A VKG Model of Bicycle Rental Services in Ghent

Candidate numbers: 100, 102, 114

May 14, 2023

1 Introduction

In this project we have constructed an ontology, mappings and queries for multiple datasets over rental bikes in Ghent, Belgium. The idea behind the application is to allow users to search for available rental bikes across different companies, and as such provide a unified portal for finding a bike to rent, without having to check the app or website for each individual company. We found this to be an excellent use-case for virtual knowledge graphs, and decided to base our semester project on it.

For the purpose of unifying the data, we created a virtual knowledge graph (VKG) using an Ontop-Protegé bundle. Furthermore we relied on database management technologies, front-end application frameworks, and APIs to create the final application.

2 Domain and datasets

The domain is rental bikes in the city of Ghent. There are several publicly available datasets for bike-sharing hubs and bikes in this city, from a few different companies. These datasets are fairly detailed, with location data on every entry, along with other useful information like availability and vehicle type.

The datasets used are publicly available on Ghents Open Data portal as CSVs, JSONs, and through APIs. We converted them to SQL insertion queries to generate tables in a PostgreSQL database. Links:

1. Blue Bike

2. Baqme

3. Donkey Republic

4. Dott

A total of 13 different datasets were initially selected for the project, which together provided a sufficiently comprehensive coverage of different suppliers of rentable bicycles for prototyping purposes. The companies produce distinct and unrelated data; furthermore, the data is heterogenous, as each company records data in different ways. As such, a knowledge graph is well suited for unifying the information contained by the datasets. Although the companies do not adhere to a unified data structure, they all contain pieces of comparable data as a consequence of dealing with entities with real-world placements. Each dataset has information about longitude and latitude, a similarity on which our application is built. We later found that some of the datasets were duplicates, where one contained slightly less information. In these cases we only mapped from the most informative duplicate. Because of this, four tables were excluded from mapping.

3 Creating a Virtual Knowledge Graph

VKGs can be used to unify information from different data sources, and to create efficient hierarchical graph representations. A VKG consists of three principal components: an ontology, a relational database schema(s), and mappings from the relational database(s) to the ontology[1].

To create and manage our VKG, we used Ontop-Protegé, which is a specialized extension to the ontology editor Protegé. It extends Protegé's features of ontology creation with the functionality to create mappings from a relational database schema, effectively translating the data contained in a relational database into an ontology representation, which can then be queried using an appropriate language, in this case SPARQL. It is important to note that the SPARQL queries are not run directly on an actual knowledge graph, but a virtual one. This means that it instead uses the mappings to translate the SPARQL queries into SQL, and then asks the relevant databases directly for the most up-to-date information.

3.1 Ontology

The ontology consists of 6 classes, 2 object properties and 9 data properties. The three main classes are Company, Hub and Vehicle, corresponding to companies offering rentable bicycle services, parking hubs for bicycles, and the bicycles themselves, respectively. Each hub and each bike is owned by a company, represented by the `:ownedBy` property (and its inverse, `:owns`). Each bicycle belongs to a subclass of Vehicle, corresponding to the exact type of bicycle. In addition, each hub accommodates a different type of bicycle, as specified by the `:vehicleType` data property.

The total of classes, object properties and data properties are shown in 1, alongside the corresponding fields from the relational schemas from which the data is extracted. Vehicles and hubs have mandatory fields `:hasLatitude` and `:hasLongitude`. These are used to calculate Manhattan distance from the users position. Afterwards, they are ran through Google Maps API to return an address that can be presented to the user. Vehicles also have mandatory fields

`:isReserved` and `:isDisabled`. Because of these, we can specify in the query that we're only looking for bikes that are not reserved or disabled. Vehicles and hubs have the optional field of `:geopoint`. Most of our source data has this property, but it is not necessary as long as we have datapoints on their latitude and longitude. Some of our datasources did not have latitude and longitude originally. In those cases they did have a geopoint value, which we extracted latitude and longitude from.

Hubs also have optional values of `:bikesAvailable` and `:hasAddress`. Optimally, each hub should have a value on how many bikes are available, but our dataset only has this for a few hubs. We chose to include it in the ontology for the hubs that have this value, but not to exclude the hubs that don't have it. Similarly, some hubs have an address, some don't. We resolved this by looking up the address through the latitude and longitude values.

