Project 1b: MONDIAL to Neo4J

Name: Timothy Zilcher
Matrikelnummer: 7004410

Documentation of the designed Schema for the MONDIAL Database in Neo4J

I. Introduction

What is Neo4J?

Neo4j is a graph database that structures data with nodes and relationships. Differently to relational databases like PostgreSQL which store data in tables, Neo4j uses a network structure to represent data.
The nodes in the graph represent entities which are tagged with labels, such as a person, product, or city, that define their role in the domain. Additionally, they can have any number of key-value pairs as properties. Relationships connect nodes with named and directed links while being also able to carry their own properties [1].

Advantages of Neo4J

Neo4J and the associated query language cypher have several advantages that make it an attractive choice for storing and analyzing networked data.
A key advantage of Neo4J is the efficient processing of networked data. While relational databases often have performance issues with complex joins and nested queries, Neo4J can handle these queries more easily with the help of direct node links.
For merging data Neo4J uses relationships that are represented directly by links between nodes. This replaces the issue of relational databases time-consuming table joins and significantly increases query speed and gives additionally a better overview.
Another advantage is the flexibility of the data model. Relational databases require a rigid schema structure while Neo4J is schema-free or schema-light. This means that new nodes or relationship types can be added without extensive changes to the existing database schema, table structure, or data migration processes. This is particularly important in dynamic environments where data requirements frequently change, as it makes it easier to adapt and evolve the data structure (especially as data volumes grow or change.)
Furthermore, Neo4J has an intuitive visualization of data networks. Data is visualized directly as a graph with actual nodes and their relationships. The latter can be visualized clearly and comprehensibly which is particularly useful for use cases that are based on analyzing complex relationship structures, such as social networks or recommendation systems.
Lastly, Neo4J works with a powerful query language called Cypher, specially developed for graph-based data. Cypher makes it possible to formulate complex patterns and paths

in a simple and intuitive way, making it not only easy to learn but also well-suited for formulating advanced queries.

II. Implementation of the MONDIAL Data into Neo4J

First Step: Preparation and Exploration

The first step was to gain an overview of the MONDIAL database. With over 40 tables, the database is quite extensive and the relationships between the tables were not immediately apparent. To get started, the database generation files were downloaded from the MONDIAL project page of the Georg-August-University of Göttingen and a locally hosted relational database via pgAdmin was created. With the use of the provided schema file and input statements, the database was filled with data. After importing the data a comprehensive review of the data and tables was carried out. During this process, it became apparent that some tables primarily serve as linking tables rather than containing standalone data. For example, the table „located“ is only used to connect tables like Country, Province, City with other tables like River, Lake and Sea.
Since data can only be imported into Neo4J in a compatible format such as csv files, the next step involved extracting and downloading the relevant tables from the relational database. Therefore, a simple script was used to export the csv files from pgAdmin.

Second Step: Develop usable Schema based on Original Schema
To develop a suitable graph schema based on the original relational structure of the MONDIAL database, the following approach was taken.
Since Neo4J models data through nodes and relationships the visual component plays a central role in understanding and designing graph structures. The initial modeling phase was carried out using an online whiteboard platform named miro. This method offered the advantage of flexibility, allowing the schema to be easily adjusted and expanded as needed.
The visual layout also provided a clear and intuitive overview of the complex relationships between various entities, making the structure easier to plan and understand. Nodes could be directly linked and potential relationships between them could be illustrated visually before implementation process began.
Once the initial structure was outlined, a detailed analysis of the relationships between these nodes followed. These connections were then modelled to reflect the logical associations of the original relational schema and form the basis for the later implementation in Neo4J.

Thrid step: Data Import
After the modelling and a critical review of the composition of the relationships, the implementation began. In order to try out different modelling approaches and find suitable link structures, the first nodes and relationships were already imported during the development of the schema. This iterative approach made it possible to gradually refine the scheme. At the end of the development process, a comprehensive and well-structured graph schema was created that is optimally adapted to the requirements of the DBMS Neo4J.

In order to smoothly import all the files and cyber scripts written for the new schema, the automated database setup script was adapted from the already existing one. It controls the start of the database system, ensures correct initialization and then automatically executes prepared scripts to prefill the Neo4j database.
After activating job control in bash, Neo4j is started in the background via the docker-entrypoint.sh script. When the database is up the script will find all .cypher files at the specified path in docker and executed one after the other. Here the initialization of the database and the import of data is done. After successful processed the originally started Neo4j process will be brought back to the foreground and the container runs as it is supposed to.
During the process the script outputs simple text messages to the console to make the current status or progress visible.

