017.

[18] G. Schreiber and F. Gandon, "RDF 1.1 XML Syntax," W3C Recommendation, Feb. 2014. http://www.w3.org/TR/2014/REC-rdf-syntax-grammar-20140225/. Accessed 08-Aug-2017.

[19] L. Feigenbaum and E. Prud'hommeaux, "SPARQL By Example," Tutorial, June 2009. http://www.cambridgesemantics.com/semantic-university/sparql-by-example. Accessed 08-Aug-2017.

[20] L. Feigenbaum and E. Prud'hommeaux, "SPARQL By Example," Tutorial, June 2009. https://www.w3.org/2009/Talks/0615-qbe/. Accessed 08-Aug-2017.

[21] A. Seaborne and S. Harris, "SPARQL 1.1 Query Language," W3C recommendation, Mar. 2013. http://www.w3.org/TR/2013/REC-sparql11-query-20130321/. Accessed 08-Aug-2017.

[22] E. Prud'hommeaux and C. B. Aranda, "SPARQL 1.1 Federated Query," W3C Recommendation, Mar. 2013. http://www.w3.org/TR/2013/REC-sparql11-federated-query-20130321. Accessed 08-Aug-2017.

[23] "Apache Jena." https://jena.apache.org/index.html. Accessed 08-Aug-2017.

[24] "An Introduction to RDF and the Jena RDF API." https://jena.apache.org/tutorials/rdf_api.html. Accessed 08-Aug-2017.

[25] T. Neumann and G. Weikum, "RDF-3X: a RISC-style Engine for RDF," *PVLDB*, vol. 1, no. 1, pp. 647–659, 2008.

[26] T. Neumann and G. Weikum, "The RDF-3X Engine for Scalable Management of RDF Data," *The VLDB Journal*, vol. 19, Feb. 2010.

[27] "AllegroGraph Graph Database." https://franz.com/agraph/allegrograph/. Accessed 08-Aug-2017.

[28] "Neo4j Graph Database." https://neo4j.com/. Accessed 08-Aug-2017.

[29] "Titan Graph Database." http://titan.thinkaurelius.com/. Accessed 08-Aug-2017.