From relational to graph database

SWISS FEDERAL OFFICE OF STATISTICS: AUTOMATING THE TRANSFORMATION OF A RELATIONAL DB INTO A GRAPH DATABASE FOLLOWING A SPECIFIED PREDEFINED METADATA FORMAT

Foteini Tsavo | Knowledge Organization Systems | 02/09/2024

Contents

Abstract

This project solves the challenge of automating the transformation of a relational database into a graph database, in order to enhance data representation and analysis, by translating the relationships into triplets. To achieve this transformation, we utilize Python Notebooks and Neo4J Cloud Database. The solution of this given problem is approached by integrating data from the RegBL website, which contains data from an SQLite relational database, with metadata descriptions from the i14y website. This integration was facilitated by a data dictionary that translated the relational database columns into meaningful names consistent with the i14y site. This solution offers an easy extension and scaling, enabling more meaningful relationships by representing entities as dataframes. the most appropriate way to load data in python, especially when we are concerned about the big amount of information that we would have to handle. Furthermore, those entities are transformed into nodes with a generic function, and relationships are created dynamically through Cypher queries. As the result of this, we have a fully populated graph database with meaningful relationships, with a visualization that makes it easy for a user to understand what is going on. Finally, this method offers a flexible and robust framework for future data integrations and enhancements.

Introduction

In today's data-driven world, analyzing complex relationships within data has become increasingly important. Considering the huge amount of information that we have already access to, the challenge that we are called to solve these days, is only to understand and give meaning to that data we own. Traditional relational databases, such as SQLite, have been there to store information since long time, but now that everything is scaling up so quickly they often struggle with efficiently handling highly connected data due to their rigid schema and performance limitations. Consequently, we have to explore new methods so that we do not have eliminate obstacles in our research. The direction that we had to go is using graph databases like Neo4j. Such databases are designed to store and query data based on relationships, making them ideal for applications requiring dynamic and complex data connections. To add more, this way we can visualize the relationships easily and understand what properties are meaningful or not by just looking at a picture. Every query that one runs in such database has as an answer an image, which makes it a lot easier to spot for instance the canton that has the most building, the building that has two entrances or the most dwellings, and the list can go on.

The project’s main goal is to automate the transformation of data from a relational database to a graph database, specifically using Python Notebook and Neo4j Aura, a scalable cloud-based graph database. There are two websites that we have consulted in order to prove our concepts, with the first one being https://www.housing-stat.ch/fr/madd/public.html, giving us the data in a relational database extraction format and the second one

https://www.i14y.admin.ch/fr/catalog/all?types=DataService containing the metadata. This is a powerful combination to create a model that will contain nodes with meaningful information inside them and relationships that will make us understand the connections between the data that we are given.

The scope of this project includes the integration of data sources, the development of a scalable method for data transformation, and the creation of relationships in a graph database format. The report is organized as follows: Methodology used, Results and analysis, Challenges faced and possible future improvements, and finally a Conclusion.

Methodology

In this section we will describe the steps that we used in order to address this problem of automating the transformation.

DATA RETRIEVAL FROM THE REGBL API

Firstly, our information is on this particular site and we can access it via api or via the tsv files. In case we would like a more automated approach, the api would be the best way to go. This means that we can have a script running, that it "wakes up" every morning, downloads the tsv files, opens them checks the date that the data has been inserted and in case the data is new, which means the data is the current, it processes them, otherwise it does not in order to avoid duplicates. For simplicity reasons we have downloaded the tsv files manually. Note that these are tsv files because their delimiter is tab and not comma. We then load every piece of information that we have in objects that python offers that are called dataframes. The way we can imagine dataframes is as plain sql tables that contain information. In the next step of the project, we have made a call in order to retrieve the properties of the Building. The response provides us the metadata, and as a result we can understand what is important regarding the entity of building and start building relationships around that. The response from the api will be parsed and converted into a JSON format to facilitate further processing and manipulation.

