Theme Park Ride Accidents - RDF & SPARQL Analysis

Project Report

Bianchi Edoardo - 20740

September 1, 2022

Semantic Technologies - Code: 73063

Contents

1 Introduction

There are thousands of amusement parks around the world that welcome millions of visitors each year. Children, families, and teenagers are ready to spend days of adrenaline and fun. Unfortunately, accidents sometimes occur. This fact raises some questions: Are amusement parks safe? Which rides are the most accident-prone? What accidents happen most often? At what time of year are accidents most common?

Possible answers can be found by analyzing data on accidents in theme parks. In this project, I will analyze RDF data on accidents using different tools. The final result is presented to the user as an interactive web application.

A running version of the final application is hosted on my Huggingface Space at:
https://huggingface.co/spaces/EdBianchi/ThemeParksAccidents_RDF-SPARQL

The repository of the project can be found on GitHub at:
https://github.com/EdoWhite/ThemeParkAccidents_RDF-SPARQL.

2 Application Domain & Motivations

Amusement parks are all over the world, and more or less serious accidents occur every year. A semantic web and linked data approach could be used to prevent and inform.

Imagine connecting accident data from all amusement parks around the world: we could get accurate and precise information that could be used to make parks safer. Not only that: thanks to the properties of technologies such as RDF, it is possible to integrate other accident-related data, such as ride popularity information (cycles of use), ride technical specs, and accurate hospital records of people involved in accidents.

The final result would lead to a data integration that allows accident-related information to be obtained and visualized with ease. This information could also be linked to national services dealing with safety in theme parks. I believe that such a system could be of value to both parks and visitors.

3 Data Source, Technologies, and Architecture

3.1 Data Source Description

The dataset used in this project comes from *Saferparks*. Saferparks is an organization that aims to limit and prevent accidents in theme parks. Although this organization is no longer research-active, it continues to share data collected over the years. Specifically, the dataset used for this project contains data from accidents that occurred between 2010 and 2017, in parks across the United States. Accident data contains information on accident categories, people involved, and rides involved. Spatial and temporal data are also present.

The dataset is available in CSV or Excel format and can be found on the *Saferparks Dataset* page. This dataset will be split and modified so that it can be loaded into a Third Normal Form (3FN) database and converted into the RDF format.

