Evolutive Graphics with Linked Data

Carlos Saito Murata
Abstract

Evolutive Graphics with Linked Data

Carlos Saito Murata

Data visualization in journalism became in the last few years an important knowledge area that mixes both journalism and computer science. This project is focused on data that evolves over time, its visualization and how it is implemented nowadays. The project proposes two kind of improvements: graphics that automatically changes when data gets updated and integration of external data to include information from knowledge databases.

This project creates a prototype that uses both data that evolves over time and data from other resources. It is created around the topic of migration, enabling users to view migrations in a map and filter those movements with filters like "migrations that happened from poor to rich countries". The prototype uses migration data stored in an accessible database combined with data about countries extracted from Wikidata. The visualization also gets updated automatically if the sources change: for example when the metrics used to guess the richness/poorness of a country change.
Contents

1 Introduction
   1.1 Data and graphics that evolves over time . . . . . . . . . . . . . 1
      1.1.1 The Weinstein scandal . . . . . . . . . . . . . . . . . . . . 2
      1.1.2 The Panama Papers . . . . . . . . . . . . . . . . . . . . . 4
   1.2 External sources and Linked Data . . . . . . . . . . . . . . . . 5
   1.3 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

2 Background
   2.1 Ontag . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
   2.2 Wikidata . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
      2.2.1 RDF . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
      2.2.2 SPARQL . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
      2.2.3 Other data sources . . . . . . . . . . . . . . . . . . . . . . 12

3 Characterization of data
   3.1 Data that evolves over time . . . . . . . . . . . . . . . . . . . . 14
   3.2 Domain specific problems . . . . . . . . . . . . . . . . . . . . . 14
      3.2.1 Partial information . . . . . . . . . . . . . . . . . . . . . . 15
      3.2.2 Contradictory data . . . . . . . . . . . . . . . . . . . . . . 15
   3.3 Other problems . . . . . . . . . . . . . . . . . . . . . . . . . . 16

4 Development
   4.1 Pre-design and high-level design . . . . . . . . . . . . . . . . . 19
   4.2 Top-level design . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
   4.3 Design and implementation . . . . . . . . . . . . . . . . . . . . . 22

5 Results
   5.1 Testing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
      5.1.1 Idempotent testing . . . . . . . . . . . . . . . . . . . . . . 27
      5.1.2 Characterization testing . . . . . . . . . . . . . . . . . . . . 27
   5.2 User interaction . . . . . . . . . . . . . . . . . . . . . . . . . . 28

6 Conclusions and Future work
   6.1 Future work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
   6.2 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
# API reference

A.1 Entity recognition ........................................ 36
  A.1.1 Global functions ........................................ 36
  A.1.2 Recognizer instance methods ......................... 36
A.2 Frontend .................................................... 37
A.3 Query ........................................................ 39
  A.3.1 Example ................................................ 39
Chapter 1

Introduction

In the last few years, data visualization and journalism became together forming a new discipline: data-driven journalism. Authors like J. Gray et al. [1] and C. W. Anderson [2] explain the concept of data journalism and its importance. In addition, collections of data have become available online (e.g. Open Government Data) and open source tools allow analyzing and visualizing the data even with little knowledge of information technology [3]. This gives journalists access to new types of data and creation of more complex and data driven visualizations, both to tell a story in better ways and. It also provides help to journalists to understand the data they handle.

This Project focuses on a specific type of data and the visualization of it, which is the data that evolves over time—which leads to graphics that evolves over time—, explained through two examples used in journalism. It is also questioned how can this be improved and for this improvement, two ideas are suggested: the automatization of the evolution of data and its graphics, and, the incorporation of external data (through Linked Data) that can also change over time.

1.1 Data and graphics that evolves over time

The data managed in this project are that data that changes over time. In short, this type of data is characterized for “giving different answers to the same question depending on the time the question is stated”. This could happen two scenarios.

1. **The question implies time.** For example, if the question like “Who is the winner of the last Tour de France?” the answer is different depending on the year because of the annual periodicity of the tournament.

2. **The question does not imply time.** For example a question like “Who is the winner of the Tour de France in 2014?” the answer is apparently fixed. However, years after the initial announcement of the winner, as
a result of a doping scandal, a different person could be declared as the
winner of the tournament.

In both scenarios there is a problem about the reliability of the data. The
source might be corrupted or the data might be not properly updated. The
correctness of the data is out of the scope of this project. However, as long as
the available information is updated and correct, the project is able to represent
it properly. Some of the problems are addressed and corrected or, at least
discovered.

Two examples of usage of data and graphics that evolve over time are shown
below.

1.1.1 The Weinstein scandal

This example is an article published on Univision\(^1\) that talks about the impli-
cations of the “#meeToo movement”, an International movement against sexual
harassment and assault spread virally in 2017 as a hashtag used on social media
to help demonstrate the widespread prevalence of sexual assault and harassment,
especially in the workplace. It shows a list of news about famous people report-
ing sexual harassment.

The list of news has a graph in its left side that shows some numbers: When
the user scrolls down, the list of news scrolls but the graph remains its position
displaying different information depending on the position of the scroll. The list
displays two types of graphics: (i) below the photo of the harasser, the number
of people been harassed according to the article that is aligned with the graphic,
and, (ii) the total number of people that has been harassed according to all the
articles from the beginning until the aligned one (figures 1.1, 1.2 and 1.3)

Figure 1.1: Screenshot of the Univision article with the scroll on top

\(^1\)See \url{http://uni.vi/z7ar100VHVT}
When the graph is aligned with the first article, it shows that 8 people are harassed according to that article, and in total 8 people are harassed. (figure 1.1)

When the graph is aligned with the second article, it shows that 2 people are harassed according to that second article, and that 8+2 sums 10, the number of people harassed according to both first and second articles (figure 1.2).