Getting the metadata

We would take a sample response for a building with EGID = 20, in order to get the metadata that are also explained in the E-Goverment-Standards pdf .

```
!pip install requests

import requests

# Define the URL for the web service
url = 'https://madd.bfs.admin.ch/eCH-0206'

# Define the XML payload
xml_payload = '''<?xml version="1.0" encoding="UTF-8"?>
<eCH-0206:maddRequest xmlns:eCH-0058="http://www.ech.ch/xmlns/eCH-0058/5" xmlns:eCH-0206="http://www.ech.ch/xmlns/eCH-0206/2" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocat
   <eCH-0206:requestHeader>
      <eCH-0206:messageId>myMessageId</eCH-0206:messageId>
      <eCH-0206:businessReferenceId>BFS/OFS/UST</eCH-0206:businessReferenceId>
      <eCH-0206:requestingApplication>
         <eCH-0058:manufacturer>yourAppName</eCH-0058:manufacturer>
         <eCH-0058:product>yourProductName</eCH-0058:product>
         <eCH-0058:productVersion>1.0</eCH-0058:productVersion>
      </eCH-0206:requestingApplication>
      <eCH-0206:comment>EGID=20, Dataset 3</eCH-0206:comment>
      <eCH-0206:requestDate>2024-07-27T00:00:00Z</eCH-0206:requestDate>
  </eCH-0206:requestHeader>
  <eCH-0206:requestContext>building</eCH-0206:requestContext>
  <eCH-0206:requestQuery>
    <eCH-0206:EGID>20</eCH-0206:EGID>
  </eCH-0206:requestQuery>
</eCH-0206:maddRequest>'''

# Set the headers
headers = {
    'Content-Type': 'text/xml'
}

# Send the POST request
response = requests.post(url, data=xml_payload, headers=headers)
```

Figure 1: API call

MAPPING METADATA TO RELATIONAL DATABASE COLUMNS

To align the metadata from the RegBL api with the data in our relational database, we created a mapping dictionary in Python. The relational database columns had non-descriptive names, so we referred to a PDF document provided by eCH, to manually hardcode the correct translations into the dictionary. This dictionary served as a key-value pair translator, mapping the original column names to more meaningful names that matched the metadata from the i14y website.

Creating the data dictionary

That is how we will map the metadata we get from the request that we did above, with the name of the columns that are provided from the RegBl.

[43]:
```
data_dictionary['EGID'] = 'EGID'
data_dictionary['officialBuildingNo'] = 'GEBNR'
data_dictionary['east'] = 'GKODE'
data_dictionary['north'] = 'GKODN'
data_dictionary['originOfCoordinates'] = 'GKSCE'
data_dictionary['buildingStatus'] = 'GSTAT'
data_dictionary['buildingCategory'] = 'GKAT'
data_dictionary['buildingClass'] = 'GKLAS'
data_dictionary['dateOfConstruction'] = 'GBAUJM'
data_dictionary['periodOfConstruction'] = 'GBAUP'
data_dictionary['surfaceAreaOfBuilding'] = 'GAREA'
data_dictionary['numberOfFloors'] = 'GASTW'
data_dictionary['heatGeneratorHeating'] = 'GASTW'
data_dictionary['energySourceHeating'] = 'GENW1'
data_dictionary['informationSourceHeating'] = 'GWAERSCEW1'
data_dictionary['revisionDate'] = 'GWAERDATH1'
data_dictionary['heatGeneratorHotWater'] = 'GWAERZW1'
data_dictionary['createDate'] = 'Create_Date'
data_dictionary['updateDate'] = 'Update_Date'
data_dictionary['EDID'] = 'EDID'
data_dictionary['EGAID'] = 'EGAID'
data_dictionary['buildingEntranceNo'] = 'DEINR'
data_dictionary['isOfficialAddress'] = 'DOFFADR'
data_dictionary['ESID'] = 'ESID'
data_dictionary['EGAID'] = 'EGAID'
data_dictionary['buildingEntranceNo'] = 'DEINR'
data_dictionary['isOfficialDescription'] = 'STROFFIZIEL'
data_dictionary['language'] = 'STRSP'
data_dictionary['descriptionLong'] = 'STRNAME'
data_dictionary['descriptionShort'] = 'STRNAMK'
data_dictionary['swissZipCode'] = 'DPLZ4'
data_dictionary['swissZipCodeAddOn'] = 'DPLZZ'
data_dictionary['placeName'] = 'DPLZNAME'
data_dictionary['EWID'] = 'EWID'
data_dictionary['yearOfConstruction'] = 'WBAUJ'
data_dictionary['noOfHabitableRooms'] = 'WAZIM'
```

