

# Zoo aggregation using semantic data: The advantages of Graph Databases

Safi Dewshi, 1559816

December 11, 2019

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Advantages of Graph Databases</b>	<b>3</b>
<b>3</b>	<b>Aim</b>	<b>3</b>
<b>4</b>	<b>Design</b>	<b>3</b>
4.1	Semantic Data Technologies . . . . .	3
4.2	Ontology for cervo.io . . . . .	4
4.3	Overview of Ontology Schema . . . . .	5
<b>5</b>	<b>Evaluation and Use</b>	<b>8</b>
<b>6</b>	<b>Implementation</b>	<b>15</b>
<b>7</b>	<b>Critical Reflection</b>	<b>16</b>
<b>8</b>	<b>Conclusion</b>	<b>16</b>
	<b>Bibliography</b>	<b>17</b>
	<b>Appendix</b>	<b>18</b>
	Appendix A . . . . .	18
	Appendix B . . . . .	23

# 1 Introduction

This report will present and discuss the advantages and disadvantages of transferring a traditional relational database into semantic graph database. Semantics is the study of meaning in language, and those codifying those meanings in a machine-readable format has several benefits for the field of computing. For example, it allows programs to effectively "understand" what data means (Szeredi et al. 2014).

;;explanation/outline of what will be explained in each section;;

# 2 Advantages of Graph Databases

Traditional relational databases, typically queried via SQL, store data as tables. These tables are rigidly define, with columns and specified data types for each piece of data. Because of this strict table layout, data will often have to be split amongst multiple tables linked through foreign keys, which then need to be accessed through multiple JOIN queries which have significant performance impact as well as becoming significantly harder to read.

In contrast, graph databases, which we will query with SPARQL (Short for SPARQL Protocol and RDF Query Language <sup>1</sup>) is used for queries graph data.

Storing data in a graph has multiple advantages for storing data with multiple relations. As an example, we can consider books and their authors. Since one author can write multiple books and one book can have multiple authors, we would need multiple tables. One table to describe the authors, one to describe the books, and one to link the first two. Then to find the what books a certain author has written we would need to query the first table to find that author, join it to the third table to get a list of IDs and then join those to the second one to find the titles of the books. It is easy to see how this type of query can get very complicated and computationally expensive. With a graph database, we can simply look for the relation between author and book.

# 3 Aim

The intent of the project was to upgrade a SQL database into a SPARQL one that takes advantage of the interconnected nature of semantic data to allow more complex questions to be asked with simpler queries and to connect to external services to easily retrieve additional or updated data. As well as this, the SPARQL database is significantly more flexible and allows data on species and zoos to be in a much less rigid way.

The application provides an easy way for users to search for species and zoos and see additional information (both local and remote) about those things.

It is planned for this SPARQL to completely replace the current SQL database that hosts the site

# 4 Design

## 4.1 Semantic Data Technologies

As we discussed earlier, semantic data revolves around storing data in graphs. Data is stored as nodes with attributes and relationships between them.

SPARQL, as the expansion of the acronym suggests, is a language for querying RDF<sup>2</sup> data. Although that data can be serialized in multiple forms, all the data takes the form of resources, properties, and statements that use properties to describe resources.

These statements are also known as "triples" and take the form **subject-predicate-object**, that is to say Resource A has Property B to Resource C<sup>3</sup>. As an example, we could store the author of a book by saying **Book hasAuthor Author**.

We can make a series of statements about our resources and build up our database this way.

In RDF, resources are identified using URIs<sup>4</sup> or IRIs<sup>5</sup> which are the internationalized versions of

<sup>1</sup>where SPARQL is short for SPARQL Protocol and RDF Query Language, which is short for SPARQL Protocol and RDF Query Language etc. etc.

<sup>2</sup>Resource Description Framework

<sup>3</sup>Resource C could also be a simple literal

<sup>4</sup>Uniform Resource Identifiers

<sup>5</sup>Internationalized Resource Identifiers

URIs. URLs, which most people will be more familiar with, are a subset of URI/IRIs. An example of an IRI would be `http://www.cervo.io/ontology#deer` (Smith, Welty, and McGuinness 2004).

The semantic web is the application of these technologies to the world wide web. Figure 1 illustrates how those layers work on top of each other. The key layers for us are:

1. URI/IRI, which are used for referencing and identifying resources
2. RDF/XML, which is a format used to store and represent information
3. SPARQL/OWL, which are used to organise the data and express it in a way that a computer can take advantage of the knowledge
4. Finally a set of layers for delivering that information to the user

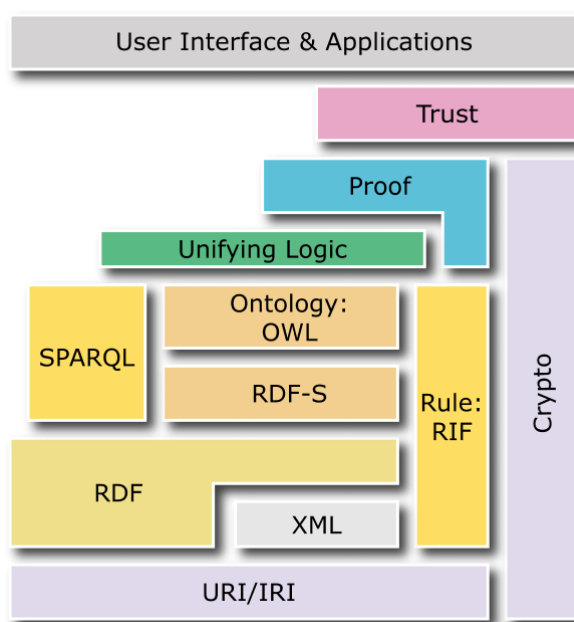


Figure 1: Semantic web stack(W3C 2007)