III. Problems that occurred while modeling the schema

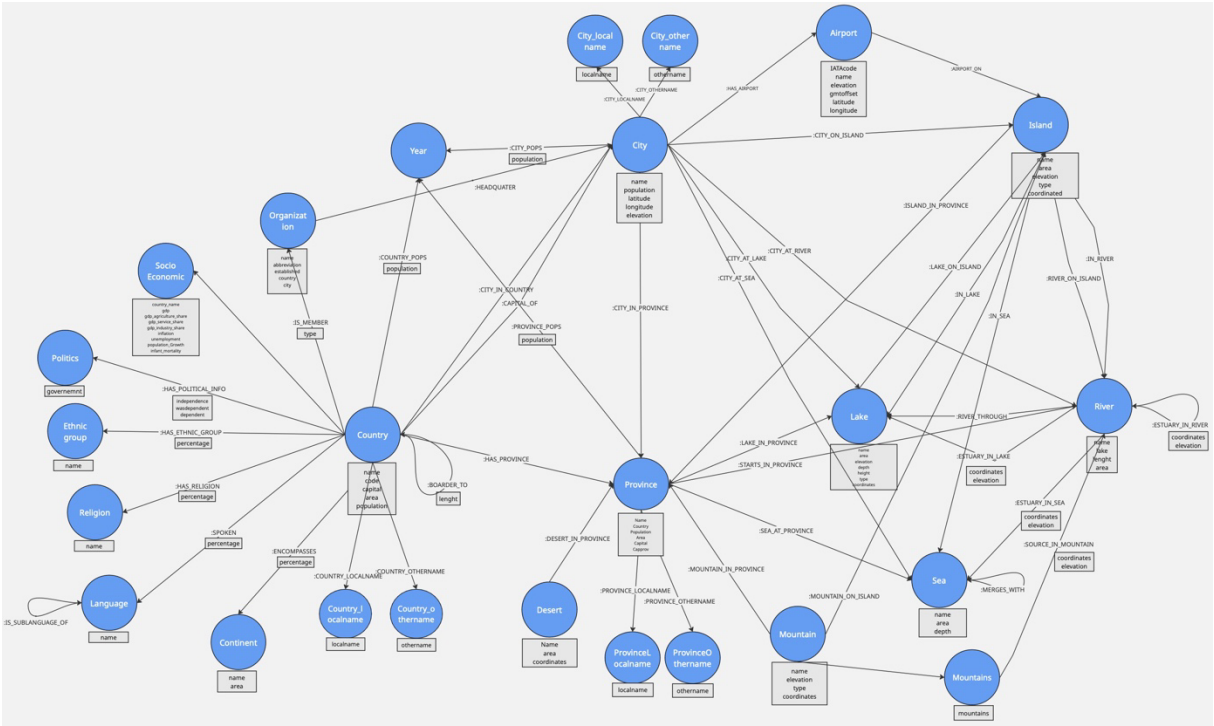
Handling of NULL-values
NULL values are handled differently in Neo4J than in relational databases. In Neo4J, null is not simply an empty value but means the property does not exist. Therefore, the use of null values requires a more thoughtful handling because importing, querying and comparing could lead to unexpected behavior in queries or data imports.
There are several approaches to address this issue when importing new data into Neo4J. The first would be to skip zero values with a WHERE clause. Only data that do not contain a null value are imported. However, this means that incomplete data records are completely ignored.
Another approach would be to definition a default value using the COALESCE function. Every missing property would then receive the same value (i.g. = 0). This brings the problem that one can no longer distinguish from actually 0 and just empty and might have other disadvantages when working with numbers.
However, for most data imports in this schema transformation, a different, more flexible and reliable method was used. By using CASE WHEN, before setting the value it is checked whether a value is empty and then make it null. In fact, Neo4j does not really see null values but rather treats them as strings with an empty text. And this approach check for exactly this.
For example, at the creation of the mountain node:
(...) SET n.name = CASE WHEN row.name IS NOT NULL AND trim(row.name) <> '' THEN trim(row.name) ELSE NULL END, (...); With this method for each attribute having a null value, importing it would have no lead to no information on a property. That really means that properties with null are not saved and the node or the relationship simply does not have this property.
This approach has the advantage that only valid, existing values are saved as a property. Missing values are skipped and not set as a property. In conclusion, node and relationships only contains relevant information.