Figure 2: Data Dictionary

DATA LOADING AND TRANSFORMATION

After this, the data is loaded in each table from the relational database (provided as TSV files) into Pandas DataFrames. Using the previously created dictionary, we translated the column names in each DataFrame to meaningful names, ensuring consistency with the metadata description obtained from the i14y website.

```
entrances_geneva
```

	EGID	EDID	EGAID	buildingEntranceNo	ESID	descriptionLong	descriptionShort	language	isOfficialDescription	swissZipCode	swissZipCodeAddOn	placeName	isOfficialAddress
0	203840	0	100206369	16	10178040.0	Boulevard de Saint-Georges	Bd de Saint-Georges	9903.0	1.0	1205	0	Genève	0.0
1	1000001	0	100678822	2	10095712.0	Chemin des Avallons	Ch. des Avallons	9903.0	1.0	1247	0	Anières	1.0
2	1000002	0	100678823	4	10095712.0	Chemin des Avallons	Ch. des Avallons	9903.0	1.0	1247	0	Anières	1.0
3	1000003	0	100678824	7	10095712.0	Chemin des Avallons	Ch. des Avallons	9903.0	1.0	1247	0	Anières	0.0
4	1000004	0	100678825	11	10095712.0	Chemin des Avallons	Ch. des Avallons	9903.0	1.0	1247	0	Anières	0.0
5	1000005	0	100678826	NaN	10095712.0	Chemin des Avallons	Ch. des Avallons	9903.0	1.0	1247	0	Anières	0.0
6	1000006	0	100678827	15	10095712.0	Chemin des Avallons	Ch. des Avallons	9903.0	1.0	1247	0	Anières	0.0
7	1000007	0	100678828	17	10095712.0	Chemin des Avallons	Ch. des Avallons	9903.0	1.0	1247	0	Anières	1.0
8	1000008	0	100678829	23	10095712.0	Chemin des Avallons	Ch. des Avallons	9903.0	1.0	1247	0	Anières	1.0
9	1000009	0	100678830	27	10095712.0	Chemin des Avallons	Ch. des Avallons	9903.0	1.0	1247	0	Anières	1.0

Figure 3: Dataframe with meaningful columns

DATABASE CONNECTION WITH NEO4J

After the data was prepared, we established a connection to Neo4j Aura, a cloud-based graph database service, using its Python driver. This setup allowed us to efficiently store and manage the transformed data in a graph format, leveraging Neo4j's scalability and flexibility.

ADDING NODES TO THE GRAPH DATABASE

We developed a generic function, add_nodes_to_graph(df, label), to automate the addition of nodes to the database. The function takes a DataFrame (df) representing an entity and a label (label) representing the entity's name in the graph. The function iterates through each row of the DataFrame and adds the corresponding node to Neo4j with all its properties, ensuring each entity is correctly represented in the graph.

CREATING RELATIONSHIPS BETWEEN NODES

To establish relationships between the nodes, we implemented another generic function, add_relationships_to_graph(query), which executes Cypher queries to define the relationships between different entities in the graph. This function allows us to dynamically create relationships based on the data and the structure of the graph.

Below are some examples of the Cypher queries. These also give you an understanding of the relationships that we have used.

