Connecting the dots...

A step towards Linked Open Data by incrementally visualizing an emergent RDF schema

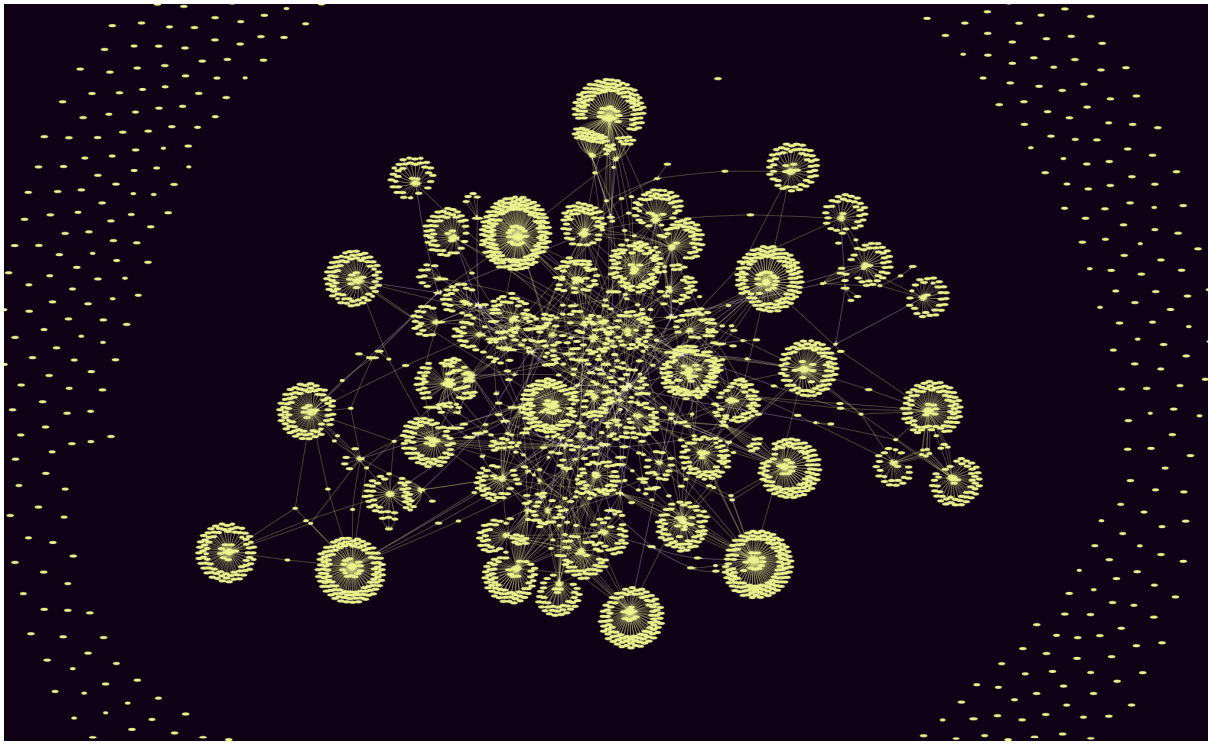
01-07-2022

Irma Mastenbroek

Master Thesis

Applied Data Science

Utrecht University



Supervisor: Dr. M. Behrisch

Second Supervisor: Prof. dr. ir. A.C. Telea

Abstract

Graphs are a popular way to store and visualize information. In the era of big data, the need for scalable and interactive visualization tools rises. In the Linked Open Data cloud, objects, events or concepts are described to have some pre-defined relationship to other entities. In a purely theoretical sense, one could create one huge knowledge graph of all interrelations of all available information known to man. However, due to historical reasons not all graph data structures are consistent and the visualization techniques for graphs do not scale up well. In this thesis, we will focus on emerging a schema of data stored in a Resource Description Framework (RDF) format. An example of data stored in RDF manner is DBpedia, a project aiming to extract all structured content created on Wikipedia. The retrieval of specific facts from an RDF dataset is often hindered by the lack of schema knowledge, further complicated by noisy data, annotation mistakes and time-outs when queries are too demanding. A schema can be seen as a clustering of the RDF-triples, downscaling the graph making interpretation and visualization possible. In this research we create an interactive node-link visualization of the global schema that enables users to discover the schema of any big RDF dataset. The tool is capable of iteratively expanding the graph as new parts of the graph arrive from the backend.

Contents

1 Introduction

In recent history a lot of our knowledge, human behaviour and interaction has been stored in various heterogeneous data structures. This ranges from non-digital books or archived files to continuously updated time series data collected by our phones. Domains such as science, telecommunications, social media and web-mining measure and extract huge amounts of digital data which are often times hard to integrate into down-stream applications.

In this section we describe how data heterogeneity problem stands in the way of working with Resource Descritpion Framework (RDF) data effectively. RDF is a general method of describing data by defining relationships between data objects without the need to adhere to a pre-defined relational schema. Being able to explore a schematic representation without prior knowledge can contribute to building effective tools to handle RDF data. Effective RDF tools, in turn, can catalyse the growth of the Linked Open Data (LOD) cloud. Lastly, we describe the contribution of this thesis; the streaming graph representation of a relational schema for RDF databases DBpedia [1] and WikiData[2].

1.1 The Data Heterogeneity Problem

In a relational database, instances of a class `Person` will always contain a subset of some pre-defined set of attributes. In case of `Person`, this set of attributes could be `\{dateOfBirth, livesIn, worksFor\}`. It is possible to store an instance of a person with missing values, however it is nearly impossible to store additional interesting information you find out about later, like this person's `favouriteMeal`. In essence, you need knowledge in advance about everyting that might be of interest later. A non-relational database structure like RDF does not require a pre-existing schema, making it easy to add unexpected information later on. As idyllic as that may sound, in practice non-relational databases are notoriously hard to perform analysis on, as it is difficult to explore data without some form of global schema knowledge. In theory continuously adding new knowledge to a graph should work, but due to the rapid pace of change, different traditions and the lack of general overview, selective data retrieval is compromised. Moreover, querying from an RDF source results in timeouts when the request is too big, adding to the importance of *knowing what you're looking for* in the data set. In order to create a global schema however, all data needs to be explored at least once.

1.2 Emerging a relational schema

This brings us to the goal of the backend part of this research project. In order for RDF-stored information to be a useful data source, we need it to have something like a semantic schema. The schema should roughly sketch what is going on inside the data set in order to be able to purposefully query for the usable data and avoid query timeouts. The idea behind RDF is that the data does not have to adhere to a particular schema, so no ground truth will be available to compare to our findings. Therefore, we need to clearly decide on our goals here. What is a good schema, when is it useful? Which instances have property sets that make them likely to represent the same *thing*? In case the reader is interested in the backend part of this research project, we refer them to Koray Poyraz's thesis. His thesis, pipeline and code are published online[3].

[1] https://www.dbpedia.org/
[2] https://www.wikidata.org/
[3] https://git.science.uu.nl/vig/mscprojects/rdf-schema-retrieval/

1.3 Vizualization of a global schema for big RDF sources.

A summary of the contribution of this thesis is given here. Once a partial schema structure is created by the backend, it is passed on to the frontend in batches of nodes and edges. In order for humans to explore and analyze the found schema, a sensible visual representation is needed. The user has the ability to get an overview of the global data structure as well as the ability to zoom into interesting subsets of the graph and interact with the nodes and edges. In this visualization it should be visible when a new partial schema is received, adding it to the existing partial schema and continuously update the visualization.

The contribution of this thesis will be visualizing the emergent schema of well known RDF sources DBpedia and WikiData. The visualization of the schema will be made using a predefined library (vis.js [4]), visualizing a graph with nodes and edges in a web-interface. The program is able to add, and simultaneously visualize, a new batch of schema nodes to the existing schema. The visualization yields an aesthetic graph, giving the user the opportunity to interact at any time with the graph by panning, zooming and clicking on nodes.

2 Background and Related Work

The following section is divided into three parts. In the first part the concept of Linked Open Data is described, and we mention how our research contributes to the interpretation of linked data. The next part describes how an RDF triple is stored. Next, the backend schema extraction problem definition is described. Lastly, we discuss different graph visualization techniques, aesthetic rules for graph drawing, scalability issues in large graph visualization and ways to incorporate interaction with the graph.

2.1 Linked Open Data

Linked Open Data (LOD) is one of the core pillars of the Semantic Web, often thought of as a virtual linked data cloud where anyone can extract and add data without disturbing the original data source. In 2006 the pioneer of the world wide web, Tim Berners-Lee, coined the term "Linked Data" as the foundation of "Web 3.0". The basic principle behind LOD is that data is more meaningful if it can be connected to other data. For sources like Wikidata, DBpedia, GeoNames and GRID, RDF is the current standard. The principles of this structure of information are that 1) all conceptual things should have an HTTP address on the web, 2) looking up any HTTP should return useful information about the thing and 3) anything else the thing has a relationship with through its data should also have an HTTP.