Note on Data Quality:
When analyzing the data, I became aware that some entries were incorrectly coded. These errors were not caused by the import or export of the data but were already present

in the original generational files of MONDIAL. For example, Düsseldorf is stored as ‘DÃ¼sseldorf’ and Cologne as ‘KÃ¶ln’. These errors could lead to problem in the queries and confusion when working with the data mode.

IV. The Designed Schema

This schema was developed as the foundation for implementing the MONDIAL dataset in Neo4J. The link to this board is attached at the end of this document.



At the end of this document, a detailed list of all files used to create nodes and relationships can be found.

V. Reasoning for Design Choices

Direction of Relationships

In Neo4J relationships are always directed, so they always have a start node and a target node. However, in many cases, modelling in both directions is possible. Here the direction should be carefully chosen as this has an effect on readability of the graph and the clarity of queries.

A great example is the relationship between country and city. This could either be modelled as ‘A country has a city’ or as ‘A city is located in a country’.

In this case, the first relationship was chosen because the starting point is often the country. In addition, this direction is consistent with other relationship such as HAS_PROVINCE or HAS_LANGUAGE.

Naming of Relationships
Another key issue in the design process was the naming of the relationships between nodes. Especially in the case of relationships that perform geographical categorizations, such as LOCATED_IN. In Neo4j, it is technically possible to use the same relationship type for different contexts, such as for a city located in a country and at the same time for a lake located in a province.
On the downside, such design could lead to difficulties in understanding the data structure and spread confusion. Furthermore, it could lead to a decrease in query performance as Neo4J must evaluate multiple contexts for identically named relationships, increasing query complexity.

Adjustment of the Node Properties
Another design decision was to outsource certain node properties which are necessary in the relational model to connect tables together, but not needed in Neo4j. This restructuring follows the basic idea of Neo4J as a graph database, in which the relationships between the nodes are an important part in structure rather than individual nodes being equipped with many properties. For example, the country property in a city node was not copied in the schema transformation process. This information can now only be found by query the relationship from city to country.
This adjustment improves the semantic meaningfulness and network visualization of the data. The focus is here on linking information through relationships, which fully reflects the strengths of Neo4J as a graph database. This adjustment was done for all relevant nodes.

Property-bearing relationships (relationship with properties)

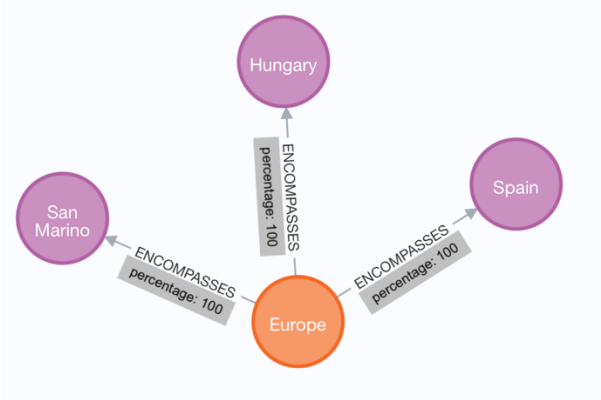
a) Example with Country and Organization (is_member)
In order to illustrate the design decisions made in the schema, it is started to describe the nodes with relationships which themselves have properties. These relationships contain additional information about the connection between the nodes.
A good example is the relationship between a country and an organization node. In PostgreSQL, these two entities are created from separate tables that can be linked together using a join command. An additional link can also be created using the ‘ismember’ table, which provides further information on the relationship.
However, in the schema for Neo4J the country and organization nodes are linked to each other in the form of a directed relationship that is named IS_MEMBER. Additionally, the relation’s name makes it intuitively clear that a country is a member of an organization.
A significant difference to the relational database structure, which has already been mentioned, is that relationships can have their own properties in Neo4J. The relationship between the nodes is not just a simple link but carries additional information. The IS_MEMBER relationship contains the property ‘type’ (of membership) and can have different forms like member’, ‘regional member’, ‘non-regional member’ etc. By storing the membership type on the relationship itself, the relationship gets an additional property for the relationship and at the same time offers better visual representation without additional complexity.
With this approach it is not necessary to create a separate organization node with an individually stored type for each country. Instead, several Country nodes can be linked to

the same organization where the respective type of membership is stored directly as a property in the relationship.