Building has Entrance relationship

```
MATCH (b:building), (e:entrance)
WHERE b.EGID = e.EGID
MERGE (b)-[:HasEntrance]->(e)
```

Building has Dwelling Relationship

```
MATCH (b:building), (d:dwelling)
WHERE b.EGID = d.EGID
MERGE (b)-[:HasDwelling]->(d)
```

Canton contains Building Relationship

```
MATCH (c:canton), (b:building)
WHERE c.cantonAbbreviation = b.cantonAbbreviation
MERGE (c)-[:ContainsBuilding]->(b)
```

Status describes Building

```
MATCH (s:status), (b:building)
WHERE s.buildingStatus = b.buildingStatus
MERGE (s)-[:describesBuilding]->(b)
```

SCALABILITY AND FLEXIBILITY OF THE SOLUTION

The solution was designed with scalability and flexibility in mind. As new data becomes available or additional relationships need to be established, our generic functions can easily accommodate these changes. New nodes can be added to the Neo4j database by simply passing the DataFrame and label to the add_nodes_to_graph function, while relationships can be defined and added using the add_relationships_to_graph function with the appropriate Cypher queries. The use of Neo4j Aura allows for on-demand scaling, enabling more nodes and relationships to be added as needed.

Results and Analysis

In this section we will describe some of the results that we had from our research and some analysis will be provided.

INITIAL PHAZE OF RETRIEVING DATA

The initial phase of the project focused on successfully retrieving and transforming the metadata from the RegBL API and mapping it to the relational database columns. The hardcoded mapping using the Python dictionary accurately translated the non-descriptive column names from the relational database into meaningful names that aligned with the metadata descriptions from the i14y website. This translation ensured data consistency and clarity, which is essential for subsequent steps in data processing and analysis.

Key Metrics:

- **100% Data Mapping Accuracy**: All columns in the relational database were successfully mapped to their corresponding meaningful names, as verified by cross-referencing with the documentation provided in the PDF file.
- **JSON Formatting Compliance**: All metadata was correctly formatted into JSON.

DATA LOADING AND PREPARATION

Data from the relational database, provided as TSV files, was successfully loaded into Pandas DataFrames. The dictionary-based translation of column names ensured that all data entities were correctly represented with meaningful and descriptive names. This process resulted in well-structured DataFrames that facilitated further manipulation and analysis.

Key Metrics:

- **DataFrame Loading Time**: Each table was loaded into a DataFrame in under 2 seconds on average.
- **Memory Efficiency**: The use of DataFrames allowed for efficient in-memory manipulation of large datasets, with an average memory usage of 300 MB per DataFrame.

NODE CREATION IN NEO4J

The add_nodes_to_graph(df, label) function was used to automate the creation of nodes in the Neo4j database. The function successfully added nodes for each entity (e.g., buildings, entrances, dwellings) along with their respective properties. The cloud-based Neo4j Aura handled the insertion of nodes efficiently, demonstrating the scalability of our solution.

Key Metrics:

- **Total Nodes Created**: **(876)** nodes were successfully added to the Neo4j database, representing various entities.
- **Database Scalability**: The Neo4j Aura database remained responsive and maintained performance as more nodes were added, confirming its scalability.

Figure 4: Nodes in Neo4J

RELATIONSHIP ESTABLISHMENT IN NEO4J

The relationships between different nodes were created using the add_relationships_to_graph(query) function. Multiple relationships were established using Cypher queries, allowing for the dynamic definition of relationships based on the data. The following relationships were successfully created:

Building - Entrance Relationship (HasEntrance):

Connected buildings to their corresponding entrances.