One can add information to the semantic web about any kind of object or concept, linking one URI to another URI by specifying their relationship. URI is an abbreviation of Uniform Resource Identifyer, a unique sequence of characters that identifies any type of object, including real-world objects like people and places, concepts, data or any information resources such as web pages and books. In his 2009 TED-talk [6], Tim Berners-Lee addresses all internet users: "I said, 'Could you put your documents on this web thing?' And you did. Thanks. It's been a blast, hasn't it? (...) Now, I want you to put your data on the web. Turns out that there is still huge unlocked potential." Since then, linked open data has indeed grown, albeit modestly. The LOD cloud has gained popularity due to the 2012 launching of the knowledge graph by Google [16]. As of june 2022, well over a decade later, 1568 data sets are interlinked. This number is

[4] https://vis.js.org/

growing yearly, but takes up less than 0.005% of all publicly known data sets [2].

So why is Linked Data still marginal? As suggested by [4] it failed to attract IT vendors, as making a profit of non-proprietary technologies is a lot harder and is therefore less appealing for investors. This is amplified by the fact that the *semantic web* is perceived as being overly academic and not yet widely applicable. The lack of investments and innovative projects has resulted in today's absence of well functioning frontend tooling. For web developers the tools are not targeting existing skill sets, they are difficult to install and they are unreliable, meaning a tool may be taken out of development within a few years. A modest turning point was 2017, when SHACL [1] and Amazon's Neptune [5] got launched. Moreover, there exist more JavaScript libraries for querying RDF, making sure that frontend developers can query directly from the source.

Traditional data integration frameworks often ingest data from various sources and store it in a unified representation, usually in a relational format. In these relational databases, a relational schema (Figure 1) represents in what way concepts relate to each other, and which attributes might be attached to each of those concepts. To create a relational database, or add new data to it, schema knowledge is necessary. This characteristic is especially fiddly when merging several relational DBs with dissimilar schemas to describe similar concepts. Databases that adhere to a pre-defined schema are sometimes referred to as a "schema first" approach. When it is preferred to have more flexibility for the user to represent and evolve data without schema knowledge, the go-to database structure is one that enables "schema last".

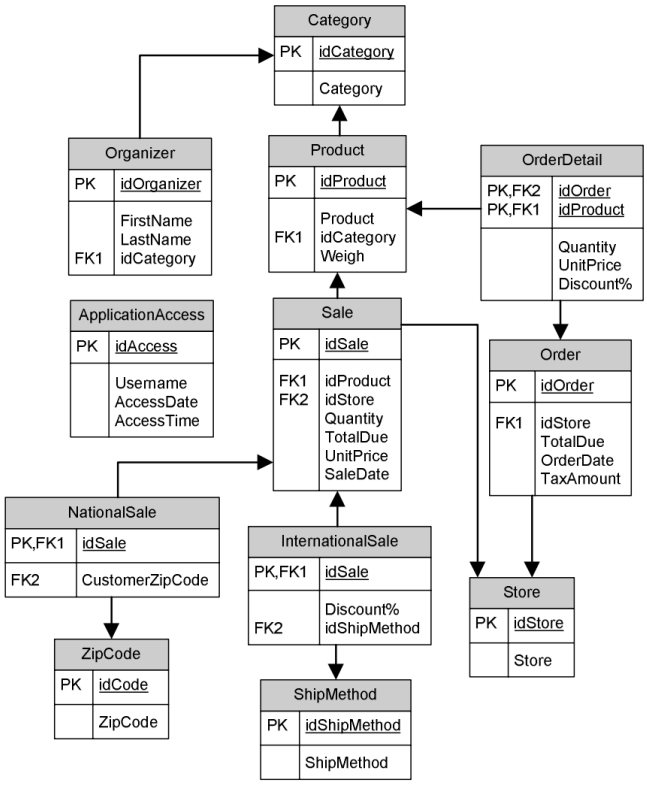


Figure 1: The typical lay-out of a relational schema. In relational databases, this structure is a blueprint of how the data set looks like. Source: [8]

Due to the vast variety of concepts, loose standards and the evolution over time, one can see why extracting a global schema is a particularly ambiguous endeavour. The lack of schema knowledge is in the way of effectively querying LOD for analytical purposes, as SQL-speaking database systems require to declare a schema upfront.

2.2 RDF

An RDF statement is often referred to as an RDF-triple. A triple can be seen as a directed labeled edge in a knowledge graph. The triple consists of two URIs, the subject and the object, and their relation, the predicate.

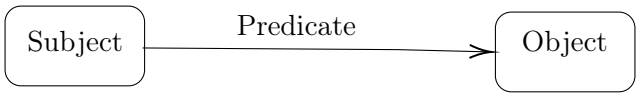


Figure 2: Caption

The nodes are one of three types, URIs, blank nodes or literals. The subject is always some concept, meaning it is an URI or a blank node. Blank nodes indicate the existence of a thing without using an URI to identify a solid concept. For instance when the attributes `street, zip-code` and `town` all point to one particular yet unspecified house. The third entry of the triple, the object, is either an URI, a blank node or a literal. A literal is not another concept or "node", but can be seen as a node property of the subject node instead.

An RDF statement is not necessarily machine readable code, conceptually it is a way of describing things and concepts using sets of three terms. Valid RDF decriptions are "Apples are green" or "Apples grow on trees". In theory, anything in this general format could be added to this abstract network of all knowledge, all it needs is an object, subject and their relation.

Perhaps confusingly, RDF is also the name of a Sementic Web schema, more commonly referred to as *vocabularies* and *ontologies*. These are sets of externally stored definitions which can be reused to describe data. In the statement "Apples are fruits" is represented as a `<subject, property, object>` triple

```
<http:/.../Apples> rdf:type <http:/.../Fruits>.
```

Here, `rdf:type` is a property stored in the RDF ontology describing that the subject (apples) is a subcategory of the object (fruits). Ontologies provide pre-defined classes, properties, individuals, and data values stored as Semantic Web documents. These definitions might look like they could function as a good base for a relational schema. However, in [13], the researchers show that actual RDF datasets exhibit only a very *partial* use of ontology classes, and subjects share triples with properties from classes defined in *multiple* ontologies. They found in DBpedia that each subject combines information from more than eight different ontology classes on average. Therefore, being faced with a schema description task, we will explore methods that find clusters of similar concepts by exploiting the similarity in property sets, therefore independent of the web-vocabularies.

2.3 Schema extraction problem description

In order to visualize a schema of knowledge data, we need to discuss what we want to learn from looking at the visualization in the frontend. The problem description that follows is the

semantic part of the task tackled in the backend. We will follow the definitions proposed in [11].

Let us first solidify what a schema for an RDF database might look like. In some data structures, a schema is a strict representation to which the data conforms, e.g. a blueprint, however this is not the case in the context of web data. Alternatively, a schema extracted from RDF data can be understood as a guide to interpret the data, without needing to explore the data by hand. The reason for not being able to find a strict representation for generic RDF data is that two entities of the same type may have a different set of properties, and an entity may have several types.

2.3.1 Problem definition

Let R denote a set of resources and B a set of blank nodes denoting anonymous resources. P denotes a set of properties and L a set of literals. An entity a is defined by $a \in R \cup B, a \notin L$. A data set in RDF format is defined as a set of triples D, where

$$D \subseteq (R \cup B) \times P \times (R \cup B \cup L)$$

In RDF data visualization this can be transformed to a graph $G(V, E)$, where a vertex $v \in V$ is drawn from resources, blank nodes or literals, and a labeled, directed edge $e \in E$ is an element from our triple set D.

The goal is to extract a schema S from an RDF data set D with incomplete schema information. The problem we face is how to infer type definitions and links between types. Such a schema S is composed of

- A set of possibly overlapping classes $C = \{C_1, ..., C_n\}$ where each C_i corresponds to a set of entities, defining their type.
- A set of links $\{p_1, ...p_m\}$ such that each p_i is a property mapping one class to another.
- A set of hierarchical links involving two types corresponding to classes in C, expressing that one is the generic type of the other.

The crutch lies in evaluating the similarity between entities in order to assign them to their "correct" class. Entities of the same type could be described by heterogeneous property sets, and a given entity may have several types. A similar problem arises when identifying the links between class-types on the basis of existing properties between entities, as not *all* instances of an entity will have direct link with instances of another entity. We face a challenge concerning the *semantic ambiguity*, where similar conceptual information are stored in dissimilar formats or vice versa. In case this problem sparks the readers interest, we refer to Koray Poyraz's thesis. In his work, the pipeline and similarity measures used are explained in depth. Therefore, in the scope of this thesis we focus solely on how the frontend will receive the partial schemas.