b) Example with Country und Continent (encompasses)

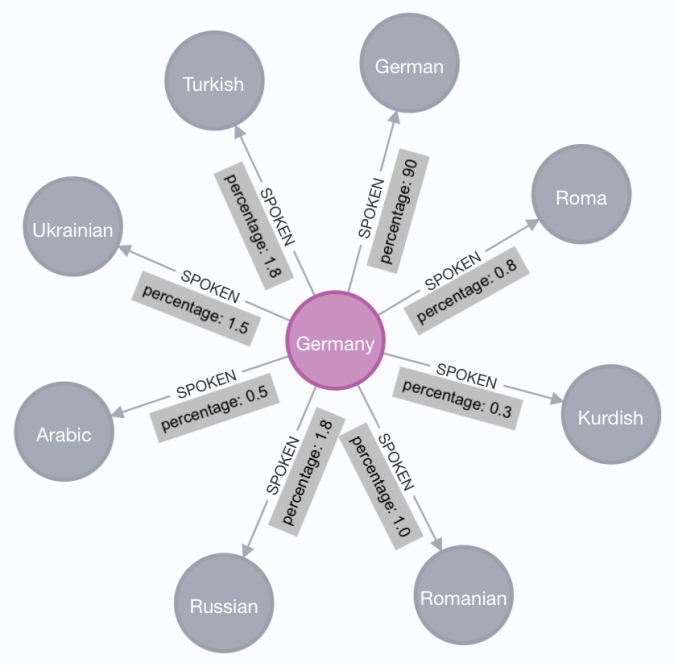
The connections between Country and Language, as well as between Country and Continent, are modeled in a similar way. In both cases the relationship was designed using a third table functioning as an intermediary and simultaneously provide an additional, meaningful property that enriches the relationship in the graph model.

The relationship between Country and Continent was modelled using the relationship „encompasses“. It gives the relationships an additional property, displaying the percentage of the country area that is located on a continent. This modeling approach allows a more informative representation of the data and therefore makes full use of Neo4J's ability to store properties directly on relationships.



c) Example with Country und Language

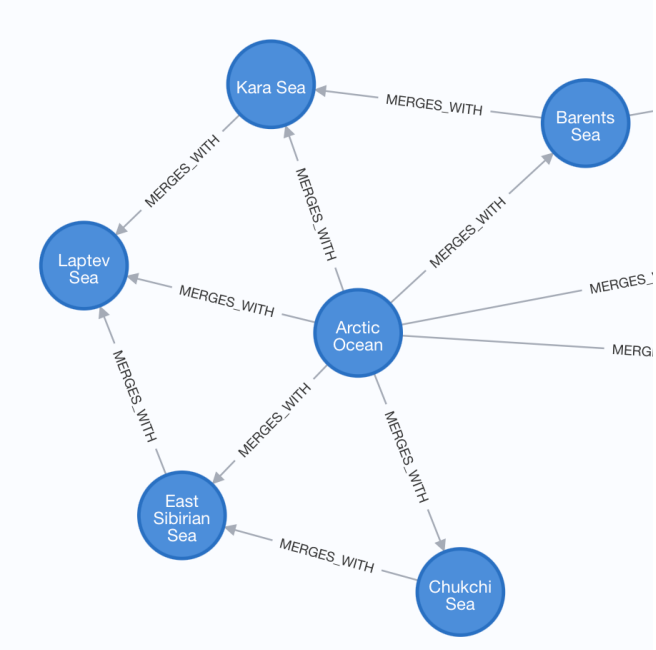
The connection between country and language was connected using a relationship with the percentage property from „spoken.csv“. This design shares the same advantages.



d) Example with Borders between Countries and Sea's merging

Another special aspect of the schema transformation involved the handling of the two tables 'mergeswith' (which indicates when two seas merge) and 'borders' (which describes the borders between two countries along with the length of the border) that only consists of the same entity in both collum's. The relationships connect the same nodes and were therefore implemented as a bidirectional connection. To reflect this symmetry, the relationship was created in both directions. Additionally, the length of the border was stored as a property directly on the relationship.

The same approach was applied to „mergeswith" between sea nodes. The relationship was also modeled in both directions to accurately reflect the mutual connection.



Relationships without extra csv Files for Properties

a) Example with Country and Religion

In some cases, there is no suitable table/csv file serving as a property for a relationship between two nodes. Nevertheless, a relationship between the nodes must be modelled. Therefore, one of the respective node files is used acting as the basis for the relationship while in most cases these relationships have no additional properties.