Generally, our approach was to focus primarily on those properties which exist in all or most of the datasets. There are properties which fall outside of this restriction, but which nevertheless contain useful information, such as the column `last_seen` which appear in one of the tables containing bicycles. Modelling more information as (optional) data and object properties would facilitate more advanced queries later on in the process, and as such could be a good avenue for future extensions of the program.

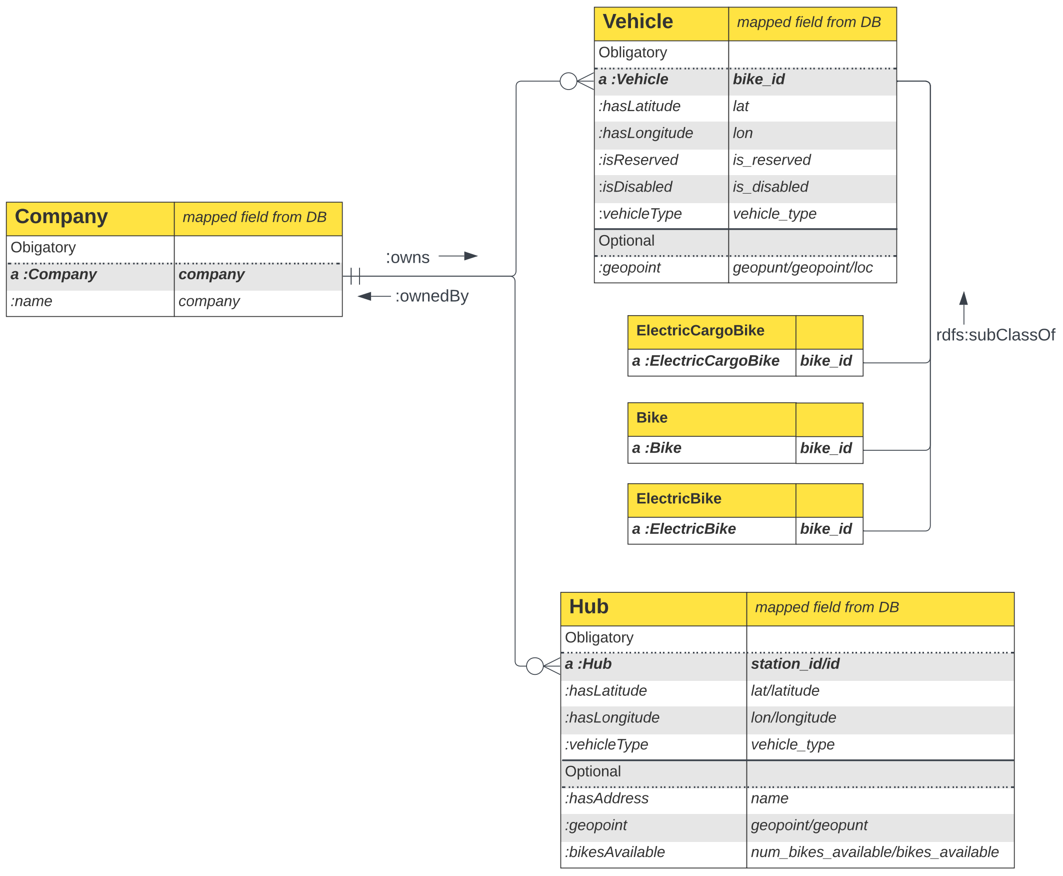


Figure 1: A model of the ontology in (modified) UML. Properties of the ontology are shown in the left columns, while the relational database columns from which the data originates are shown in the right columns.