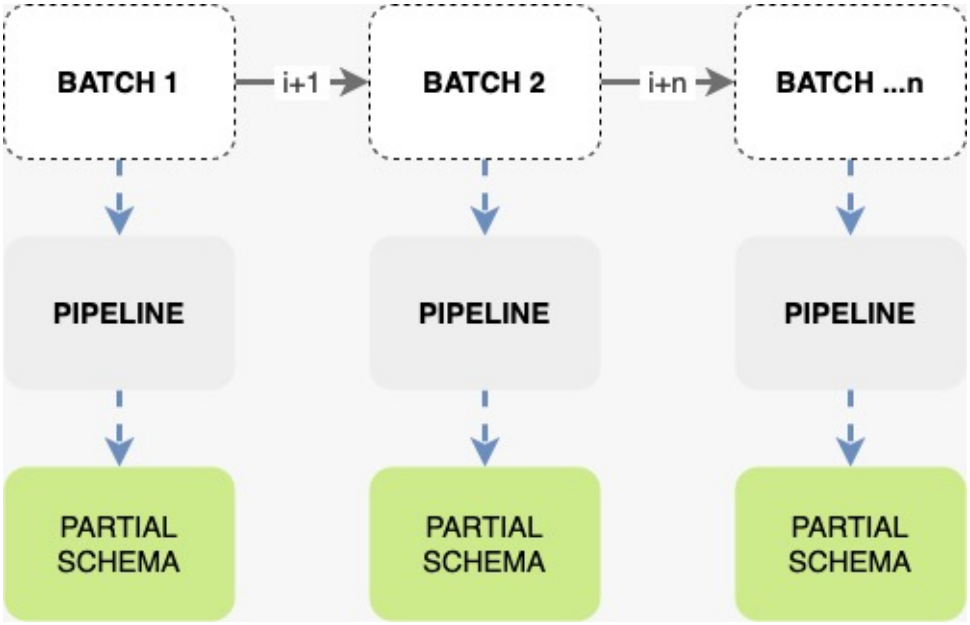


Figure 3: The frontend will receive partial schemas created independently by the backend. It is important that our visualization is capable of building a graph iteratively, adding new batches of nodes to the existing graph.

2.4 Graph visualization

The following section will give an impression of different graph visualization techniques and the objective and subjective measures that come into play. The graph drawing problem can be described as follows: given a set of nodes and edges, calculate the position of each node and the curve of each edge.

2.4.1 Aesthetics

Naturally, we need a set of quality constraints to objectively compare graph layouts. Examples of such objective goals are to minimize number of edge crossings or total edge curvature. Other, more *aesthetic* rules can be imposed on the final layout: nodes and edges must be evenly distributed, edges must be straight lines, etc. For trees, an additional aesthetic rule is to place nodes of equal depth on the same horizontal line. Sometimes, these aesthetic rules arise because of the clear benefit they yield for the viewer, with respect to interpretation and usability of the layout. This is not the case for each objective aesthetic measure. For instance, when drawing a graph in a non-deterministic way, an aesthetic constraint is used as an objective function in minimization problems. The measurable aesthetic objective can be of practical nature.

The absoluteness of these measures have been questioned by conducting usability studies. In usability studies, the focus lies on the end-user perspective and they aim to solidify the relative importance of aesthetic objectives. In a survey conducted by Purchase [15], the recipients are confronted with tasks that require relational reading of the graph, such as finding the shortest path between two nodes or removing the smallest set of edges in order to disconnect two given nodes. She concludes that "reducing the crossings is by far the most important aesthetic, while minimizing the number of bends and maximizing symmetry have a lesser effect". These studies have recently gained credibility in the graph visualization community, recognizing their contribution to help focus on important issues in the area.

2.4.2 Scalability

Many graph drawing algorithms are computationally expensive, which causes issues when drawing graphs with a big amount of nodes. Not only does the rendering of a graph take too long, many methods are not scalable in general. They produce good results when the number of nodes is in the hundreds, but they are unusable for graphs with thousands of nodes. An example is the classic hierarchical tree structure, where the leafs become unreadable because the nodes are densely packed. In studies [14] and [12], alternative layouts for trees are proposed to overcome these shortcomings.

Figure 4: A collection of creative euclidean tree layouts. The left hand side displays four layouts proposed by [14]. Rectangular layouts highlight variation in branch length, whereas slanted layouts facilitate comparison of branch order. Circular layouts are most efficient when visualizing large numbers of nodes but make it more difficult to compare branch lengths. Radial layouts do not convey ancestral information and are most appropriate for unrooted trees. The right displays a novel tree map of research topics (containing over 5000 nodes) produced by [12] in 2022. It effectively shows a high-level structure, while staying compact and planar with no edge crossings. This method is said to be scalable up to hundreds of thousands of nodes.

With big graphs, the scalability of the method is further compromised if the method is bad at continuously updating the graph data. For nearly all layout algorithms, in worst case the entire layout needs to be recalculated upon adding new nodes and edges. Which is $\mathcal{O}(N)$ at least, with N being the number of nodes in the graph. The cheapest non-deterministic force-directed method, `BarnesHut`, runs in $\mathcal{O}(N \log N)$.

2.4.3 Navigation and interaction

Facilitating navigation and interaction is essential in information visualization. No layout algorithm can overcome the problems raised by the large sizes of graphs we encounter in numerous research domains. Zooming is almost always essential in big graphs. Zooming can take on two forms. *Geometric zooming* provides a blow up of the graph content. *Semantic zooming* enables information to change upon zooming. We see an example of this type of zoom in Figure 4 in the image on the right, where the information density is kept constant upon zooming, showing more detail when zoomed in. This naturally requires underlying clustering techniques, which is

an NP-hard problem in Euclidian space.

Moreover, traditional zooming and panning can be challenging for the end-user. To illustrate: imagine having to go from one part of the graph to the other, like going from Amsterdam to Kuala Lumpur on Google maps. If the program is lagging, the user overshoots when zooming out and panning, which then needs correcting. The problem with zooming in is that the view expands exponentially fast, making it difficult to zoom into the correct area and keeping the target point in view. This results in a non-monomatic approach to the target point, meaning we move further away from the target before moving closer. Moreover, one loses global context when zooming in. When this is considered problematic in terms of interpretation, methods referred to as *focus+context* can be applied. An example is fisheye distortion, where Euclidian space is distorted around a focal point, blowing up the information close to the mouse to give it more space.

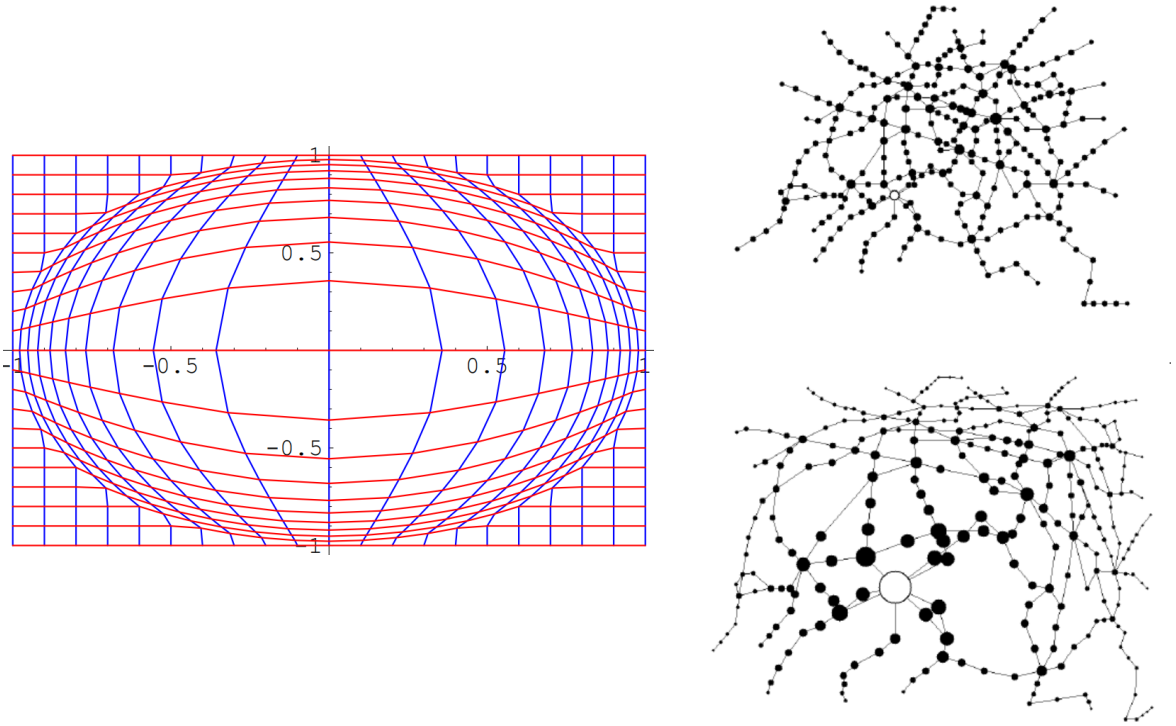


Figure 5: On the left a fisheye distortion of a regular grid is depicted with distortion factor 4. On the right is an example of the metro map of Paris with and without fisheye distortion. The area around the focal point is blown up, while the rest shrinks down but stays in view. Source: [10]

Moreover, extra interaction with the graph is possible by clicking on nodes and edges. As an example, edges connected to a node will change color upon clicking the node, or trigger a popup with node information.