In some cases, however, one of the tables used for creating a node, contains information that is much better placed in the context of a relationship than in the node itself. In the linkage between country and religion this was realized as follows. In addition to the country and religion name, the underlying religion table also contains percentages that indicate how strongly a religion is represented in a particular country.

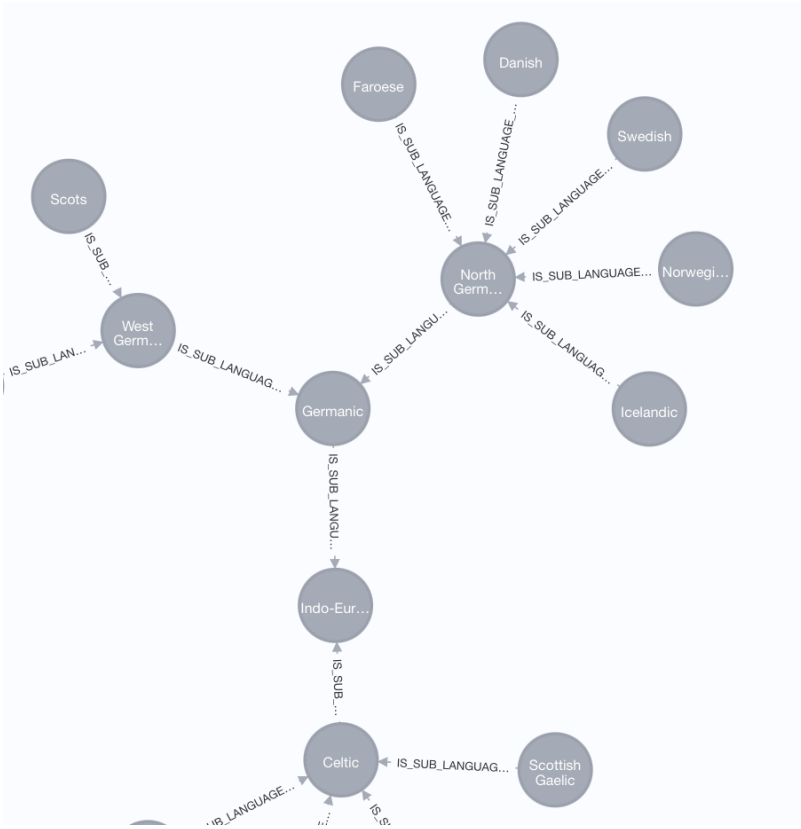
Since countries and religions both appear several times in the table to show the proportion of each religion within a country, it makes much more sense for the Neo4J schema to model the religions as independent nodes and use the respective share as a property in the relationship. Consequently, each religion node is only created once and

can be linked to any number of countries. Like in the examples above the percentage share is now visible on the edge making the schema more efficient and less redundant.

b) Example with Country and EthnicGroup
The same was done with the relationship between country and ethnicgroup, using the percentage share from the table ethnicgroup as property for the relation.

Network structures and Hierarchical Relations

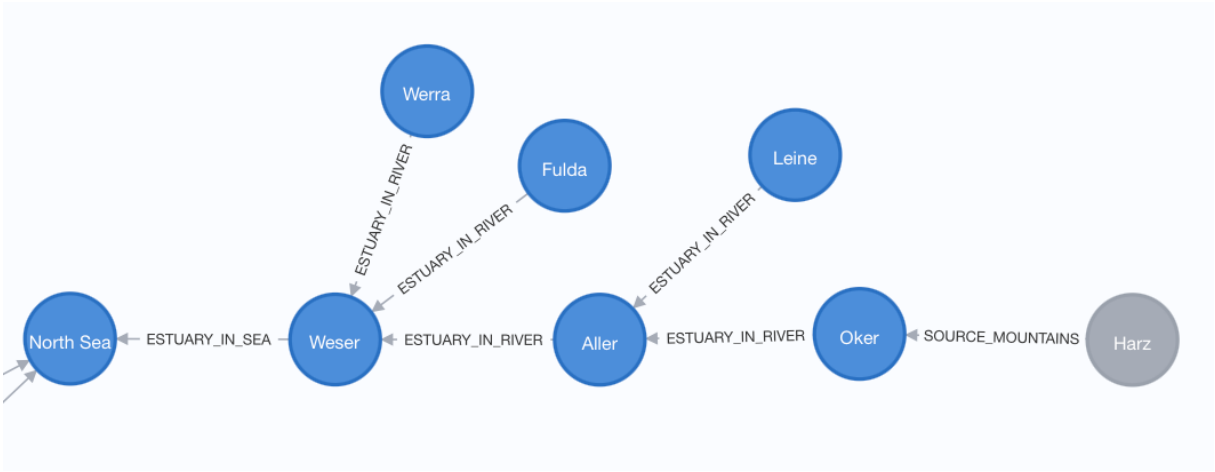
a) Modelling Hierarchical Relations: Language Structure
In the modelling of the language structure in Neo4J, the hierarchical relationship between languages was done by a directed relationship of the type 'IS_SUB_LANGUAGE_OF'.
Each language is represented as a separate 'language' node, while the information about the superordinate language (superlanguage) is no longer stored as a property in the node itself but put in the relationship between two language nodes.
This modelling follows the strengths of graph databases, using the connections between data points. By mapping the language hierarchy with explicit relationships between the language nodes it is easier comprehensible and has a more "natural" graph structure.
In this way, language families can be represented in the form of a tree or network in which both subordinate languages and super languages can be found efficiently.