When the user scrolls to the end of the page, the graph shows the total number of people that has been reported as harassed in total (figure 1.3). As more articles are added, the number would change making that data non-constant.
1.1.2 The Panama Papers

This example is an ICIJ article\(^2\) that shows the relationship between the people that have close relationships with Donald Trump —the president of the United States of America in 2018— and the “Panama Papers” scandal\(^3\).

It shows a graphic with Donald Trump in the center and lines that end in circles that are the people close to him (figure 1.4).

![Figure 1.4: Screenshot of the ICIJ article when the user enters to the page](Image)

When the user click one of the people, it shows a biography in the right side and a graph with the connections described in the biography in the left side (figure 1.5). As the user scrolls through the different parts of the biography in the right, the graphic in the left changes showing the information that is written in the part that the user is reading (figure 1.6). The information includes relationships between people and organizations (private companies and public organizations).

![Figure 1.5: Screenshot of the ICIJ article when user clicks on “Randal Quarles”](Image)

---

\(^2\)See [https://projects.icij.org/paradise-papers/the-influencers/#/](https://projects.icij.org/paradise-papers/the-influencers/#/)

\(^3\)The Panama Papers are 11.5 million leaked documents that detail financial and attorney–client information for more than 214,488 offshore entities.
1.2 External sources and Linked Data

This Project explores the inclusion of data from multiple sources that enables the access to more data and the creation of more meaningful visualizations using those data.

Specifically, this project uses two types of data: internal data and external data.

Internal data are strings extracted from news articles with semantics annotations. This extraction is done with the tool Ontag described in Section 2.1. In Ontag, data is curated and validated by the community.

With the semantic annotations it is possible to join the data with external sources like Wikidata.

To avoid contradictions between internal and external data, different information is extracted from one and other source. In case of having more than one external source it is necessary to have a mechanism to address conflicts between sources (either choosing one over another or having some aggregation). The implementation of this mechanism is out of the scope of the project.

The project also assumes that all the data are facts: the sources have their own mechanisms to guarantee its correctness before inserting them into the system.

These improvements can be applied in the examples shown in section 1.1.
One example of improvement using external sources in the “Weinstein scandal” scenario could be retrieve the data from various news sources or complementing the information with other databases like IMDb\(^4\), the Internet Movie Database and discover in which cases the harasser and victim worked in the same movie.

In the “Panama Papers” scenario, a developer might want to retrieve information of the people involved from general knowledge databases like Wikipedia or other databases like Data.gov\(^5\), the collection of datasets published by the Government of the United States.

The major requirement when dealing with this type of data (data from different sources) is to be able to connect the different databases together. Linked Data is a concept aimed to solve this problem.

Linked Data is about employing two technologies: (i) Resource Description Framework (RDF) — a family of specifications of the W3C (see [4]) — to describe and model information, and, (ii) the Hypertext Transfer Protocol (HTTP) to publish structured data on the Web and to connect data between different data sources, effectively allowing data in one data source to be linked to data in another data source [5]. The principles of Linked Data were defined by Tim Berners-Lee in 2006 [6] and its guidance has been extended by documents like [7] that provides recipes on which publishing systems can be based.

The mechanisms and technologies behind Linked Data and the usage in this particular Project are discussed in Section 2.2.1.

1.3 Contributions

1. Design a system to create graphics that changes automatically as data evolves.

2. Integrate external data sources with stored data.

\(^4\)See https://imdb.com
\(^5\)See https://data.gov
Chapter 2

Background

The solution of visualizing data that evolves over time and data from different sources is proposed in this Project through the development of a software prototype. The prototype involves the design and development of a software that shows relationships between migration movements and the properties of the places where those migrations happen.

The prototype takes those information from different sources: (i) migration movements are taken from Ontag and they are considered the internal data of the project. (ii) properties of places where the migrations happen are taken from Wikidata and they are considered the external data of this project.

![Diagram of External and Internal Data](image)

Figure 2.1: External and internal data in this project

2.1 Ontag

Ontag\(^1\) is a tool that converts news articles into machine-readable data. It is promoted and developed by Common Action Forum in collaboration with the Ontology Engineering Group of the Technical University of Madrid.

Ontag works joining the concepts of question, tag, annotation and answer in four steps:

\(^1\)See [https://ontag-face.herokuapp.com](https://ontag-face.herokuapp.com)
1. **Create the question.** The community creates questions with periodical relevance. For example: *Describe the migration flow of refugees.*

2. **Tag the question.** The author of the question creates tags, which are the structure that answers of the question should have. For example, the question may have the tags: *place of origin, destination, amount, date.*

3. **Propose content.** Users propose content that may answer the question. For example, *news articles.*

4. **Highlight the content.** Users highlight parts of the content creating annotations.
   
   Then, users put the question tags on the annotations. For example, in an article, a user can highlight *Syria* and tag it with *place of origin*; highlight *Lesbos* and put the tag *destination* and so on.

   All the annotations (with the tags) can be group together to form an answer for the question. See figure 2.2.

![Figure 2.2: How data are related in ontag](image)

The data in Ontag is stored as text and can be read from a public API. The relevant endpoint for this project is **GET /answers**. It gives a list of answers, where each answer is a list of annotations.

```json
{
  id: 3,
  question_id: 1,
  annotations: [
    {text: 'Syria', tag: 'origin'},
    {text: 'Lesbos', tag: 'destination'},
    {text: '38760', tag: 'amount'}
  ]
}
```
2.2 Wikidata

As a source of “properties of places”, this Project uses Wikidata. Wikidata is a free and open knowledge base that can be read and edited by both humans and machines. Wikidata acts as central storage for the structured data of its Wikimedia sister projects including Wikipedia, Wikivoyage, Wikisource, and others [8].