2.4.4 Dynamic graph drawing

While static graphs arise in many applications, dynamic processes give rise to graphs that evolve through time. Nearly all existing approaches to visualization of dynamic graphs are based on the force-directed method. Early approaches by Brandes and Wagner [7] adapt the force-directed

model to a dynamic model using a Bayesian framework. Diehl and Görg [9] consider graphs in a sequence to create smoother transitions. Most of these early approaches, however, are limited to special classes of graphs and usually do not scale to graphs over a few hundred vertices.

Two important criteria to consider for the layout of evolving and dynamic graphs are:

- the *readability* of intermediate layouts, depending on how well these intermediate graphs adhere to aesthetic criteria; and
- the *mental map preservation* in the evolution of the graph over time. That is, in two consecutive graphs we want the nodes and edges to be more or less on the same place.

These two criteria are often contradictory. If we obtain individual layouts for each graph, without regard to other graphs in the series, we optimize readability at the expense of mental map preservation, and vice versa.

2.4.5 Visualizing RDF schemas

To our knowledge, no scientific papers have discussed a method to visualize a global RDF schema of big RDF databases. Some basic schema visualizations are available for small subsets of bigger RDF datasources. These schemas are not automatically generated, but require domain knowledge and cleaning by hand. An example of a relational schema for a biomedical subset of WikiData is given in Figure 6 below.

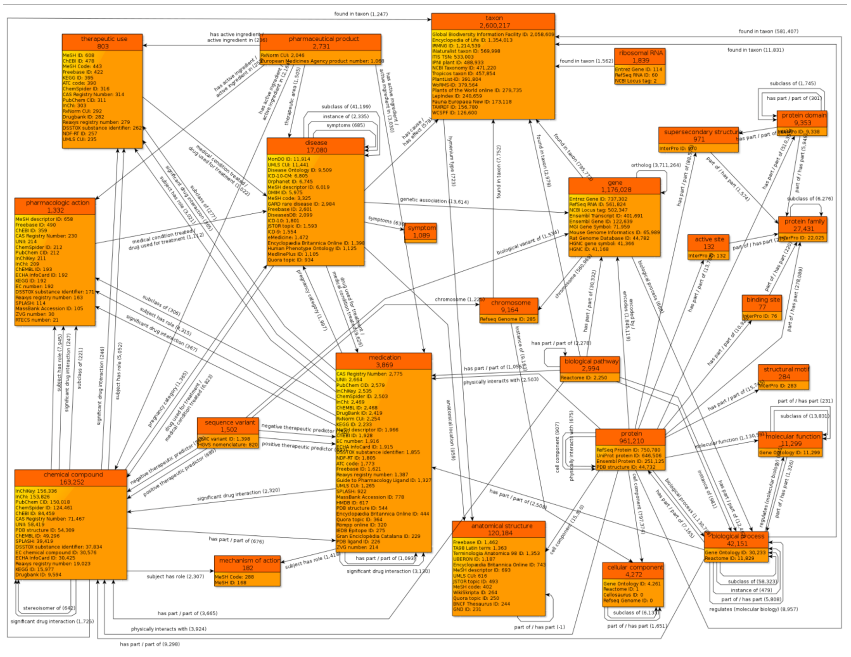


Figure 6: Each box represents one type of biomedical entity. The header displays the name of that entity type, as well as the count of Wikidata items of that type. The lower portion of each box displays a partial listing of attributes about each entity type, together with the count of the number of items with that attribute. Edges between boxes represent the number of Wikidata statements corresponding to each combination of subject type, predicate, and object type. Data are generated using the code by Andrew Su [5].

This type of visualization is not scalable to work for very large RDF sources.

3 Incremental RDF Schema Visualization

As mentioned previously, to our knowledge no scientific papers have tried to visualize a schema for big RDF data sets. This research aims to provide a visualization of the schema created by the backend. The following section describes how the data arrives in the frontend and the way the data needs to be transformed to build a schema graph incrementally using the vis.js library.

3.1 Data transformation

The backend will send batches of nodes to the frontend. In this project, the communication between these two algorithms is not realized as it is outside of the scope of the thesis. Therefore, we will mimic the stream of data to be coming in batches. First, let us consider the data structure in which the frontend receives a batch of nodes. Each sub-graph is stored as a list of objects of type `SchemaNode`. We will commence by describing the structure and meaning of each attribute in the `SchemaNode` class.

- `node.id(String)`, stores the unique node ID
- `node.label(String)`, stores the label. The label represents the name of the schema node class.
- `node.class_type(String)`, denotes whether this schema node class is "implicit" or "explicit". Explicit classes are always inferred from `typeOf` relations, whereas implicit classes are instances matched on similar property sets.
- `node.class_overlaps([String])`, an array of node IDs to indicate which classes this class overlaps with. That is, classes that share instances.
- `node.properties([String])`, an array of properties to denote the set of literals this class is associated with.
- `node.out_edge([SchemaEdge])`, an array of outgoing edges, denoted as `SchemaEdge` type objects.
- `node.in_edge([SchemaEdge])`, an array of incoming edges, denoted as `SchemaEdge` type objects.

The edges are stored as attributes of a node. An edge is of type `SchemaEdge` conisting of three attributes:

- `edge.id(String)`, denoting the id of the edge. For an edge going from `fromNode` to `toNode`, the ID will be "`fromNodeXtoNode`"
- `edge.label(String)`, storing the edge label. The edge label is the predicate corresponding to the edge. Say we have a class `Author` and a class `Book`, the label of the connecting edge is `Writes`.
- `edge.to_node(String)`, denoting the node ID of the node that shares an edge with.

As mentioned above, the frontend will be assumed to receive batches of nodes in this format. We will have to mimic this by cutting up the entire JSON file into k batches. For each batch, the data is transformed from a list of `SchemaNodes` to lists of nodes and edges. We have to take into account that we only want to add edges that are connected to an already existing node. This is because a portion of the edges stored in a `SchemaNode` point to literals, and literals are

considered to be attributes of a schema node and should therefore not appear as nodes in our graph visualization. Therefore, when reading the incoming and outgoing edges for a node, we should check if the from-nodes and to-nodes have already been added to the list of nodes.

This means that we cannot simply cut up the JSON into k batches and create k independent sub-graphs to later merge them, as we would have skipped over edges that share connections with nodes from other batches. We need a *complete* list of nodes first, before we can add edges. Algorithm 1 below shows how we transform the JSON file into a list of nodes and edges.

Algorithm 1 Read in the data from JSON and rewrite to $G(V, E)$

```
Input: nodeJSON([SchemaNode])
Output: nodes, edges
nodeIDlist ←∅
nodelist, edgelist ←∅
for node ∈ nodeJSON do
    add node to nodelist
    add node.id to nodeIDlist
for node ∈ nodelist do
    for edge ∈ outgoing edges do
        if toNode ∈ nodelist then
            add edge to edgelist
    for edge ∈ incoming edges do
        if fromNode ∈ nodelist then
            add edge to edgelist
```

Although merging independent sub-graphs is not an option, this luckily does not mean that we cannot add nodes in batches. A new set of incoming nodes can be added like any node that was in the first batch. Once we have a list of nodes and a list of edges, all that remains is to create a graph capable of continuously expanding. For this thesis we use a graph visualization library called vis.js. In order to create a visualization, we create batches of size k and add k nodes and edges per iteration. A timeout function is used to prevent the server from crashing when adding large batches. In practice we have noticed that the performance is best with a trivial batch size of $k = 1$. When choosing a larger k, say $k = 20$ and a timeout of 2 seconds, the graph renders nicely in the beginning but eventually takes more time per batch as the size of the graph increases, eventually crashing because the timeout duration is not longer sufficient.