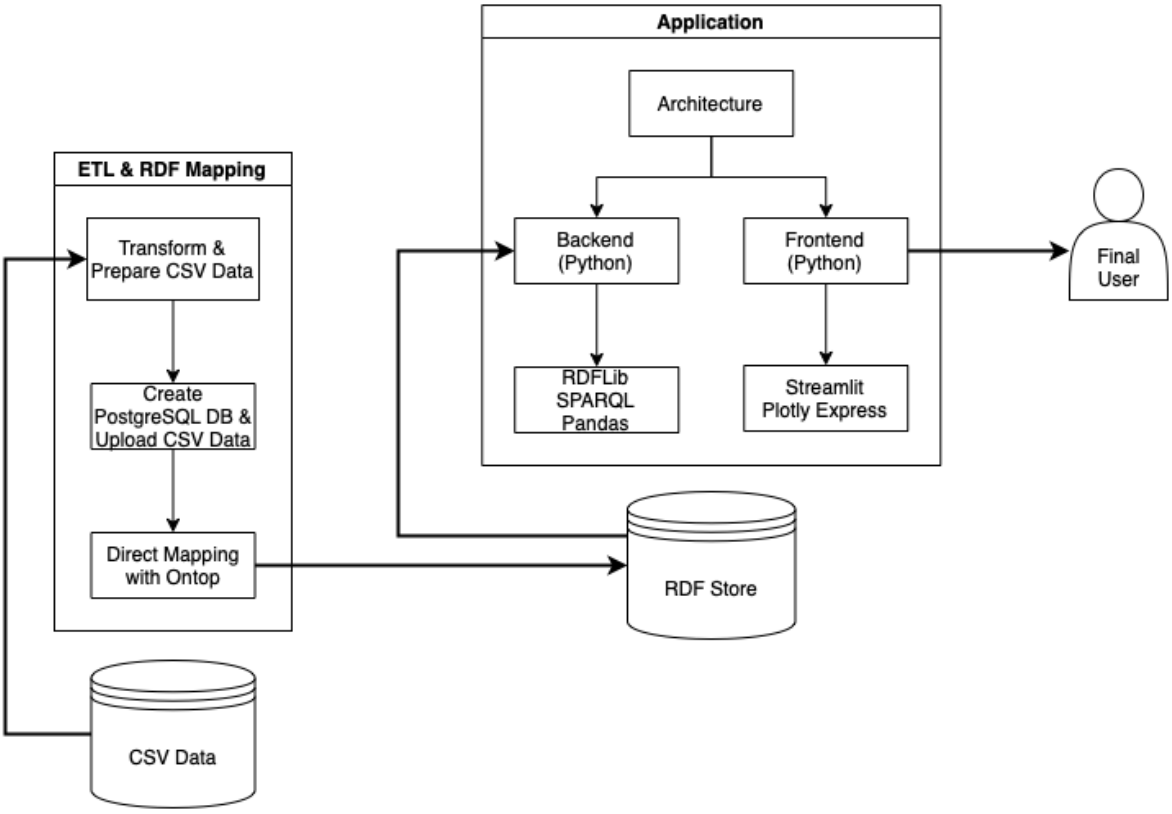

Figure 1: Architecture of the system. Image by the author.

3.2 Technologies and Overall Architecture

In this subsection I present the technologies used to develop the system and the general architecture (represented in figure 1). In particular, the following tools and frameworks are used:

- ***RDFLib***: This Python library is designed to work with RDF. It provides the ability to load, save, and navigate RDF graphs. It is also possible to query graphs with SPARQL.
- ***SPARQL***: SPARQL is the query language used to query the RDF graph.
- ***Ontop***: Ontop is a tool used to map a relational database source to an RDF data source.
- ***Streamlit***: Streamlit is an open-source Python framework used to develop simple but effective web apps. Streamlit provides a set of components that can be placed on the pages of the application. These components range from titles to user inputs, dataframes, and graphs.

  Creating an app with Streamlit is easy and doesn't require any front-end experience. The backend of the app is written in plain Python, and the frontend is generated using Streamlit-specific functions. These functions render elements (title, text, charts, results, ...) on the app page(s).
- ***Plotly Express***: This Python library is used to create figures and visualizations.
- ***Huggingface***: Hugging Face is an AI-oriented community. On their website, they host tons of models, datasets, and spaces. Users can create spaces for free to host and share their applications. Apps to share must be developed using Gradio or Streamlit.
- ***PostgreSQL + pgAdmin 4***: Tools used to create & operate the database, and load data from CSV tables.

All these components are used together and are indispensable for the correct functioning of the application.

4 Development Stages

4.1 Overview

Figure 1 represents the architecture of the system, which reflects the development stages. In particular, building the application required the following steps:

1. Get the dataset about accidents from Saferparks.
2. Prepare the data: selecting the attributes of interest, split the dataset, export transformed CSV tables.
3. Create a PostgreSQL database and upload the data from the exported tables.
4. Use the Ontop tool to map the relational database and materialize the final RDF dataset.
5. Use SPARQL to query the RDF and plot the results on a web app made with Streamlit.

Steps 2 and 3 are referred as Extract, Transform, Load (ETL). All the steps are briefly explained in the following sections.

4.2 Data Preparation

To prepare the data, I used a *Jupyter* Notebook. In particular, I used the *Pandas* library to split the dataset into different tables while maintaining referential constraints. The purpose is to export CSV tables ready to be loaded into a PostgreSQL database. The Third Normal Form (3FN) should be respected. The notebook with all the operations is contained in the project repository at *data/data_preparation.ipynb*. The tables exported to CSV are also present in the *data/prepared_data* folder.