The human-readable part of wikipedia are HTML pages, each one describing a concept and readable as a physical encyclopedia.

To make the data computer-readable, Wikidata implements the principles and technologies of Linked Data.

The term Linked Data was coined by Tim Berners-Lee. He outlined four principles of linked data: [6]

1. Use URIs as names for things.
2. Use HTTP URIs so that people can look up those names.
3. When someone looks up a URI, provide useful information using the standards.
4. Include links to other URIs, so that they can discover more things.

For this Project it is relevant to know how data are conceptually stored and how can data be read. Data are stored implementing RDF and they can be read using SPARQL. The following section (2.2.1) only describes RDF as a concept. The actual implementation of both RDF and SPARQL is not covered here and it is not relevant for this Project.

2.2.1 RDF

Resource Description Framework (RDF) is a family of specifications of the World Wide Web Consortium (See [4]) used to describe and model information. This section explains how a page in Wikidata describing Douglas Adams is transformed into computer-readable data conformed to the RDF specs.

The article in Wikidata about Douglas Adams contains (among others) the information shown in the Table 2.1:

<table>
<thead>
<tr>
<th>Douglas Adams</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native language</td>
</tr>
<tr>
<td>Place of birth</td>
</tr>
<tr>
<td>Educated at</td>
</tr>
</tbody>
</table>

Table 2.1: Human readable information about Douglas Adams

In RDF all the information is stored in triples. Every triple is a subject-predicate-object tuple. The information shown in table 2.1 is equivalent to the triples shown in table 2.2 where each row is a triple.

\footnote{See https://wikidata.org/wiki/Q42}
Table 2.2: Information about Douglas Adams expressed in triples

<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Douglas Adams</td>
<td>Native language</td>
<td>British English</td>
</tr>
<tr>
<td>Douglas Adams</td>
<td>Place of birth</td>
<td>Cambridge</td>
</tr>
<tr>
<td>Douglas Adams</td>
<td>Educated at</td>
<td>St John’s College</td>
</tr>
</tbody>
</table>

Then, following the principle of Linked Data that says that all the URIs are used as names for things, every concept (thing) should be identified by an URI as shown in the table 2.3:

<table>
<thead>
<tr>
<th>Concept</th>
<th>URI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Douglas Adams</td>
<td><a href="https://wikidata.org/wiki/Q42">https://wikidata.org/wiki/Q42</a></td>
</tr>
<tr>
<td>British English</td>
<td><a href="https://wikidata.org/wiki/Q7979">https://wikidata.org/wiki/Q7979</a></td>
</tr>
<tr>
<td>Cambridge</td>
<td><a href="https://wikidata.org/wiki/Q350">https://wikidata.org/wiki/Q350</a></td>
</tr>
<tr>
<td>Place of birth</td>
<td><a href="https://wikidata.org/wiki/Property:P19">https://wikidata.org/wiki/Property:P19</a></td>
</tr>
</tbody>
</table>

Table 2.3: Concepts as URIs

It is important to note that the predicates in the triples (“Native language”, “place of birth”, “educated at”) are also concepts and because of this, they are identified by URIs.

In conclusion, RDF represents data in triples, where each element that is not a simple datatype (number, boolean, string) is identified by an URI.

2.2.2 SPARQL

SPARQL is an RDF query language, that is, a semantic query language for databases, able to retrieve and manipulate data stored in RDF format. SPARQL allows for a query to consists of triple patterns, conjunctions, disjunctions and optional patterns [9].

SPARQL queries allow to query data from a triples database. The queries can search for triples given any part of them. For example, knowing the URI for “Douglas Adams” (written in the code as wd:Q42) and the URI for “Native language” (wdt:P103), it is possible to perform a query to look for the object of triples where subject is “Douglas Adams” and predicate is “Native language”. This query, in SPARQL language is:

```sparql
SELECT ?language WHERE {
  wd:Q42 wdt:P103 ?language
}
```

This returns the language “British English” bound to the variable “?language” defined in the query.
It is possible also to make queries that return more than one result. The following query returns all the cities stored in Wikidata, or equivalently, all the subjects (bound to the variable “?city”) of triples where predicate is “instance of” and object is “city” (In the code shown below, for simplification, the actual URIs for “instance of” and “city” are replaced by “wdt:instance_of” and “wdt:city” respectively).

```
SELECT ?city WHERE {
}
```

The result is a list of all the cities of the world (table 2.5 shows 5 elements of the actual list returned by Wikidata contains more than 11000 elements).

<table>
<thead>
<tr>
<th>?city</th>
</tr>
</thead>
<tbody>
<tr>
<td>Berlin</td>
</tr>
<tr>
<td>London</td>
</tr>
<tr>
<td>Toronto</td>
</tr>
<tr>
<td>Nuuk</td>
</tr>
<tr>
<td>Vatican City</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

Table 2.5: Extract of the list of “All cities in the world” returned by wikidata

It is possible to make more complex queries to retrieve at the same time a list of all the countries in the world and some data of those countries like the GDP per capita or its country code.