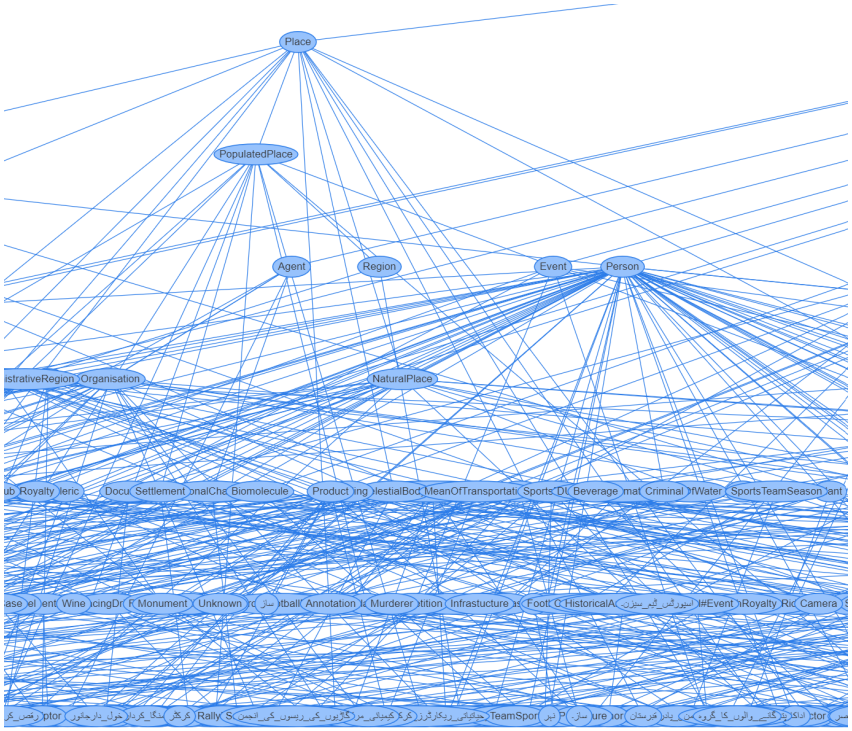
Vis.js gives four options for solvers in the physics menu: `barnesHut, repulsion, hierarchicalRepulsion` and `forceAtlas2Based`. All of these options are non-deterministic graph drawing algorithms, based on attraction and repulsion of pairs of nodes. Out of these options, with a running time of $\mathcal{O}(N \log N)$, `barnesHut` is computationally least expensive. In these non-deterministic algorithms, in each iteration nodes and edges are incrementally moving towards an equilibrium, much like in gradient descent. The graph is *done* when a local minimum is reached, e.g. no small movements will further improve the objective. The method generally does not find the global minimum, but the resulting graph is easy on the eye nevertheless. Force-based algorithms tend to create organic looking structures when increasing the amount of nodes, generally outperforming deterministic methods.

3.2 Hierarchical graph rendering

vis.js gives the option to create hierarchical graphs in a tree structure. To illustrate the shortcomings of this method for our purpose, we will consider a small subset of the DBpedia RDF database. A hierarchical subset of the RDF triples can be obtained by querying for all triples with a `rdfs:subClassOf` predicate. We obtain all classes that are sub-classes of other classes in the ontology. Figure 7 depicts a hierarchical representation.

Figure 7: A strictly hierarchical representation of classes in DBpedia. The leafs are extremely dense and overlapping, making them impossible to read. The hierarchical structure within the tree gives a good representation of the ontology structure

Rendering this graph takes up to an hour, and the results are mediocre at best. Even upon zooming, a spaghetti of edges makes it difficult to interpret the structure. The visualization could be cleaned up significantly by removing all redundant edges by using transitivity laws. For example, if `TennisPlayer` is a subclass of `Athlete`, and `Athlete` is a subclass of `Person`, there is no need for to draw an edge from `TennisPlayer` to `Person`, as this could be deduced. However, the problem of unreadable leafs remains to be an issue.



The same graph can be drawn non-hierarchical or not-strictly hierarchical. Both layouts are cleaner than the one above, but it is difficult to see the overall structure. The force-directed method is considerably faster than the other two, more stable and therefore more satisfying to watch being built.

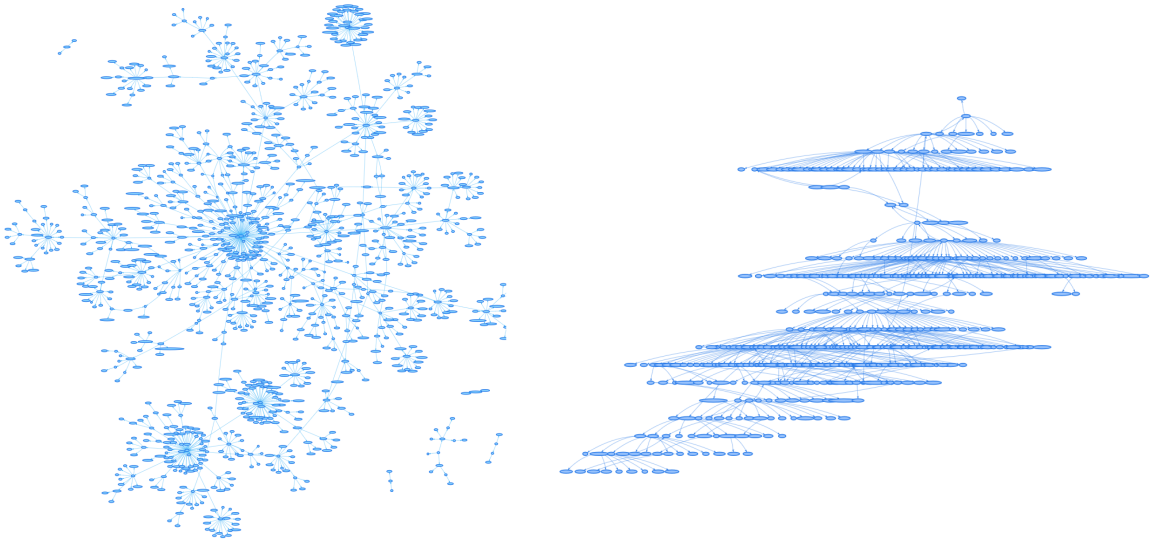


Figure 8: A global view of two alternative visualizations of the same DBpedia class hierarchy. The graph on the left is built with a force-based method, *forceAtlas2Based*. The *clusters* of related ontology classes are clearly visible, but the hierarchy is lost. The graph on the right is a non-strictly hierarchical view, taking up less space horizontally than the strictly hierarchical graph. This makes it easier to read but also makes it harder to interpret the hierarchy.

Let us consider taking an hierarchical graph as a base, and adding on the rest of the triples in the DBpedia database as *add-ons*. To clarify: imagine the class hierarchy being a Christmas tree, then the non-hierarchical clusters of classes are added to branches as decoration, possibly in a different color. The idea is appealing: we would see the class hierarchy which makes it easier to locate, pan and zoom to our preferred branch. However, there are several downsides to this plan. One being that vis.js does not give the option to make partially hierarchical structures. Even if we would find a library that would support this feature, or if we build it ourselves, the spaghetti of overlapping nodes and edges would get progressively worse. We would need to create more space for the nodes, however, the parameters of the hierarchical method would be impossible to tune due to inpredictability of incoming nodes.

3.3 Force-directed graph visualizations

In the following section, we will be using the schemas created by the backend. The pipeline is described in detail in Koray's thesis. We used DBpedia and WikiData RDF data. The source codes of our implementations are available online[6].

In the scope of this thesis, we stick to the force-directed visualization methods due to the large amount of information we want to visualize. More specifically, we stick to the `barnesHut` algorithm. To showcase the way this algorithm builds the schema incrementally, three screenshots of the same graph at different moments in time are shown in the image below. The list of nodes we received from the backend is a schematic representation of the DBpedia and WikiData RDF databases. For DBpedia, these 4381 classes cover roughly 3.5 million triples. WikiData is represented by 3447 classes, covering roughly 5.5 million triples. Below, the emergent schemas for DBpedia and WikiData are visualized.