4.3 Database Creation & Population

The mapping functionality provided by the Ontop tool requires data to be in a relational database. For this purpose, I created a custom PostgreSQL database with all the required

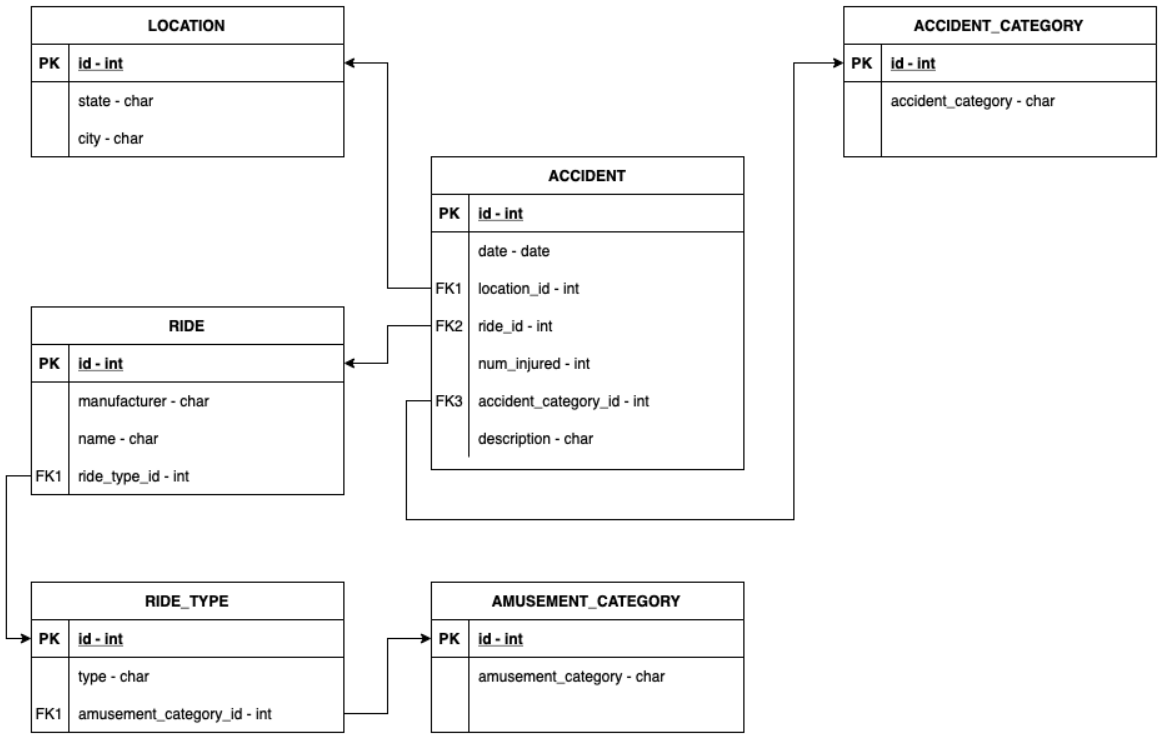

Figure 2: The database schema. Image by the author.

tables and constraints. Figure 2 represents the database schema. Database tables are created using the SQL DDL language. As an example, the following code represents the creation of the accident table:

```
CREATE TABLE accident (
    id INTEGER PRIMARY KEY,
    date DATE,
    num_injured INTEGER,
    description CHARACTER VARYING,
    location_id INTEGER,
    accident_category_id INTEGER,
    ride_id INTEGER,
    FOREIGN KEY (location_id) REFERENCES location(id),
    FOREIGN KEY (acc_cat_id) REFERENCES accident_cat(id),
    FOREIGN KEY (ride_id) REFERENCES ride(id)
)
```

After this step, I used the PSQL tool to upload the CSV tables exported during the *Data Preparation* phase into the database. To be uploaded, the CSV tables and the database tables must share the same names and attributes. The following general command is used for loading the data:

```
COPY <table_name> FROM '<csv absolute path>' DELIMITER ',' CSV HEADER
```

For example:

```
COPY accident_category FROM '/User/Data/acc_cat.csv' DELIMITER ',' CSV HEADER
```

At the end of this step, the database is fully populated and all the constraints are respected.