Knowing the URIs of the correct terms (shown in table 2.6), the following code will return a table of all the countries in the world with its GDP per capita and its 2-digit country code. The result of the query is shown in table 2.7.

```
SELECT ?country, ?countryCode, ?gdp WHERE {
    ?country wdt:P31 wd:Q3624078.
}
```
### Table 2.6: Concepts and their URIs in Wikidata

<table>
<thead>
<tr>
<th>Concept</th>
<th>URI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance of</td>
<td>wd:P31</td>
</tr>
<tr>
<td>Sovereign country</td>
<td>wd:Q3624078</td>
</tr>
<tr>
<td>ISO 3166-1 alpha-2 code</td>
<td>wdt:P297</td>
</tr>
<tr>
<td>GDP per capita</td>
<td>wdt:2299</td>
</tr>
</tbody>
</table>

### Table 2.7: Countries, country codes and GDP

<table>
<thead>
<tr>
<th>?country</th>
<th>?countryCode</th>
<th>?gdp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canada</td>
<td>CA</td>
<td>45066</td>
</tr>
<tr>
<td>Ireland</td>
<td>IE</td>
<td></td>
</tr>
<tr>
<td>Spain</td>
<td>ES</td>
<td>33629</td>
</tr>
<tr>
<td>Luxemburg</td>
<td>LU</td>
<td>101926</td>
</tr>
</tbody>
</table>

For this query, three properties are used as examples: *Sovereign country*, *ISO 3166-1 alpha-2 code* and *GDP per capita*. Later on the Project (see section 4.1) when the actual properties are used to make the prototype, a proper definition will be given.

### 2.2.3 Other data sources

The external sources for this Project could be another database. Wikidata is a general knowledge database and not special domain. This means that Wikidata can offer a broad knowledge on diverse areas but not deep knowledge on any of them. For the scope of this project, and the target of the application, the knowledge offered by this type of database is enough.

Another database that was taken into consideration was DBPedia\(^3\). DBpedia is a “crowd-sourced community effort to extract structured content from the information created in various Wikimedia projects. This structured information resembles an open knowledge graph (OKG) which is available for everyone on the Web” \([10]\).

The main difference between DBPedia and Wikidata is how concepts are defined in each. Since DBPedia is a knowledge base extracted from Wikipedia and Wikipedia is multi-lingual (different languages have different versions of wikipedias), DBPedia results in a multi-lingual knowledge base.

In the other hand Wikidata is a unique knowledge base where each defined concept can have multiple “labels” (one per language) associated to that concept.

Because of this, in DBPedia, the same concept (i.e. “Greece”) may have different URIs (one for each Wikipedia article). All the URIs are linked to each other by a property “same-as”.

---

\(^3\)See [https://wiki.dbpedia.org](https://wiki.dbpedia.org)
Even with the efforts to unify concepts and URIs (explained by Kontokostas et al. in [11]), for this Project, the approach taken by DBPedia is more problematic and Wikidata is preferred. However, having in mind this issue, DBPedia is a great alternative that offers more quantity of information than Wikidata. Further and detailed comparison of more knowledge databases are done in other publications like [12] which also compares Wikidata and DBPedia with other services like YAGO\textsuperscript{4}, Freebase\textsuperscript{5} and OpenCyc\textsuperscript{6}.

\textsuperscript{4}See http://yago-knowledge.org
\textsuperscript{5}See https://freebase.com
\textsuperscript{6}See http://www.cyc.com/opencyc/
Chapter 3

Characterization of data

At this point of the Project it is important to define properly what are the kind of data that is faced (data that evolves over time) and do an analysis on the potential problems that can arise with these data in general and within the domain of the prototype in particular.

3.1 Data that evolves over time

Data that evolves over time could be defined as data “that gives different answers to the same question depending on the time the question is stated”. This could happen two scenarios.

1. The question implies time. For example, if the question like “Who is the winner of the last Tour de France?” the answer is different depending on the year because of the annual periodicity of the tournament.

2. The question does not imply time. For example a question like “Who is the winner of the Tour de France in 2014?” the answer is apparently fixed. However, years after the initial announcement of the winner, as a result of a doping scandal, a different person could be declared as the winner of the tournament.

In both scenarios there is a problem about the reliability of the data. The source might be corrupted or the data might be not properly updated. The correctness of the data is out of the scope of this project. However, as long as the available information is updated and correct, the project is able to represent it properly. Some of the problems are addressed and corrected or, at least discovered.

3.2 Domain specific problems

The system queries data with information about migration movements between places. The table 3.1 is an example of an entry.
Table 3.1: Example of migration data taken from Ontag

<table>
<thead>
<tr>
<th>Origin</th>
<th>Destination</th>
<th>Date range</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syria</td>
<td>Lesbos</td>
<td>2015-01-01 to 2015-06-30</td>
<td>38 760</td>
</tr>
</tbody>
</table>

The system has to query data following certain criteria. In this operation there are some problems that may happen depending on the algorithm used in each operation.

For all the problems described in this section, the developer that implements the system should also ensure that the correct data is returned when queried under those circumstances.

### 3.2.1 Partial information

In some cases, the query matches partially with the data. For example, if a query is “read all the migration that happened in 2016” and the stored data is the data of table 3.2:

<table>
<thead>
<tr>
<th>Origin</th>
<th>Destination</th>
<th>Date range</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td>C</td>
<td>2015-12-20 to 2016-03-20</td>
<td>5000</td>
</tr>
<tr>
<td>CD</td>
<td>AB</td>
<td>2015-12-20 to 2017-03-02</td>
<td>10000</td>
</tr>
</tbody>
</table>

Table 3.2: Example of partial information problem

Some implementations might ignore both data because the date range is out of 2016, which is the most restrictive approach. However, others may try to interpolate and calculate how many people among the people in both rows corresponds to year 2016.

### 3.2.2 Contradictory data

In some cases, different registers show contradictory information. In the example in table 3.3, the same migration from A to B is happening at the same time but different amount of people is doing the migration.

<table>
<thead>
<tr>
<th>Origin</th>
<th>Destination</th>
<th>Date range</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>2015-12-20 to 2016-03-20</td>
<td>10000</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>2015-12-20 to 2016-03-20</td>
<td>1000</td>
</tr>
</tbody>
</table>

Table 3.3: Example of contradictory data

The most restrictive implementation discards both data and also includes tests to detect this types of contradiction. However, other implementations may try to extract a conclusion from these registers, for example, returning the average amount.
Sometimes, the information could be partially contradictory. Consider the example in table 3.4 where the date ranges of rows (1) and (2) overlap some days.