SPARQL has four basic query types:

1. SELECT, which is used to extract data from the database in the form of a table
2. CONSTRUCT, which is similar to SELECT but returns the data as an RDF graph.
3. ASK, which gives a true/false result for a query
4. DESCRIBE, which returns a RDF graph that 'describes' a resource.

We will primarily be using SELECT as that returns data in the most convenient way for our purposes.

## 4.2 Ontology for cervo.io

Currently the website runs on a MySQL database, with several different tables containing the information needed to construct a taxonomic tree, populate it with species, and link those species to the zoos that keep them as shown in Figure 2.

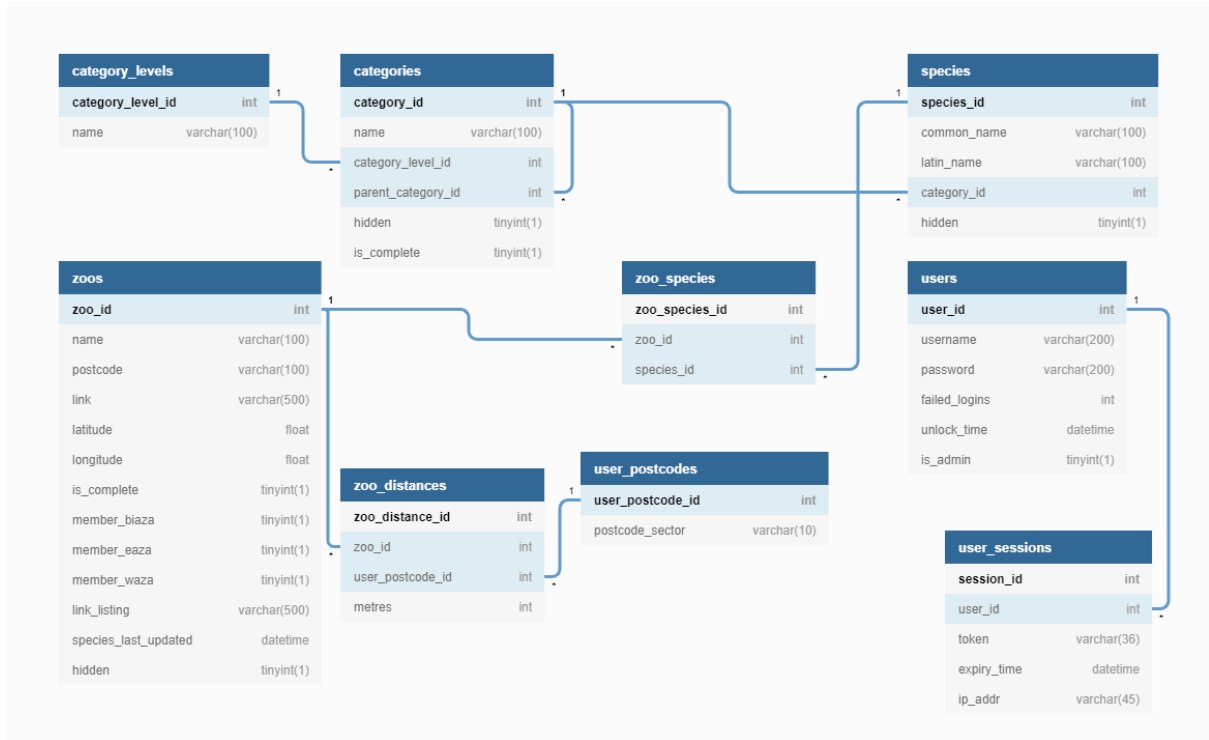


Figure 2: Current database schema diagram

However this SQL solution had several problems that were easily remedied by switching to a SPARQL database:

1. It was inflexible and would require the schema to be rewritten to accommodate new types of data
2. Certain queries, such as retrieving all animals under "Mammalia" require complicated and expensive SQL queries
3. Supplementary data would require manual data entry and would be susceptible to becoming outdated

Several other queries became significantly easier to write by taking advantage of the triples. Additionally the use of common keys such as postcode and species names makes the incorporation of external data sets significantly easier even within a single query

⌋⌋EXPLAIN SPARQL⌋⌋

### 4.3 Overview of Ontology Schema

The ontology for the database was created using Protégé, a free open-source ontology editor developed at the Stanford Center for Biomedical Informatics Research (Musen 2015). Protégé implements a GUI to design, visualise, and ensure the internal consistency of an ontology. Once completed the ontology can be imported into a database such as Apache Jena Fuseki