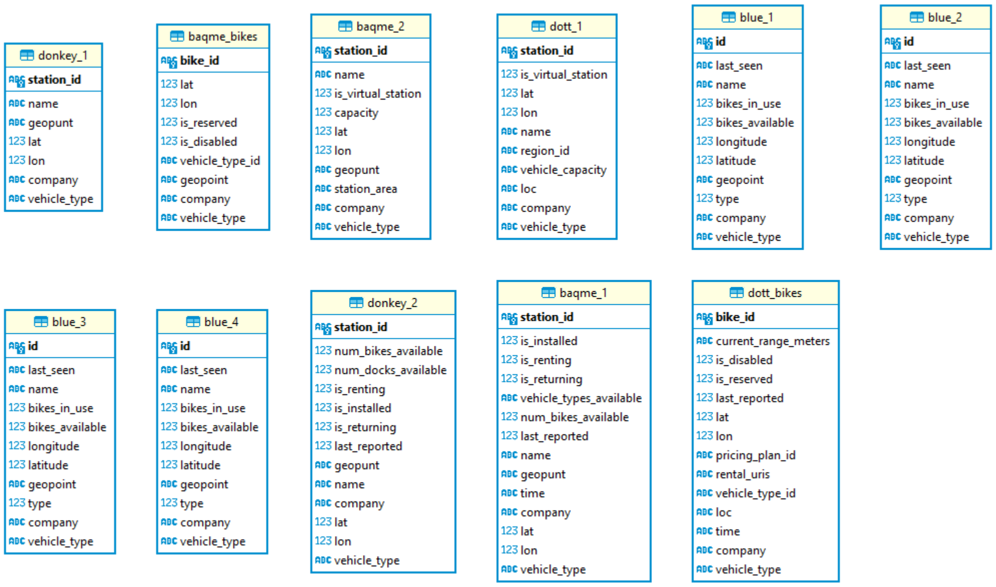


Figure 2: ER diagram of the database.

3.1.1 Relations

The main purpose of the program is not to implement a comprehensive tree which links together related schema in a hierarchical structure, but rather to unify heterogeneous data from several distinct data sources into a single queriable VKG. As a result, there are few relations required in the VKG, and none in the database. The one which is ubiquitous is `:owns`/`:ownedBy`, which specifies the ownership relationship between hubs/vehicles and companies.

3.2 Mappings

Figure 3 shows an example of a mapping from the relational database to the ontology. The instance identifier is collected from the primary key in the table, in this case `station_id`. Hubs have many properties that are collected from their corresponding fields in the database. There is a single object property, `:ownedBy` that connects hubs and vehicles to companies. For this object property we also defined an inverse `:owns` to complete the relation both ways, giving more flexibility when writing queries.

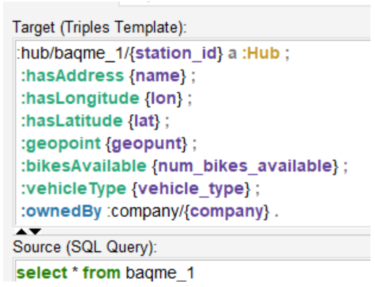

Target (Triples Template):
```
:hub/baqme_1/{station_id} a :Hub ;
:hasAddress {name} ;
:hasLongitude {lon} ;
:hasLatitude {lat} ;
:geopoint {geopunt} ;
:bikesAvailable {num_bikes_available} ;
:vehicleType {vehicle_type} ;
:ownedBy :company/{company} .
```
Source (SQL Query):
```
select * from baqme_1
```

Figure 3: Hub mapping from our project

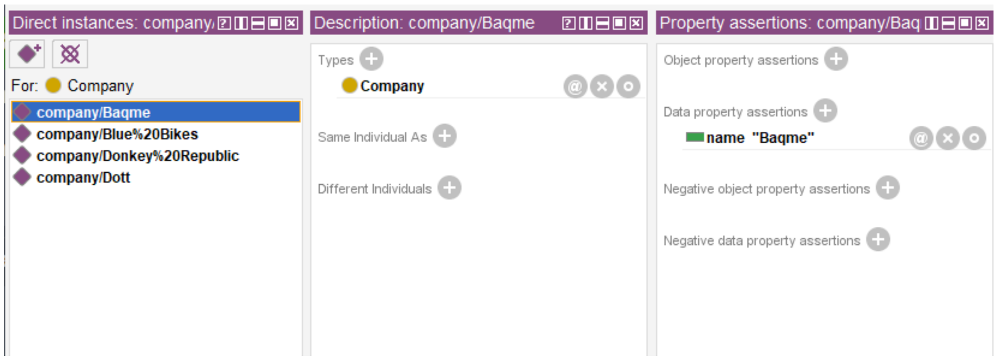

Direct instances: company
For: Company
company/Baqme
company/Blue%20Bikes
company/Donkey%20Republic
company/Dott
Description: company/Baqme
Types
Company
Same Individual As
Different Individuals
Property assertions: company/Baq
Object property assertions
Data property assertions
name "Baqme"
Negative object property assertions
Negative data property assertions