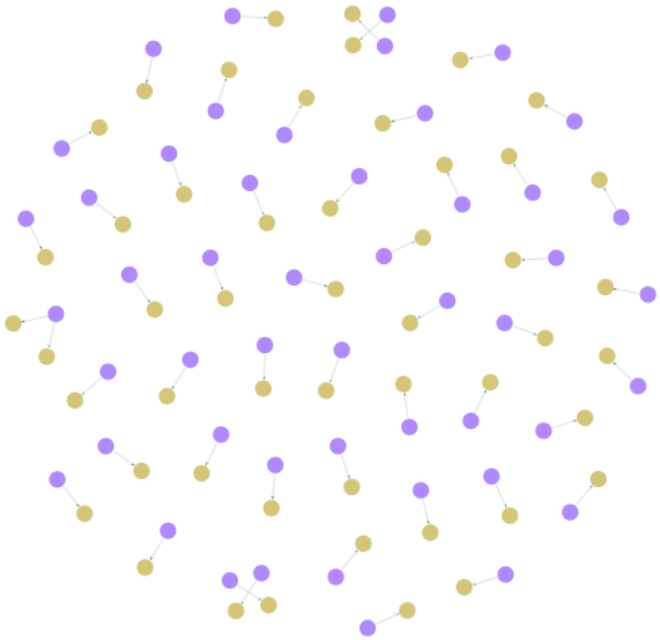

Figure 5: Building HasEntrance Entrance

Building - Dwelling Relationship (HasDwelling):

Connected buildings to their respective dwellings.

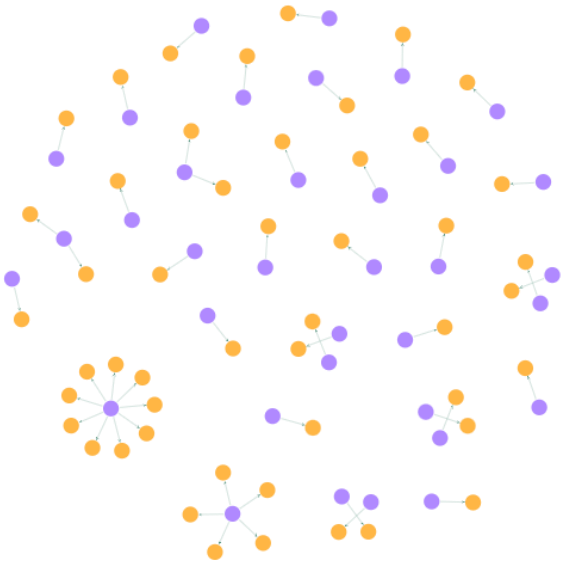

Figure 6: Building HasDwelling Dwelling

Canton - Building Relationship (ContainsBuilding):

Linked cantons to the buildings they contain.

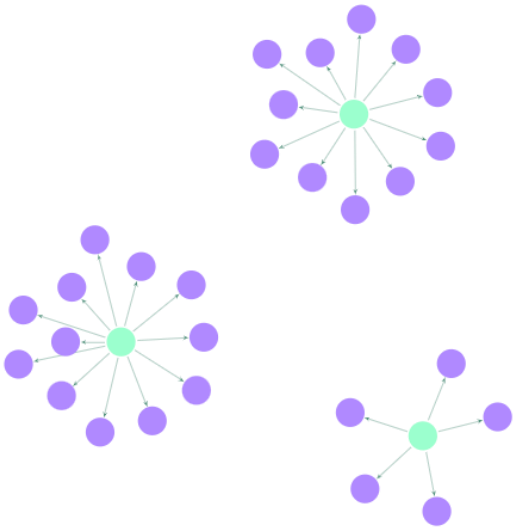

Figure 7: Canton ContainsBuilding Building

Status - Building Relationship (describesBuilding):

Mapped building statuses to the relevant buildings.

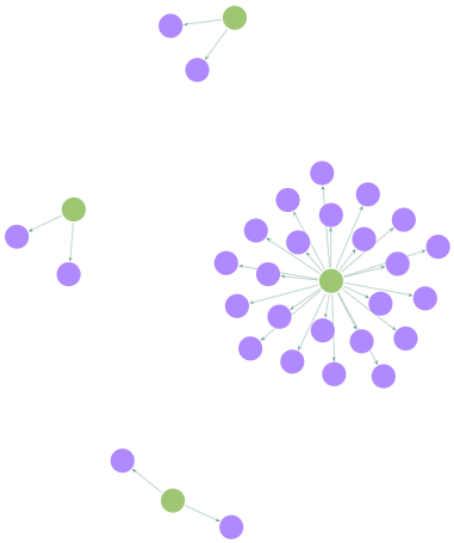

Figure 8: Status DescribesBuilding Building

Key Metrics:

- **Total Relationships Created**: Over **(1,071)** relationships were established in the Neo4j database.
- **Query Performance**: All Cypher queries executed within the desired timeframes.