Figure 3 shows the metrics of the completed ontology from Protégé. It has over 2400 individuals, which were imported from the SQL database and formatted using Python, split amongst 35 classes. The class hierarchy is shown in Figure 4. The "Location" class represents, as the name suggests, locations where animals are kept such as zoos or wildlife refuges and the "Species" class represents the animals that can be kept in those locations. These two classes represent most of the individuals stored in the database. The "phylogeneticTree" class and its subclasses represent a way of categorising species based off their evolutionary relationships (Rintoul and Bear 2016). Whilst this is by necessity somewhat imprecise<sup>6</sup>, it nevertheless provides a convenient way to group species. In our case it means we can, select all mammals by their relation back to the class "Mammalia", all big cats by their relation to the genus "Panthera", or all cats by the family "Felidae". The "collectiveTerm" subclass is an extension of this that both allows us

<sup>6</sup>nature does not like fitting into our neat little boxes

to more easily select one of those groups or to make more arbitrary groupings such as "Fluffy animals". Whilst this database only contains a subset of the kingdom "Animalia", the framework is there for it to be expanded other kingdoms or even to all living organisms<sup>7</sup>.

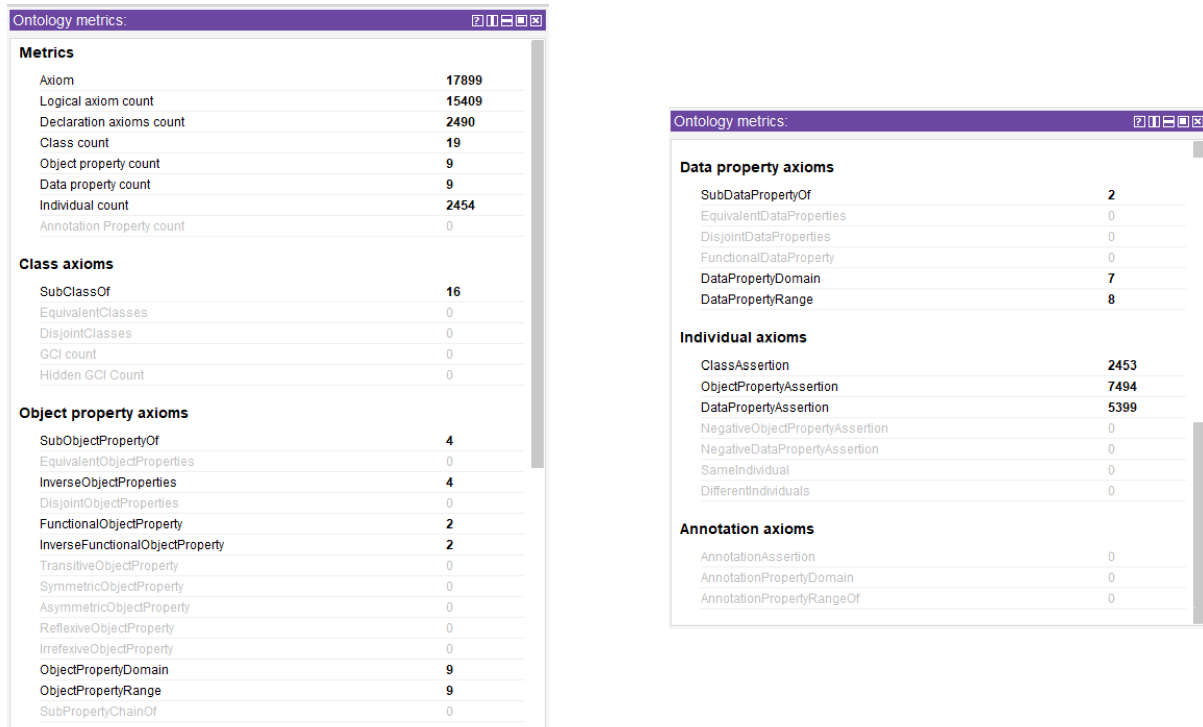


Figure 3: Ontology Metrics from Protégé

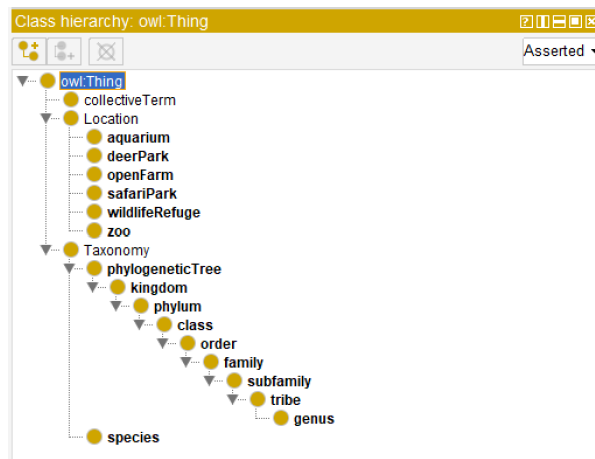


Figure 4: Class Hierarchy

Figure 5 shows a visual overview of the classes and their object property relations (Which are expanded in Figure 6). Species **are-kept-at** locations and inversely a location **keeps-animals** species. Animals **are-known-as** collective terms and the inverse; collective terms **can-refer-to** animals. The taxonomy for the phylogenetic tree is grouped under **taxonomic-relation**, where **is-species-in** and **contains-species** relate species to their genus and **contains-clade** and **is-clade-in** relate sub-classes under the phylogenetic tree (kingdom to phylum to order and so on). **is-species-in** and **is-clade-in** are functional properties to enforce the tree structure.