4.4 Mapping & Materializing the RDF

After loading the data into the dataset, I used the Ontop tool to generate the RDF dataset. In particular, the conversion is done with an R2RML mapping, that specifies a mapping from a relational database to RDF (RDB to RDF). This operation can be done from the terminal:

```
$ ./ontop bootstrap -m ./RDF/mapping.obda -p ./basic.properties
      -t ./RDF/ontology.ttl -b http://example.org/

$ ./ontop mapping to-r2rml -p ./basic.properties
      -i ./RDF/mapping.obda -o ./RDF/mapping.ttl

$ ./ontop materialize -m ./RDF/mapping.ttl -p ./basic.properties
      -t ./RDF/ontology.ttl -o ./RDF/rdf-dataset.ttl -f turtle
```

The first command generate an OBDA (Ontology Based Data Access) file and an ontology. The second command takes as input the OBDA file and produces a mapping. The third command uses the mapping and the ontology to output the final RDF data. To summarize, the result of this step includes an ontology, a mapping file and the RDF data.

4.5 Querying & Presenting Results

The final application is built with Streamlit. The layout includes several containers, each container consists of a chart and a section that allows you to see the query code. More details and images about the GUI are contained in the *Functionalities* section.

Queries are written in SPARQL and executed using the RDFLib library. The result of the queries is converted to a Pandas dataframe, which is later used for displaying the results in the form of charts. The charts are built with the Plotly Express library.

5 Using the System

In this subsection, I describe the usage of the system. A running version of the application is hosted on my Huggingface at:
https://huggingface.co/spaces/EdBianchi/ThemeParksAccidents_RDF-SPARQL.

Alternatively, one can run the application locally. In this case, please note that the proposed Command Line Interface (CLI) commands may vary depending on the platform used. It is also important to point out that the platform on which this application was developed and tested is *osx-arm64* and it may be necessary to install or remove additional libraries to make the application work on other platforms.

To run the application locally, one needs first to clone the GitLab/GitHub repository and make sure to have all the requirements necessary to use the application. To simplify this operation one can create a new virtual environment and install the required libraries. The *conda_env.txt* file contains all the required libraries in a format that is ready to install in an Anaconda Virtual Environment.

This operation can be done from the (CLI) with the following command:

```
# navigate into the cloned folder
$ conda create --name my-env --file conda_env.txt
$ conda activate my-env
```

Next, launch the actual application using Streamlit. This can be done easily with:

```
(my-env) $ streamlit run app.py
```

If the command is executed correctly, it will open a browser page with the application running. To stop the application, simply close the terminal running Streamlit.