<table>
<thead>
<tr>
<th>Origin</th>
<th>Destination</th>
<th>Date range</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>2015-12-20 to 2016-03-20</td>
<td>10000</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>2016-02-20 to 2016-08-20</td>
<td>1000</td>
</tr>
</tbody>
</table>

Table 3.4: Example of partially contradictory data

The most restrictive implementation would discard both data in case of a query including them together. However, if the query is performed to get only migrations in 2016, some implementations would either take or not the second row.

Algorithms that try to aggregate the data to return calculated data should be designed to consider queries that include one, the other or both rows and other cases where more than two rows are partially contradictory.

Other types of contradiction are harder to detect. Consider the data in table 3.5:

<table>
<thead>
<tr>
<th>Origin</th>
<th>Destination</th>
<th>Date range</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Paris</td>
<td>2015-12-20 to 2016-03-20</td>
<td>10000</td>
</tr>
<tr>
<td>A</td>
<td>France</td>
<td>2015-12-20 to 2016-03-20</td>
<td>1000</td>
</tr>
</tbody>
</table>

Table 3.5: Example of semantic contradiction

In this case, both data are not contradictory from a strict formal point of view. However, it is not possible that more people move from the same place (A) to Paris than to France as Paris is part of France.

This type of contradiction needs a deep knowledge of the database and skills including advanced entity recognition that are out of the scope of this Project.

The implementations that are mentioned but are not included in the Project have different implications: they might require a deep knowledge in statistics, geographics, anthropology or sociology among others. Some of them also opens ethical issues leading to mis-information or bias of the designer of the algorithm. All these implications are outside the scope of this project.

### 3.3 Other problems

Working with data from external sources have mainly two problems: availability and reliability.

Availability problems may happen with actual unavailability of the service or because of some network problem. A similar problem to this is performance. If the software must handle multiple requests and each of them take some amount of time, the result could be a bad performing system. Both problems can be
solved by some solutions like implementing a cache. These solutions are out of
the scope of this project.

Reliability issues happen if the data from external sources is incomplete,
contradictory or not true. It is completely out of the scope of this project to solve
this issue. In this Project, it is assumed that all the information provided by
all sources, i.e. Wikidata and Ontag, are facts. This is possible because those
projects have their own methods to verify the information.
Chapter 4

Development

To design the software, the methodology chosen is an adapted version of incremental agile. The steps of this methodology are: Pre-design (including collection of user requirements), Design (high-level and low-level), Implementation and Test.

1. **Pre-design.** In this phase, a series of preconditions are set. These preconditions are chosen as limits of the prototype and include technical decisions: programming languages, tools, frameworks.

   Based on the limitations of the prototype, a number of interviews are conducted to users in order to have the point of view of the potential users of the system. The outcome of this phase is an *initial design of the application*.

2. **High-level design.** A design of the system is made based on the requirements taken from the previous phase. The design includes both software architecture design and user interface design. After this phase, the software is divided into parts that can be developed incrementally.

   After the pre-design and high-level design phases, a loop of the phases design, implementation and test is done for each part of the software:

   1. **Low-level design.** Design of one part of the system. It include the details of the architecture and details of the user interface.

   2. **Implementation.** The actual code for that part of the system is written in this phase.

   3. **Test.** In this phase, all the written code is tested. First, it is tested using unit tests. Then, the integration with the existing parts of the software is also tested (integration tests). Finally, if necessary, the user interface is tested against real users.
4.1 Pre-design and high-level design

The prototype of the system is a “dashboard”-like web application. The dashboard accepts two inputs from the user: a date and a filter. Based on those two inputs and having the migrations and countries data, the dashboard will show a visualization of the migration data that meets the filter chosen by the user.

For example, a user might want to see migrations that happened in 2015 from poor to rich countries. In this example, 2015 is the date and from poor to rich countries is the filter.

To discuss which filters are better to have in this prototype, a series of interviews are performed in order to have feedback from potential users.

Three interviews are conducted in this phase. The interviewed people are: (i) a Professional Journalist working in a Non-Profit Organization, (ii) a Student of Master in Human Rights at Uppsala University and (iii) a Student of Bachelor in Peace and Development at Uppsala University.

To all them, a brief explanation of the app is given with the stated question above. After that, possible filters are discussed. These are the filters that interviewed people found interesting to have:

- **Languages spoken in a place.** Not only first language but also second and third.
- **Form of government.** Monarchy, dictatorship, parliament...
- **Climate.** Average temperature, average precipitation, number of natural catastrophes...
- **Human Development Index (HDI).** It is an indicator that aggregates life expectancy, education and income per capita, which are used to rank countries. It is used to measure countries development by the United Nations Development Program. [13]
- **Gross Domestic Product per capita made on basis of purchasing power parity** (or GDP (PPP) per capita) is the value of goods and services produced within a nation in a given year, converted to U.S. dollars divided by the population and adjusted for differences in the cost of living in different countries. [14]
- **Country freedom** according to the Freedom in the World Report, which is a yearly survey and report made by the non-governmental organization Freedom House that measures, among others, the degree of political rights around the world. [15]
- **Peacefulness of a country** depending on if the country is in a war.

To choose which filters to include in the dashboard, a search in Wikidata is done to check which ones appear as properties on countries. Among those, two are discarded: languages spoken in a place, climate and peacefulness of countries.
• Wikidata do not store any climate values in its page. It would be possible to implement this filter by consulting other databases like climate agencies.

• Wikidata has information about the official languages spoken in a country. This data is different from the languages spoken by its population as it excludes the languages taught at school.