b) Modelling Networks: Rivers with their Source and Estuaries

With the river table the flow of rivers could get transferred and visualized in the graph network. The table contains various information on rivers, including their name, length and the estuary of the river that can be a sea, a lake or another river. Depending on the type of estuary a relationship for each was created, river flows into a sea, river flows into a lake and river flows into another river. The information on the estuary itself like coordinates and altitude are not attached to the river node but saved directly as properties of the relationship. A great advantage of this modelling is that it can be visualized because the graph model makes the geographical and logical connections visible. One can immediately see which river flows into where or how water systems extend across country borders.

The modelling of the river network with its sources and estuaries shows how well network data can be represented graphically with Neo4j. This example shows that not only a simple relationship can be created but also rather complex and natural networks. With the river network starting at a source, running through various estuaries into other rivers and finally ending in a sea or lake. This structured modelling is much clearer and more intuitive than the tabular representation in a relational schema. It enables a great visuality and easily comprehensible understanding of the flow structure of watercourses.



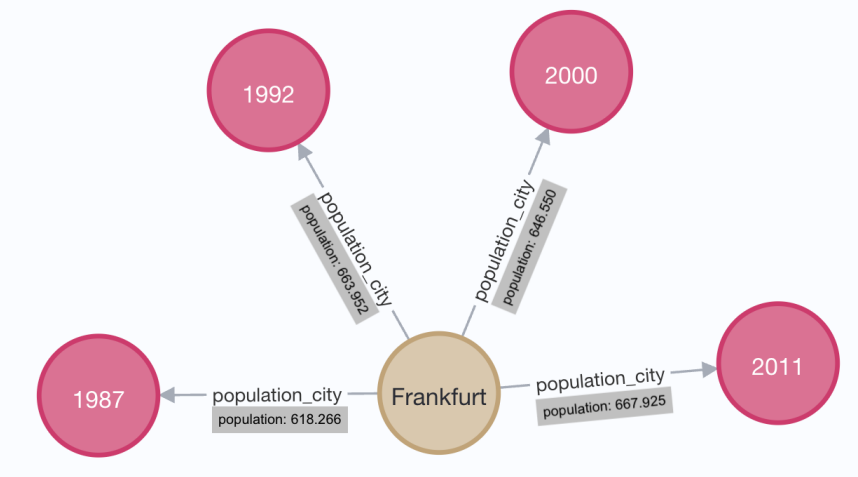
c) Restriction when attaching Province Data to Relationships

During the modelling a problem occurred up with the geo_estuary.csv file which contains the province in which the estuary of a river is located. Initially, this location information should get to attach as an additional property to the existing relationship between the river and its opposite. However, this approach was not optimal. The province attribute is not part of the relationship between a river and its estuary and rather describes a specific location in geographical space. With the province as property there wouldn't be a connection with the province node possible and therefore also no queries. To still be able to model this information in the graph network a separate relationship between the river and the province node was created.

Structuring Time in Graphs: Introducing Year Nodes

Dynamic Data with the 'Year' Node

In the PostgreSQL scheme there are dynamic population data for cities (City), provinces (Province) and countries (Country). These are stored in the citypops, provpops and countrypops tables. However, this data is not uniformly structured: Each entitity has a different number of population entries, each of which is linked to a specific year. In addition, the year numbers themselves differ depending on the city, which makes structuring for the transformation difficult.