6 Functionalities

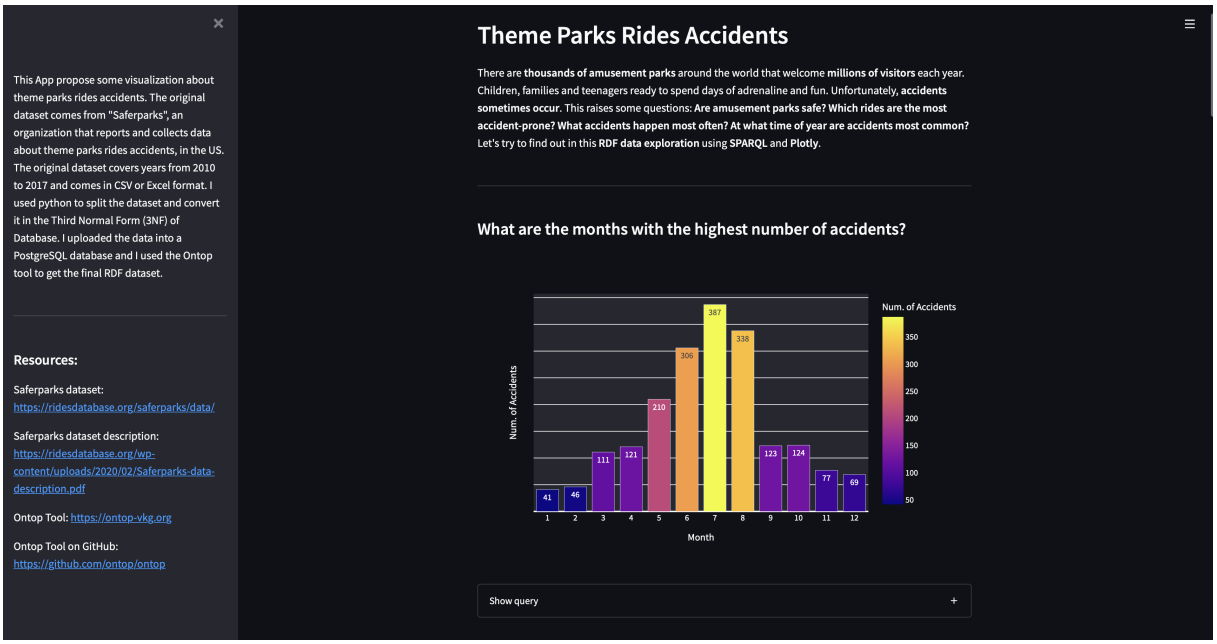

Figure 3: A part of the final app to the user. Image by the author.

The system is used to compute analytical queries on the data. The results are proposed to the final user as visualizations of different kinds, mainly bar charts, pie charts, and treemaps.

Visualizations can be dynamically zoomed in, zoomed out, and navigated. Below each graph is a box that allows the user to view the query executed. The query box is represented in figure 4. At the bottom of the page, a section allows the user to write and execute custom queries. The application is accessed from a web browser.

The basic structure has eight sections, six of which are visualizations. Specifically:

- **Accident Monthly Analysis**: Represents the months with the highest number of accidents.
- **Accidents Geographic Analysis**: Represents states and cities where more accidents occur.

Figure 4: A visualization and the related query box. Image by the author.

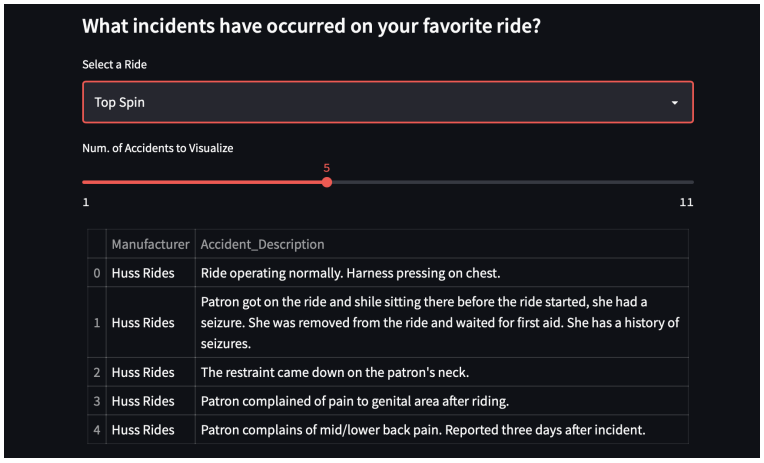

Figure 5: The parametric query, with select box input and slider. Image by the author.

- **Description of Accidents occurred on a Particular Ride**: This parametric query allows the user to select a ride from a list of rides and returns a variable number of accidents occurred on that ride. The number of results is selected using a slider. The parametric query is represented in figure 5.
- **Most Common Accidents**: Represents the most common types of accidents

- **Most Dangerous Ride Categories**: Represents the categories of rides on which the most accidents occur.
- **Most Dangerous Ride Types**: Represents the types of rides on which the most accidents occur.
- **People Involved in Accidents**: Represents the number of injured people that are generally involved in an accident.

The last section, represented in figure 6, allows the user to type in a personalized query and see the results in a tabular format.