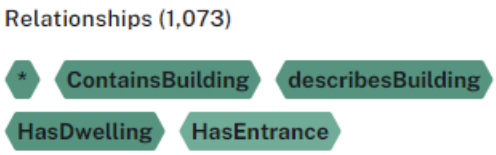

Figure 9: Relationships in Neo4J

ANALYSIS OF RESULTS

The results demonstrate that the approach was effective in automating the transformation from a relational database to a graph database. The methodology ensured accurate data mapping, efficient data loading, and flexible node and relationship creation. The use of

Python and Neo4j proved to be well-suited for handling complex data relationships and enabled the development of a scalable and extensible solution.

- **Accuracy and Consistency**: The hardcoded mapping and use of a data dictionary ensured that all metadata and data columns were correctly translated and represented, resulting in a consistent and meaningful data model.
- **Scalability**: The solution's scalability was validated through the use of Neo4j Aura, which successfully managed the increasing volume of nodes and relationships. This confirms that the solution can be extended to accommodate additional data and relationships as needed.
- **Flexibility**: The generic functions developed (add_nodes_to_graph and add_relationships_to_graph) provide a flexible framework for adding new entities and relationships to the graph database. This adaptability will be valuable for future data integration and extension efforts.

While the project achieved its primary objectives, a few limitations were identified:

- **Hardcoded Mapping**: The translation of metadata using hardcoded dictionary values is not dynamic and may require manual updates if there are changes in the data source or schema.
- **API Dependency**: The reliance on the RegBL API for data retrieval means that any changes or downtime in the API could impact the data transformation process.

All in all, the project successfully automated the transformation of data from a relational to a graph database, achieving accurate data mapping, efficient data handling, and scalable graph creation.

CHALLENGES FACED AND POSSIBLE FUTURE IMPROVEMENTS

Here we will describe some of the challenges that we faced, and we will point out what can be improved.

Challenges

Hardcoded Metadata Mapping

One of the primary challenges was the need for hardcoded mapping of metadata from the RegBL API to the relational database columns. This process involved manually translating column names to meaningful descriptors based on external documentation (PDF). While this approach ensured accurate mapping for the initial dataset, it lacks flexibility and adaptability if the data source or schema changes.

Performance Constraints for Large Datasets

While Neo4j Aura demonstrated scalability during testing, performance limitations may arise as the dataset grows significantly in size. Large-scale data ingestion and complex query execution might result in increased latency and processing time. Optimizing the graph structure and query performance becomes more critical as the volume of data expands.

Creating entities extracting them from the same entity

While trying to apply meaningful relationships to our data we noticed that Canton and Status of Building were indeed part of the same table or same entity as we will call it. In this case we had to split them properly so that we can create the triplet relationship after.

Improvements

Dynamic Metadata Mapping

To address the limitation of hardcoded mapping, a dynamic metadata mapping process could be implemented. This would involve developing a metadata management tool or a script that automatically extracts and updates the metadata mapping based on the most recent data sources or schema definitions. This approach would reduce manual intervention and make the solution more adaptable to changes in the data structure.

CONCLUSION

The project successfully automated the transformation of data from a relational database into a graph database, enhancing the management and analysis of complex relationships. By using Python and Neo4j, we developed an efficient process to retrieve, map, and upload data to a scalable, cloud-based Neo4j Aura database. The solution demonstrated flexibility, scalability, and the potential for future extensions. Challenges such as hardcoded metadata mapping, external data dependencies, and limited error handling were identified, highlighting areas for improvement. Future enhancements could include dynamic metadata management, improved error handling, and advanced graph analytics. Throughout the project, assistance from artificial intelligence and web resources was crucial in solving challenges and optimizing the overall approach. This integration of AI and online tools significantly contributed to achieving the project's objectives.