Individuals also have data properties associated with them, and these are listed in Figure 7. All individuals have names, **hasName** which is broadly interchangeable with **rdfs:label** or **foaf:name**. Species

<sup>7</sup>Although how one would go to a zoo to see a bacteria or other single-celled organism is a separate matter



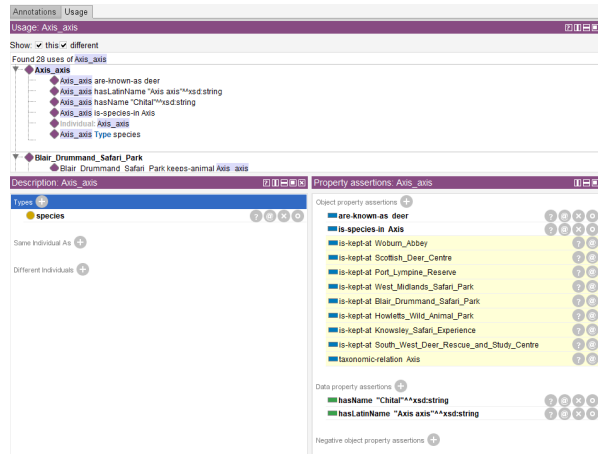


Figure 8: An example species individual

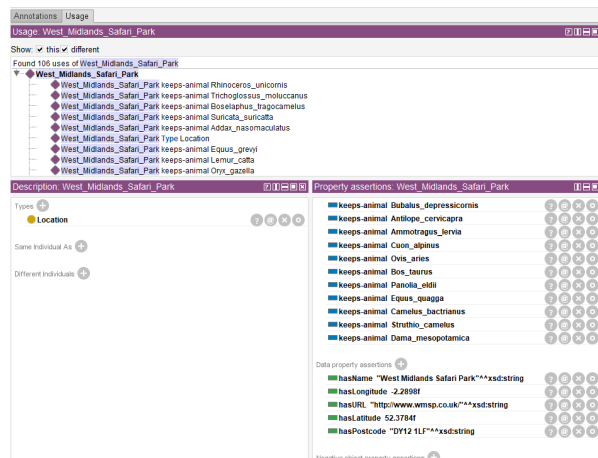


Figure 9: An example zoo individual

## 5 Evaluation and Use

This section will show and explain some of the SPARQL queries used.



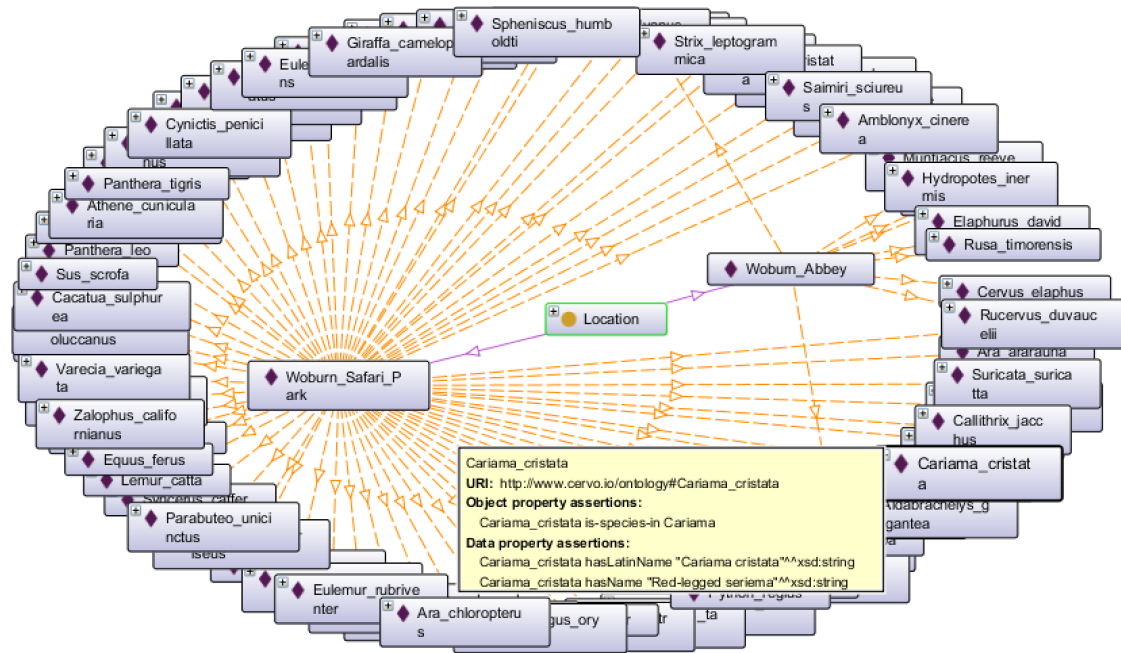


Figure 10: Ontograph showing some individuals

```
PREFIX cervo: <http://www.cervo.io/ontology#>
```

```
SELECT ?animal ?name ?latinname
```

```
WHERE
```

```
{{
```

```
    ?animal a cervo:species;
```

```
    cervo:hasName ?name;
```


```
    cervo:hasLatinName ?latinname
```

```
}}
```