[6] https://git.science.uu.nl/vig/mscprojects/rdf-schema-retrieval

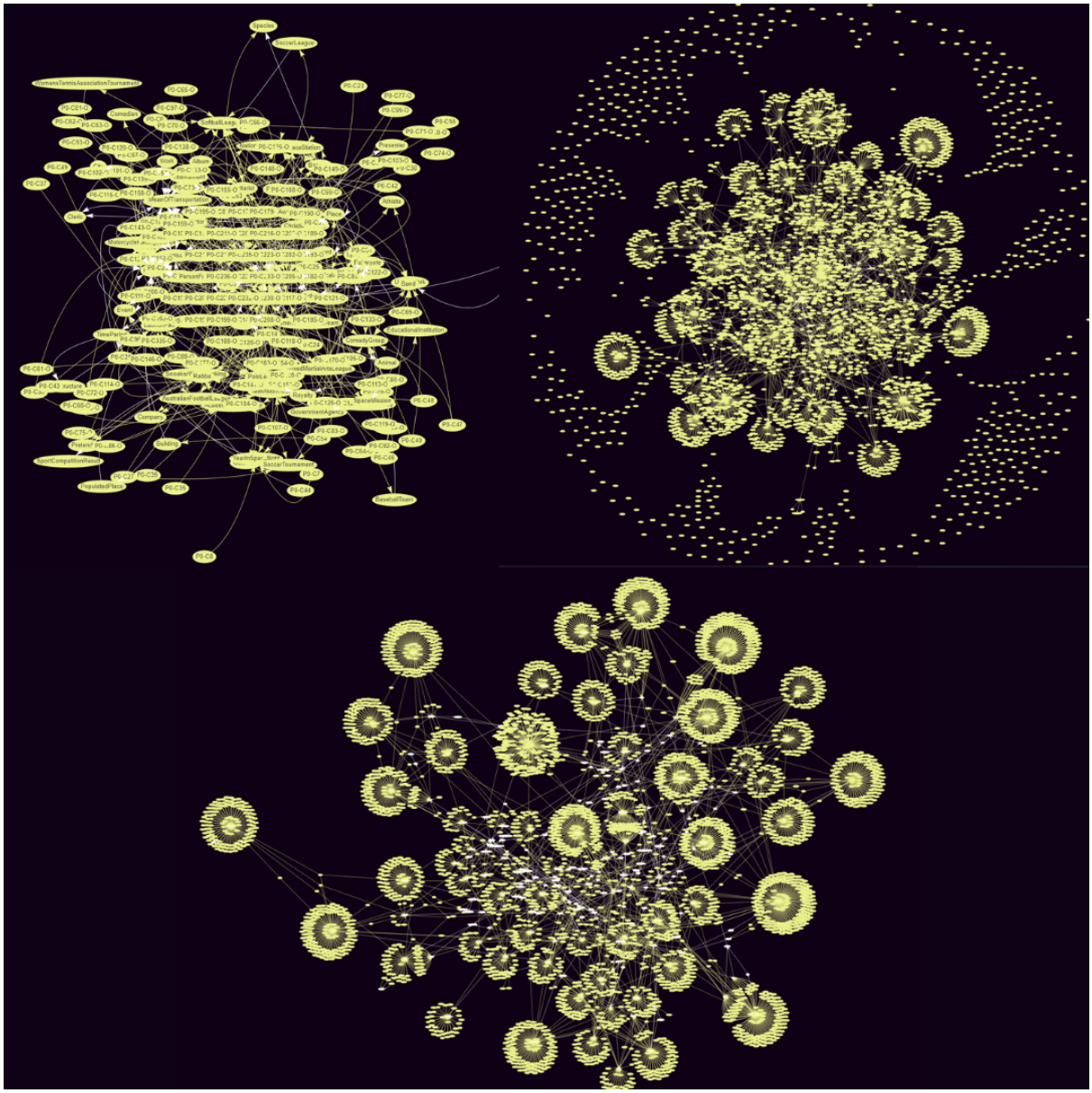


Figure 9: Three different stages in visualizing the DBPedia schema with the *BarnesHut* algorithm. The top left depicts the state after a couple of seconds. The top right is what the graph looks like after about an hour of rendering. The bottom depicts the DBpedia schema after a full night. The *BarnesHut* algorithm continued to adjust the position of the nodes, showing it had not yet reached an equilibrium. This DBpedia schema consists of 213 explicit classes (white) and 4168 implicit classes (yellow).

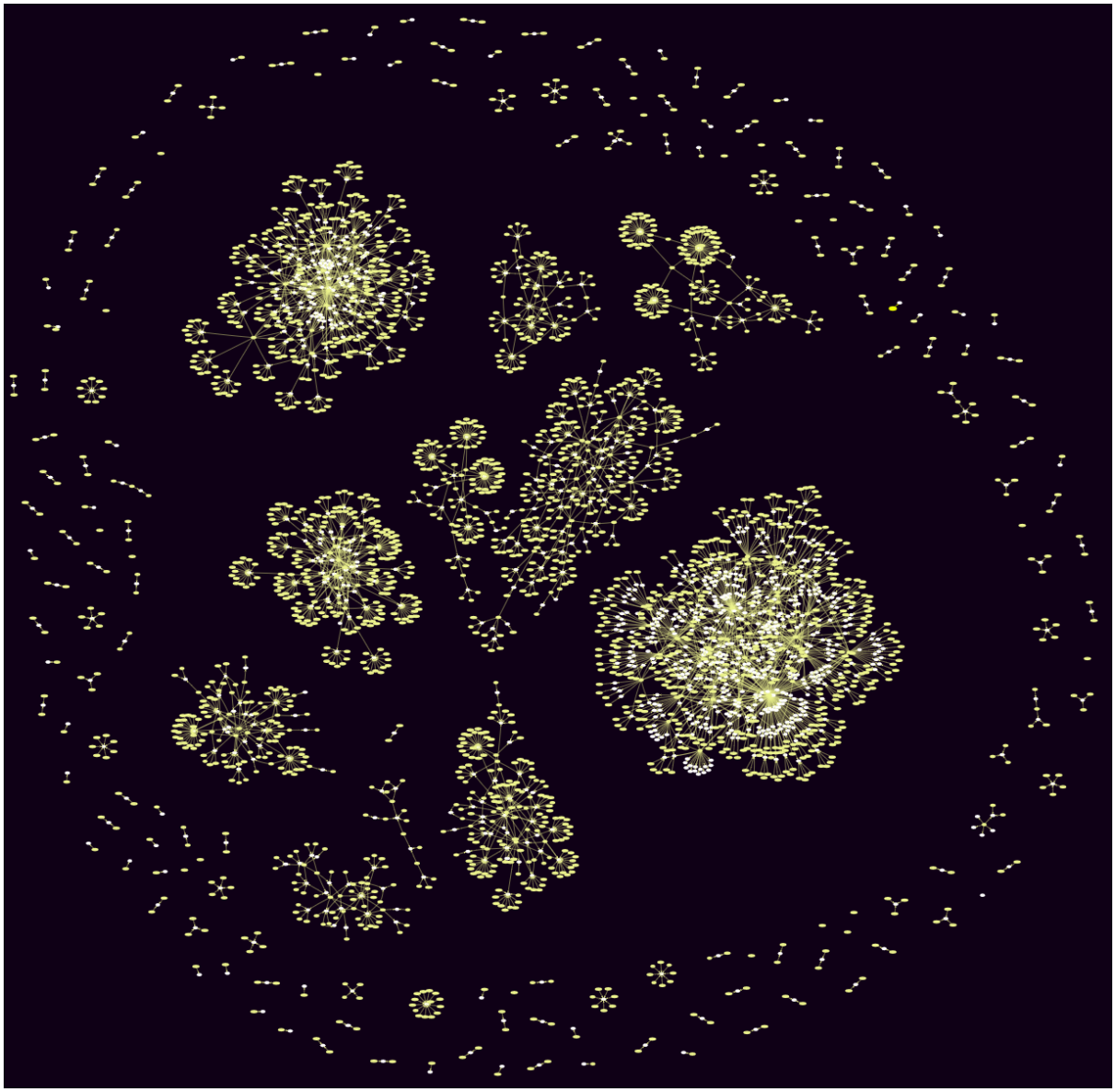


Figure 10: A schematic representation of the WikiData database. The schema was incrementally built using the *BarnesHut* algorithm, and covers roughly 5.5 million triples in roughly 3.5 thousand classes. These are divided into 1161 explicit classes (white) and 2286 implicit classes (yellow).

3.3.1 Interaction

Interaction with the graph is realized by the built-in ability to pan and zoom to navigate the graph. Moreover, if the user feels like the layout can be improved or detangled, it is possible to drag a node across the screen, after which the `barnesHut` algorithm will make its neighbouring nodes follow, improving the layout until it reaches a new local minimum. Further interactive adaptations made by this study are:

- when clicked, a node changes color,
- edges connected to the clicked node change color,
- and a popup opens upon clicking a node, displaying:
  - class name,
  - class properties, e.g. node properties generally associated with nodes adhering to this class,
  - class type, denoting whether the clicked node represents an explicit or implicit class,
  - class overlaps, e.g. classes that share instances with this class,
  - classes to which the class relates via an outgoing edge,
  - classes to which the class relates via an incoming edge.