• In Wikidata, ambiguous data are not correctly defined. Conflicts that do not have specific and objective starting and end dates are difficult to formalize and they are not present in Wikidata.

After this, to be able to implement the filters, those are formalized in terms of the data about migrations. The definitions of the filters are:

• **Filter by Human Development Index.** Movements such that HDI of the origin is less than 0.50 and the HDI of the destination is higher than 0.75.

• **Filter by GDP (PPP) per capita.** Movements such that GDP (PPP) per capita value of the origin is less than the value on destination.

• **Filter by country freedom.** Movements such that origin is a non-free country and the destination is a free country.

The filter “form of government” is finally discarded given the complexity of formalizing it because of the numerous forms of governments around the world and their classification.

### 4.2 Top-level design

To make the data to flow through the system from the beginning (migration data from Outag and data about countries from Wikipedia) to the end (the dashboard) with the inputs from the user, one more element is required: a way to link the places contained in Outag data (strings) with Wikidata concepts (URIs). To do this, an “entity recognition” module is placed in the system (see figure 4.1).
In addition, the link between Ontag strings and Wikidata URIs is stored in a database\(^1\) (accessed via the “psql” module); the user inputs —year and filter— are grouped into a “query” module which is the responsible of reading the data from the database given the user inputs; finally, the dashboard is divided into web components. This data flow leads into a top level architecture of the system (figure 4.2), where the modules are grouped into different layers.

---

\(^1\)The database chosen for this Project is PostgreSQL, a relational database. However, given that the data handled in the prototype is stored in a single table, there choice has no implications.
data access layer only responsibility is to access —write and read— to external resources (Wikidata and psql) and the logic of operating with the data is handled by the other layers of the application. Further details of testing are in section 5.1.

4.3 Design and implementation

Entity recognition

The Entity recognition module has two functions: (a) a **recognition function** that transforms strings like “Paris” into concepts, and, (b) an **insertion function** that inserts the migration data with the places converted into URIs into the database. The latter function only calls the psql module in the data access layer and handle the possible errors on insertion.

The recognition function is more complex and its sequence is shown is the following (also shown in figure 4.3):

1. The recognition function receives the input as a string.
2. It calls the `search()` function in the Wikidata module which searches the concept using the Wikidata search API. The query returns a list of URIs matching the string.
3. For each concept of the list, the `getType()` function is called which performs a query to Wikidata to get the type of the concept, specifically to guess if the concept is instance of Place or any subtype of Place.
4. Discard all the concepts that are not places and return the first element of the list.

---

3See https://www.wikidata.org/w/api.php?action=help&modules=wbsearchentities
This recognition function is a very elementary implementation of a string-to-concept function. Its based on the assumption that the list of entities returned by the Wikidata search API is ordered being the first element the closest to the search string.

The function would fail if the concept to be matched is not present in Wikidata, the concept is not correctly classified as “Place” and also if the search is performed for places with homonyms. It is also ignoring the context of the word. Covering all this means an advanced implementation of a Natural Language Processing function which is out of the scope of this Project.

Query

The query function returns the movements contained in the database giving a year and a filter. Its sequence is described below and shown in figure 4.4.

Figure 4.4: Sequence of query function

1. The function receives two inputs: year and filter.
2. It calls the psql module which performs a query in the database to get the movements that happened in the input year. The psql module return a list of movements.
3. The query function calls the getCountryData() function from the Wiki-data module to get the data of a specific country (its Gross Domestic Product per capita based on PPP, its Human Development Index and whether the country is free or not).
4. The query function filter the movements having the actual filter input and the data returned by the previous step.
5. The query function returns the filtered movements.
To enhance the performance of the system, the `getCountryData()` function called here is a good place to put a cache of the countries data retrieved from Wikidata.

**Frontend components**

The front-end part of the Web Interface is a component tree formed by several components:

- **Dashboard** is the root component. This stores the internal state of all the application and also makes queries to the back-end.

- **Map** shows a graphic representation of the data (a list of origin-destination-amount tuples).
  
  This component could have different children depending on how to represent the data. If the children needs a specific input, the transformation from the origin-destination-amount triple to that specific input is done in this component.

  For example, in the prototype, it has a *Cloropeth* component which is a map in which areas are shaded in proportion to some measurement. In this case, the cloropeth has two colours (red and blue) where red means “country with people moving out” and blue “country with people moving in”. The more saturated red or blue is a country, the bigger the amount of the people moving in/out.

- **Date Picker** allows users to choose a date.

- **Filter Selector**. With this component users can choose between the filter options: “Human Development Index”, “Poor to rich” and “Non-free to free”.

Following the principle of single source of truth, the information that is relevant to one component is stored in its internal state. However, if that information affects to other components, it is stored in its common ancestor.

For example, the retrieved data from the backend is stored in **Dashboard**. The date chosen by the user is also stored in **Dashboard** because that data is needed in both **Map** and **Date Picker** components. However, the zoom level which is only relevant in the map, is stored in the **Map** component.

The figure 4.5 shows the steps taken by each component in the beginning, when the user access to the app. This example includes a Cloropeth component, which is a child of the Map component.
1. The Dashboard component performs a query to the backend, to the /movements endpoint to get all the movements done in 2016.