QUERY RESULTS

Table

Raw Response



Showing 1 to 50 of 1,122 entries

Search:

Show 50 entries

	animal	name	latinname
1	<a href="#">cervo:Axis_axis</a>	"Chital"	"Axis axis"
2	<a href="#">cervo:Dama_dama</a>	"Fallow deer"	"Dama dama"
3	<a href="#">cervo:Cervus_elaphus</a>	"Red deer"	"Cervus elaphus"
4	<a href="#">cervo:Rangifer_tarandus</a>	"Reindeer"	"Rangifer tarandus"
5	<a href="#">cervo:Dama_mesopotamica</a>	"Persian fallow deer"	"Dama mesopotamica"
6	<a href="#">cervo:Cervus_albirostris</a>	"White-lipped deer"	"Cervus albirostris"
7	<a href="#">cervo:Cervus_canadensis</a>	"Wapiti"	"Cervus canadensis"
8	<a href="#">cervo:Cervus_nippon</a>	"Sika deer"	"Cervus nippon"
9	<a href="#">cervo:Elaphurus_davidianus</a>	"Milu"	"Elaphurus davidianus"
10	<a href="#">cervo:Hyelaphus_calamianensis</a>	"Calamian deer"	"Hyelaphus calamianensis"

Figure 11: A SPARQL query retrieving all animals stored in the database

Figure 11 shows a basic query, simply finding all `?animal` which are under the subclass `cervo:animal`, and then retrieving their IRI, Name, and Latin name

```
PREFIX cervo: <http://www.cervo.io/ontology#>

SELECT ?name ?latinname
WHERE
{
  {
    ?animal a cervo:species;
            cervo:hasName ?name;
            cervo:hasLatinName ?latinname.
    FILTER(NOT EXISTS{
      SELECT ?animal
      {?animal ^cervo:keeps-animal|cervo:is-kept-at ?zoo}})
  }
}
```

QUERY RESULTS

Table Raw Response

Showing 1 to 34 of 34 entries Search:  Show 50 entries

	name	latinname
1	"Calamian deer"	"Hyelaphus calamianensis"
2	"Bawean deer"	"Hyelaphus kuhlii"
3	"Siberian roe deer"	"Capreolus pygargus"
4	"Marsh deer"	"Blasteocerus dichotomos"
5	"Taruca"	"Hippocamelus antisensis"
6	"Huemul"	"Hippocamelus bisulcus"
7	"American red brocket"	"Mazama americana"
8	"Bororo"	"Mazama bororo"
9	"Merida brocket"	"Mazama bricenii"
10	"Dwarf brocket"	"Mazama chunyi"

Figure 12: All animals with no location object property

Figure 12 extends this and shows all animals not found at a location by using the `NOT EXISTS` field to filter out any animals with the `cervo:is-kept-at` or the inverse of `cervo:keeps-animal` relation.

```

PREFIX cervo: <http://www.cervo.io/ontology#>

SELECT ?name ?colterm ?zooname
WHERE
{{
    ?animal a cervo:species;
            ^cervo:keeps-animal ?location;
            cervo:hasName ?name.
    ?location cervo:hasName ?zooname.
    OPTIONAL{?animal cervo:are-known-as|^cervo:can-refer-to ?colterm}
}}ORDER BY DESC (?colterm)

```

QUERY RESULTS

Table Raw Response

Showing 1 to 50 of 5,154 entries

Search:  Show 50 entries

	name	colterm	zooname
1	"Moose"	cervo:deer	"Highland Wildlife Park Kincaig"
2	"Moose"	cervo:deer	"Knowsley Safari Experience"
3	"Moose"	cervo:deer	"Scottish Deer Centre"
4	"Moose"	cervo:deer	"Wildwood Trust"
5	"Chital"	cervo:deer	"Blair Drummand Safari Park"
6	"Chital"	cervo:deer	"Howletts Wild Animal Park"
7	"Chital"	cervo:deer	"Knowsley Safari Experience"
8	"Chital"	cervo:deer	"Port Lympine Reserve"
9	"Chital"	cervo:deer	"Scottish Deer Centre"
10	"Chital"	cervo:deer	"South West Deer Rescue and Study Centre"

Figure 13: All animals sorted by collective term

Figure 13 finds all animals that are kept in a location, and lists them in order of their collective term. Since not every animal has a collective term, this query is kept in an `OPTIONAL` query, so that animals without an assigned term will still show up.

```

PREFIX cervo: <http://www.cervo.io/ontology#>

SELECT ?genusname ?speciesname
WHERE
{
    ?taxon cervo:hasName "Mammalia";
        ^cervo:is-clade-in* ?y.
    ?y ^cervo:is-species-in ?x;
        cervo:hasName ?genusname.
    ?x cervo:hasName ?speciesname
} } ORDER BY ?genusname

```

QUERY RESULTS

Table Raw Response

Showing 1 to 50 of 426 entries

Search:  Show 50 entries

	genusname	speciesname
1	"Acinonyx"	"Cheetah"
2	"Acomys"	"Asia Minor spiny mouse"
3	"Addax"	"Addax"
4	"Aepyceros"	"Impala"
5	"Ailuropoda"	"Giant panda"
6	"Ailurus"	"Red panda"
7	"Alces"	"Moose"
8	"Allochrocebus"	"Mountain monkey"
9	"Alouatta"	"Black howler"
10	"Alouatta"	"Venezuelan red howler"
11	"Amblonyx"	"Asian small-clawed otter"