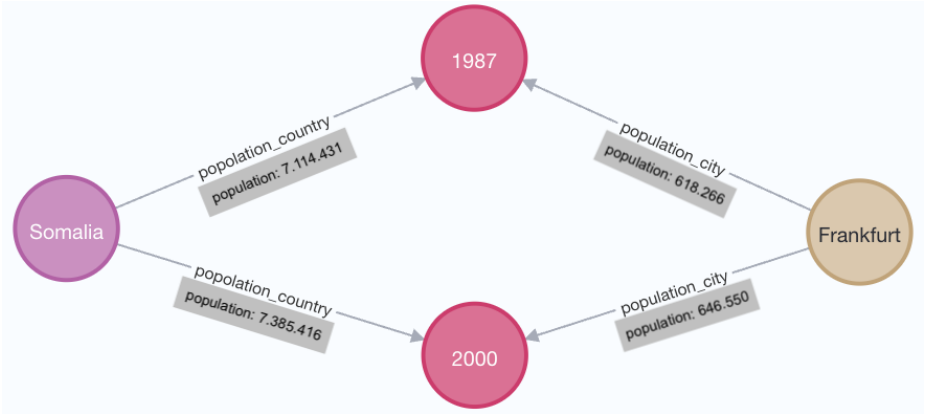


In Neo4J, the year dates must be modelled either as separate nodes or as properties within relationships. However, the inconsistent structure of the data makes a clear and consistent implementation in the graph schema difficult. Challenging is that the years are not standardized and vary depending on the city, province or country. This leads to increased effort in managing the data and would make time-related queries significantly more complex.

In order to model the population in relation to the respective year in Neo4j it was decided to create a new separate year node. This node contains all the years used and offers a structured allocation. For all three nodes (city, province and country) a respectively relationship to the year node was created and each given the property of the population (e.g. countrypops, provincepops, citypops). Because of the low number of years with population data (no more than 6, average 4) that a country, city, province has this approach is still well organized and not chaotic.

This solution allows the dynamic population data to be represented efficiently without unnecessarily complicating the node structure.

It is particularly efficient as it avoids redundancy. Each year value need to exist just once (as a node) and several countries, provinces and city's can be linked to it. At the same time the schema remains clear and time-related queries can be carried out more easily and with better performance.



Nodes and properties

a) Merging Tables together: SocioEconomic Node

When transferring the data from the PostgreSQL schema to Neo4j, an important focus was on avoiding redundancy and increasing the clarity of the graph model.
Instead of creating a separate node for each individual statistic it was tried to not create an unnecessarily large number of nodes without them having a value.
As there were two tables in the original SQL schema, one with economic and one with demographic information it was decided to combine both in a single node named 'SocioEconomic'. Economic and demographic data are often analyzed together and are closely linked.
The new node contains economic attributes such as GDP, composition of the GDP inflation rate and unemployment rate from the economy table as well as demographic characteristics such as population growth and infant mortality from the population table.
Both tables contained data for only a single year (current year?), making them easy to combine.

b) Adding Node Properties to another Node

In the relational schema of the MONDIAL database, information about local and alternative names (xx_localname, xx_othername) is stored in separate tables.
In the schema transformation to Neo4J these tables were modeled as an independent node and connected to the corresponding main node via clearly named relationships. These relationships do not carry any additional properties.
However, it was considered whether integrating this information directly as properties into the existing City, Province, and Country nodes might be more efficient. This idea was based on the fact that the original tables contain only one relevant attribute for the Neo4J schema. This approach would have had several advantages like reducing the overall number of nodes. Additionally, accessing local or alternative names would have been faster and more direct.
On the other hand, multiple entries of several local or other names per node would have led to modelling problems. For example would there have been several cities with more than one local name. That why the decision was made to not integrate these names into the nodes, clearly separate them and letting the nodes only contain centralized information.

VI. Conclusion

The transformation of the Mondial dataset into a Neo4j data model proved to be very successful overall.
The ability to store properties directly on relationships between nodes helped with a smooth transformation and it possible to transfer many tables from the relational schema to a more intuitive and flexible structure.
However, challenges also arose, particularly in the integration of time-dependent data. The use of year nodes as a solution was functional, but not ideal in terms of clarity and performance for larger time series.
The modelling worked best with the river system. Here the full potential of the graph-based representation was revealed. Unlike in the relational approach, where the relationships within the river table were obscure, the natural network of sources, estuaries and connected rivers were modelled clearly and were visually comprehensible with Neo4j.