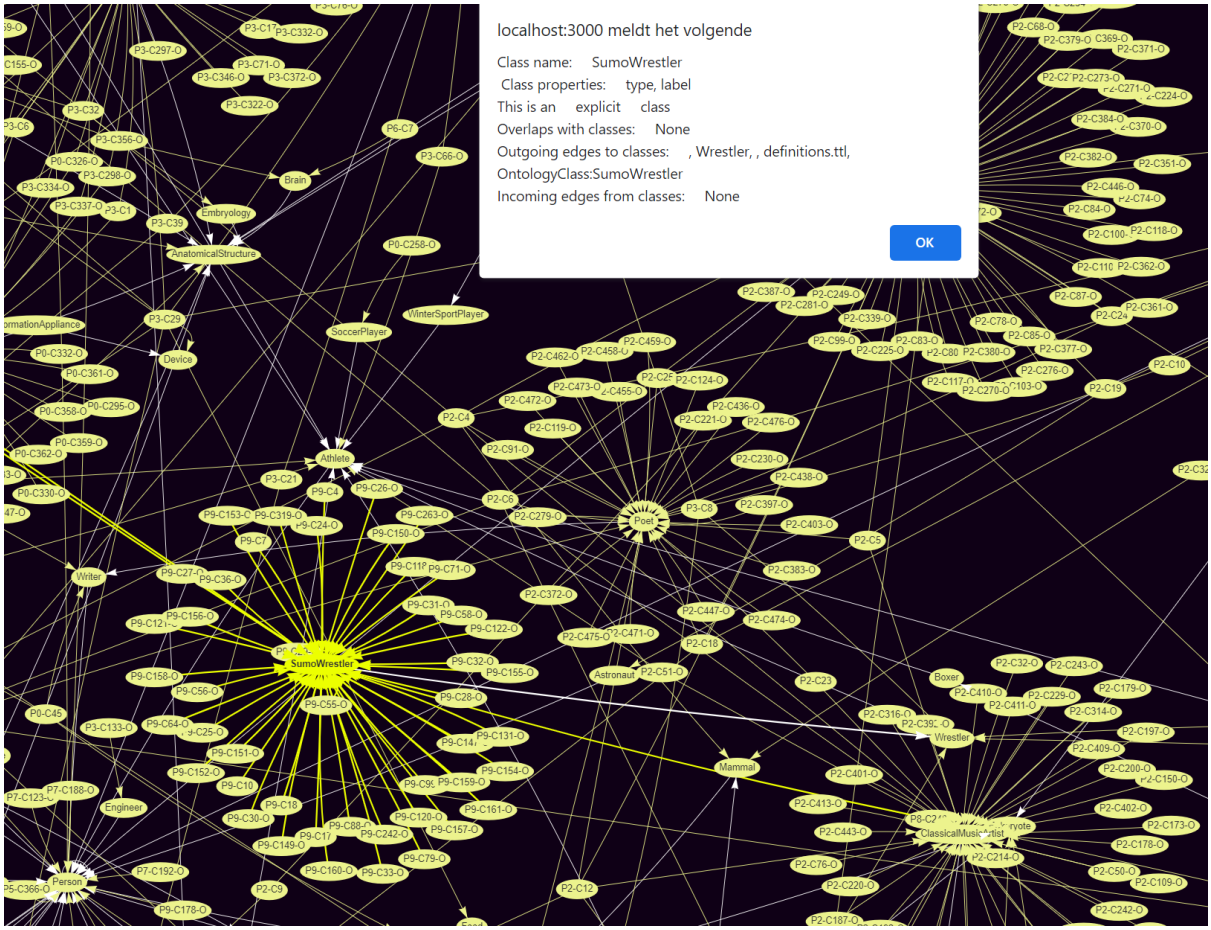


Figure 11: A screenshot of the visualization, zoomed into the *SumoWrestler* class. After clicking on the class node, a popup with additional node information is displayed.

4 Evaluation

The evaluation consists of two parts. The first part shows how this schema visualization can be used to explore and compare RDF data sets. By means of example, we compare the global structure of DBpedia to WikiData. In the section thereafter, limitations with respect to the scalability of the visualization tools are mentioned.

4.1 Datasets

DBpedia extracts data from Wikipedia and builds an RDF graph. This project, created in 2007, aims to extract structured information from Wikipedia and to make it available on the Linked Data Web. The other database we used in this thesis is WikiData. The WikiData project was announced in 2012, and its main goal is providing high-quality structured data acquired and maintained collaboratively to be directly used by Wikipedia to enrich its contents. Although the two projects have somewhat different goals, in an ideal situation, DBpedia and Wikidata should contain equivalent content. In a comperative study between the two datasets [3], the researchers note: "Knowledge bases and ontologies as DBpedia, Freebase, OpenCyc, Wikidata, and YAGO have been experimented in a large number of projects and are now adopted in commercial applications. Therefore, it is important, not only for the academic but also for the industrial community, to have a description of the main features of these knowledge sources. To the best of our knowledge, even if a complete description of these sources is available in many papers, an approach providing a critical comparison is still missing."

Our algorithm is capable of giving the user a clear overview of implicit and explicit classes and how they relate, as well as giving a global structure to navigate and explore. Therefore, our implementation is an approach providing critical visual comparision between RDF sources.

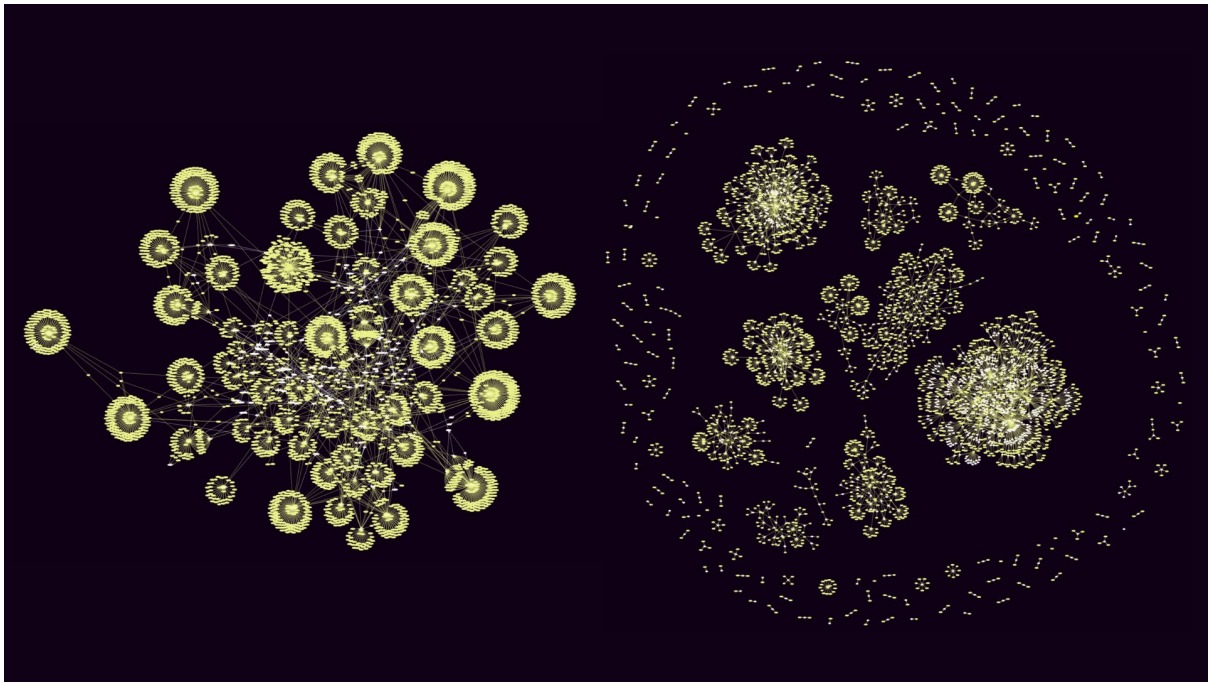


Figure 12: A side by side comparison of the global schemas of DBpedia (left) and WikiData (right). In one glance, the viewer can spot differences in structure. DBpedia is a lot more structured, connected and centers around the explicit classes (white), whereas Wikidata is more messy, disjoint, and explicit classes function as bridges between implicit classes, instead of taking a central position.

4.2 Case Study Findings

The code we use is very slow, but we eventually end up with a aesthetically pleasing, organic looking graphs for DBpedia and WikiData. This is surprising, as the utility of the basic force-directed approach is limited to small graphs and results are known to be poor for graphs with more than a few hundred vertices. There are multiple reasons why traditional force-directed algorithms do not perform well for large graphs. One main obstacle to the scalability of these approaches is the aforementioned fact that the physical model typically has many local minima. Even with the help of sophisticated mechanisms for avoiding local minima the basic force-directed algorithms are not able to consistently produce good layouts for large graphs. However, the DBpedia graph ends up looking clear, albeit after a full night of rendering.