Figure 14: A query matching all mammals

In Figure 14, we find the individual with the name "Mammalia", then the asterisk matches any number of `cervo:is-clade-in` until it arrives at one that has the property `cervo:is-species-in`. At that point it finds the name of the genus and the species and sorts the results by the genus name. An `INSERT` clause could then be used to add the collective term of "mammal" for all those species

```
PREFIX cervo: <http://www.cervo.io/ontology#>
```

```
SELECT ?latinname ?name
```

```
WHERE
```

```
{{
```

```
  ?animal a cervo:species;
```

```
          cervo:hasLatinName ?latinname;
```

```
          cervo:hasName ?name.
```

```
  FILTER (REGEX (?name, "Ch", "i"))
```

```
}}
```

QUERY RESULTS

Table Raw Response

Showing 1 to 50 of 72 entries

Search:  Show 50 entries

	latinname	name
1	"Axis axis"	"Chital"
2	"Muntiacus reevesi"	"Chinese muntjac"
3	"Hydropotes inermis"	"Chinese water deer"
4	"Myotis bechsteinii"	"Bechstein's bat"
5	"Netta rufina"	"Red-crested pochard"
6	"Pyrrhonorax pyrrhonorax"	"Red-billed chough"
7	"Kobus megaceros"	"Nile lechwe"
8	"Kobus leche"	"Lechwe"
9	"Acinonyx jubatus"	"Cheetah"
10	"Struthio camelus"	"Common ostrich"

Figure 15: Using regex to filter names that match a string

Figure 15 shows an example of using regular expressions to filter the query. In this case it is matching all animals where the name matches the string "Ch" case insensitively. This is vital for searching the database without an exact knowledge of how the names have been entered into the database.

```

PREFIX cervo: <http://www.cervo.io/ontology#>

SELECT ?name (COUNT(?zoo) AS ?zoos)
WHERE
{
    ?animal a cervo:species;
            cervo:hasName ?name;
            ^cervo:keeps-animal ?zoo.
} } GROUP BY ?name
HAVING (?zoos <=2)
ORDER BY DESC (?zoos)

```

QUERY RESULTS

Table Raw Response

Showing 1 to 50 of 579 entries

Search:  Show 50 entries

	name	zoos
1	"African civet"	"2"^^xsd:integer
2	"African grey hornbill"	"2"^^xsd:integer
3	"African spoonbill"	"2"^^xsd:integer
4	"African wood owl"	"2"^^xsd:integer
5	"Alligator snapping turtle"	"2"^^xsd:integer
6	"American kestrel"	"2"^^xsd:integer
7	"American mink"	"2"^^xsd:integer
8	"Aplomado falcon"	"2"^^xsd:integer
9	"Arctic fox"	"2"^^xsd:integer
10	"Ashy-faced owl"	"2"^^xsd:integer

Figure 16: Finding species kept at  $\leq 2$  zoos

Figure 16 shows the number of zoos an animal is kept at, and then uses the **HAVING** criteria to narrow the results down to species found in  $\leq 2$  zoos.

```

PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
SELECT *
WHERE {
  ?x dbpedia2:taxon "Axis axis"^^rdf:langString.
  ?x dbpedia2:name ?z.
  ?x rdfs:comment ?y.
FILTER (LANG(?y)='en')
  ?x ^foaf:primaryTopic ?q
}

```

Figure 17: Querying DBPedia

Figure 17 is now querying DBPedia’s SPARQL endpoint. It was noted that dbpedia’s `taxon` property matched the `hasLatinName` from our database. This means that we can query this external resource and get additional data from the remote resource. This highlights one of the key advantages of semantic technologies since our application can “understand” the information contained in DBPedia.

```

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX postcode: <http://data.ordnancesurvey.co.uk/ontology/postcode/>

SELECT ?districtname ?countyname ?countryname
WHERE {{
  ?postcodeUri rdfs:label "L34 4AN".
  OPTIONAL{{?postcodeUri postcode:district ?district.
    ?district rdfs:label ?districtname}}.
  OPTIONAL{{?postcodeUri postcode:county ?county.
    ?county rdfs:label ?countyname}}.
  OPTIONAL{{?postcodeUri postcode:country ?country.
    ?country rdfs:label ?countryname}}
}}
```

Figure 18: Querying Ordnance Survey’s postcode database

Similarly to Figure 17, Figure 18 is querying an external resource. This time we are asking Ordnance Survey’s endpoint for more information about the postcodes we have associated with our location individuals. Many `OPTIONAL` operations were used in this query as it is uncertain how much of that information exists for each postcode.

## 6 Implementation

The GUI was built using QT Designer, which produced a `.ui` file which was then referenced in the main python code with functions added to handle button pushes and so on.

SPARQL queries were handled using the python `requests` library to send `GET` request to both the local and remote endpoints. Their responses were converted into json and then parsed through the application and presented to the user in table views.