VII. Listing of all used Nodes and Relationships

Nodes	Used .csv Files
Airport	airport.csv
City	city.csv
City_localname	citylocalname.csv
City_othername	cityothername.csv
Continent	continent.csv
Country	country.csv
Country_localname	countrylocalname.csv
Country_othername	countryothername.csv
Desert	desert.csv
Ethnicgroup	ethnicgroup.csv
Island	island.csv
Lake	lake.csv
Language	language.csv
Mountain	mountain.csv
Mountains	mountain.csv
Organization	organization.csv
Politics	politics.csv
Province	province.csv
Province_localname	provincelocalname.csv
Province_othername	provinceothername.csv
Religion	religion.csv
River	river.csv
Sea	sea.csv
SocioEconomic	economy.csv / population.csv / country.csv
Year	year.csv

Relationship	Connected trough
BORDER_TO	Country -> Country
COUNTRY_LOCALNAME	Country -> Country_localname
COUNTRY_OTHERNAME	Country -> Country_othername
CITY_LOCALNAME	City -> City_localname
CITY_OTHERNAME	City -> City_othername
PROVINCE_LOCALNAME	Province -> Province_localname

PROVINCE_OTHERNAME	Province -> Province_othername
HAS_ETHNIC_GROUP	Country -> Ethnicgroup
HAS_RELIGION	Country -> Religion
SPOKEN	Country -> Language
POPULATION_CITY	City -> Year
POPULATION_COUNTRY	Country -> Year
POPULATION_PROVINCE	Province -> Year
ENCOMPASSES	Country -> Continent
IS_MEMBER	Country -> Organization
DESERT_IN_PROVINCE	Desert -> Province
MOUNTAIN_IN_PROVINCE	Mountain -> Province
PROVINCE_IN_COUNTRY	Province -> Country
CITY_IN_COUNTRY	City -> Country
HAS_ECONOMY	Country -> Economy
HAS_POLITICS	Country -> Politics
SEA_IN_PROVINCE	Sea -> Province
MERGES_WITH	Sea -> Sea
AIRPORT_NEAR_CITY	Airport -> City
AIRPORT_IN_COUNTRY	Airport -> Country
AIRPORT_ON_ISLAND	Airport -> Island
ESTUARY_PROVINCE	River -> Province
STARTS_IN_PROVINCE	River -> Province
CITY_ON_ISLAND	City -> Island
LAKE_IN_PROVINCE	Lake -> Province
ISLAND_IN_PROVINCE	Island -> Province
RIVER_THROUGH	River -> Lake
CITY_AT_RIVER	City -> River
CITY_AT_LAKE	City -> Lake
ISLAND_IN_SEA	Island -> Sea
CITY_AT_SEA	City -> Sea
ISLAND_IN_RIVER	Island -> River
ISLAND_IN_LAKE	Island -> Lake
RIVER_ON_ISLAND	River -> Island
HAS_SOCIOECONOMIC	Country -> SocioEconomic
CAPITAL_OF	City -> Country
HEADQUARTERS_IN	Organization -> City
IS_SUB_LANGUAGE_OF	Language -> Language

PART_OF	Mountain -> Mountains
SOURCE	Mountains -> River
ESTUARY_IN_RIVER	River -> River
ESTUARY_IN_SEA	River -> Sea
ESTUARY_IN_LAKE	River -> Lake

Link to the miro board:

https://miro.com/app/board/uXjVIJYr9Eg=/?share_link_id=503601218203

https://miro.com/welcomeonboard/Q2Rpd0ZneFR1eVhwNkpVRXo3NVE2ckw4VUFEa2Y2dkpNbXV1emVwVFgrWFBnRUlrUUtIVjFGcmZXdzRPSjRkOTFZUk00d2xsSHBxRDl5czNmOWY3dFdINTNuYm96QWJ4Mk10VzJNnVoN2hISkN1NkNjaHVEbjhmYUJubm5FTTVhWWluRVAxeXRuUUgwWDl3Mk1qRGVRPT0hdjE=?share_link_id=116228941639

Sources:

[1] https://neo4j.com/docs/getting-started/whats-neo4j/