The big schemas in Figure 12 indeed look aesthetically pleasing from afar. Upon zooming in however, it becomes clear how information-packed the schema is. To illustrate, the first 300 classes in the DBpedia schema are visualized in Figure 13 below. The edges create a spaghetti, making it difficult to read the labels.

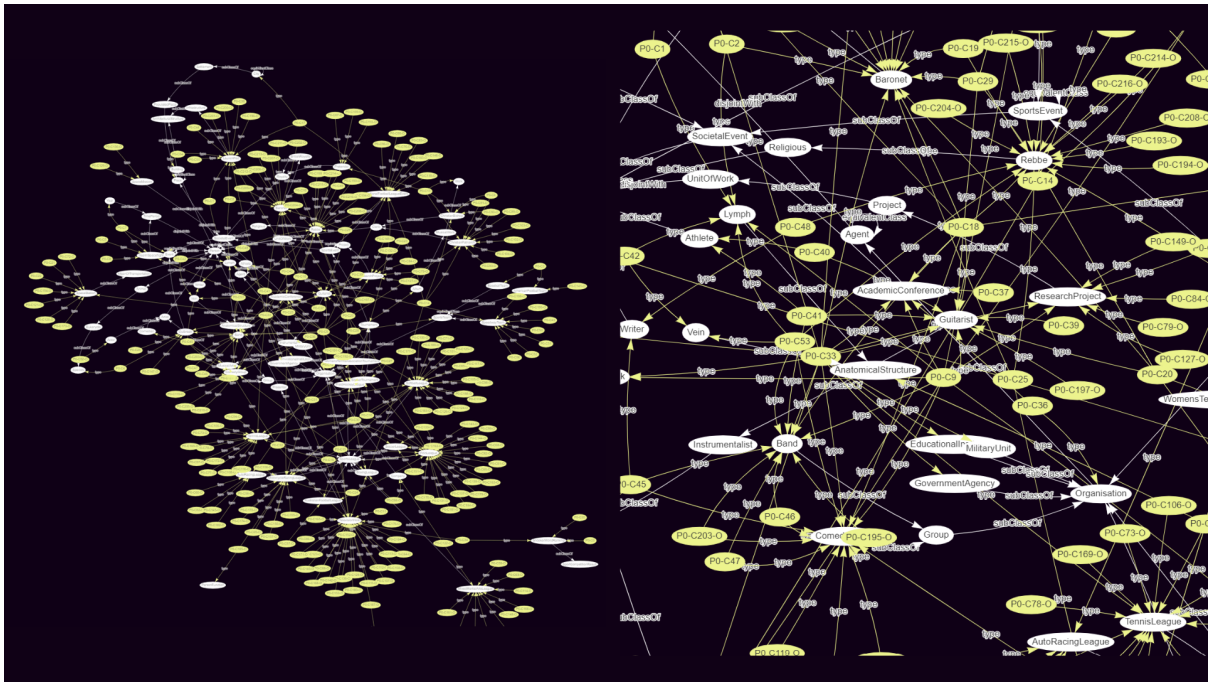


Figure 13: A schema of the first 300 classes in the DBpedia schema. The edge labels are visible, however they are difficult to interpret. The edge labels disappear upon zooming out.

One explanation for the slow rendering is that in normal use, all nodes are randomly placed on the screen with the edges connected, after which the `barnesHut` algorithm runs about a thousand iterations to get everything into place at once. In our adaption however, nodes are added one by one. It is nontrivial to extrapolate how vis.js is updating the graph and the visualization, as little documentation can be found on the performance of adding new nodes to an existing graph rendered with the a force-based algorithm. Further research can be done on adapting the `barnesHut` algorithm to handle incoming nodes and simultaneously cutting down on the computation time for the force-based updating.

The visualization can be used to further improve the algorithm that creates the semantic schema. Notice how in Figure 11, the class `SumoWrestler` has a lot of incoming edges. We are interested in seeing *classes*, and not in *instances* of specific classes. If a sub-graph is shaped like a dandelion, that is, a center with many incoming edges from nodes of degree 1, it is likely that at least some of the leaves could be merged. In future research, one could ask how many different classes *should* point to `SumoWrestler` and change hyperparameters accordingly. The downside of this approach is that building a schema in the backend takes about an hour. However, if successful, less nodes are passed on to the frontend which would speed up the visualization significantly.

Instead of tuning hyperparameters in the backend, further exploration can be done with frontend clustering algorithms such as `k-means Clutering`. Another approach is to hard-code an anti-dandelion effect by limiting how many classes of degree 1 are allowed to point to one given class. If that number exceeds a certain threshold, the leafs can be folded in. The downside of this approach is that it requires even more computation power in the frontend.

5 Conclusion

In this thesis project we introduce a novel approach to incrementally build and visualize a global schema for big RDF data sources. This facilitates the understanding of the information contained in RDF data. This thesis in particular incrementally builds an interactive schema graph using the vis.js library. The visualization method uses a scalable force-directed layout to visualize a large schema, yielding an aesthetically pleasing graph. The tool can be used to discover big RDF data sources such as DBpedia and WikiData by panning, zooming, and clicking on classes to obtain insight about how the data is interlinked. Moreover, the visualization tool can be helpful for backend developers to test and improve the backend pipeline used to emerge semantic schemas for linked data.

As future work, we suggest to look for methods to improve the scalability of the visualization method. Adding nodes in a stream is necessary when handling large graphs in a frontend environment. Methods to build huge streaming graphs in a web environment can be applied to this visualization problem.

Moreover, giving the user the ability to change node and edge labels might help to improve the semantic quality of the schema. Using domain knowledge from graphic design and information studies might further improve the readability of the schema. Incorporating adaptations of the user to the schema structure might also be beneficial for the structure of the RDF data itself. The schema can be used to see *how* new triples should be added, essentially using the schema as a blueprint, where the blue lines can be added or erased over time. This will be beneficial and potentially critical to use RDF sources in the future.

References

[1] Shapes constraint language (shacl), July 2017. https://www.w3.org/TR/shacl.

[2] Big data - statistics facts, June 2022. https://www.statista.com/topics/1464/big-data.

[3] D. Abián, F. Guerra, J. Martínez-Romanos, and Raquel Trillo-Lado. Wikidata and dbpedia: A comparative study. In Julian Szymański and Yannis Velegrakis, editors, *Semantic Keyword-Based Search on Structured Data Sources*, pages 142–154, Cham, 2018. Springer International Publishing.

[4] Florian Bauer and Martin Kaltenböck. *Linked Open Data: The Essentials: A quick start guide for decision makers*. January 2012.

[5] Bradley R Bebee, Daniel Choi, Ankit Gupta, Andi Gutmans, Ankesh Khandelwal, Yigit Kiran, Sainath Mallidi, Bruce McGaughy, Mike Personick, Karthik Rajan, et al. Amazon neptune: Graph data management in the cloud. In *ISWC (P&D/Industry/BlueSky)*, 2018.

[6] Tim Berners-Lee. The next web. https://www.ted.com/talks/tim_berners_lee_the_next_web.

[7] Ulrik Brandes, Marco Gaertler, and Dorothea Wagner. Experiments on graph clustering algorithms. In *European symposium on algorithms*, pages 568–579. Springer, 2003.

[8] Andrea Carmè, Jose-Norberto Mazón, and Stefano Rizzi. A model-driven heuristic approach for detecting multidimensional facts in relational data sources. volume 6263, pages 13–24, September 2010.

[9] Stephan Diehl and Carsten Görg. Graphs, they are changing. In *International Symposium on Graph Drawing*, pages 23–31. Springer, 2002.

[10] Ivan Herman, Guy Melançon, and M Scott Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on visualization and computer graphics*, 6(1):24–43, 2000.

[11] Kenza Kellou-Menouer and Zoubida Kedad. Schema discovery in rdf data sources. In *International Conference on Conceptual Modeling*, pages 481–495. Springer, 2015.

[12] Masood Masoodian and Saturnino Luz. Map-based interfaces and interactions. In *Proceedings of the 2022 International Conference on Advanced Visual Interfaces*, pages 1–4, 2022.

[13] Minh-Duc Pham, Linnea Passing, Orri Erling, and Peter Boncz. Deriving an emergent relational schema from rdf data. In *Proceedings of the 24th International Conference on World Wide Web*, pages 864–874, 2015.

[14] Jim Procter, Julie Thompson, Ivica Letunic, Chris Creevey, Fabrice Jossinet, and Geoffrey Barton. Visualization of multiple alignments, phylogenies and gene family evolution. *Nature methods*, 7:S16–25, March 2010.

[15] Helen Purchase. Which aesthetic has the greatest effect on human understanding? In *International Symposium on Graph Drawing*, pages 248–261. Springer, 1997.

[16] Amit Singhal. Introducing the knowledge graph: things, not strings, May 2012.