Figure 4: Individuals by class

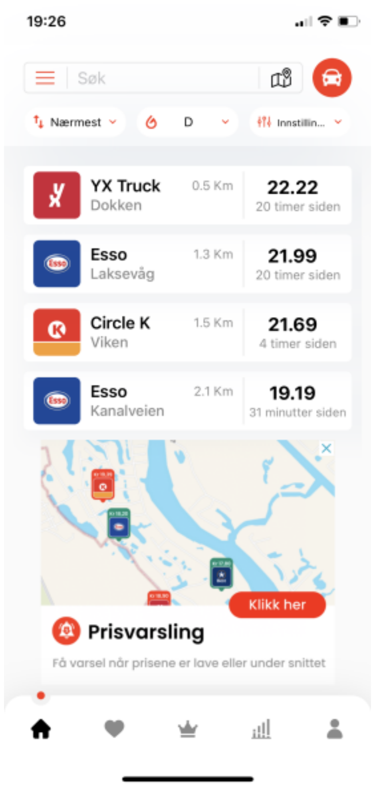
We chose to define companies explicitly, as
individuals by class, as opposed to mapping them from tables, which we tried at first but found it to produce impractically complex SQL translations when querying the VKG. Our system response time saw massive improvements when switching to explicit declarations as shown in figure 4.

4 Application

To fully utilize VKGs in an application setting, we looked to our own needs and how the properties of VKGs could suit them. As an example of a similar application, "Drivstoffappen" gathers information from several fuel providers to let the user consider multiple options at the same time 5 6. Although their technology is not open source, it is easy to imagine using APIs or web-scraping technologies in combination with a VKG to create the back-end architecture for a similar project. In much the same way, the goal of our application is to unify information from several companies, to provide easy-to-understand options for the user. The fact that company applications displays only the location of its own vehicles means that users have to check multiple apps to find the closest option. We leverage our own VKG to streamline this process.

4.1 Database

For the relational database server, we started out using H2 for its simplicity and portability. We then moved on to Denodo as we wanted more functionality in a better equipped environment. Denodo proved to be uncooperative with Protegé, particularly with `limit` statements. Instead of spending valuable time solving these obscure problems, we switched to the well-established, reliable database server and management system, PostgreSQL. It provides sufficient data handling tools and ease of use for our purposes. For instance, two of the data sets provided only a "geopoint", a concatenation of latitude and longitude


19:26

Søk

Nærmest | D | Innstillin...

YX Truck Dokken	0.5 Km	22.22 20 timer siden
Esso Laksevåg	1.3 Km	21.99 20 timer siden
Circle K Viken	1.5 Km	21.69 4 timer siden
Esso Kanalveien	2.1 Km	19.19 31 minutter siden

Klikk her

Prisvarsling

Få varsel når prisene er lave eller under snittet

Figure 5: Drivstoffappen - An overview of nearest options, though you can sort by price to view the cheapest options

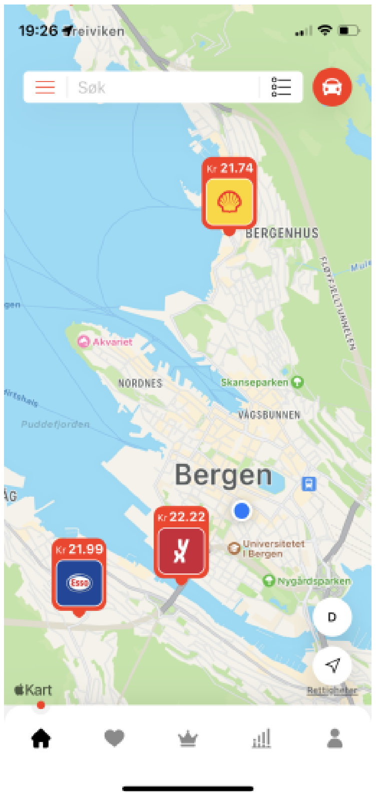


Figure 6: Drivstoffappen - Mapping out the options, an easy interface

which was impossible to compare directly to other datasets where they were defined individually. PostgreSQL allowed us to extract values from geopoint into individual columns. Furthermore, we faced very slow queries resulting from latitude and longitude having differently defined types between the datasets. Depending on the dataset these values could be denoted as floats, numerics or even text. This meant that our VKG management tool, Ontop, had to cast all latitude and longitude values to a common type, causing extensive rewrite times. We resolved this issue by rewriting the datatypes.