## 7 Critical Reflection

The graph database provides clear advantages over the previous relational MySQL database. It is significantly more flexible and able to account for an animal having multiple names as well as adding extra information about the animals without needing to significantly rework the underlying data structure. Queries are also significantly more intuitive and self-explanatory compared to an SQL database. Queries such as the one shown in Figure 14 would require multiple queries and recursive joins in the SQL database to travel down the taxonomy but is the matter of a few lines in SPARQL. Similarly, we can ask more complex questions such as "What animals are kept in fewer than 2 zoos?"

Additionally the use of shared relations makes it simple to query external resources and update the database.

There is scope for significantly more complex queries. For example, a user may wish to know which zoos are in a certain range. In this case we can query our database for a list of zoos and then query Ordnance Survey to get the location information to filter that list.

This means that if we want to get all the species that are inside a parent category, we have to first get the sub-categories, and then the sub categories of those and so on until we reach the base level

;;Something about hosting solutions? ;;

Additionally

## 8 Conclusion

This report explains the problems inherent to the existing relational database and outlines the benefits of switching to graph data structures



## References

- Musen, Mark A. (June 2015). “The protégé project”. In: *AI Matters* 1.4, pp. 4–12. DOI: 10.1145/2757001.2757003. URL: <http://protege.stanford.edu/>.
- Rintoul, David and Robert Bear (Sept. 2016). *Taxonomy and phylogeny*. OpenStax CNX. URL: <http://cnx.org/contents/12696f5e-80cb-4399-9449-b753db45280b@8>.
- Smith, Michael K., Chris Welty, and Deborah L. McGuinness (2004). *OWL Web Ontology Language Guide*. URL: <https://www.w3.org/TR/owl-guide/>.
- Szeredi, Péter et al. (2014). *The Semantic Web Explained: The Technology and Mathematics behind Web 3.0*. Cambridge University Press. ISBN: 978-0-521-70036-8.
- W3C (2007). *Semantic Web, and Other Technologies to Watch, slide 24*. URL: [https://www.w3.org/2007/Talks/0130-sb-W3CTechSemWeb/#\(24\)](https://www.w3.org/2007/Talks/0130-sb-W3CTechSemWeb/#(24)).

## Appendix

### Appendix A

---

```

from PyQt5 import uic, QtWidgets
import sys
import requests
from PyQt5.QtWidgets import QAbstractItemView, QTableWidgetItem

Ui_MainWindow, QtBaseClass = uic.loadUiType("cervo.ui")
localurl = "http://localhost:3030/Cervo/query"
dbpedia = "http://dbpedia.org/sparql"
ordnancesurvey = "http://data.ordnancesurvey.co.uk/datasets/os-linked-data/apis/sparql"

class CervoUI(QtWidgets.QMainWindow, Ui_MainWindow):
    def __init__(self):
        QtWidgets.QMainWindow.__init__(self)
        Ui_MainWindow.__init__(self)
        self.setupUi(self)
        self.searchButton.clicked.connect(self.species_search)
        self.refineSearchButton.clicked.connect(self.refine_species_search)
        self.zooSearchButton.clicked.connect(self.zoo_search)
        self.refineZooSearch.clicked.connect(self.refine_zoo_search)
        self.currentSpecies = []
        self.currentZoos = []

    def species_search(self):
        searchtext = self.searchBar.text()
        print("Searching for: " + searchtext)
        url = localurl
        if self.visitableCheck.isChecked():
            sparql = """
PREFIX cervo: <http://www.cervo.io/ontology#>

SELECT ?name ?latinname
WHERE
{{{
    ?animal cervo:hasLatinName ?latinname.
    ?animal cervo:hasName ?name.
    FILTER(EXISTS{{SELECT ?animal {{?animal ^cervo:keeps-animal|cervo:is-kept-at ?zoo.}}}})
    FILTER(REGEX(?name, "{}", "i"))
}}}
UNION
{{{
    ?animal cervo:are-known-as|^cervo:can-refer-to ?collectiveterm.
    ?animal cervo:hasName ?name.
    ?animal cervo:hasLatinName ?latinname.
    FILTER(EXISTS{{SELECT ?animal {{?animal ^cervo:keeps-animal|cervo:is-kept-at ?zoo.}}}})
    FILTER(REGEX(?collectiveterm, "{}", "i"))
}}}
"""
        else:
            sparql = """
PREFIX cervo: <http://www.cervo.io/ontology#>

SELECT ?name ?latinname
WHERE
{{{
    ?animal cervo:hasLatinName ?latinname.
    ?animal cervo:hasName ?name