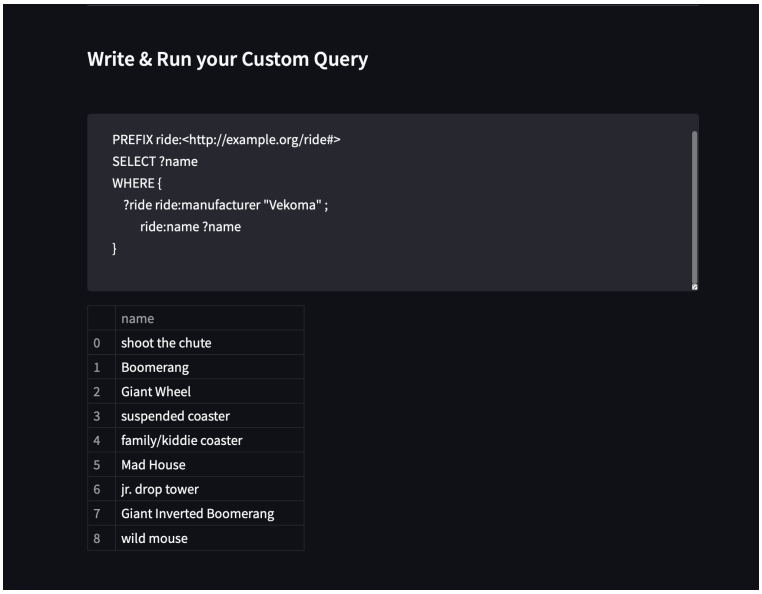

Figure 6: The section of the app that allow users to write custom queries. Image by the author.

7 Lessons Learned and Conclusions

The vision proposed by the Semantic Web and the use of Linked Data allows for merging and integrating data, thus giving access to a large amount of information. All this requires open standards and interoperability, properties not easy to achieve. The difficulty when integrating data resides in the fact that databases rarely adopt the same attributes to represent the same objects.

As a result, an object is represented differently in different databases.

To overcome this problem we need to add new definitions to data. This new information is known as vocabulary (or ontology). Consequently, standardization of vocabularies is also a practical difficulty.

The benefits, however, concern greater efficiency in terms of search: the more data connected, the more rich results are obtained. In addition, the schema flexibility of RDF is an advantage over relational databases, where a change to the schema can pose difficulties.

That said, relational databases are widely used today. Therefore, to take advantage of the semantic web technologies, we can use tools to automatically convert relational data into RDF data with the respective ontologies. An example is Ontop, used in this project.

Regarding the system I developed, I tried to use the different tools and technologies introduced during the course. Although the application itself is not particularly complex, I tried to take care of all the details, especially those concerning the graphical interface.

Further steps may include, for example, an integration with accident data from other nations to create one unique database. In addition, hospital accident records and park attendance data could be connected to the source. These data together would provide a single, comprehensive view that could be used to prevent further accidents.

References

Free University of Bolzano/Bozen. *Ontop*. URL: https://ontop-vkg.org.
Huggingface Team. *Huggingface*. URL: https://huggingface.co.
Jupyter Team. *Jupyter*. URL: https://jupyter.org.
Pandas Team. *Pandas*. URL: https://pandas.pydata.org.
pgAdmin Team. *pgAdmin 4*. URL: https://www.pgadmin.org.
Plotly Team. *Plotly Express*. URL: https://plotly.com/python/plotly-express/.
PostgreSQL Team. *PostgreSQL*. URL: https://www.postgresql.org.
RDFLib Team. *RDFLib*. URL: https://rdflib.readthedocs.io/en/stable/.
Rides Database. *Saferparks*. URL: https://ridesdatabase.org/saferparks.
– *Saferparks Dataset*. URL: https://ridesdatabase.org/saferparks/data/.
Streamlit Team. *Streamlit*. URL: https://streamlit.io.
W3C. *SPARQL*. URL: https://www.w3.org/TR/rdf-sparql-query/.