The query for that is

GET /movements?year=2016

2. The backend (the query module) responds with a list of all the movements done in 2016.

An example of response is the following, representing movements among Syria, Morocco, Spain and France.

```
[
    {origin: 'sy', destination: 'fr', amount: 10000},
    {origin: 'sy', destination: 'es', amount: 3000},
    {origin: 'mo', destination: 'es', amount: 12000},
    {origin: 'mo', destination: 'fr', amount: 5000},
    {origin: 'es', destination: 'fr', amount: 1000}
]
```

3. The Dashboard component pass the response to the Map component.

4. The Map component takes the response and adapts it to data that matches with the inputs of the actual map, in this case the Cloropeth component. The result of this conversion will be an array of countries and how much population they earn/loss due to the migrations:
{country: 'sy', amount: -13000},
{country: 'mo', amount: -15000},
{country: 'fr', amount: 16000},
{country: 'es', amount: 12000}
]

5. The Cloropeth component draws the map with the input of the previous step

The figure 4.6 shows the steps taken by each component when the user chooses a different year (they click on a year).

![Sequence diagram when user selects a date](image)

Figure 4.6: Sequence diagram when user selects a date

1. **A click is dispatched in the `<YearSelector>` component.** The `onClick` prop is called, which is actually a function passed by `Dashboard`.

2. **The `<Dashboard>` component** check if it has the data of the chosen year, stored.
   - If they are stored, steps 3 to 5 shown previously are taken. If not, all the steps described before are taken.

   Refer to Appendix A for the full API reference of all the modules of the system.
Chapter 5

Results

5.1 Testing

Given the specific constrains of the system (the type of data that is being managed and the user interaction with it), testing of the system is done by mixing idempotent tests and characterization testing. Not the parts of the system are being tested.

5.1.1 Idempotent testing

The goal of these tests are to ensure the good functioning of the system. These tests must not test any external services and as a rule, every time tests are run, they must give the same results.

Tested modules are the entity recognition module and the query module. Those modules depend on external elements (Wikidata and psql respectively). In the context of the tests, those external elements are mocked. This behaviour is easy to do since the program separated into layers as seen in the figure 4.1 of the section 4.2. The data access layer only purpose is to access to external data without doing any intermediate operation and it is easy to replace with a layer that simulates an external service for testing purposes.

This kind of tests are useful to test the algorithms chosen by the developer. Specially to detect the problems addressed previously (see section 3.2).

To test the integration of the system with the external services, instead of preparing a test suite comparing expected and returned results, an approach based on characterization testing is followed.

5.1.2 Characterization testing

Characterization testing is a technique that consists in two steps.

1. In a first step, the test run the functions to be tested and their results are saved in the system. Before saving the tests, the results should be
manually checkid and only saved if they are the expected ones. In case of having a version control system with the code of the program, the results are also part of it.

2. In a second step, the test run the functions again and this time the results are compared with the previously saved ones, raising errors if they are different.

In short, this means that the results of the functions run in the step one are the “expected” results for the second step. All validations are done in this second step.

If there is an error, the results should be checked manually to conclude that: (a) the returned results are not the expected ones so the error is correct or (b) the returned results are valid and the saved version must be updated for future testing.

These tests are slow to run because they make actual queries to the external services. Also, like the integration tests mentioned before, those tests can fail due to changes in the services (their API, the implementation) and other external causes (network loss, bad configuration, etc.).

The intention of these tests are not to ensure the good functioning of the system. The tests also do not detect any errors in the system which contradicts in some way the intention of any software testing.

Even having all the mentioned drawbacks, characterization testing is useful to –more or less– ensure that under certain circumstances the system behaves in the same way. It is also an approach to test that the external services have updated their data, a specific thing that is relevant in this Project.

5.2 User interaction

When the user enters to the system, they see the dashboard divided in three regions: a map in the center occupying almost all the screen; the filter selector in the right and a year selector in the bottom (See figure 5.1). By default the chosen year is “2017” and the selected filter is “all” meaning that the map is showing all the movements that happened during 2017.
Then, the user can choose another filter, for example, “from non-free to free” and the map would show only the movements that happened from non-free to free countries as shown in figure 5.2.

If the user chooses a “poor to rich” filter, it shows the movements that match with the GDP (PPP) per capita filter (figure 5.3). Some results may look strange since this filter shows movements from “poorer to richer countries” meaning that a movement from a “poor” country to a “not-so-poor” country is included in this filtering.
If the user chooses the “low HDI to high HDI” filter in year 2017 (figure 5.4), the map is completely blank because Wikidata does not offer any data about HDI in 2017. Notice that, if Wikidata gets an update and it include the HDI of the countries for 2017, the map would show the movements correctly without any human manipulation needed.

By choosing another year, for example 2014, (figure 5.5), the map shows the migrations that match with the filter criteria.
Figure 5.5: Map showing movements in 2014 from low HDI to high HDI countries

Finally, the user can click on a country to display only the movements from and to that country, for example, the United States (figure 5.6).

Figure 5.6: Map showing movements from and to the U.S.
Chapter 6

Conclusions and Future work

6.1 Future work

This Project opens opportunities to expansions in various directions, some of them making small differences to it and others making more deep changes. Some of those directions are:

- Implement technical enhancements like different levels of cache or other performance improvements
- Change the components in the frontend to visualize data in different ways, maybe including different types of maps or graphs that are not maps at all.
- Add a layer of customization, letting the users to “modify” the criteria of the filters, for example letting them to decide what are the limits for HDI to be considered low or high.
- Use other properties found in Wikidata to make more filters. Formalize and implement the ones proposed in the design chapters.
- Use the properties in Wikidata in other ways like grouping countries by continent and be able to visualize not only country-to-country movements but also continent-to-continent or similar.
- Use different external sources: other general knowledge databases or other domain-specific databases to obtain other knowledge.
- Use other entity recognition system to link strings to concepts, or go further and not recognize only strings but images or another type of media.
6.2 Conclusions

This Project made possible to create graphics to be transformed automatically when data from external resources change. It also finished with a tool that could be useful for journalists.

Several technical and non-technical skills that are needed to make this type of project possible.