```

```

        FILTER(Regex(?name, "{}", "i"))
    }}
    UNION
    {{
        ?animal cervo:are-known-as|^cervo:can-refer-to ?collectiveterm.
        ?animal cervo:hasName ?name.
        ?animal cervo:hasLatinName ?latinname.
        FILTER(Regex(?collectiveterm, "{}", "i"))
    }}}
    """

sparql = sparql.format(searchtext, searchText)
r = requests.get(url, params={'format': 'json', 'query': sparql})
results = r.json()
print(results)

formatted_species = self.formatresults(results)
self.currentSpecies = formatted_species

self.format_view(formatted_species, self.speciesView)
self.refineSearchButton.setEnabled(True)
self.showZoos.setEnabled(True)
self.showExtraInfo.setEnabled(True)
self.speciesView.setCurrentCell(0, 0)

def refine_species_search(self):
    current_row = self.speciesView.currentRow()
    print(current_row)
    print(self.currentSpecies[current_row])

if self.showZoos.isChecked():
    url = localurl
    sparql = """
PREFIX cervo: <http://www.cervo.io/ontology#>

SELECT ?zoonaame ?postcode ?zoo_website
WHERE
{{
    ?animal cervo:hasLatinName "{}".
    ?zoo cervo:keeps-animal|^cervo:is-kept-at ?animal.
    ?zoo cervo:hasName ?zoonaame.
    ?zoo cervo:hasPostcode ?postcode.
    ?zoo cervo:hasURL ?zoo_website
}}
    """

elif self.showExtraInfo.isChecked():
    url = dbpedia
    sparql = """
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
SELECT ?name ?comment ?wikiarticle
    """

```

```

        WHERE {{
            ?animal dbpedia2:taxon "{}"^^rdf:langString.
            ?animal dbpedia2:name ?name.
            ?animal rdfs:comment ?comment.
            FILTER (LANG(?comment)='en')
            ?animal ^foaf:primaryTopic ?wikiarticle
        }}
        """
    else:
        return

    sparql = sparql.format(self.currentSpecies[current_row]['latinname'])
    r = requests.get(url, params={'format': 'json', 'query': sparql})
    results = r.json()
    formatted_extra_data = self.formatresults(results)
    self.format_view(formatted_extra_data, self.extraInfoView)

def zoo_search(self):
    searchtext = self.zooSearchBar.text()
    print("Searching for: " + searchtext)
    url = localurl
    sparql = """
    PREFIX cervo: <http://www.cervo.io/ontology#>

    SELECT ?zooName ?postcode ?zooWebsite
    WHERE
    {{
        ?zoo cervo:hasName ?zooName.
        ?zoo cervo:hasPostcode ?postcode.
        ?zoo cervo:hasURL ?zooWebsite
        FILTER(REGEX(?zooName, "{}", "i"))
    }}
    """
    sparql = sparql.format(searchtext)
    # todo: split the repeated section into its own function
    r = requests.get(url, params={'format': 'json', 'query': sparql})
    results = r.json()
    print(results)

    formatted_zoos = self.formatresults(results)
    self.currentZoos = formatted_zoos

    self.format_view(formatted_zoos, self.zooView)
    self.refineZooSearch.setEnabled(True)
    self.locationInfo.setEnabled(True)
    self.speciesKeptHere.setEnabled(True)
    self.zooView.setCurrentCell(0, 0)

def refine_zoo_search(self):
    current_row = self.zooView.currentRow()
    print(current_row)
    print(self.currentZoos[current_row])

    if self.speciesKeptHere.isChecked():
        url = localurl
        sparql = """
        PREFIX cervo: <http://www.cervo.io/ontology#>

```

```

        SELECT ?speciesKept
        WHERE
        {{
            ?zoo cervo:hasPostcode "{}".
            ?zoo cervo:hasName ?zooName.
            ?zoo cervo:keeps-animal|^cervo:is-kept-at ?species.
            ?species cervo:hasName ?speciesKept
        }}
        """
    elif self.locationInfo.isChecked():
        url = ordanancesurvey
        sparql = """
        PREFIX owl: <http://www.w3.org/2002/07/owl#>
        PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
        PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
        PREFIX postcode: <http://data.ordnancesurvey.co.uk/ontology/postcode/>

        SELECT ?districtname ?countyname ?countryname
        WHERE {{
            ?postcodeUri rdfs:label "{}".
            OPTIONAL{{?postcodeUri postcode:district ?district.
                ?district rdfs:label ?districtname}}.
            OPTIONAL{{?postcodeUri postcode:county ?county.
                ?county rdfs:label ?countyname}}.
            OPTIONAL{{?postcodeUri postcode:country ?country.
                ?country rdfs:label ?countryname}}
        }}
        """
    else:
        return

    sparql = sparql.format(self.currentZoos[current_row]['postcode'])
    r = requests.get(url, params={'format': 'json', 'query': sparql})
    results = r.json()
    formatted_extra_data = self.formatresults(results)
    self.format_view(formatted_extra_data, self.zooExtraInfo)

def format_view(self, formatted_data, view):
    if len(formatted_data) == 0:
        formatted_data = [{'error': 'Query returned no results'}]
    row_count = (len(formatted_data))
    column_count = (len(formatted_data[0]))
    view.setColumnCount(column_count)
    view.setRowCount(row_count)
    view.setSelectionMode(QAbstractItemView.SingleSelection)
    view.setHorizontalHeaderLabels((list(formatted_data[0].keys())))
    for row in range(row_count):
        for column in range(column_count):
            item = (list(formatted_data[row].values())[column])
            view.setItem(row, column, QTableWidgetItem(item))

def formatresults(self, data):
    keys = data['head']['vars']
    results = data['results']['bindings']
    return [{key: result[key]['value'] for key in keys} for result in results]

if __name__ == "__main__":

```

```
app = QtWidgets.QApplication(sys.argv)
window = CervoUI()
window.show()
sys.exit(app.exec_())
```

## Appendix B