4.2 Hosting

Once PostgreSQL was set up for hosting and managing our database locally, we used the port forwarding functionality of ngrok to make the data remotely accessible for querying.

4.3 Endpoint

Using Ontop Command Line Interface, we exposed the VKG locally, which was accessed as an API through the front-end application.

We used the following program to run Ontop CLI:

```
ontop endpoint -t bikeOntology.rdf
-m bikeOntology.obda -p bikeOntology.properties
--cors-allowed-origins=http://localhost:3000/
```

4.4 Front end

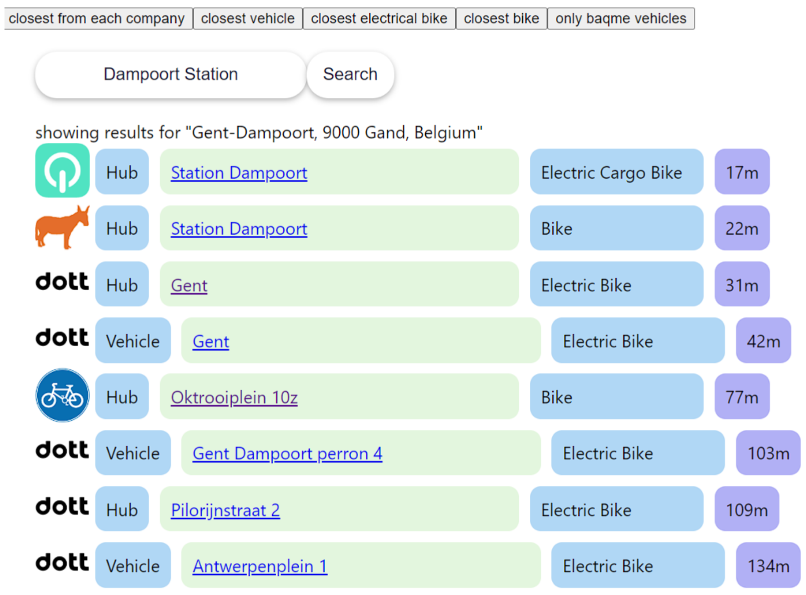
To create a neat user experience we created a React application. The front-end connects to the Ontop CLI endpoint and sends queries based on user input. In this way, the front-end application simplifies the complexity of the VKG and presents information to the user in a way that meets their needs.

4.5 Website

The website we created 8 provides a streamlined way of interacting with the VKG. It has a field for input, where the user can write any valid address. When the search button is pressed, a GET request is made to Google's Geocoding API, which returns a host of information about the user input, including a fully formatted address, longitude, and latitude. The numerical values for longitude and latitude are inserted into a predefined SPARQL query. The SPARQL query is then sent to the Ontop CLI, and the React application renders the response. Whenever a candidate vehicle or hub is displayed, the user can click on its address and be redirected to Google Maps, where they are shown the quickest path to the target.

```
PREFIX : <http://www.example.org/bikeOntology#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

select ?distance ?lat ?lon ?cname ?class ?vt ?x
where {
{select ( abs(3.691 - ?lon) + abs(51.02 - ?lat) as ?distance) ?lat ?lon ?cname ?class ?vt ?x
where {
    ?x :ownedBy ?c .
    ?x :hasLatitude ?lat .
    ?x :hasLongitude ?lon .
    ?c :name ?cname .
    ?x a ?class .
    ?x :vehicleType ?vt .
    filter (?c = <http://www.example.org/bikeOntology#company/Baqme>) .

}
order by asc (?distance)
limit 1}
union
{select ( abs(3.691 - ?lon) + abs(51.02 - ?lat) as ?distance) ?lat ?lon ?cname ?class ?vt ?x
where {
    ?x :ownedBy ?c .
    ?x :hasLatitude ?lat .
    ?x :hasLongitude ?lon .
    ?c :name ?cname .
    ?x a ?class .
    ?x :vehicleType ?vt .
    filter (?c = <http://www.example.org/bikeOntology#company/Blue%20Bikes>) .

}
order by asc (?distance)
limit 1}
union
{select ( abs(3.691 - ?lon) + abs(51.02 - ?lat) as ?distance) ?lat ?lon ?cname ?class ?vt ?x
where {
    ?x :ownedBy ?c .
    ?x :hasLatitude ?lat .
    ?x :hasLongitude ?lon .
    ?c :name ?cname .
    ?x a ?class .
    ?x :vehicleType ?vt .
    filter (?c = <http://www.example.org/bikeOntology#company/Donkey%20Republic>) .

}
order by asc (?distance)
limit 1}
union {select ( abs(3.691 - ?lon) + abs(51.02 - ?lat) as ?distance) ?lat ?lon ?cname ?class ?vt ?x
where {
    ?x :ownedBy ?c .
    ?x :hasLatitude ?lat .
    ?x :hasLongitude ?lon .
    ?c :name ?cname .
    ?x a ?class .
    ?x :vehicleType ?vt .
    filter (?c = <http://www.example.org/bikeOntology#company/Dott>) .

}
order by asc (?distance)
limit 1}
}
order by asc (?distance)
```