It also involves ethical and social issues that are not possible to solve from the Computer Science. As an example, depending on the “definition of country” that the developer chooses, it could result in different results and in sending wrong data to the users. This Project put an emphasis on the usage of structured data, but these type of issues and ambiguous definitions need to be taken into account carefully, specially in cases where definitions are ambiguous on purpose.

This Project is only a small approach into the topic. The Project solve a problem and in the journey of solving it, it discovers more problems, some of them with complex solutions and some of them unsolved.
Bibliography


Appendix A

API reference

A.1 Entity recognition

The API of the Entity recognition module allows users to introduce information about migration (the origin and destination of the migration, the amount of people that perform that migration and the date range when the migration has been done). Both origin and destination are provided as strings and stored as concepts. The transformation from string to concept is also performed by this module.

A.1.1 Global functions

Recognizer([options])

Returns an instance of Recognizer.

A.1.2 Recognizer instance methods

r.recognize(text, [type])

Perform a search of the text in Wikidata and retrieve an array of all the possible concepts that are close to that text. Parameters:

- String text. The text to look for.
- String type optional. Accepts the value “place”. If specified, it returns only the concepts that are actually places.

r.insert(data, [sources])

Insert an information about a migration into the database. Accepts two parameters:
• **Object data** is an object with 5 fields containing the information to be inserted. The fields are:
  
  – **String origin.** The origin of the migration
  
  – **String destination.** The destination of the migration
  
  – **Number amount.** The amount of people that perform the migration.
  
  – **Date startDate.** The initial date when the migration happened.
  
  – **Date endDate.** The end date when the migration happened.

• **Object sources optional** is an object with 5 fields containing references to the locations where the data are found. These fields should be compliant to the W3C Annotation standard and includes information referencing the URI, position and similar information. The 5 fields have the same names as the fields in the `data` parameter and each correspond to the source of each data.

### A.2 Frontend

The components in the frontend part are implemented using the React framework. React is a framework that allows the creation of web components in JavaScript. Each component have inputs (so called *props*) and an internal information (called *state*). All components are part of a Component Tree which is similar to the resulting DOM tree after rendering all the Component Tree.

Props are used to pass information through the tree downwards to its child(ren). Props can be also functions that act as callbacks. For example, a Button component may specify an onClick prop which is a function that is called when the button is clicked.

React also allows the creation of a *context*, information that is passed down to all the components of the Component Tree.

**Dashboard component**

Is the root component of all the application. It is *stateful*. It makes queries to the backend and stores internally the data needed across its children. It process the data and pass it to its children components. This component has no props.

This component, internally, stores the fetched data from the backend (i.e. vectors of movements) and has methods to fetch data and handle errors (e.g. connection errors).

This component also stores the user choices that are relevant across the entire application: the chosen year.

**Map component**

Render a Map given vectors of movements (origin-destination-amount tuples). This component has one prop:
• **Array movements** an array of objects with three fields:
  
  – **Object origin** a “Country” object representing the origin of the migration
  
  – **Object destination** a “Country” object representing the destination of the migration
  
  – **Number amount** the amount of people that move from “origin” to “destination”

The **Country** object is an object that represents a country. It has two fields:

• **String code** a two letters country code.

• **String name** the name of the country

The user can choose a single country. In that cases, only the movements from/to that country are shown. The chosen country is stored in this component. This component filter the movements according to this criteria.

**Cloropeth component**

Renders a choropleth map of the Earth. It paints countries using a bi-polar color progression. Countries with negative values are painted in blue and countries with positive values are painted in red. This component has one prop:

• **Array series** which is an array with the information of countries and a number to represent. Each element is an object with three fields:

  – **String code** is a two letters country code.
  
  – **String name** is the name of the country
  
  – **Number amount** is the amount that has to be represented in the map.

• **String selectedCountry**. If specified, this country is “highlighted”.

• **Function onSelect**. This function is called when a country is clicked on the map. The function should has one argument

  – **String country**. The country code of the clicked one.

This component is also stateful. It fetches GeoJSON data from an external site to get the polygons of the shape of the World map and save it as internal state. This operation is done only the first time the component is rendered. In this way, no more HTTP requests are necessary even if the props change.

**Table component**

Renders a table of countries and a number associated to each country. It is a table version of the cloropeth. It’s mainly created for debugging purposes. It has the same props as the Cloropeth component.
RangeSelector component

Renders a date selector. The user can choose what year to represent. Props:

- Function `onChange(selectedYear)`. This function is called when the user chooses a different year. The function has one argument:
  - `Number selectedYear`. The selected year in 4-digits format.

A.3 Query

It is an HTTP method under the `GET /movements` endpoint. Path parameters:

- `year`. Four-digits year. Returns the data only of a certain year period.
- `Optional filter`. If specified, return only the data that satisfies the filter. It accepts the values `hdi`, `gdppp` and `free`.

It returns an array of `movements`, where a movement is an object with three fields: origin, destination and amount.

The Query module perform the following operations

1. Reads, from the internal database, the migrations happened in the specified year.
2. Makes a query to Wikidata in order to get the places where the filter match, e.g. a list of origin countries and a list of destination countries.
3. From the results read from the database, filter the results (e.g. maintain in the list if the migration is from a country included in the list of origin countries).
4. Aggregate the results.

In all the steps, there are some edge-cases and non correct outputs that may happen depending on the quality of the data. This is discussed in Section 3.2.

A.3.1 Example

Get the migrations in 2016:

`GET /movements?year=2016`

Returns

```
[
  {origin: 'es', destination: 'fr', amount: 10000},
  {origin: 'mo', destination: 'es', amount: 15000},
  {origin: 'mo', destination: 'fr', amount: 8000},
  {origin: 'sy', destination: 'fr', amount: 20000},
  {origin: 'sy', destination: 'es', amount: 3000}
]
```