Figure 7: An example of a SPARQL query of the VKG. This finds the closest hub or vehicle from each company.


closest from each company | closest vehicle | closest electrical bike | closest bike | only baqme vehicles

Dampoort Station | Search

showing results for "Gent-Dampoort, 9000 Gand, Belgium"

	Hub	Station Dampoort	Electric Cargo Bike	17m
	Hub	Station Dampoort	Bike	22m
dott	Hub	Gent	Electric Bike	31m
dott	Vehicle	Gent	Electric Bike	42m
	Hub	Oktrooiplein 10z	Bike	77m
dott	Vehicle	Gent Dampoort perron 4	Electric Bike	103m
dott	Hub	Pilorijnstraat 2	Electric Bike	109m
dott	Vehicle	Antwerpenplein 1	Electric Bike	134m

Figure 8: 320Bikes website

5 Use case: Finding the closest available vehicle/hub from each company

Here, we aim to locate the closest vehicle or hub from each of the four companies in our data. This might be useful in a situation where a user carries a subscription to one of the companies' services, which makes it desirable to locate the closest vehicle or hub from that company. At the same time, it might be desirable to check nearby locations from other companies, in case the location from the preferred company is particularly far away, or a location from a different company is particularly close, which might override the initial preference.

Currently, this functionality is implemented as a separate button in the web application interface. When the user inputs their location and presses the button, the predefined SPARQL query depicted in 7 is passed to the SPARQL endpoint. This query finds the union of four subqueries, each of which finds the closest hub or vehicle of one of the companies; the results are subsequently ordered from closest to furthest away. The app then presents the four ordered results to the user, who is then given the option to navigate to an external site providing precise directions.

6 Summary

The program we have created allows the user to search for the location of rentable bicycles and bicycle hubs, from several different companies, within the city of Ghent. It makes use of a SPARQL endpoint which queries a VKG implemented in Ontop-Protegé, which again connects to a PostgreSQL database. The data used was collected from the Open Data Portaal Ghent and converted to a relational database format, which Ontop-Protegé can query based on SPARQL queries sent through the SPARQL endpoint, translated into SQL using Ontop's built-in functionality.

Using these tools, it has been possible to transform a sample of unrelated, heterogeneous data sources, produced by different bicycle rental companies, into a dynamic and flexible virtual knowledge graph, to which complex queries can be made.

As future work, we would aim to expand the VKG by implementing more of the information from the data sources in our ontology. For instance, we made the decision to focus primarily on those fields which exist, in some format, in most or all of the datasets. There are some which fall outside of this restriction which nevertheless contain potentially useful information. Furthermore, we would look to enhance the usability of the web application, by providing more options to the user; particularly, a modular search functionality where the user could include or exclude specific properties from their search.

References

[1] Guohui Xiao, Linfang Ding, Benjamin Cogrel, and Diego Calvanese. Virtual Knowledge Graphs: An Overview of Systems and Use Cases. *Data Intelligence*, 1(3):201–223, 06 2019.